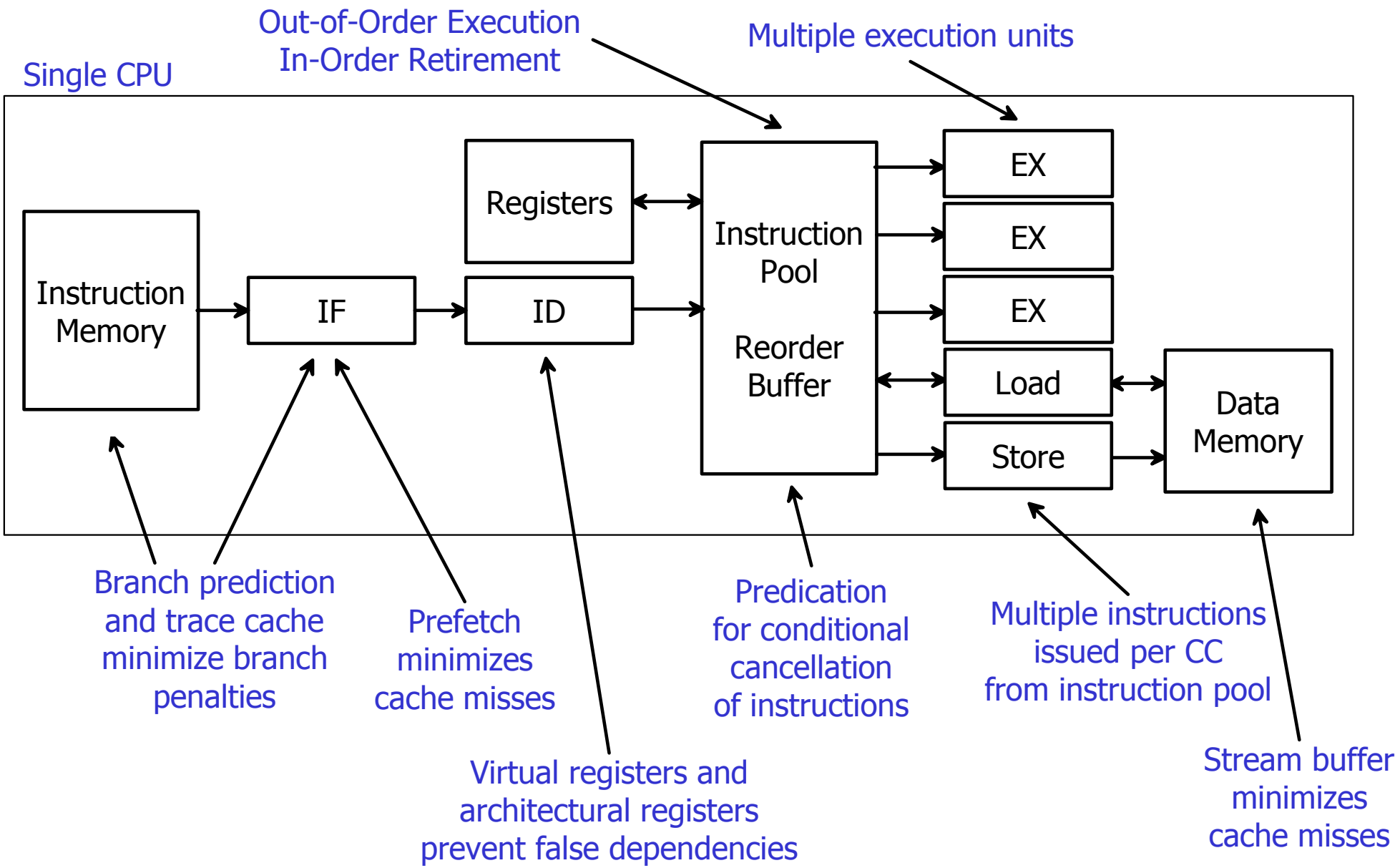
Beyond Instruction Level Parallelism

Summary of Superscalar Processing



ILP Scalability Limit

Scaling instruction window and decoder rate

execution units $u_i \rightarrow u_i' = \alpha_u u_i$

pipeline stages $s_i \rightarrow s_i' = \beta_s s_i$

instruction window $IC_{EU} \rightarrow IC_{EU}' = \alpha_u \beta_s IC_{EU}$

$$\lambda^{ideal} \rightarrow \lambda^{ideal\,\prime} = \frac{\left(\beta_s \alpha_u^2\right) s \times u^2}{1 + \left(\beta_s \alpha_u\right) s \times u}$$

Scaling 6 → 15 EUs with 2 → 8 superpipelined stages

$$\alpha_u = \frac{15}{6} \quad \beta_s = \frac{8}{2} \quad \Rightarrow \quad \alpha_u \times \beta_s = 10$$

$IC_{EU} = 120$ instructions executing in parallel

$15 > \lambda^{ideal} \geq 14.9$ instructions decoded per CC

Difficulties

Decode 15 instructions per CC

- Despite cache misses, mispredictions, …

Maintain window of 120 independent instructions

- Branches ≈ 20% of instructions
- 25 – 30 branches in window ⇒ large misprediction probability

Require larger source of independent instructions

Exploit inherent parallelism in software operations

Sequential and Parallel Operations

Programs combine parallel + sequential constructs

- High-level job → model-dependent sections
  - Processes
  - Threads
  - Classes
  - Procedures
  - Control blocks
- Sections compiled → ISA = low level CPU operations
  - Data transfers
  - Arithmetic/logic operations
  - Control operations

High-level job → execution

- Machine instructions — small sequential operations
  - Local information on 2 or 3 operands
- CPU cannot recognize abstract model-dependent structures
- Information about inherent parallelism lost in translation to CPU

Parallelism in Sequential Jobs

Concurrency in high-level job

Two or more independent activities in defined to execute at same time

- Parallel — execute simultaneously on multiple copies of hardware
- Interleave — single hardware unit alternates between activities

Example

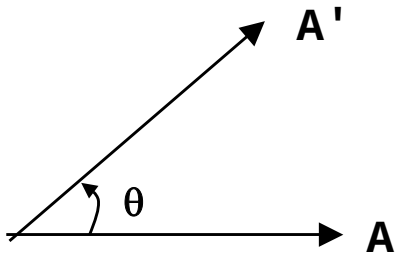
- Respond to mouse events
- Respond to keyboard input
- Accept network message

Functional concurrency

Procedure maps $\mathbf{A' = R(\theta) \times A}$

Code performs sequential operations

$$A_x' = A_x \cos\theta + A_y \sin\theta$$
$$A_y' = -A_x \sin\theta + A_y \cos\theta$$

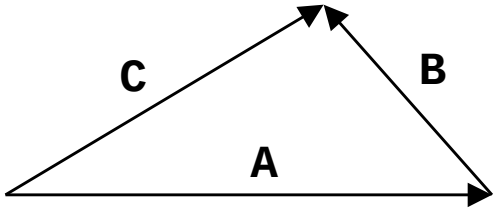


Data concurrency

Procedure maps `C = A + B`

Code performs sequential operations

```
for (i = 0, i < n, i++) C[i] = A[i] + B[i]
```



Extracting Concurrency in Sequential Programming

Programmer

- Codes in high level language
- Code reflects abstract programming models
  - Procedural, object oriented, frameworks, structures, system calls, ...

Compiler

- Converts high level code to sequential list
  - Localized CPU instructions and operands
- Information about inherent parallelism lost in translation

Hardware applies heuristics

- Partially recover concurrency as ILP

Technique	Concurrency Identified / Reconstructed
Pipelining	Parallelism in single instruction execution
Dynamic scheduling superscalar	Operation independence
Branch and trace prediction	Control blocks
Predication	Decision trees

Extracting Parallelism in Parallel Programming

Programmer

- Identifies inherently parallel operations in high level job
  - Functional concurrency
  - Data concurrency
- Translates parallel algorithm into source code
- Specifies parallel operations to compiler
  - Parallel threads for functional decomposition
  - Parallel threads for data decomposition

Hardware

- Receives deterministic instructions reflecting inherent parallelism
  - Code + threading instructions
- Disperses instructions to multiple processors or execution units
  - Vectorized operations
  - Pre-grouped independent operations
  - Thread Level Parallelism

The "Old" Parallel Processing

1958 — research at IBM on parallelism in arithmetic operations

1960 – 1980

- Mainframe SMP machines with N = 4 to 24 CPUs
- OS dispatches process from shared ready queue to idle processor

1980 – 1995

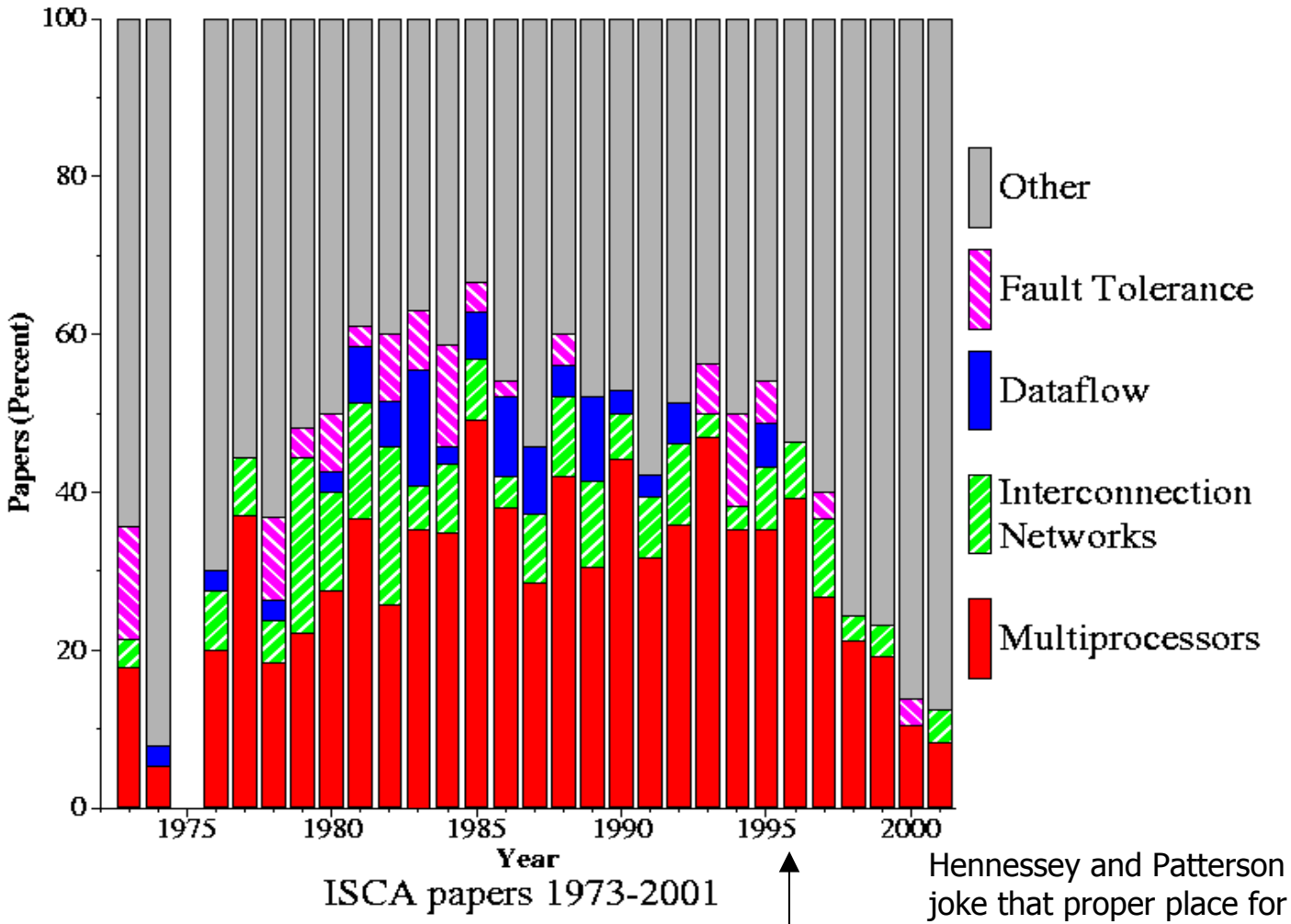
- Research boom
- Automated parallelization by compiler
  - Limited success — compilers cannot identify inherent parallelism
- Parallel constructs in high level languages
  - Long learning curve — parallel programmers are typically specialists
- Inherent complexities
  - Processing and communication overhead
    - Inter-process message passing — spawning/assembling with many CPUs
    - Synchronization to prevent race conditions (data hazards)
  - Data structures
    - Shared memory model
    - Good blocking to cache organization

1999 — fashionable to consider parallel processing a dead end

Rise and Fall of Multiprocessor R&D

Topics of papers submitted to ISCA 1973 to 2001

Sorted as percent of total



ISCA papers 1973-2001

Hennessey and Patterson joke that proper place for multiprocessing in their book is Chapter 11 (a section of US business law on bankruptcy)

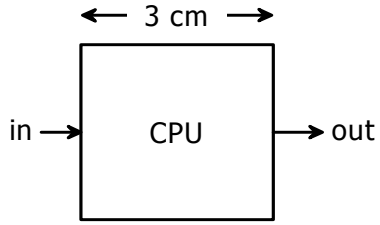
ISCA — International Symposium on Computer Architecture

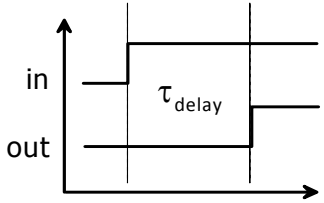
Ref: Mark D. Hill and Ravi Rajwar, "The Rise and Fall of Multiprocessor Papers in the International Symposium on Computer Architecture (ISCA)", http://pages.cs.wisc.edu/~markhill/mp2001.html

It's Back — the "New" Parallel Processing

Crisis rebranded as opportunity

Processor clock speed near physical limit (speed of light = 3×10^{10} cm/s)





$$\tau_{delay} > \frac{3 \text{ cm}}{3 \times 10^{10} \text{ cm/sec}} = 10^{-10} \text{ sec}$$

$$\tau_{clock} \approx 10^{-10} \text{ sec} \Rightarrow R_{max} < 10^{10} \text{ Hz} \approx 10 \text{ GHz}$$

Heating

- Clock rate ↑ ⇒ heat output ↑
- CPU power ↑ ⇒ chip size ↑ ⇒ heat transfer rate ↓ ⇒ CPU overheats

Superscalar ILP cannot rise significantly

- Instruction window ~ 100 independent instructions

"Old" parallel processing is not sufficient

Some interesting possibilities

- Multicore processors cheaper and easier to manufacture
- User level thread management
- Multithreaded OS kernels and OS level thread scheduling
- Compiler support for thread management APIs
- New debugging tools

Processes and Threads

Process

One instance of an independently executable program

Basic unit of OS kernel scheduling (on traditional kernel)

Entry in process control block (PCB) defines resources

- ID, state, PC, register values, stack+memory space, I/O descriptors, …

Process context switch → high volume transfer operation

Organized into one or more owned threads

Thread

One instance of independently executable instruction sequence

Not organized into smaller multitasked units

Limited private resources — PC, stack, and register values

- Other resources shared with other threads owned by process

Scheduled by kernel or threaded user code

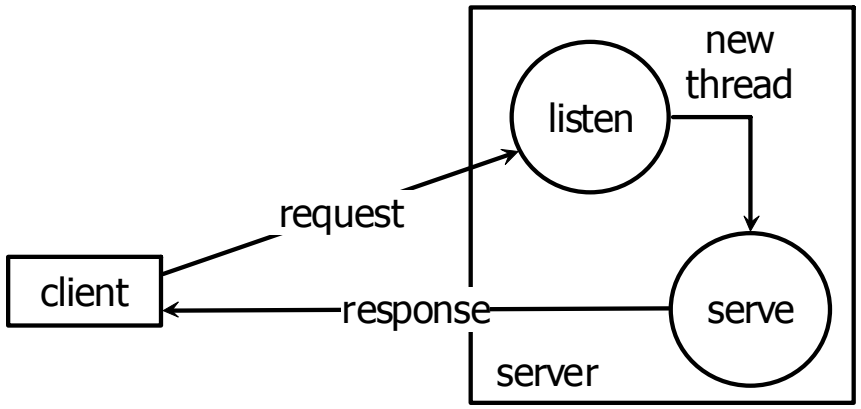
Thread switch → low volume transfer operation

Multithreaded Software

Threaded OS kernel

- Process = one or more threads
- Multithreaded application
  - Organized as more than one thread
  - Threads scheduled by OS or application code
  - Not specific to parallel algorithms

Classic multithreading example



- Multithreaded web server
  - Serves multiple clients
  - Creates thread per client
- Server process creates **`listen`** thread
  - **`Listen`** thread blocks — waits for service **`request`**
  - Service **`request`** → **`listen`** thread creates new **`serve`** thread
- **`Serve`** thread handles web service **`request`**
- **`Listen`** thread returns to blocking

Decomposing Work

Decomposition

Break down program into basic activities

Identify dependencies between activities

"Chunking" — choose size parameters for coded activities

Functional Decomposition

Each thread assigned different activity

Example — 3D game

- Thread 1 updates ground
- Thread 2 updates sky
- Thread 3 updates character

Data Decomposition

Each thread runs same code on separate block of data

Example — 3D game

- Divide sky into n sections
- Threads 1 — n update section of sky

Hardware Implementation of Multithreading

No special hardware requirements

Multithreaded code runs on single / multiple CPU system

Run-time efficiency depends on hardware/software interaction

Coarse-grained multithreading

Single CPU swaps among threads on long stall

Fine-grained multithreading

Single CPU swaps among threads on each clock cycle

Simultaneous multithreading (SMT)

Superscalar CPU pools instructions from multiple threads

Enlarges instruction window

Hyper-Threading

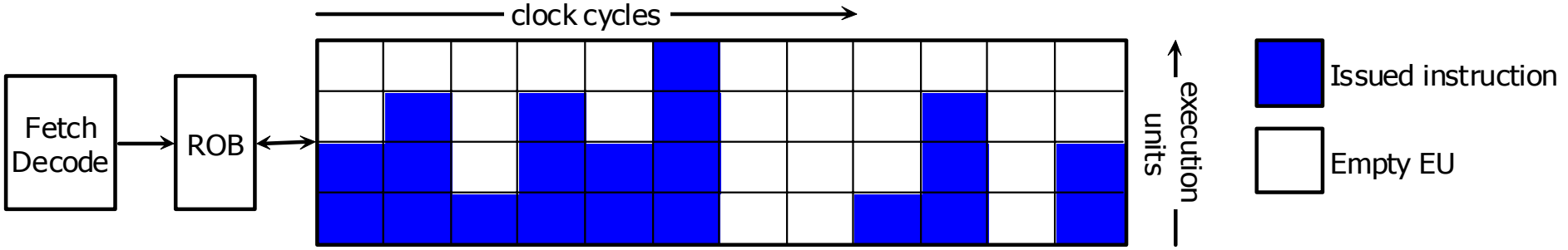
Intel technology combining fine-grained multithreading and SMT

Multiprocessing

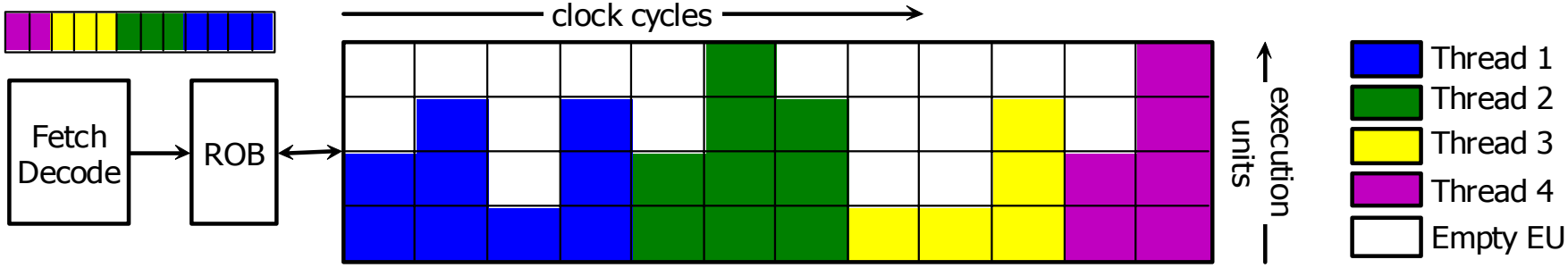
Dispatches threads to CPUs

Superscalar CPU Multithreading

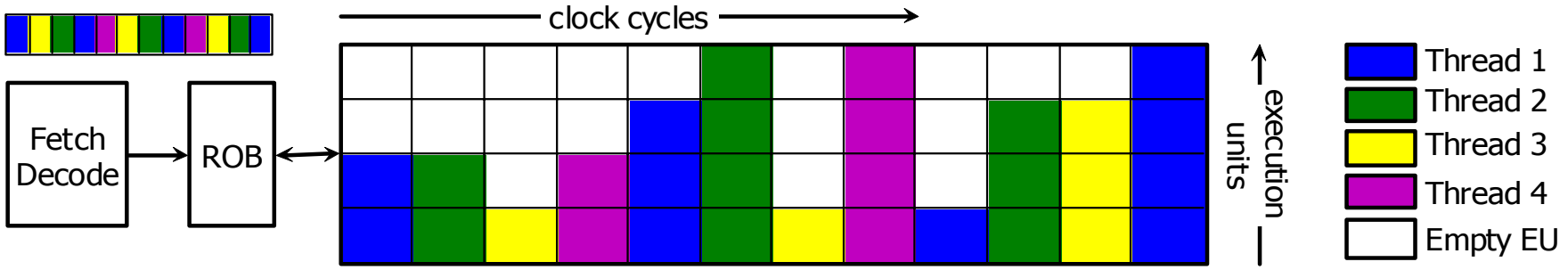
Single thread on superscalar



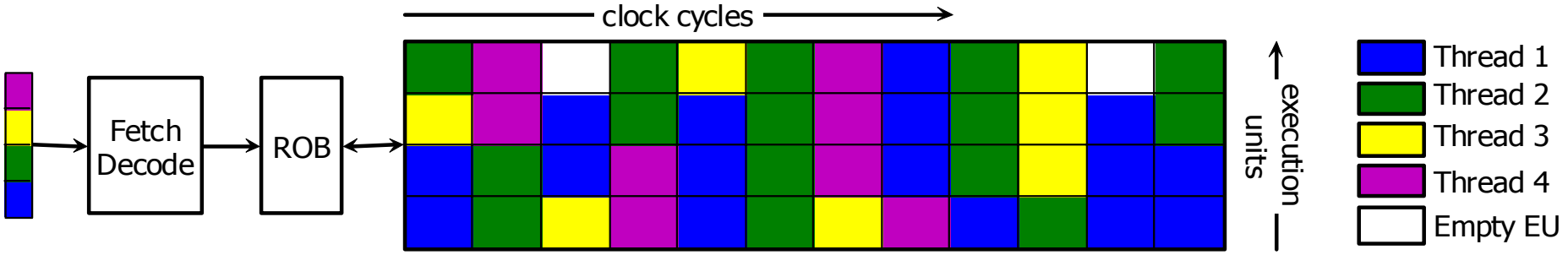
Course grained multithreading on superscalar



Fine grained multithreading on superscalar



Simultaneous Multithreading



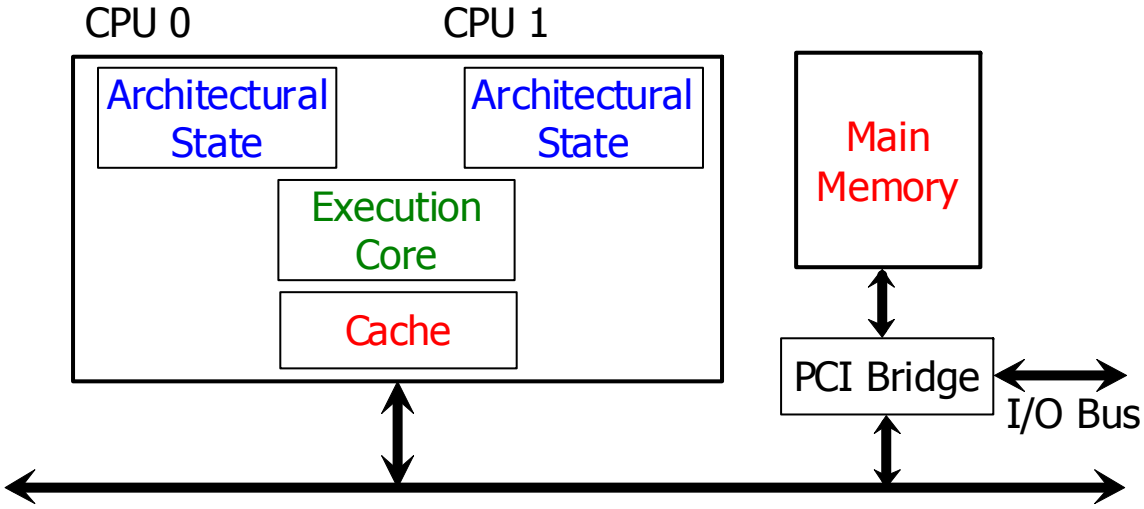
Simultaneous multithreading on superscalar

- Pool instructions from multiple threads
- Instructions labeled in reorder buffer (ROB)
  - PC
  - Thread number
  - Operands
  - Status
- Large instruction window

Advantage on mispredictions

- Only thread with misprediction is cancelled
  - Other threads continue to execute
- Cancellation rate from mispredictions → ¼ single-thread cancellation rate

Hyper-Threading



Architectural State
Registers, stack pointers and program counter

Execution Core
ALU, FPU, vector processors, memory unit

Two copies of architectural state + one execution core

- Fine grained N = 2 multithreading
- Interleaves threads on In-Order fetch/decode/retire units

Issue instructions to shared Out-of-Order execution core

- Simultaneous N = 2 multithreading (SMT)
- Executes instructions from shared instruction pool (ROB)

Stall in one thread ⇒ other thread continues

- Both CPUs keep working on most clock cycles
- Advantage of course-grained N = 2 multithreading

Thread Coexistence

Multiprocessor code

- Provides source of independent instructions
- Permits high processor utilization

Independent applications running in parallel

- Unrelated instructions with no data dependencies
- Independence can create resource conflicts
  - Require different data blocks in cache
  - Use different branch prediction cache and trace cache

Parallel threads of single application

- Different pieces of same program
- Run in coordinated fashion
  - Communicate, synchronize, exchange data
- Stall in thread can stall related threads
  - Cache miss, page fault, branch misprediction, ...

Helper Thread Model

Performs no committed work

- Do not change any program result
- Results not committed to memory
- Require no additional hardware support

Performs loads and branches that appear in work thread

- Encounter cache misses before work thread
- Prepares caches
- Prevents costly misses

Helper Thread Example

Example

```
L:   MUL R4, R6, R8
     ADD R4, R6, R9
     ADD R1, R2, R3
     SUB R3, R4, R5
     LW R6, 0(R1)       ; cache miss
     ADD R6, R3, R2
     BEZQ R6, L         ; misprediction
```

Work Thread	Helper Thread
`L: MUL R4, R6, R8`	`L: ADD R1, R2, R3`
`ADD R4, R6, R9`	`LW R6, 0(R1)` ; cache miss
`ADD R1, R2, R3`	; cache update
`SUB R3, R4, R5`	`BEQZ R6, L` ; misprediction
`LW R6, 0(R1)` ; no cache miss	; update predictor
`ADD R6, R3, R2`	
`BEZQ R6, L` ; no misprediction	

Flynn Taxonomy for CPU Architectures

	Instruction	
Data	Single Instruction Single Data **SISD**	Multiple Instruction Single Data **MISD**
	Single Instruction Multiple Data **SIMD**	Multiple Instruction Multiple Data **MIMD**

SISD

Standard single CPU machine with single or multiple pipelines

SIMD

Vector processor or processor array

Performs one operation on data set on each CC

MISD

Perform multiple operations on one data set each CC

Few products — IBM Watson IA applies multiple algorithms to same data

MIMD

Multiprocessor or cluster computer

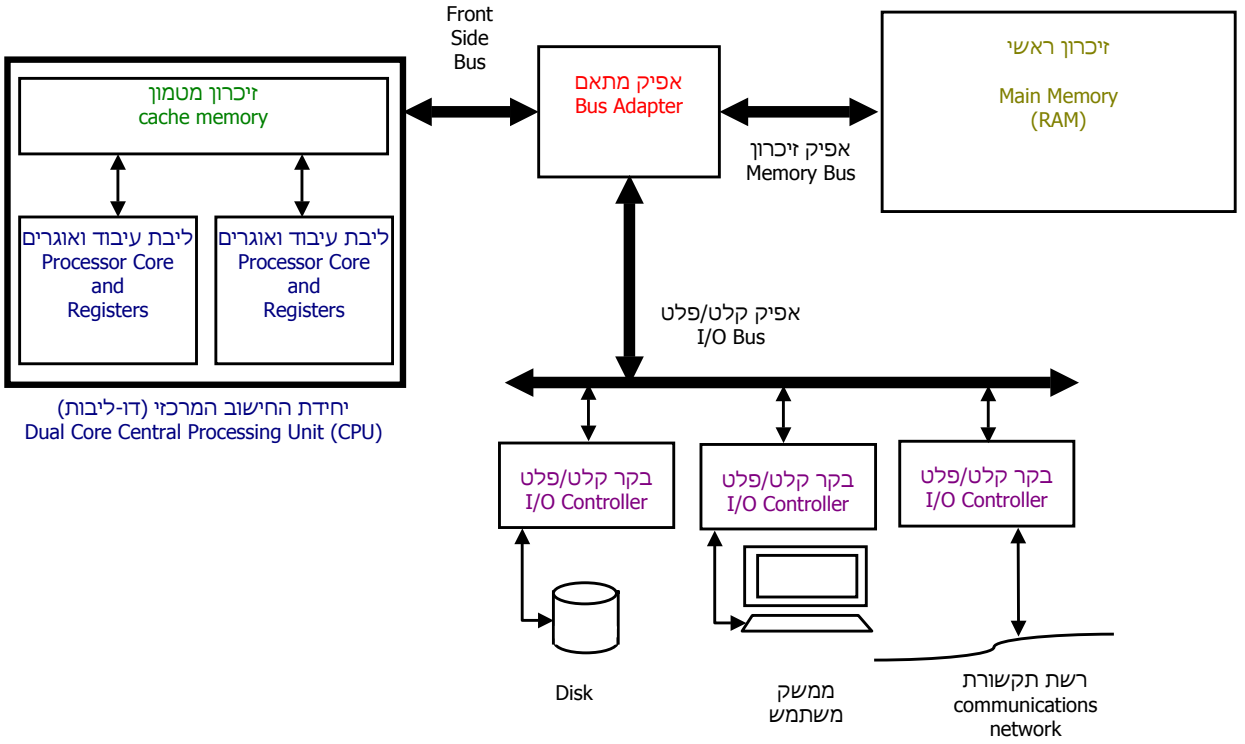
Perform multiple operations on multiple data sets on each CC

Ref: M.J. Flynn, "Very High-Speed Computers", Proceedings of the IEEE, Dec. 1966.

Multiprocessor Architecture

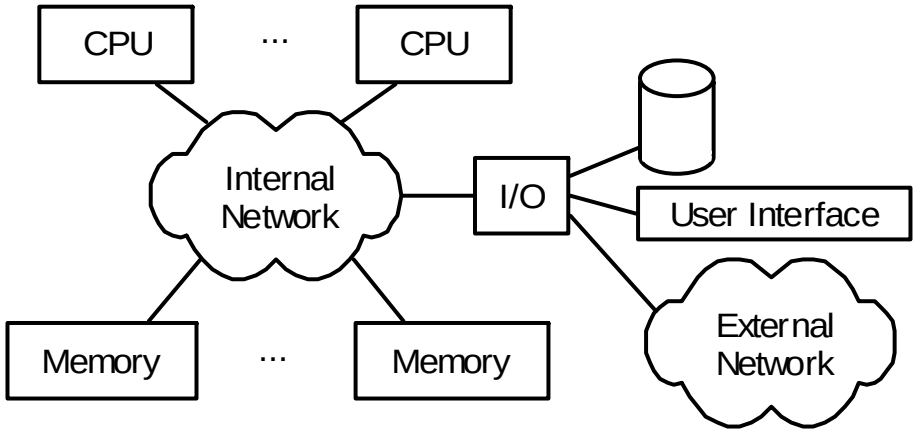
SISD/SIMD workstation

- Dual core CPU
  - Architectural registers
  - Cache
  - Execution units
- I/O system
  - Long-term storage
  - Peripheral devices
  - System support functions
- Main memory
- Internal network system



MIMD multiprocessor

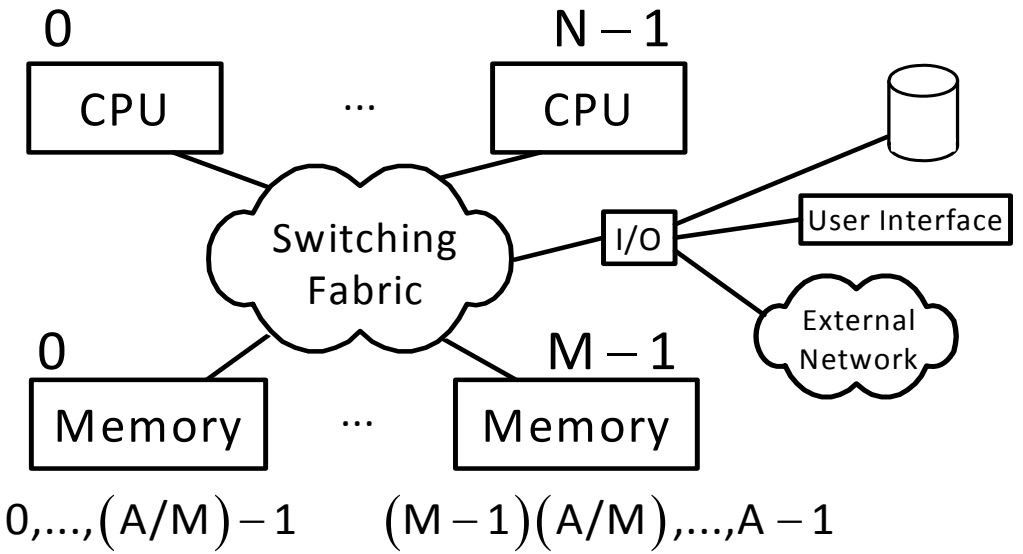
- Multiple CPUs
- I/O system
- Main memory
  - Unified or partitioned
- Internal network
  - From simple bus to complex mesh



Network Topology → Parallelization Model

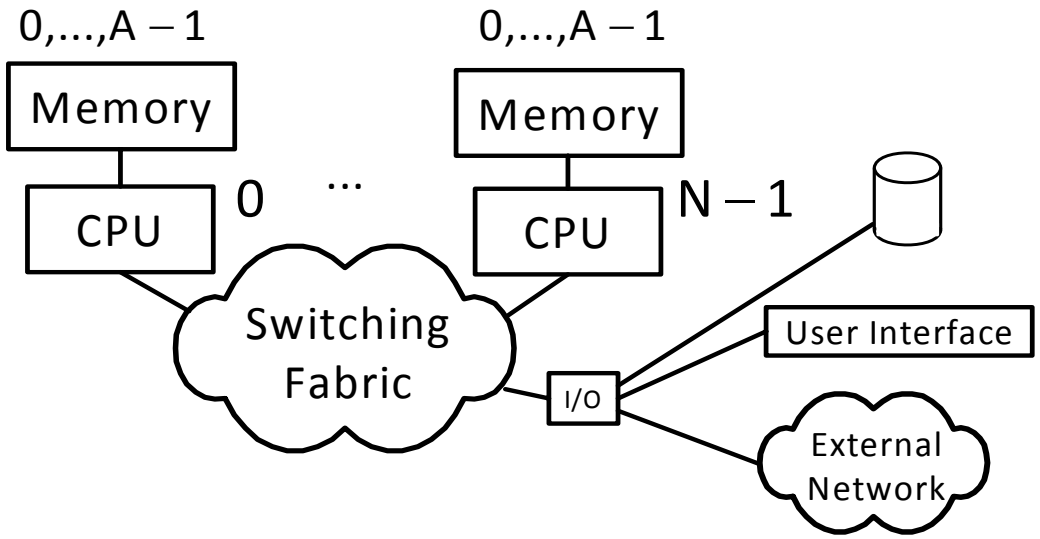
Shared Memory System

- Global memory space *A* physically partitioned into *M* blocks
- *N* processors access full memory space via internal network
- Processors communicate by write/read to shared addresses
- Synchronize memory accesses to prevent data hazards



Message Passing System

- *N* nodes — processors with private address space *A*
- Processors communicate by passing messages over internal network
- Messages combine data and memory synchronization



Flynn-Johnson Taxonomy

<table>
<tr><td></td><td>Single Instruction</td><td colspan="3">Multiple Instruction</td></tr>
<tr><td>Single Data</td><td>SISD</td><td colspan="3">MISD</td></tr>
<tr><td rowspan="3">Multiple Data</td><td rowspan="3">SIMD</td><td>GMSM</td><td>GMMP</td><td>Global Memory</td></tr>
<tr><td colspan="2">MIMD</td><td></td></tr>
<tr><td>DMSM</td><td>DMMP</td><td>Distributed Memory</td></tr>
<tr><td></td><td>Shared Memory</td><td>Message Passing</td><td></td></tr>
</table>

Ref: E. E. Johnson, "Completing an MIMD Multiprocessor Taxonomy", Computer Architecture News, June 1988.

Shared Memory versus Message Passing

	Shared Memory	Message Passing
Interprocess communication	Multiple CPUs access shared addresses in common address space	Multiple CPUs exchange messages
Communication overhead	Cache / RAM updates Cache coherency	Message formulation Message distribution Network overhead
Scalability	Limited by complexity of CPU access to shared memory	Independent of number of CPUs Limited by network capacity
Applicability	Fine grain parallelism Light parallel threads Short code length Small data volume	Course grain parallelism Heavy parallel threads Long code length Large data volume
API	OpenMP	Message Passing Interface (MPI)

Amdahl's Law for Multiprocessors

Parallelization

Divide work among N processors

F_P = fraction of program that can be parallelized $= \frac{IC_P}{IC} \Rightarrow IC_P = F_P \times IC$

For parallel work $CPI \rightarrow CPI_{\text{parallel}} = CPI / N$

$$S = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{CPI \times IC}{\left[CPI \times \left(IC - IC_P \right) + \frac{CPI}{N} \times IC_P \right]}$$

$$= \frac{CPI}{\left(1 - F_P\right) \times CPI + F_P \times \frac{CPI}{N}} = \frac{1}{\left(1 - F_P\right) + \frac{F_P}{N}}$$

With contemporary technology, for most applications, $F_P \approx 80\%$

$$S = \frac{1}{\left(1 - 0.8\right) + \frac{0.8}{N}} \xrightarrow[N \to \infty]{} 5 \qquad CPI^{ideal} = \left(1 - 0.8\right) \times 1 + 0.8 \times \frac{1}{N} \xrightarrow[N \to \infty]{} = 0.2$$

MP and HT Performance Enhancements

MP Without Hyper Threading

CPUs	S	S/CPU
2	1.7	0.85
4	2.6	0.65

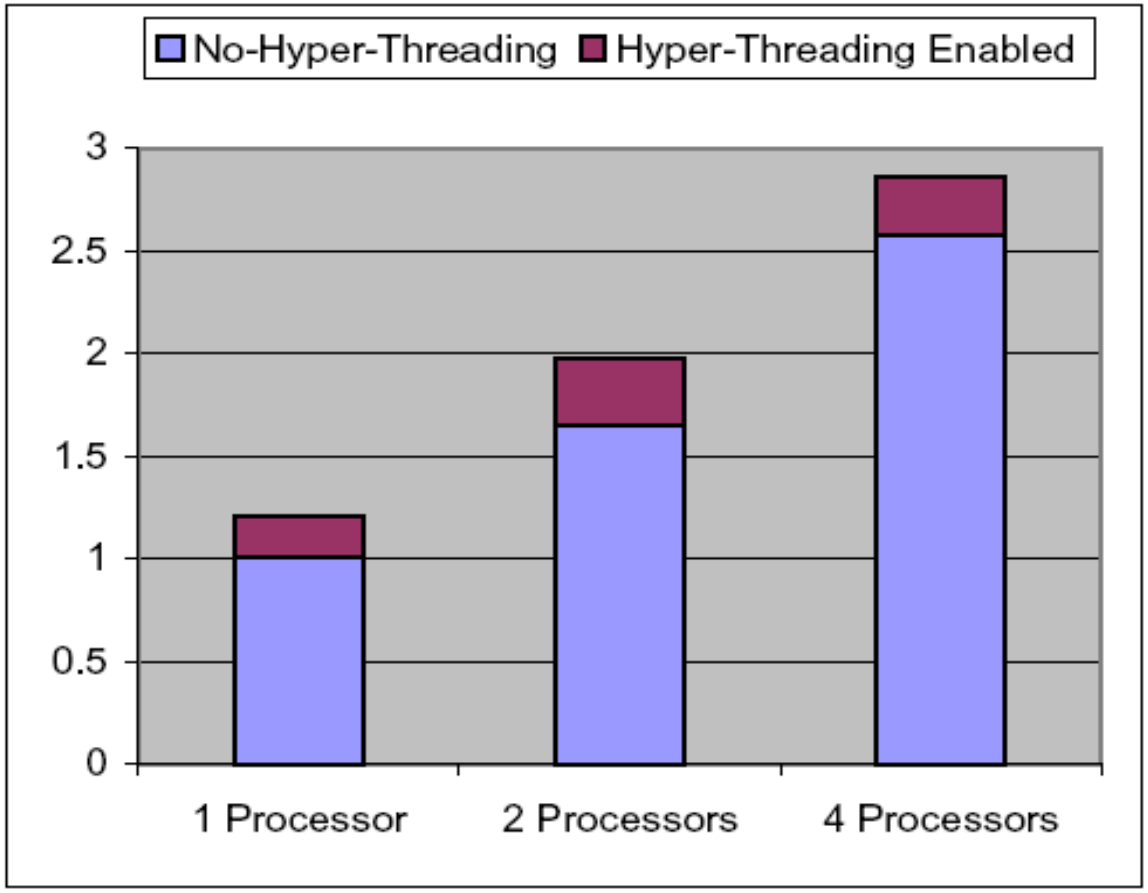
$$1.7 = \frac{1}{(1 - F_P) + \frac{F_P}{2}}$$

$$2.6 = \frac{1}{(1 - F_P) + \frac{F_P}{4}}$$

$$F_P \approx 0.8$$

Hyper Threading Without MP

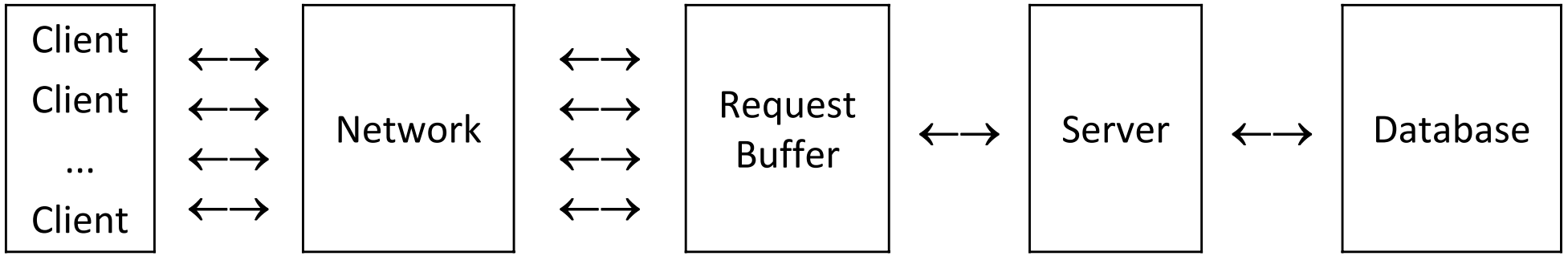
CPUs	S	S/CPU
1	1.2	0.60



Speed-up for On Line Transaction Processing (OLTP)

On Line Transaction Processing (OLTP)

Model



Transactions

Client requests to server + database

- Banking, order processing, inventory management, student info system

Independent work — inherently multithreaded

- 1 thread per request
- Server sees large batch of small parallel threads

Short sequential code

- SQL transactions — short accesses to multiple tables

Complex (DB) access ⇒ memory latency ⇒ CPU stalls per thread

- CPI^{OLTP} = 1.27 on 8-pipeline dynamic scheduling superscalar
- CPI^{SPEC} = 0.31 on same hardware

Memory Access Complexities in OLTP

SQL thread

- Access multiple tables
- Example
  - Order processing ⇒ customer account, inventory, shipping, ...
- Tables in separate areas of memory
  - Cache conflicts
- Generates multiple memory latencies per thread

Multiple threads

- Threads access same tables
- Requires atomic SQL transaction
- Requires thread synchronization
- Synchronization ⇒ locks on parallel threads ⇒ memory latencies

SMT advantage

- Process many threads to hide memory latency

Multiprocessor Efficiency

Ideal speedup

$$S_{F_P=1} = \left.\frac{1}{(1-F_P)+\frac{F_P}{N}}\right|_{F_P=1} = N$$

Efficiency

Actual speedup relative to ideal (linear) speedup

Speedup per processor

$$E = \frac{S}{S_{F_P=1}} = \frac{S}{N} = \frac{1}{N} \times \frac{1}{(1-F_P)+\frac{F_P}{N}} = \frac{1}{(1-F_P)N + F_P}$$

Efficiency of large system

$$E \xrightarrow[N\to\infty]{} 0$$

Grosch's Law versus Amdahl's Law

Computers enjoy economies of scale

Claim formulated by Herbert R. J. Grosch at IBM in 1953

Performance-to-price ratio rises as price rises

$$\text{performance} = k_G \times C^s \qquad k_G = \text{constant},\ C = \text{cost},\ s = \text{constant} \sim 2$$

$$\text{performance / cost} \sim k_G \times C$$

If cost of multiprocessor system is linear in unit price of CPU

$$Cost(N) = \alpha \times N$$

$$\text{performance}(N) = k_G \times (\alpha \times N)^2 = k_G \alpha^2 N^2 \Rightarrow S = \frac{\text{performance}(N)}{\text{performance}(1)} = N^2$$

Amdahl's law implies

$$\text{performance}(N) = \frac{k_{Amdahl}}{(1-F_P)+\frac{F_P}{N}} = \frac{k_{Amdahl}}{(1-F_P)+\frac{F_P}{Cost(N)/\alpha}} = \frac{Cost(N)}{(1-F_P)Cost(N)+\frac{\alpha}{k_{Amdahl}}F_P}$$

$$\frac{\text{performance}(N)}{Cost(N)} = \frac{1}{(1-F_P)Cost(N)+\frac{\alpha}{k_{Amdahl}}F_P},$$ for some constant k_{Amdahl}

Claims Against Amdahl's Law

Assumption in Amdahl's law

FP = constant

Suppose instead

$$F_P = F_P(N) \text{ with } F_P(N) \xrightarrow[N\to\infty]{} 1$$

$$S = \frac{1}{1 - F_P(N) + \frac{F_P(N)}{N}} \xrightarrow[N\to\infty]{} \frac{1}{1 - 1 + \frac{1}{N}} = N$$

$$E = \frac{S}{N} \xrightarrow[N\to\infty]{} 1$$

Gustafson-Barsis Law

Parallel part of large problem can scale with problem size

run time in serial execution $= s + p \times n$, $n =$ size of problem

speedup compared to serial execution $= \frac{s + p \times n}{s + p} \xrightarrow[n \text{ large}]{} n$

Interconnection Network Types

Static	Permanent point-to-point connections between end nodes		
	Full connectivity		Requires N × (N-1) point-to-point connections
	Limited connectivity		Requires multiple hops between end nodes
Dynamic	**Bus**	Nodes perform arbitration for bus access	
		Single	Simplest implementation with standard I/O bus types VME, SCSI, PCI, datakit, etc
		Multiple	End nodes connect to N identical buses in parallel
	Switch	Switch elements configured specifically for each transfer	
		Single Stage	N × N switch with limited connectivity Data makes multiple node-to-node hops between end nodes (source to destination)
		Multistage	Full connectivity switch assembled from multiple single stage switches Not simultaneously non-blocking
		Crossbar	N × N simultaneous non-blocking connections

Communication Overhead and Amdahl's Law

Parallelization with overhead

F_P = fraction of program that can be parallelized $\Rightarrow IC_P = F_P \times IC$

Ideally $CPI \rightarrow CPI_{\text{parallel}} = CPI / N$

Including communication overhead T^{comm} in speedup

$$T^{comm} = CPI^{comm} \times IC_P \times \tau = CPI^{comm} \times F_P \times IC \times \tau$$

CPI^{comm} = processor clock cycles devoted to communication per instruction executed in parallel

$F_{overhead}$ = overhead factor = CPI^{comm} / CPI

$$S = \frac{CPI \times IC}{CPI \times IC \times (1 - F_P) + \frac{CPI}{N} \times F_P \times IC + CPI^{comm} \times F_P \times IC}$$

$$= \frac{CPI}{CPI \times (1 - F_P) + \frac{CPI}{N} \times F_P + CPI^{comm} \times F_P} = \frac{1}{(1 - F_P) + F_P \left(\frac{1}{N} + F_{overhead} \right)}$$

Large Communication Overhead

Parallelization with large overhead

$$S = \frac{1}{(1-F_P) + F_P\left(\frac{1}{N} + F_{overhead}\right)}$$

$$F_{overhead} = \text{overhead factor} = \frac{CPI^{comm}}{CPI} = \frac{\text{communication activity}}{\text{processing activity}}$$

$$S_{\max} = \lim_{N\to\infty} \frac{1}{(1-F_P) + F_P\left(\frac{1}{N} + F_{overhead}\right)} = \frac{1}{(1-F_P) + F_P F_{overhead}}$$

$$= \frac{1}{1 - F_P(1 - F_{overhead})}$$

$$S_{\max} \xrightarrow[F_{overhead}\to 1]{} 1$$

Communication overhead can eliminate benefits of parallelization