

COMPUTATIONAL PERFORMANCE

Notes prepared for EE 6481

by

Professor Cyrus D. Cantrell

May–August 2005

PERFORMANCE

- Performance is a measure of how fast something works
- Who cares?

Understanding it enables you to get the best deal

Which system has the most “bang for the buck”?

Performance determines whether a problem is numerically solvable and what methods can be used to solve it

Program takes too long to run can’t solve problem

Leads to understanding of the underlying motivation for a given computer organization

Why is one system better than another?

Which hardware components are a problem?

How does the instruction set affect performance?

Why is some hardware better than others for different programs?

PERFORMANCE IS TIME

- Transportation performance metrics

Throughput

Passenger miles per month

Performance improves with higher volume and/or distance

Passengers (users) measure performance in terms of time in-flight

- Long-distance telecommunication performance metric

BL = bits per second (B) times distance (L)

Performance improves with higher speed (B) and greater distance (L)

- Computer performance metrics

For users: Execution time or response time

Transaction processing systems: Throughput

A USER'S DEFINITION OF PERFORMANCE

- Execution time := total time required to run program
“wall-clock time”
Most significant parameter for:
Product development
Research
- Performance := (time unit)/(execution time)
Number of executions per time unit
Affected by:
Processor speed
Concurrency of execution
I/O speed
Type of program
System workload

RELATIVE PERFORMANCE

- Relative performance is the performance of one system compared to another, and is given by the performance ratio.

The performance ratio of System A to System B is:

$$\frac{\text{Performance}_A}{\text{Performance}_B} = n = \frac{\text{Execution}_B}{\text{Execution}_A}$$

System A is n times faster than System B.

EXAMPLE OF COST-PERFORMANCE ANALYSIS (1)

- Consider:

Systems S_1 , S_2 , with costs C_1 and C_2

Programs P_1 and P_2 , with execution times T_1 and T_2

- The **cost-performance ratio** for program P_j on system S_i is

$$\frac{\text{cost of } S_i}{\text{performance on } P_j} = C_i T_j$$

A *lower* value of $C_i T_j$ means better performance for a given cost, or lower cost for a given performance

- Example:

System	C_i (\$)	T_1 (s)	$C_i T_1$	T_2 (s)	$C_i T_2$
S_1	1,500	10	15,000	5	7,500
S_2	3,000	3	9,000	3	9,000

UNIX TIME COMMAND (1)

- `wotan% time linpackd.exe`
`12.7u 0.1s 0:13 99% 0+764k 1+0io 0pf+0w`

- Interpretation of this example:

12.7 seconds **user time** (u) — time spent running the program `linpackd.exe`

0.1 seconds **system time** (s) — time spent by the Unix kernel on behalf of `linpackd.exe`

13 seconds **elapsed time**

CPU time := user time + system time = 99% of elapsed time

Average memory used = 0 kb unshared + 764 kb shared

block I/O (io): 0 blocks input, 1 blocks output

0 **page faults** (pf)

0 **swaps** (w)

UNIX TIME COMMAND (2)

- `tulia% time linpckd.exe`
8.5u 0.3s 0:13 66% 0+804k 4+0io 41pf+0w

- Interpretation of this example:

8.5 seconds user time

$$\text{tulias CPU performance} = \frac{12.7}{8.5} \times \text{wotans CPU performance}$$

13 seconds elapsed time

tulia and wotan had the same performance on this job

tulias CPU time for this program = 66% of elapsed time

34 % of the elapsed time was spent waiting for other jobs

41 page faults (pf)

The hard disk was accessed 41 times for data in virtual memory

COMPUTER CZAR'S DEFINITION OF PERFORMANCE

- Throughput := total work done per unit time
- Utilization := % of available time spent executing user jobs
 - Increased by:
 - Having jobs enqueued, awaiting execution
 - Decreasing parallelization (no. of processors used by each job)
- Maximum throughput means:
 - High utilization
 - Low performance from user's point of view
 - Longer times to develop products or obtain research results

CLOCK CYCLES

- Instead of using seconds to measure execution time, often we use clock cycles, aka clock ticks, clock periods, clocks, or cycles.

Clock rate (frequency) = cycles per second.

Measured in Hertz (1 Hz = 1 cycle/s).

- **Clock period** is the time between ticks of the clock and is measured in seconds per cycle.

Period = 1/frequency

- Example: A 200 MHz (MegaHertz) clock has a clock period of

$$\frac{1}{200 \times 10^6 \text{ Hz}} = 5 \times 10^{-9} \text{ seconds} = 5 \text{ nanoseconds.}$$

- **Warning:** Some people refer to the clock period as the clock rate; they are not the same thing!

FUNDAMENTAL EQUATION FOR CPU TIME (1)

- CPU time required to run a program:

$$\begin{aligned} \text{CPU execution time} &= \frac{\text{Instructions}}{\text{Program}} \\ &\times \frac{\text{Clock periods}}{\text{Instruction}} \\ &\times \frac{\text{Seconds}}{\text{Clock Period}} \end{aligned}$$

- $\frac{\text{Instructions}}{\text{Program}}$ is determined by:
 - The instruction set architecture
 - The compiler
 - The program

FUNDAMENTAL EQUATION FOR CPU TIME (2)

- $\frac{\text{Clock periods}}{\text{Instruction}}$ is determined by:

Architecture and logic design (cost tradeoffs)

Instruction mix

- $\frac{\text{Seconds}}{\text{Clock period}}$ is determined by:

Semiconductor device properties

Logic design

CLOCK PERIODS PER INSTRUCTION (CPI) (1)

- **CPI** is the *average* number of clock periods per instruction:

$$\text{CPI} := \frac{\text{clock periods}}{\text{instruction count}} = \frac{C}{I}$$

I = total number of instructions of all types

C = total number of clock periods

- Suppose there are N types of instructions

I_k = number of instructions of type k ; $I = \sum_k I_k$

CPI_k = clock periods to execute *one* instruction of type k

$$C = \sum_{k=1}^N I_k \times \text{CPI}_k \qquad \text{CPI} = \frac{C}{I} = \sum_{k=1}^N \left(\frac{I_k}{I} \right) \times \text{CPI}_k$$

$$\text{CPI} = \sum_{k=1}^N F_k \times \text{CPI}_k$$

where F_k = fraction of type k instructions

$$F_k = \frac{I_k}{I} \qquad 0 \leq F_k \leq 1$$

CLOCK PERIODS PER INSTRUCTION (CPI) (2)

- Example for a MIPS R2000 processor with a particular instruction mix:

k	Instr. type	F_k	CPI_k	$F_k \times CPI_k$
1	Load	0.30	2	0.60
2	Store	0.15	2	0.30
3	ALU op.	0.40	1	0.40
4	Branch	0.15	2	0.30
CPI = $\sum_{k=1}^4 F_k \times CPI_k$				1.6

NUMBER OF CPU CLOCK CYCLES

- Two independent methods of computing NC = number of cycles in one program:

In terms of execution time and clock frequency:

$$NC = ET \cdot CF$$

In terms of instruction count and clocks per instruction:

$$NC = IC \cdot CPI$$

From the equation

$$ET \cdot CF = IC \cdot CPI$$

one can find any one of ET , CF , IC or CPI , given the other three

CLOCK PERIODS PER INSTRUCTION (CPI) (3)

- Computation of CPI using CPU time for two different systems S_1 , S_2 running a program with 100×10^6 instructions

System	Clock rate (MHz)	CPU time (s)	Clock periods(NC)	CPI
S_1	100	10	1000×10^6	10
S_2	200	3	600×10^6	6

A PEAK PERFORMANCE METRIC: PEAK MIPS (1)

- MIPS := **Millions of Instructions Per Second**

The **peak MIPS** rating is calculated using the theoretical **peak rate** at which instructions can be issued

- For **average MIPS**, use the fundamental performance equation:

$$\begin{aligned} \text{MIPS} &= \frac{\text{Instruction Count}}{\text{Execution Time} \times 10^6} = \frac{\text{Instruction Count}}{\text{Clock Periods} \times \text{Cycle Time} \times 10^6} \\ &= \frac{\text{Instruction Count} \times \text{Clock Frequency}}{\text{Instruction Count} \times \text{CPI} \times 10^6} \end{aligned}$$

Then

$$\text{average MIPS} = \frac{\text{Clock Frequency}}{\text{CPI} \times 10^6}$$

A PEAK PERFORMANCE METRIC: PEAK MIPS (2)

- MIPS := **Millions of Instructions Per Second**

The **peak MIPS** rating is calculated using the theoretical **peak rate** at which instructions can be issued

Neglects most real software and hardware properties:

- Memory-cache or memory-CPU bandwidth

- I/O speed

- Type of program (locality of instruction references)

- System workload

Also known as “Meaningless Indicator of Processor Speed”

MEGAFLOPS

- MFLOPS := Millions of Floating-Point Operations Per Second

Peak (“guaranteed not to exceed”) MFLOPS:

Theoretical maximum rate at which floating-point operations can be performed (assuming all FP operations take equal time)

Example: Cray Y-MP, 1 processor

Clock period = 6 ns

1 addition + 1 multiplication possible on each cycle

Peak MFLOPS = $2 \times$ clock frequency = 333 MFLOPS

MFLOPS measured by a benchmarking program:

A program that can do useful work, *e.g.*, LINPACK

A synthetic benchmark, such as:

Livermore loops

SPEC

THE LINPACK BENCHMARK (1)

- LINPACK solves a system of linear equations

$$\mathbf{Ax} = \mathbf{b}$$

using the Gaussian elimination algorithm

Generates a random coefficient matrix \mathbf{A} and right-hand side \mathbf{b}

Timing carried out in the program (not externally):

```
call matgen(a,lda,n,b,norma)
t1 = secnds(0.0)
call dgefa(a,lda,n,ipvt,info)
time(1,1) = secnds(0.0) - t1
:
```

Executes $2n^3/3 + 2n^2$ floating-point operations (where matrix \mathbf{A} is $n \times n$)

THE LINPACK BENCHMARK (2)

- 2 standard benchmark problem sizes:

100 × 100:

No code changes allowed

Array size so small that pipeline latency is significant

May be the most indicative standard benchmark for computationally intensive engineering/scientific programs

1000 × 1000:

Any code change is allowed that does not change the problem being solved

THE SPEC BENCHMARK

- SPEC := Standard Performance Evaluation Corporation

Current CPU intensive benchmark suites:

SPECint95

integer

SPECfp95

floating point

Other benchmarks:

SDM

UNIX Software Development Workloads

SFS

System level file server (NFS) workload

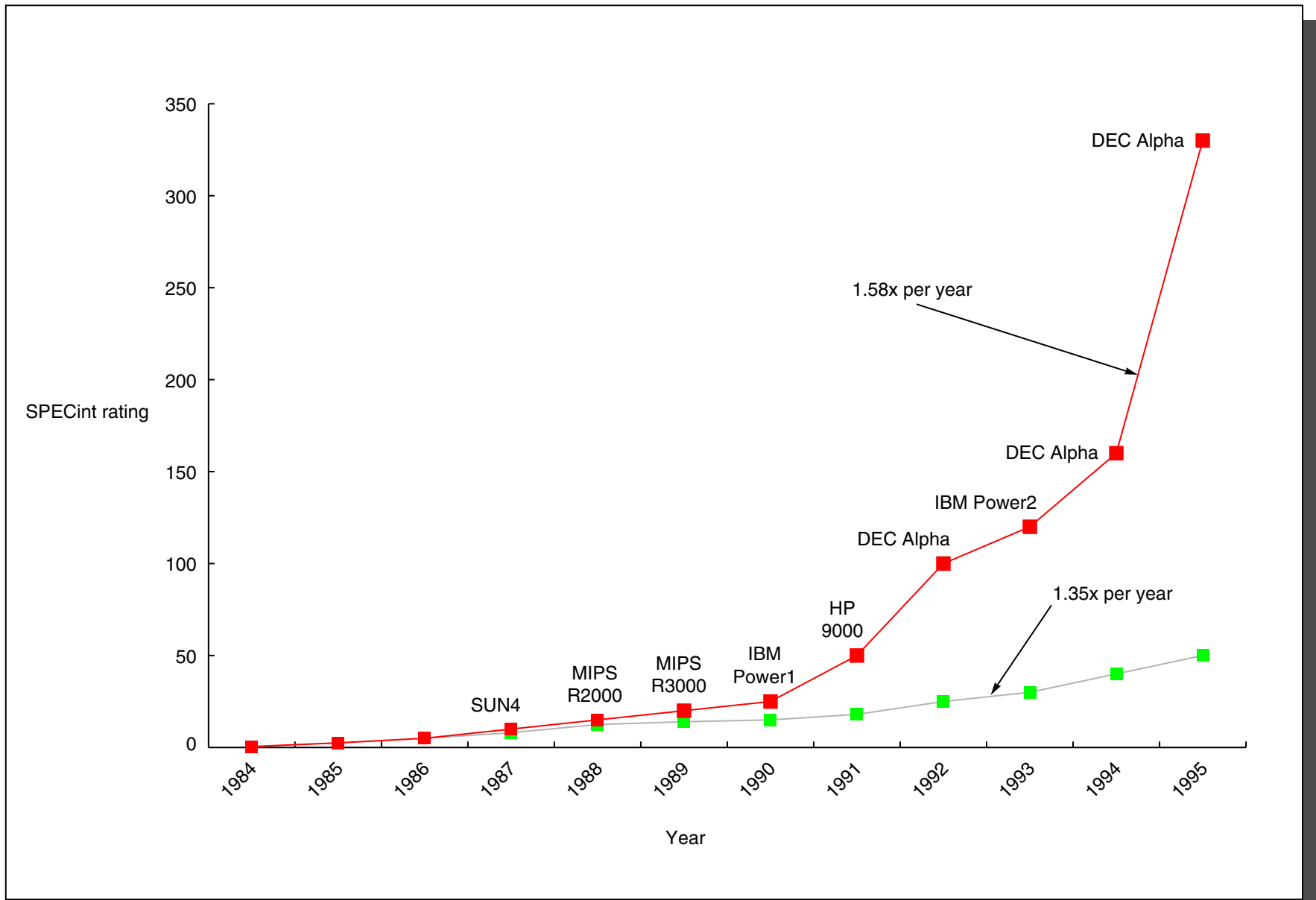
SPEChpc96

High-performance computing benchmarks

- SPEC benchmarks are small “kernels” of code extracted from real programs

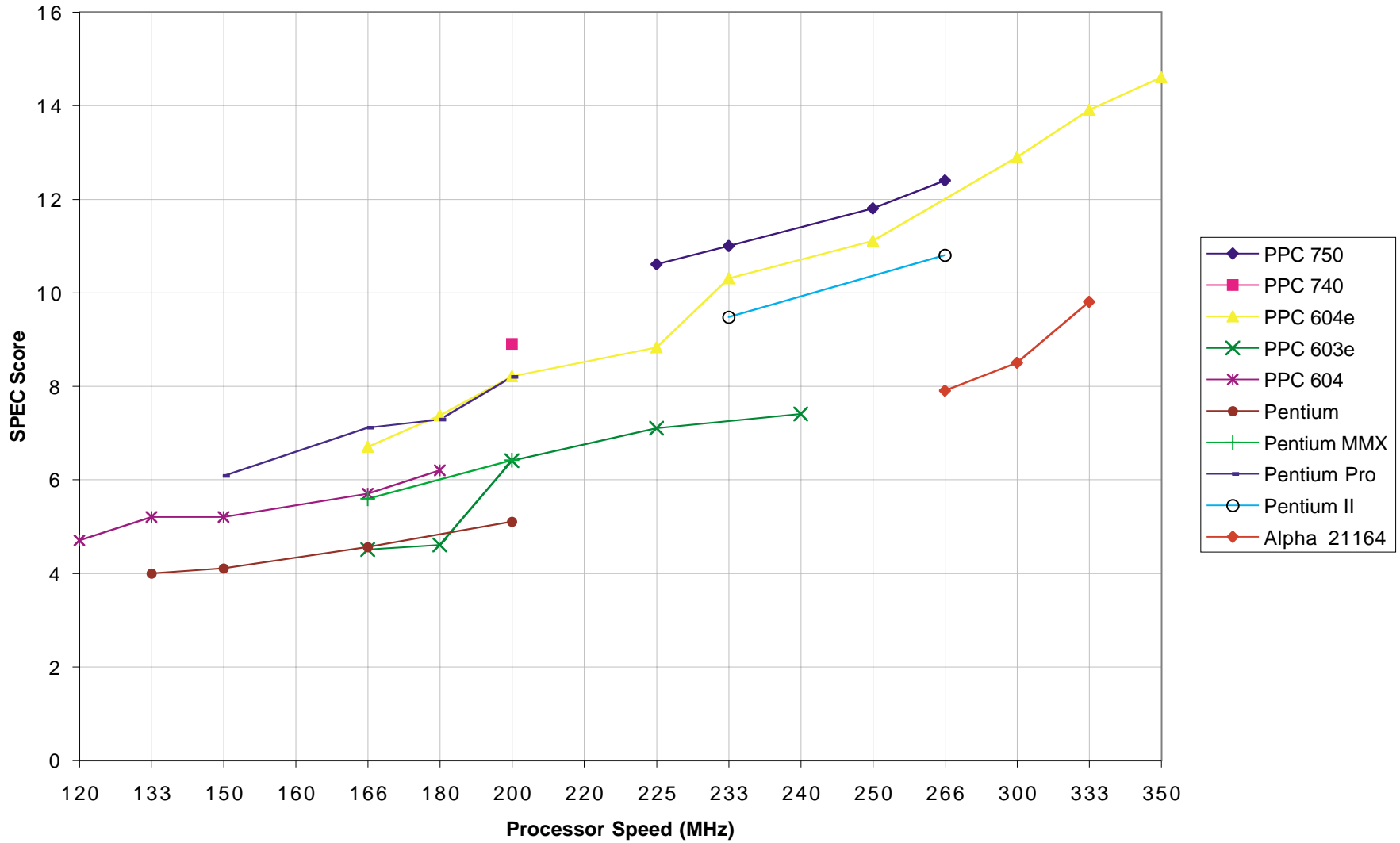
May run unrealistically fast

Programs with large arrays that are accessed using a large stride in memory are likely to run much more slowly than the same program with smaller arrays and a smaller stride

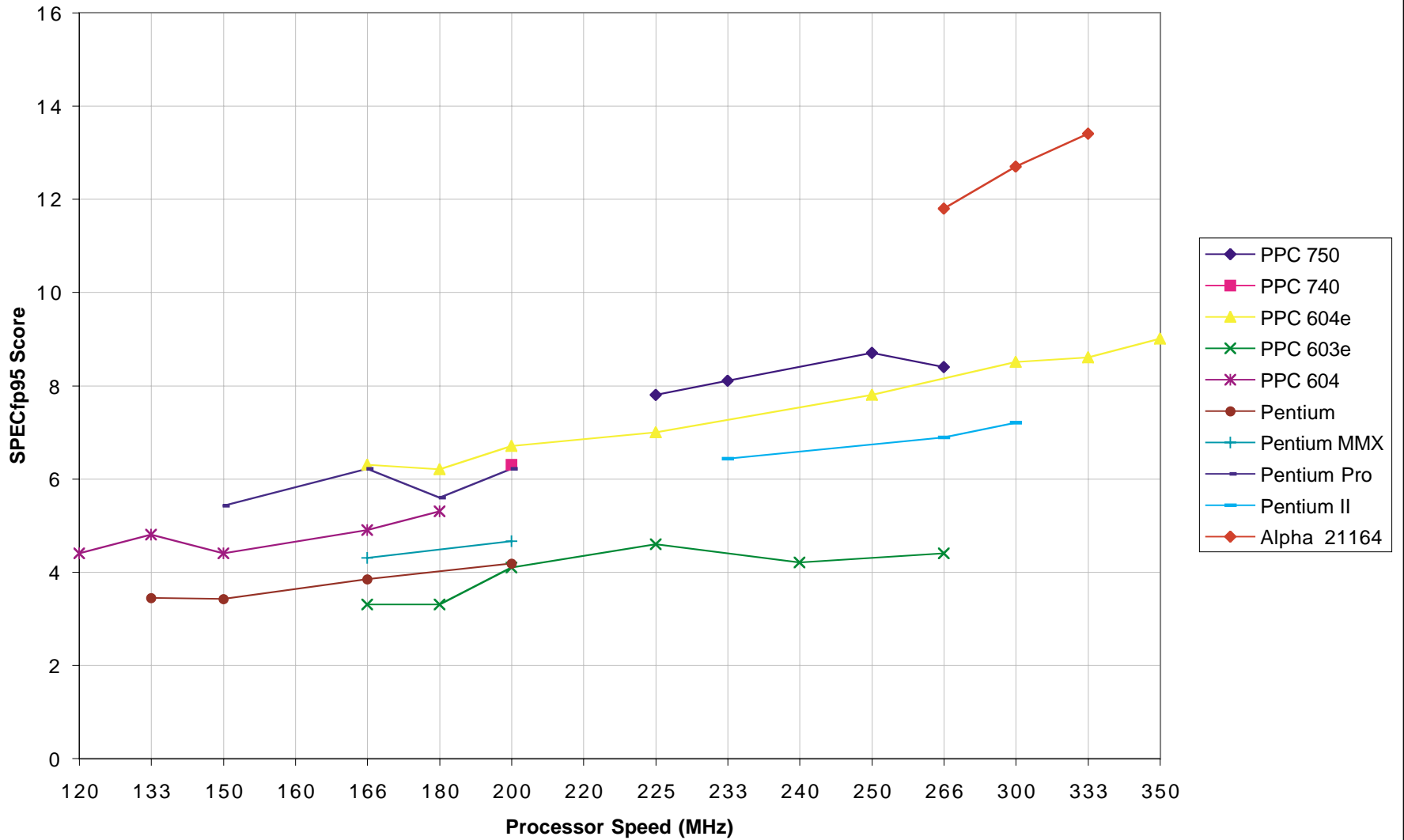


Growth in microprocessor performance since the mid 1980s has been substantially higher than in earlier years

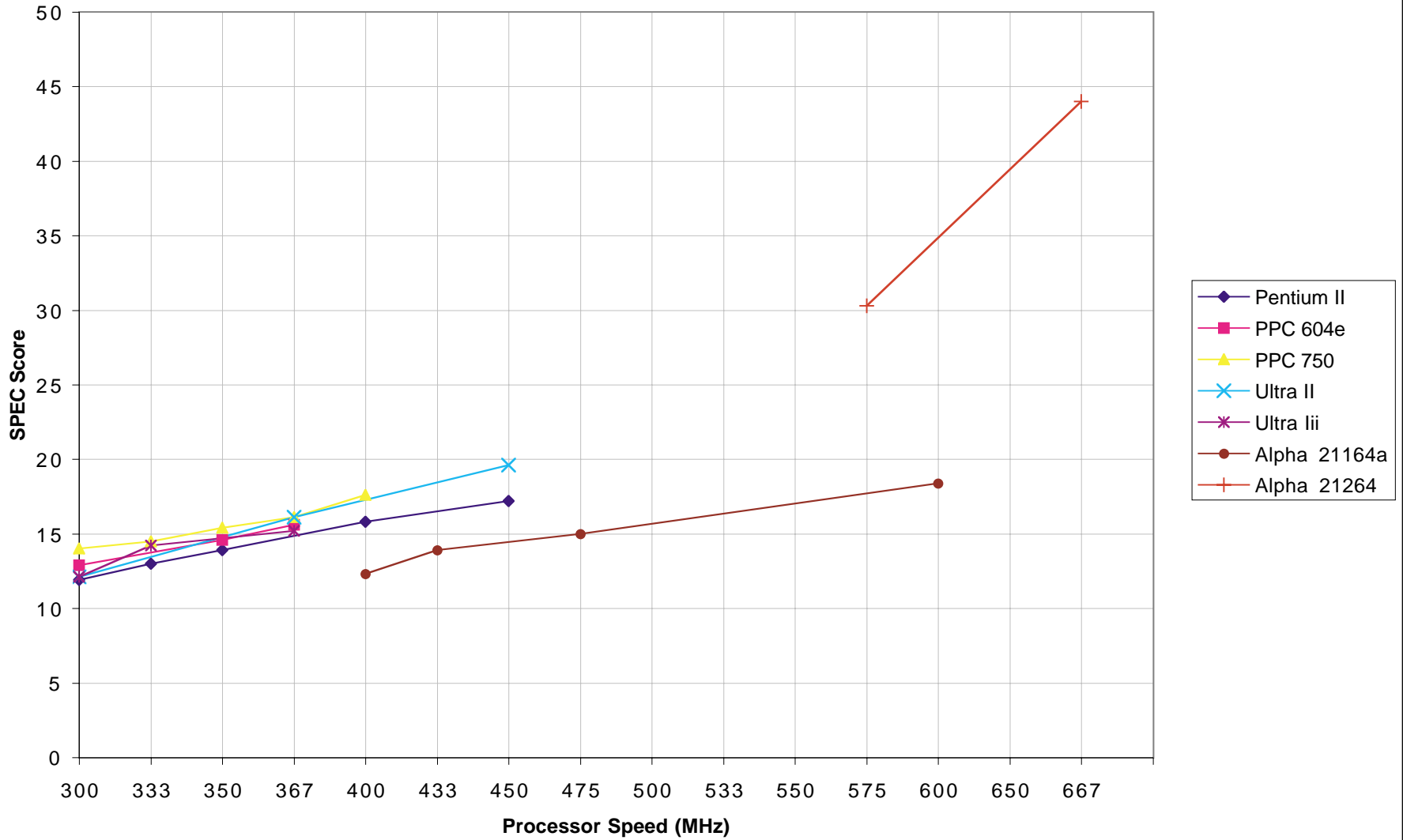
SPECint95 Comparison



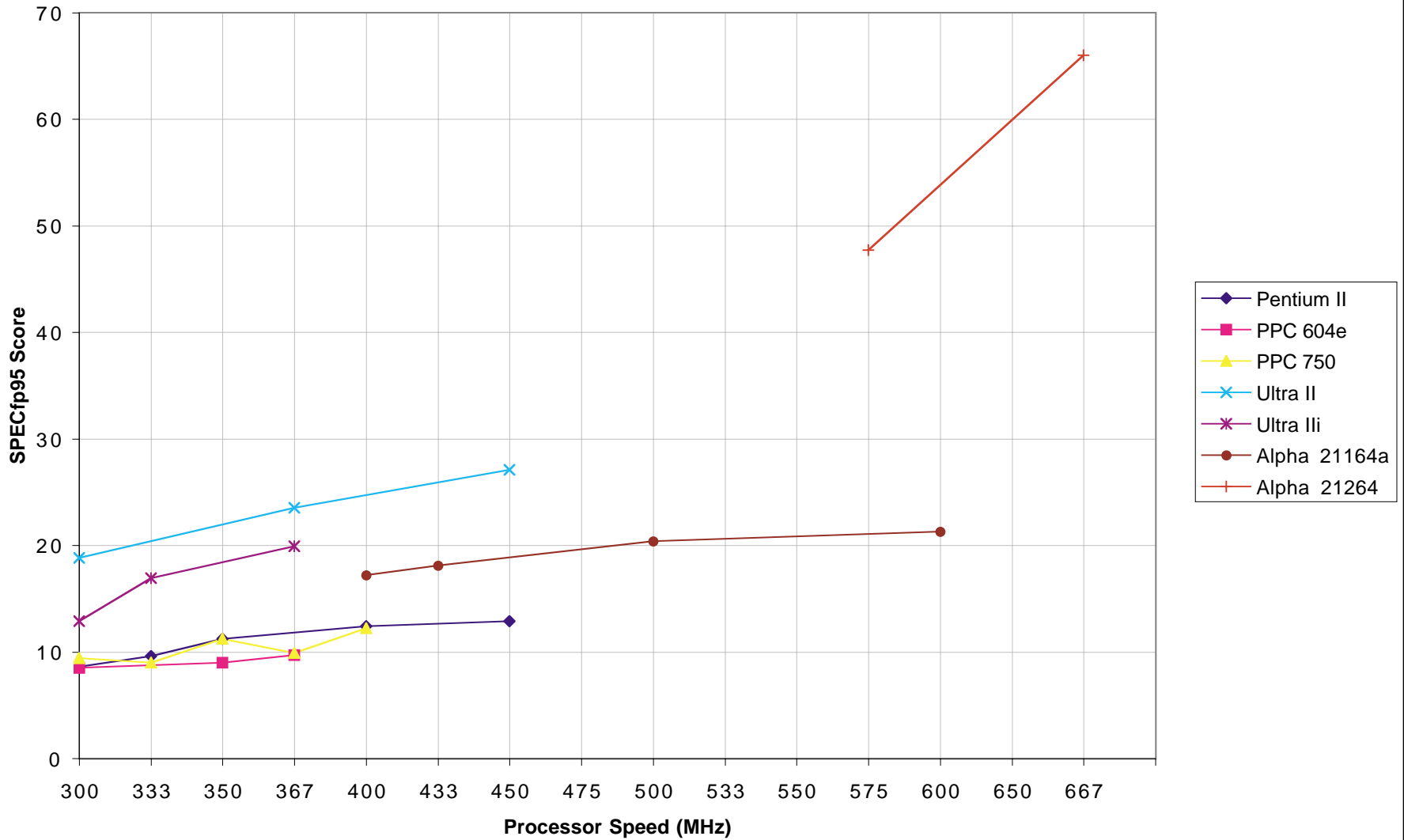
SPECfp95 comparison



SPECint95 Comparison



SPECfp95 comparison



SPEEDUP

- Definition of speedup:

$$\text{Speedup} := \frac{\text{Original Execution Time}}{\text{Improved Execution Time}}$$

- The maximum speedup that can be obtained by using a faster mode of execution is limited by the fraction of the time the faster mode can be used.

AMDAHL'S LAW (1)

- Calculation of speedup due to an enhancement of part of a system or program:

$$\text{Overall speedup} = \frac{\text{Old execution time}}{\text{New execution time}}$$

where

New execution time

= Execution time of unenhanced part

+ Execution time of enhanced part

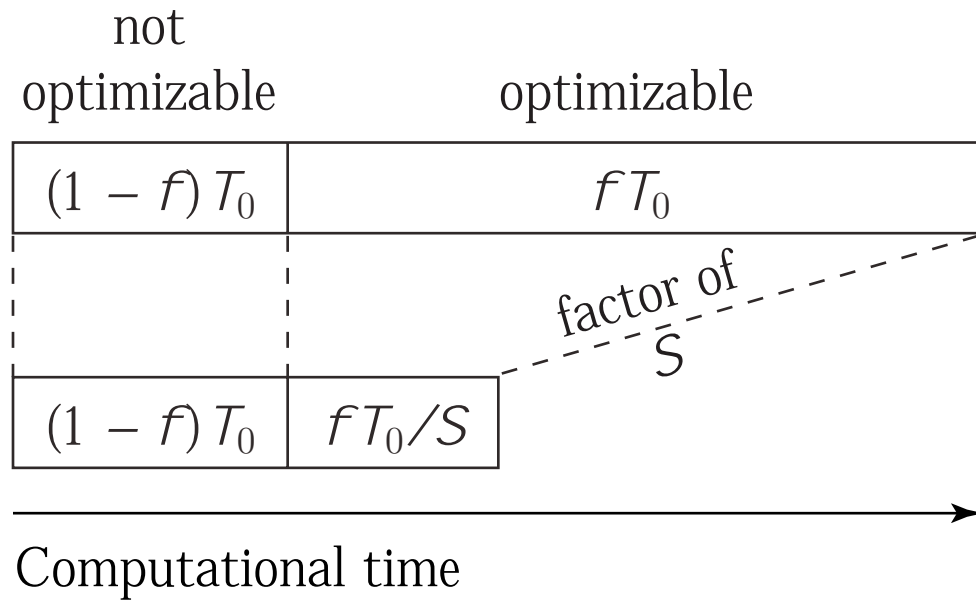
= (Old execution time) \times (1 – Fraction enhanced)

+ $\frac{(\text{Old execution time}) \times (\text{Fraction enhanced})}{\text{Speedup of the enhanced part}}$

= (Old execution time)

$\times \left((1 - \text{Fraction enhanced}) + \frac{\text{Fraction enhanced}}{\text{Speedup of the enhanced part}} \right)$

PICTORIAL EXPLANATION OF AMDAHL'S LAW



AMDAHL'S LAW (2)

- Overall speedup due to an enhancement:

$$S_{\text{overall}} = \frac{1}{(1 - f) + \frac{f}{S_{\text{enhanced}}}}$$

S_{enhanced} = speedup of the enhanced part

f = fraction of executable program that can be enhanced

- The maximum speedup that can be obtained by using a faster mode of execution is limited by the fraction of the time the faster mode can be used.

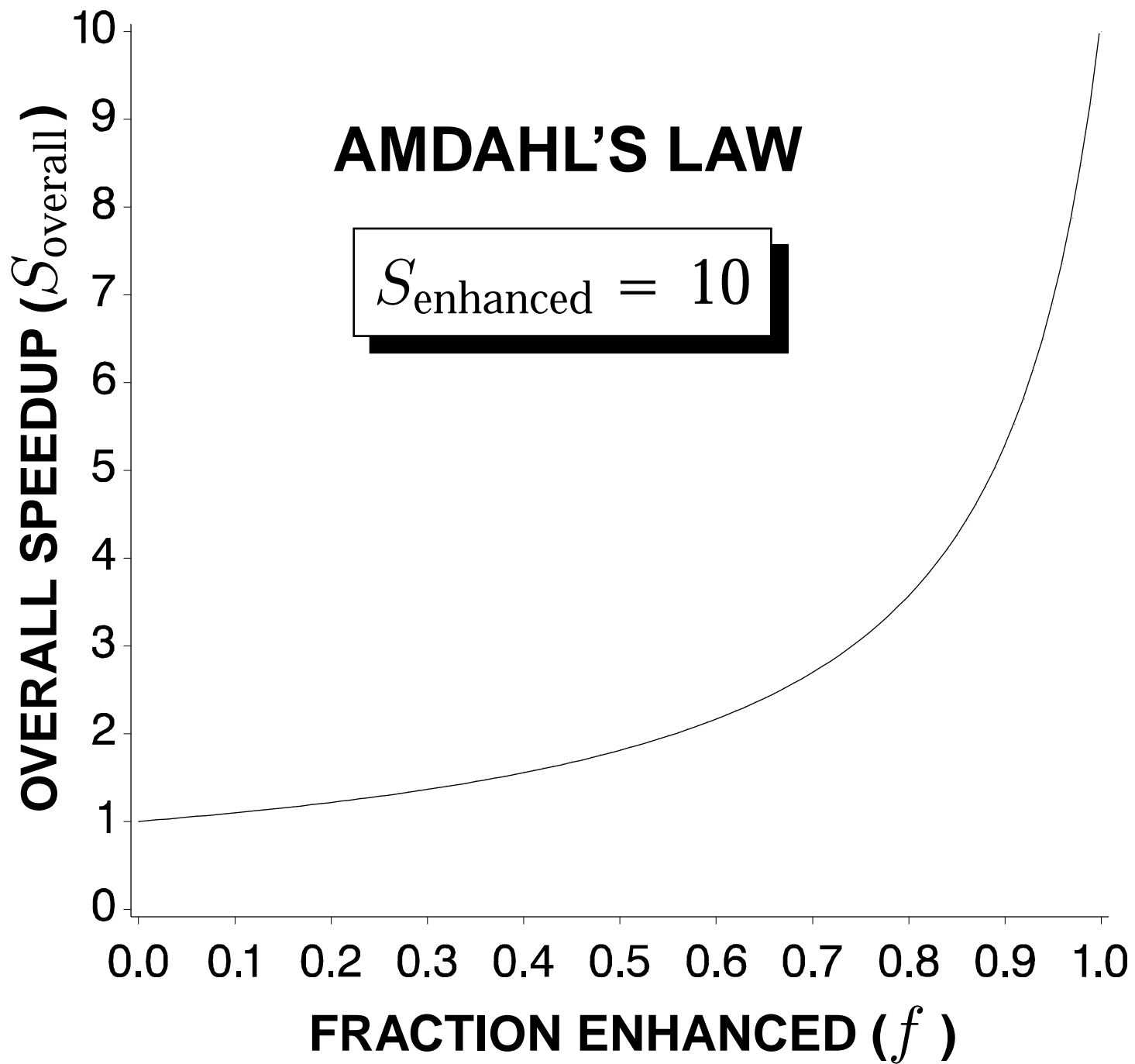
Examples include:

Improving your Web-surfing speed

Parallelization of computer programs

Performance of symmetric-multiprocessor systems

Design of an instruction set architecture



AMDAHL'S LAW (3)

- Example 1: Speedup from the use of cache memory, assuming that
 - Cache is 5 times faster than main memory
 - Cache can be used 90% of the time

Result:

$$\text{Overall speedup} = \frac{1}{(1 - 0.9) + \frac{0.9}{5}} = 3.57$$

- Example 2: Same as example 1, but assuming that cache can be used only 50% of the time:

$$\text{Overall speedup} = \frac{1}{(1 - 0.5) + \frac{0.5}{5}} = 1.67$$

AMDAHL'S LAW (4)

- Example 3 (Real-world):

At the Los Alamos and Lawrence Livermore National Laboratories, 10 years of work by highly skilled programmers resulted in only 70% vectorization on most of the workload

Vectorized code runs up to 10 times faster than scalar code

Amdahl's Law predicts that the average speedup from vectorization is

$$\text{Overall speedup} = \frac{1}{(1 - 0.7) + \frac{0.7}{10}} = 2.70$$

Result: The “killer micros”, whose scalar speed is equal to or better than the scalar speed of the CRAY vector processors, took over a significant share of CRAY's market at the national laboratories

BENCHMARKING A PARALLELIZED PROGRAM

- Benchmark results for a program running on one processor:
6629.0u 5.0s 1:51:21 99% 0+0k 0+0io 0pf+0w
- Benchmark results for the same program running in parallel on four processors (same system):
7779.0u 9.0s 32:57 393% 0+0k 0+0io 0pf+0w

- Analysis:

One processor: User time = 6629 seconds

Four processors: Sum of user times = 7779 seconds

Sum of times on 4 CPUs = 393% of elapsed time

Overall speedup = ratio of elapsed times = 3.38

Theoretical maximum speedup = $S_{\text{enhanced}} = 4$

Actual speedup = $S_{\text{overall}} = 3.38 = 84\%$ of theoretical maximum

$$\text{Implied fraction enhanced} = f = \frac{1 - \frac{1}{S_{\text{overall}}}}{1 - \frac{1}{S_{\text{enhanced}}}} = 0.9$$

AMDAHL'S LAW (5)

- Example 4 (the **RISC revolution**):

Several different groups “discovered” more or less independently that 10 instructions account for more than 90% of the executions

Suppose that one can design hardware that executes almost all of the top 10 instructions in one clock period, instead of 10

Amdahl's Law predicts that the overall speedup is

$$\text{Overall speedup} = \frac{1}{(1 - 0.9) + \frac{0.9}{10}} = 5.26$$

Result: **REDUCED INSTRUCTION SET COMPUTER (RISC)** architectures such as the MIPS R2000

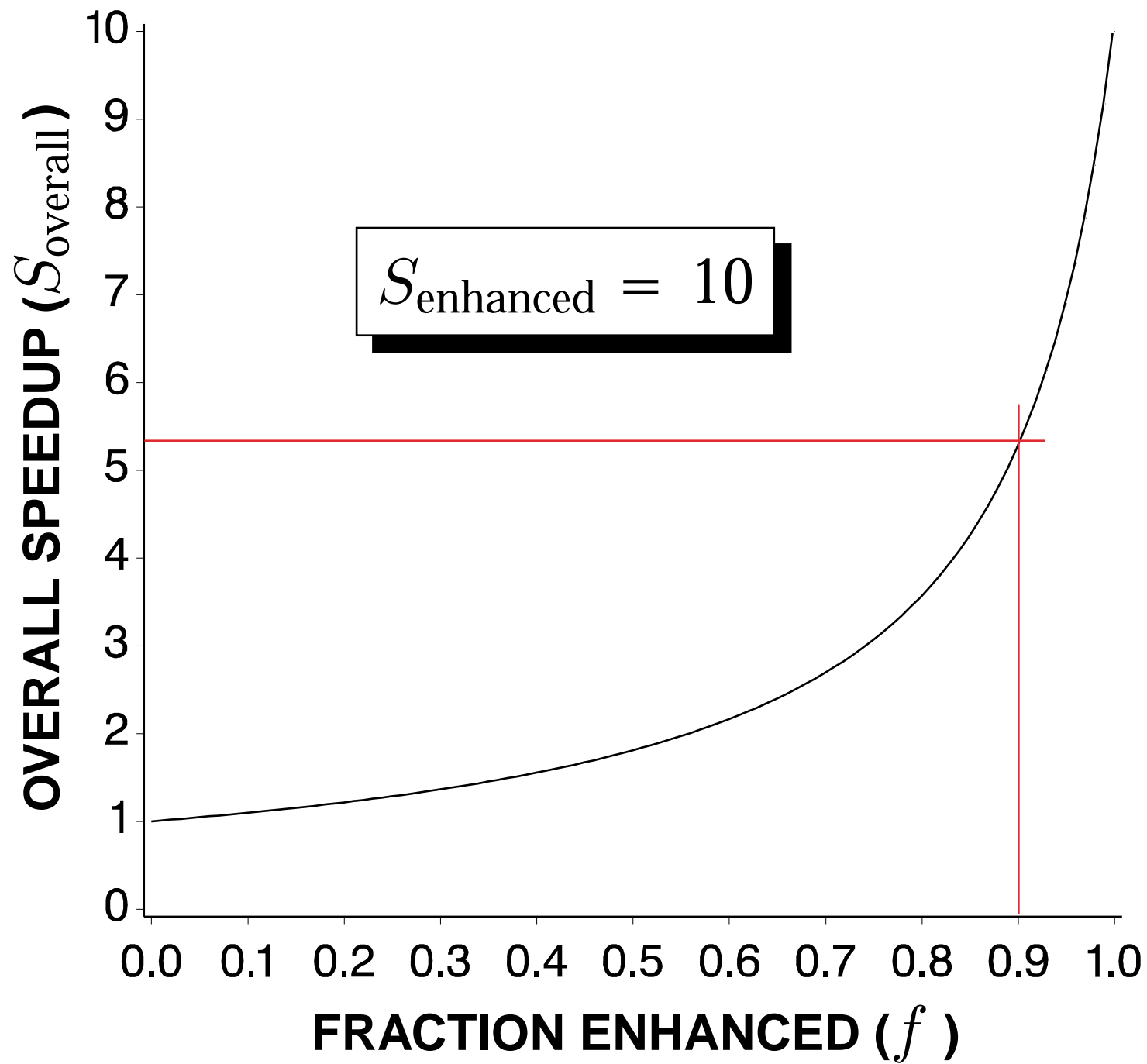
General principle: “**Make the common case fast**”

When Sun Microsystems changed from Motorola 68020 (CISC) processors to SPARC (RISC) processors *at the same clock frequency*, performance improved by a factor of 5



TOP 10 INSTRUCTIONS FOR THE 80x86

Rank	80x86 instruction	% of total integer executions
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move (register-register)	4%
9	call	1%
10	return	1%
Total for top 10 instructions		95%



AMDAHL'S LAW (6)• Example 5 (**Parallel execution**):

Let f be the fraction of a program that is parallelizable, and let P be the number of processors

The speedup in wall-clock time due to execution with P processors instead of 1 processor is

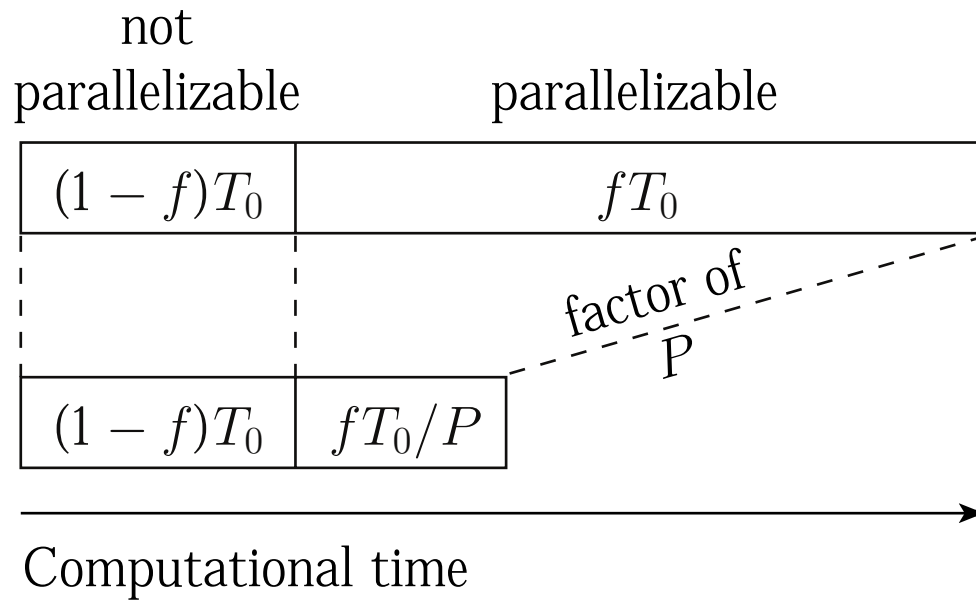
$$S(P) = \frac{1}{(1 - f) + \frac{f}{P}}$$

This version of Amdahl's law ignores the dependence of f on P due to the dependence of I/O and memory/CPU latency on P

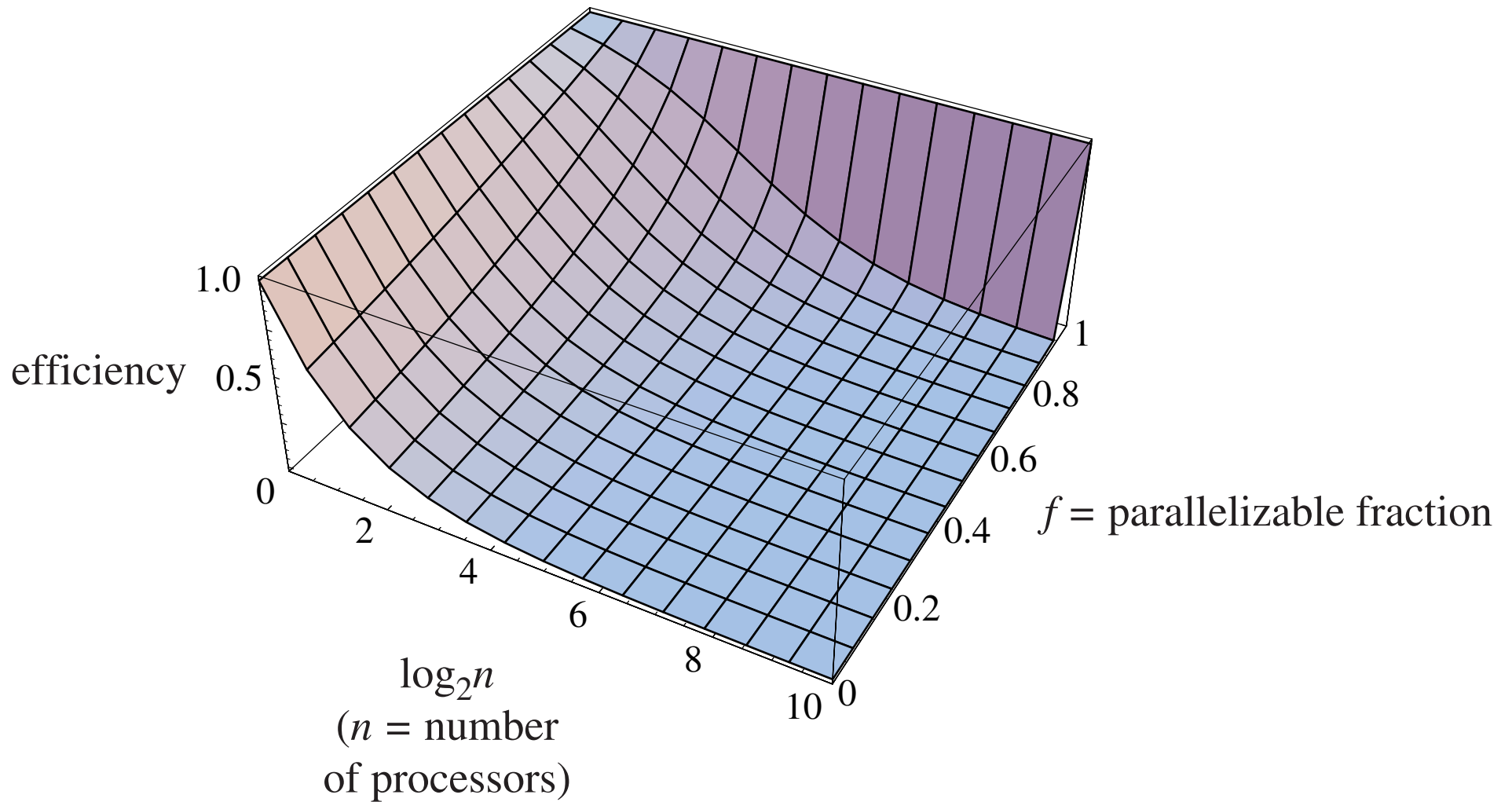
The actual speedup is always less than the asymptotic speedup,

$$S_{\text{asympt}} = \lim_{P \rightarrow \infty} S(P) = \frac{1}{1 - f}$$

PICTORIAL EXPLANATION OF AMDAHL'S LAW



AMDAHL'S LAW FOR PARALLEL COMPUTING



AMDAHL'S LAW (7)

- Number of processors (P_F) needed to achieve a fraction F of the asymptotic speedup:

$$S(P_F) = \frac{1}{(1 - f) + \frac{f}{P_F}} = F S_{\text{asympt}} = \frac{F}{1 - f}$$

Then

$$P_F = \left(\frac{F}{1 - F} \right) \left(\frac{f}{1 - f} \right)$$

- ▷ Estimate of the point of diminishing returns that can be obtained by using additional processors to reduce a computation's wall-clock time:

$$P_{1/2} = \frac{f}{1 - f}$$

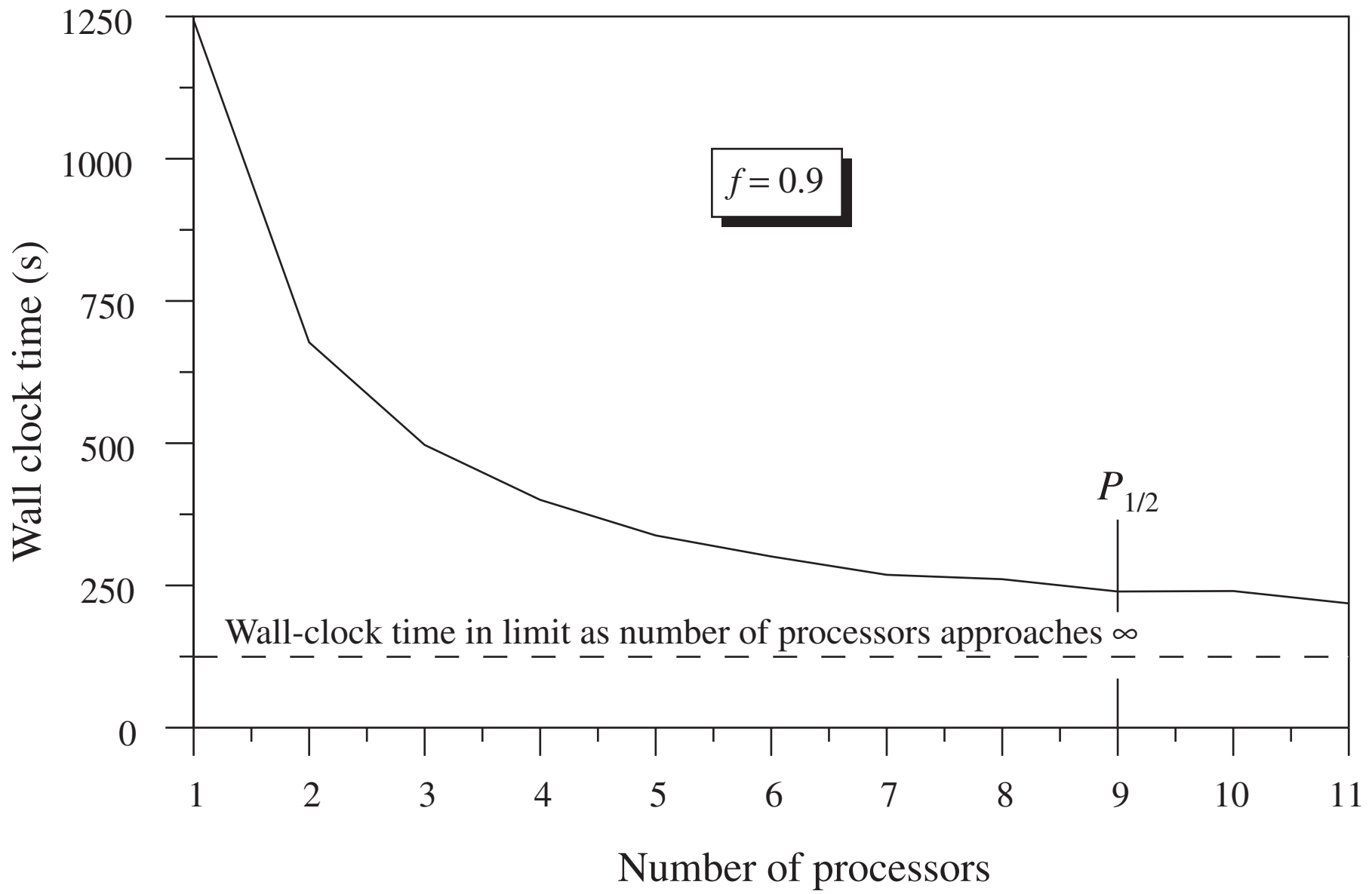
It almost never pays to use more than $P_{1/2}$ processors

Some of the following slides show experimentally measured execution time as a function of number of processors

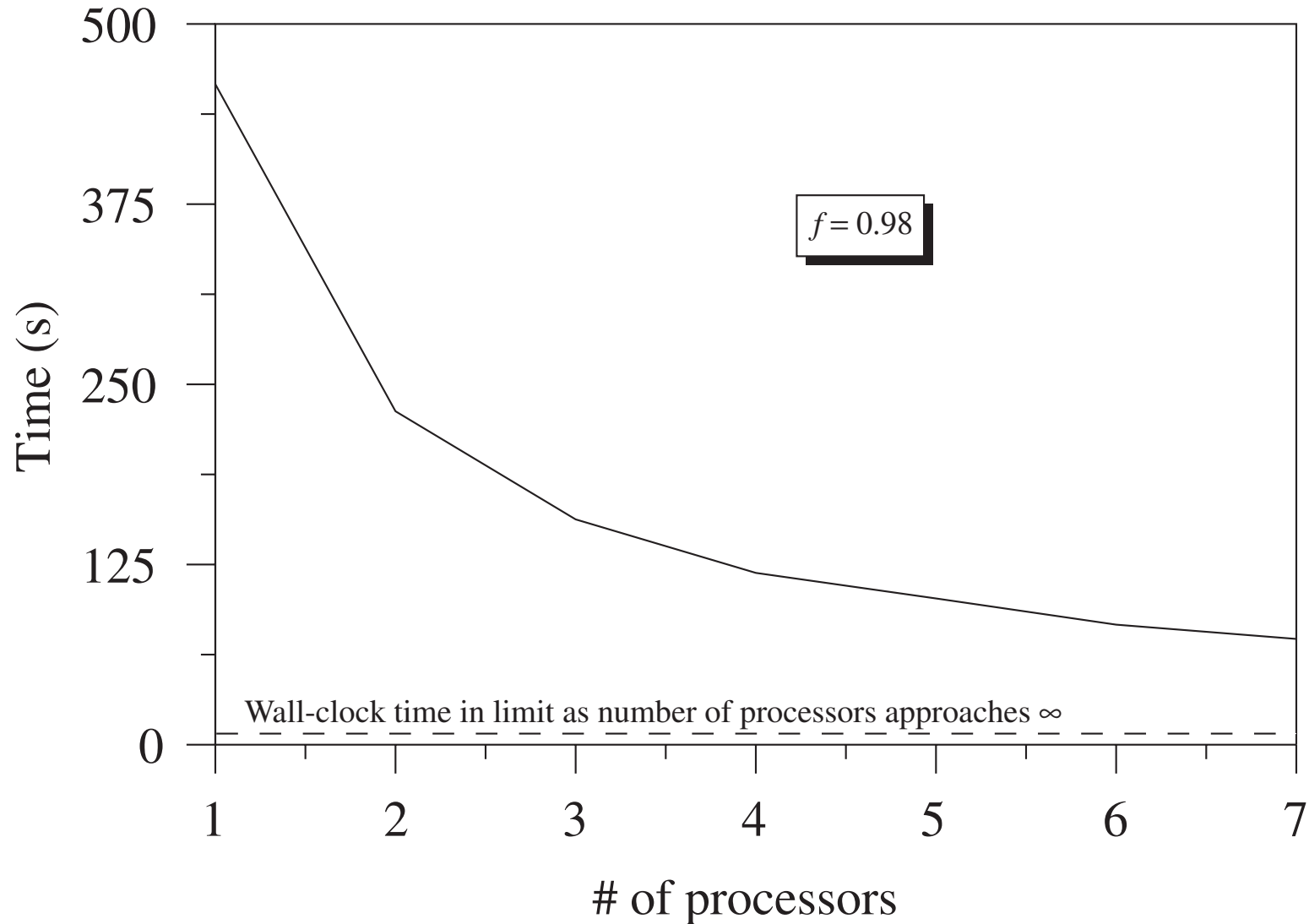
AMDAHL'S LAW (8)

- Examples of asymptotic speedup as a result of parallel execution:

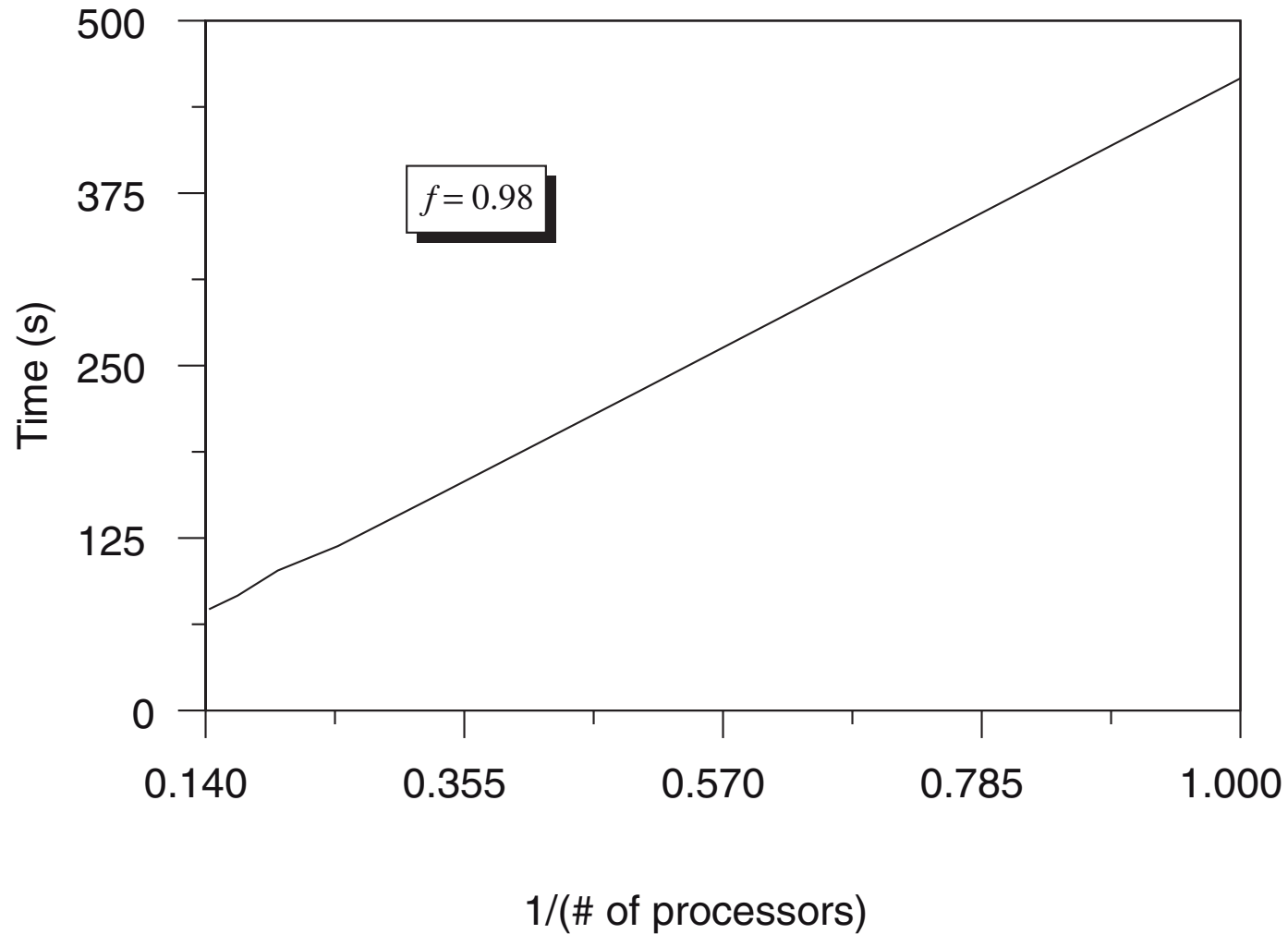
f	S_{asympt}	$P_{1/2}$
0.10	1.11	0.11
0.50	2.00	1
0.75	4.00	3
0.80	5.00	4
0.90	10.0	9
0.95	20.0	19
0.98	50.0	49
0.99	100	99



TIME vs. NUMBER OF PROCESSORS (PB METHOD)



TIME vs. (NUMBER OF PROCESSORS)⁻¹ (PB METHOD)



PREPARING A PROGRAM FOR PARALLEL EXECUTION

- Study your computational algorithm to understand which parts of the computation can benefit from parallelism
 - ▷ Use profiling tools to estimate what fraction of the execution time occurs in the parallelizable parts, and estimate S
- Study what you need to do (within your programming language and compiler) to ensure that the parallelizable part is compiled as parallel code
- Measure $P_{1/2}$ experimentally, by measuring execution time vs. number of processors for your program and on the target machine
 - ▷ It's unfair to other users to request more than $P_{1/2}$ processors
 - ▷ If $P_{1/2} = 1$:
 - Understand why your program or algorithm is not benefitting from parallelism
 - Parallelizable fraction too small (compile for 1 processor)
 - Execution time of a parallel thread > setup/communication time
 - (choose a symmetric multiprocessor, not a cluster)

PROGRAMMING IMPLICATIONS OF HIERARCHICAL MEMORY

- Maintain instruction and data locality in the innermost loop

Program a short innermost loop

No conditional statements

No function or subprogram calls

Inline code instead of calling a function

Use no statements that throw interrupts

No I/O in the innermost loop

To maximize data locality:

Arrange unit stride (or the shortest possible stride) for all arrays that are referenced in the innermost loop

LOOP ORDERING IN MATRIX MULTIPLICATION

- The operations in the computation of a matrix product $\mathbf{C} = \mathbf{AB}$ are

```
/* (ijk) loop */  
for (i=0; i<n; i++)  
{  
  for (j=0; j<n; j++)  
  {  
    for (k=0; k<n; k++)  
    {  
      c[i][k] = c[i][k] + a[i][j] * b[j][k];  
    }  
  }  
}
```

- There are 6 possible loop orderings:
ijk, ikj, jki, kij, kji, jik

MATRIX MULTIPLICATION (1)

- Loop orderings for multiplication of $n \times n$ matrices:

$$\begin{aligned}
 \mathbf{C} &= \sum_{i=1}^n \sum_{k=1}^n c_k^i \mathbf{e}_i \boldsymbol{\epsilon}^k = \mathbf{AB} = \left(\sum_{i=1}^n \sum_{j=1}^n a_j^i \mathbf{e}_i \boldsymbol{\epsilon}^j \right) \left(\sum_{k=1}^n \sum_{l=1}^n b_k^l \mathbf{e}_l \boldsymbol{\epsilon}^k \right) \\
 &= \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n a_j^i b_k^j \mathbf{e}_i \boldsymbol{\epsilon}^k \\
 &= \sum_{i=1}^n \sum_{k=1}^n \mathbf{e}_i \boldsymbol{\epsilon}^k \left[\underbrace{\left(\sum_{j=1}^n a_j^i \boldsymbol{\epsilon}^j \right)}_{\text{row } i} \underbrace{\left(\sum_{l=1}^n b_k^l \mathbf{e}_l \right)}_{\text{col } k} \right] \quad \text{dot product form (ikj and kij)}
 \end{aligned}$$

- Definitions:

\mathbf{e}_i = column vector with 1 in row i and 0 elsewhere

$\boldsymbol{\epsilon}_i$ = row vector with 1 in column i and 0 elsewhere

MATRIX MULTIPLICATION (2)

- More loop orderings for multiplication of $n \times n$ matrices:

$$\begin{aligned}
 \mathbf{C} &= \sum_{k=1}^n \left\{ \left[\sum_{j=1}^n b_k^j \left(\underbrace{\sum_{i=1}^n a_j^i \mathbf{e}_i}_{\text{col } j} \right) \right] \epsilon^k \right\} && \text{gaxpy-dyadic form } (kj \hat{i}) \\
 &= \sum_{j=1}^n \left(\underbrace{\sum_{i=1}^n a_j^i \mathbf{e}_i}_{\text{col } j} \right) \left(\underbrace{\sum_{k=1}^n b_k^j \epsilon^k}_{\text{row } j} \right) && \text{dyadic form } (jki \text{ and } jik) \\
 &= \sum_{i=1}^n \mathbf{e}_i \left[\sum_{j=1}^n a_j^i \left(\underbrace{\sum_{k=1}^n b_k^j \epsilon^k}_{\text{row } j} \right) \right] && \text{gaxpy-dyadic form } (ijk)
 \end{aligned}$$

COMBINED VECTOR AND SCALAR OPERATIONS

- SAXPY (Scalar A times X Plus Y)

$$\mathbf{z} = \mathbf{x} + \mathbf{y}$$

- GAXPY (General A times X Plus Y)

$$\mathbf{z} = \mathbf{Ax} + \mathbf{y}$$

EFFECTS OF STRIDE ON MEMORY ACCESSES (1)

- Stride = 1 + number of words between successively accessed words

Usually used to refer to array accesses

If stride = 1, then words are accessed successively (unit stride)
(since the number of intervening words = 0)

Hierarchical memory:

If stride > cache line size (in words), then there is no data locality
in a practical sense

For large strides, each array access usually incurs the full
latency of the first word accessed

Small benchmark programs give unrealistically high performance estimates

Interleaved memory:

If the stride is not relatively prime to the number of memory banks,
there will be problem sizes at which performance is sharply lowered

EFFECTS OF STRIDE ON MEMORY ACCESSES (2)

- Multiplication of $n \times n$ matrices: $\mathbf{C} = \mathbf{AB}$

Component form: $c_{ik} = \sum_{j=1}^n a_{ij} b_{jk}$

Different loop orderings in matrix multiplication correspond to different points of view about the organization of a matrix

Dot-product interpretation:

c_{ik} = matrix product of row i of \mathbf{A} with column k of \mathbf{B}

Dyadic interpretation: \mathbf{C} = sum of dyadics of the form
(matrix product of column j of \mathbf{A} with row j of \mathbf{B})

Gaxpy ($\mathbf{Ax} + \mathbf{y}$) interpretation: \mathbf{C} = sum of dyadics, each of which contains a linear combination of columns of \mathbf{A} or rows of \mathbf{B}

- For computational purposes, there are 3 loops (on i , j , and k)

Dot-product interpretation applies when the loop on j is innermost

When the i loop is innermost, it computes $b_{jk} \times$ (column j of \mathbf{A})

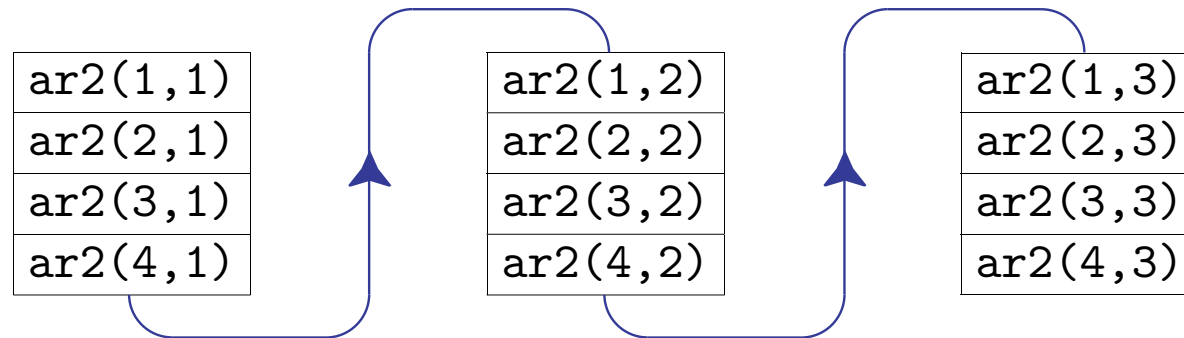
When the k loop is innermost, it computes $a_{ij} \times$ (row j of \mathbf{B})

EFFECTS OF STRIDE ON MEMORY ACCESSES (3)

- FORTRAN loop orderings jki and kji: In the innermost (i) loop in

$$c(i,k) = c(i,k) + a(i,j) * b(j,k),$$

the matrix elements $c(i,k)$ and $a(i,j)$ are accessed successively by rows, which implies stride 1 given the FORTRAN storage order:

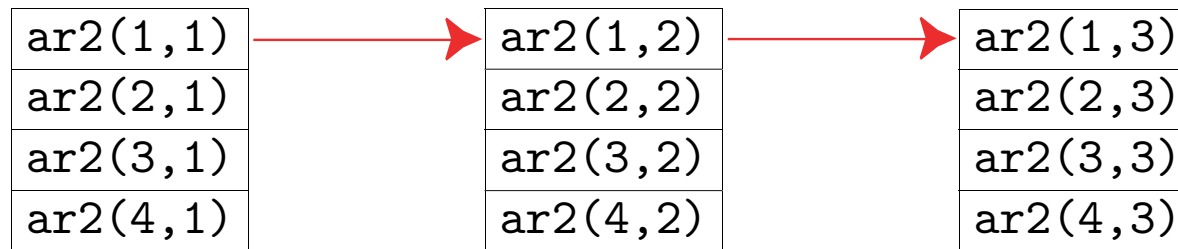


EFFECTS OF STRIDE ON MEMORY ACCESSSES (4)

- FORTRAN loop orderings *ijk* and *jik*: In the innermost (*k*) loop in

$$c(i,k) = c(i,k) + a(i,j) * b(j,k),$$

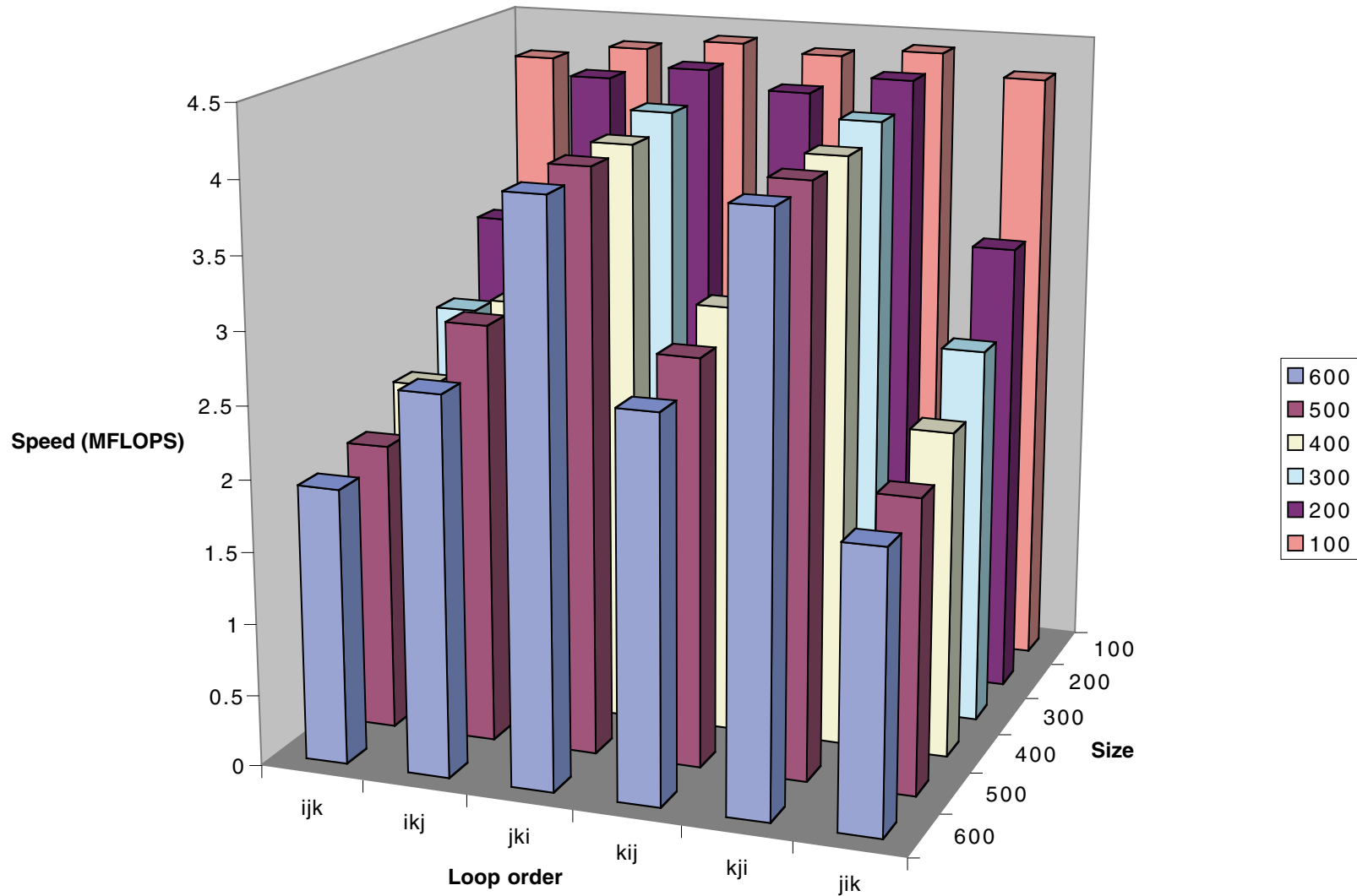
$c(i,k)$ and $b(j,k)$ are accessed successively by columns, which implies stride n given the FORTRAN storage order:



- The following plot was made on an Ultra 10 workstation with 512 kB L2 cache

Cache line size = 16 B

Speed vs. FORTRAN loop order

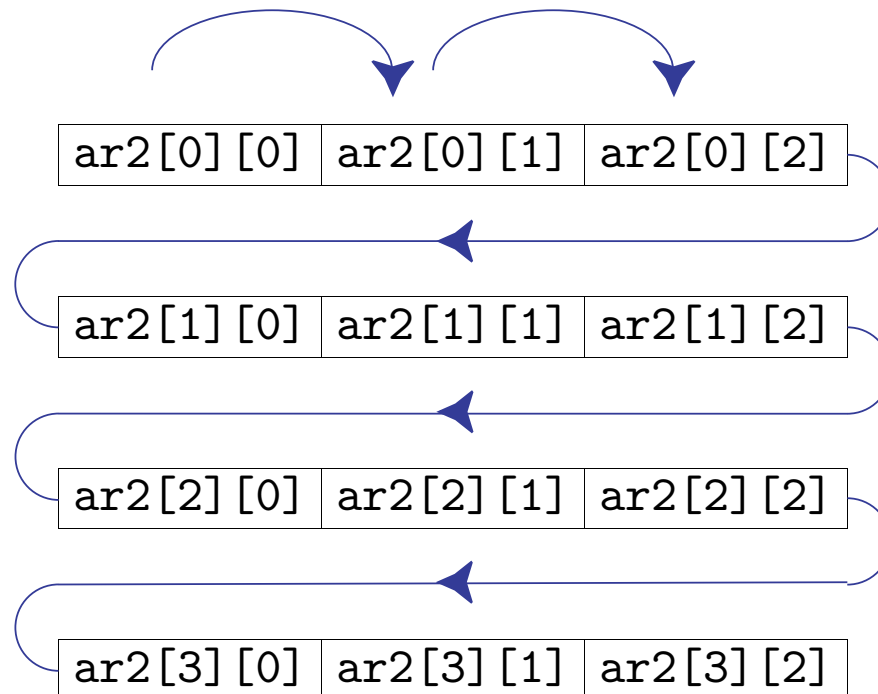


EFFECTS OF STRIDE ON MEMORY ACCESSES (5)

- C loop orderings *ijk* and *jik*: In the innermost (*k*) loop in

$$c[i][k] = c[i][k] + a[i][j] * b[j][k],$$

the matrix elements $c[i][k]$ and $b[j][k]$ are accessed successively by columns, which implies unit stride given the C storage order:

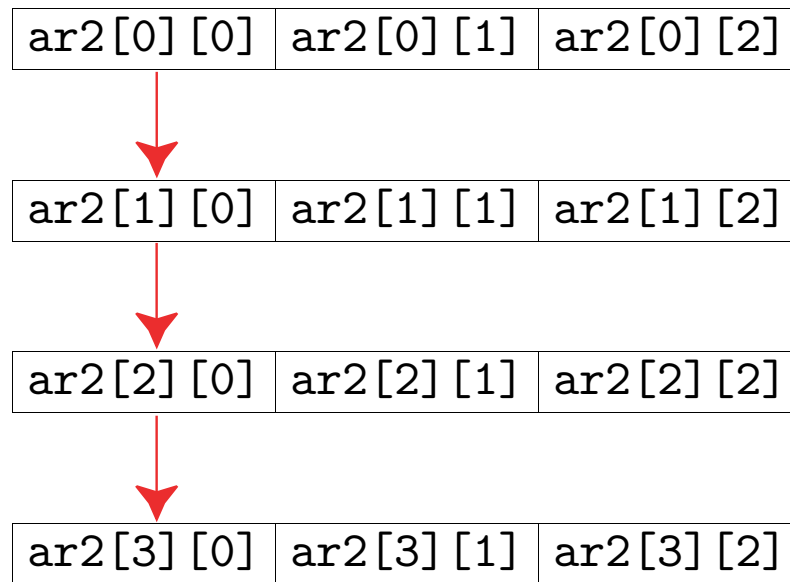


EFFECTS OF STRIDE ON MEMORY ACCESSES (6)

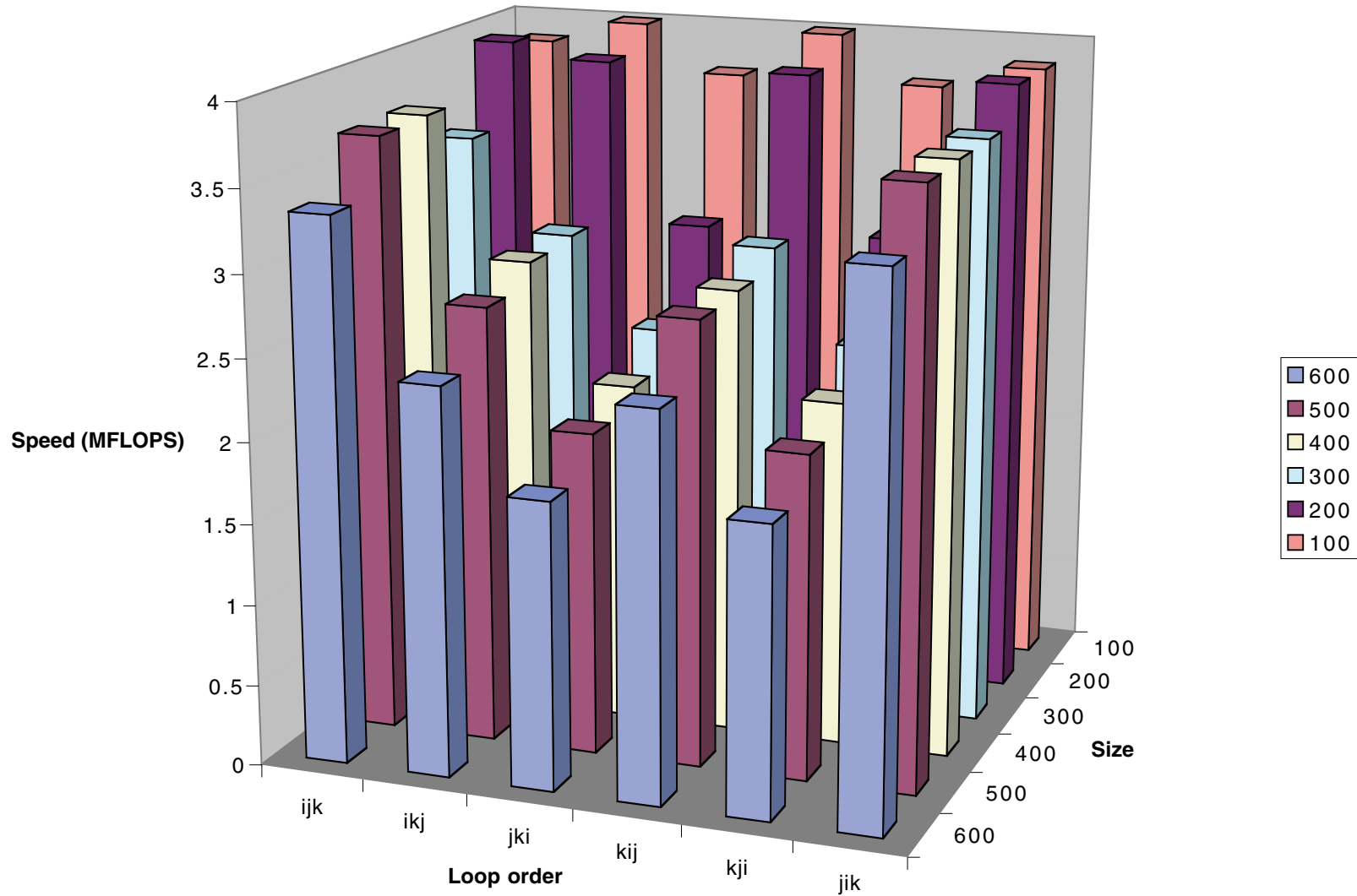
- C loop orderings jki and kji: In the innermost (i) loop in

$$c[i][k] = c[i][k] + a[i][j] * b[j][k],$$

the matrix elements $c[i][k]$ and $a[i][j]$ are accessed successively by rows, which implies stride n given the C storage order



Speed vs. C loop order (static arrays)



Speed vs. C loop order (pointers)

