Chapter 3

Quantitative Performance Theory

בפרק 2 ראינו שהמחשבים הקלאסיים שתכננו וייצרו בין 1950 — 1990 היו מסוג CISC, ארכיטקטורות מורכבות עם מימושים מורכבים. המורכבות הזאת נבעה חלקית מאילוצים טכנולוגיים כמו מחסור בזכרון ומהדרים (קומפיילרים) לא מתוחכמים. סיבה נוספת הייתה השערה ששפה עשירה דומה לשפה טבעית תאפשר תכנות יותר טובה. אבל ממחקר על ביצועי מחשבי CISC התברר שחלק מהיכולות של CISC לא בא לידי ביטוי בזמן ריצה ולכן המורכבות תרמה להאטה של המערכת ללא תמורה. הפרק הזה מציג מבוא לחקר הביצועים. בפרקים הבאים נשתמש בכלים האלה לחקור מבנים שהחליפו את ה- CISC, גישה שנקראת RISC — Reduced Instruction Set Computer.

Slides 2 — 3 Amdahl Equation for Multiprocessors

One of the first studies of computer performance was made by **Gene Amdahl** at IBM who studied the disappointing performance of parallel computing in the 1960s.

A **Symmetric Multiprocessor (SMP)** divides a programming task among N identical and equivalent CPUs. A communication network connects the processors, so that the operating system runs on one processor (or more) and the OS assigns tasks to the other processors by some scheduling method. A multicore CPU is a modern example of an SMP. It was once naively expected that programs would run N times as fast on N CPUs, but this goal was never reached.

Amdahl noticed that run time is divided between processes that can be run in parallel and processes that must run sequentially. Only the processes that can be run in parallel are enhanced by increasing the number of processors; the sequential code is the limiting factor in speedup. **Amdahl's equation** expresses this observation:

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

F_P = fraction of work that can be enhanced (parallelized)

N = speedup for part to be enhanced (number of processors)

Applying Amdahl's equation to a small server with 8 Intel Xeon processors and a workload for which 80% of the work can be run in parallel, the calculated speedup is

$$S = \frac{1}{\left(1 - 0.80\right) + \frac{0.80}{8}} = \frac{1}{0.20 + 0.10} \cong 3.33$$

Even if the number of processors were unlimited, the speedup would be limited by the 20% of the workload that runs sequentially

$$S = \frac{1}{\left(1 - F_P\right) + \frac{F_P}{N}} \xrightarrow{N \to \infty} \frac{1}{1 - F_P} = \frac{1}{1 - 0.80} = 5$$

and the maximum speed would be 5. This result shows that future enhancements to programming speed must come from increasing the degree of parallelization, that is, from better parallel programming.

Slide 4 Basic Performance Measures

As said in chapter 1, the most important performance characteristic is processing speed, measured by **Response Time**, which is the elapsed time T from start to finish of a defined program task. **Run time** is the response time for a complete program, and includes both useful work performed by the CPU and time spent waiting for external events.

Latency is the time spent waiting. Which part of run time is useful work and which part is latency depends on the situation.

Throughput is the number of defined tasks performed per unit time, that is, the task rate including latency between tasks (such as OS scheduling).

Any change to system is called an **enhancement** (even if the change degrades performance). After an enhancement, the new run time is written T '.

The **speedup** is the ratio $S = \frac{T}{T'}$ which is greater than 1 when the enhancement improves run time, in which case T' < T.

Slide 5 Processor Performance Enhancements

For a single CPU, performance can be improved by making changes to the processor hardware and to the software. Some possible changes are:

Hardware Enhancements

Increasing the clock rate

Improving instruction implementation to reduce the number of clock cycles per instruction

Improving memory organization to reduce latency in data access

Increasing the number of processing elements — multiple CPUs, multiple ALUs in each CPU, additional registers to reduce memory access

Software Enhancements

Use of run time optimizations that use CPU features more effectively

Improving the compiler to produce more efficient code

Improving the operating system to reduce run time of system code and more effectively manage threads

General outlook

It may already be clear to students of computer science that time-to-market and generic code reusability are basic goals in software development today. Software performance and code efficiency are generally of interest in specific sectors — gaming, high performance scientific computing and embedded processor-based systems. Hardware must become increasingly faster to permit increasingly bloated code to run at all.

Enhanced Run Time

The total run time is the sum of all the partial run times. Any given enhancement may affect partial run times so that some are longer, some are shorter, and some are unchanged. The speedup will be greater than 1 (net improvement) as long as the sum of new partial run times is less than the sum of old partial run times. This is seen in slide 6, where blue sections of run time are enhanced while green sections remain unchanged.

Slides 7 — 8 Amdahl Equation

Another form of Amdahl's equation can be found by comparing the sums of partial run times. The basic definitions are:

T	total run time of a task = the sum of the partial run times
T'	total run time of a task after enhancement = the sum of the partial run times after enhancement
t_e	partial run time that can be enhanced = sum of partial run times that can be enhanced
t_e'	partial run time that can be enhanced, after enhancement = sum of partial run times that were enhanced $t_e' < t_e$
t_0	partial run time that cannot be enhanced and does not change

From these definitions, we define two relative quantities:

F_e	Relative fraction of the run time that can be enhanced = $t_e / T \leq 1$
S_e	Speedup of the portion of run time that can be enhanced = $t_e / t_e' > 1$

By simple algebra we find Amdahl's equation in the form

$$S = \frac{T}{T'} = \frac{T}{t_0 + t'_e} = \frac{1}{\frac{t_0}{T} + \frac{t'_e}{T}} = \frac{1}{\frac{T - t_e}{T} + \frac{t'_e}{t_e}\frac{t_e}{T}} = \frac{1}{1 - F_e + \frac{1}{S_e}F_e}$$

which expresses the speedup in terms of relative quantities. The actual run times are not required to calculate speedup if the relative enhancements are known.

The form of Amdahl's equation for SMP is found by assigning

F_e = fraction of program that can be parallelized = F_p

S_e = speedup of parallelizable part of program = number of processors N

As an example of how Amdahl's equation works, consider the following enhancement to the float instructions:

	Before		After	
Integer	t_{INT}	100 ms	t_{INT}	100 ms
Float	t_{FP}	300 ms	t_{FP}'	200 ms
Total	T	400 ms	T'	300 ms

The partial run time of the floats is decreased from 300 ms to 200 ms and so the total run time is decreased from 400 ms to 300 ms. The speedup is S = T/T' = 400 ms / 300 ms = 4/3.

The relative quantities and speedup from Amdahl's equation are

$$F_e = \frac{t_{FP}}{T} = \frac{300\ \text{ms}}{400\ \text{ms}} = \frac{3}{4} = 75\%$$

$$S_e = \frac{t_{FP}}{t_{FP}'} = \frac{300\ \text{ms}}{200\ \text{ms}} = \frac{3}{2} = 1.50$$

$$S = \frac{1}{1 - F_e + \frac{F_e}{S_e}} = \frac{1}{1 - \frac{3}{4} + \frac{3/4}{3/2}} = \frac{1}{\frac{1}{4} + \frac{1}{2}} = \frac{4}{3}$$

Slide 9 Application of Amdahl Equation

To see how the Amdahl equation can be applied, we consider two alternative enhancements to a CPU and use Amdahl's equation to determine which provides better speedup.

For the CPU under study, Float (FP) instructions account for 50% of total run time. Among all the FP instructions, just the square root (FP) alone accounts for 20% of total run time.

Two enhancements are proposed:

Speedup of $S_e = 2$ for all FP instructions

Speedup of $S_e = 10$ for just the square root instruction

The information provided is in the form required for Amdahl's equation.

Enhancement 1

$F_e = 0.50$ and $S_e = 2$ leads to

$$S_1 = \frac{1}{1 - F_e + \frac{1}{S_e} F_e} = \frac{1}{1 - 0.50 + \frac{0.50}{2}} = \frac{1}{0.50 + 0.25} \cong 1.33 \Rightarrow 33\% \text{ speedup}$$

Enhancement 2

$F_e = 0.20$ and $S_e = 10$ leads to

$$S_2 = \frac{1}{1 - F_e + \frac{1}{S_e} F_e} = \frac{1}{1 - 0.20 + \frac{0.20}{10}} = \frac{1}{0.80 + 0.02} \cong 1.22 \Rightarrow 22\% \text{ speedup}$$

This shows a general rule of thumb known informally as **Amdahl's "law"** — a smaller partial speedup with a wide area of application provides a larger overall speedup than a large partial speedup with a small area of application.

Slides 10 — 11 Generalized Amdahl Equation

Most practical enhancements will **degrade** some partial run times.

Besides t_e = the run time that can be **enhanced** ($t_e' < t_e$) and t_0 = the **unchanged** run time there will also be t_d = the run time that is **degraded** ($t_d' > t_d$).

In relative terms, $F_d = t_d / T$ (the relative fraction of run time that is degraded) and $S_d = t_d / t_d' < 1$ (the speedup of the degraded partial run time). Now Amdahl's equation is

$$S = \frac{T}{T'} = \frac{T}{t_0 + t'_e + t'_d} = \frac{1}{\frac{T - t_e - t_d}{T} + \frac{t'_e}{t_e}\frac{t_e}{T} + \frac{t'_d}{t_d}\frac{t_d}{T}} = \frac{1}{1 - (F_e + F_d) + \frac{F_e}{S_e} + \frac{F_d}{S_d}}$$

As an example, the shift from a CISC ISA to a RISC ISA makes 94% of run time 5 times faster ($S_e = 5$), degrades 1% of the run time by a factor of 10 ($S_d = 1/10$), and leaves 5% of run time unchanged. From the Amdahl equation we find

$$S = \frac{1}{1 - (F_e + F_d) + \frac{F_e}{S_e} + \frac{F_d}{S_d}} = \frac{1}{1 - (0.94 + 0.01) + \frac{0.94}{5} + \frac{0.01}{1/10}} = \frac{1}{0.05 + 0.19 + 0.10} \cong 2.94$$

The speedup is about 3 times faster than CISC, even though some operations are slower.

Slides 12 — 13 Detailed Analysis of CPU Run Times

The use of the Amdahl equation requires us to know the relative run time data. This may be useful in qualitative comparisons, but the relative numbers are not generally available.

The detailed run time information — as seen in the SPEC benchmark — is generally found from measurements on running programs. But to make measurements on running programs requires an implementation of the CPU, and we are not usually interested in building the CPU until we can predict its performance. **CPU analysis** predicts run time without having to build the CPU.

Assumptions in CPU analysis

1. All instructions can be grouped together according to resource usage. For example, ADD R1, R2, R3 and SUB R1, R2, R3 differ only in the ALU operation
2. All instructions in a group run in the same number of clock cycles
3. Every clock cycle measures same the unit of time
4. Instruction run time = seconds per clock cycle × number of clock cycles per instruction
5. Group run time = instruction run time × number of instructions in the group
6. Total run time = sum of instruction group run times

The **basic definitions** are

$T =$ total run time of program

$t_i =$ total run time of instructions in group i

$IC_i =$ number of instructions in group i (Instruction Count)

$CPI_i =$ number of clock cycles to run 1 instruction in group i (Cycles Per Instruction)

$N_i =$ number of clock cycles to run all instructions in group i

$\tau =$ seconds per clock cycle = clock period

$R =$ clock rate $=$ clock frequency $=$ clock cycles per second $=$ Hertz (Hz) $= 1/\tau$

Using these definitions we may calculate the following quantities:

$IC =$ total number of instructions in the program

$N =$ total number of clock cycles to run the program

$CPI =$ average number of clock cycles per instruction for the program = $\frac{N}{IC}$

In general, we write the value of some *quantity* after enhancement as *quantity*'.

Slides 14 — 15 CPU Equation

The **average CPI** (cycles per instruction) characterizes the effort required by a CPU in processing instructions. A CPU with a low CPI requires fewer clock cycles to run a typical instruction than a CPU with a high CPI. The **CPU equation** permits calculation of the average CPI (cycles per instruction) from detailed information about the CPU.

The **number of clock cycles** required to run all instructions of **type *i*** is given by

$$N_i = \text{number of instructions of type } i \times \frac{\text{clock cycles}}{\text{instruction of type i}} = IC_i \times CPI_i$$

Summing over all of the instruction groups, the **total number of clock cycles** required to run all instructions is

$$N = \sum_{\text{all groups } i} N_i = \sum_i IC_i \times CPI_i$$

The average number of clock cycles per instruction is defined as

$$CPI = \frac{\text{total number of clock cycles to run program}}{\text{total number of instructions in program}} = \frac{N}{IC}$$

and using the expression for N above leads to

$$CPI = \frac{1}{IC}\sum_i N_i = \frac{1}{IC}\sum_i IC_i \times CPI_i = \sum_i CPI_i \times \frac{IC_i}{IC}$$

Notice that the ratio $\frac{IC_i}{IC}$ is the proportion (percentage) of instructions from group i in the full program and since the sum of these ratios is 1, the set of ratios forms a **probability distribution** for the instructions in the program. The average CPI is a weighted average of the individual values of CPI_i for each group i.

Slide 15 shows an **example** of how these calculations fit together. A certain **program** has 5000 integer instructions, 1000 branch instructions, and 4000 load/store instructions. The total number of instructions is the sum IC = 10,000. The program is thus 50% integer, 10% branch and 40% load/store.

On a certain CPU the CPI_i (number of clock cycles required to run one instance of an instruction from group i) is 4 CC for integer, 12 CC for branch, and 8 CC for load/store. All the integer instructions together require 4 CC per instruction × 5000 instructions = 20,000 CC to run. Similarly, the branch instructions run in 12,000 CC and the load/store instructions run in 32,000 CC. Adding these together, the complete program will run in N = 64,000 CC.

The average number of cycles per instruction is then CPI = N / IC = 60,000 / 10,000 = 6.4

The same number is found from the CPU equation as

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = 4 \times 0.50 + 12 \times 0.10 + 8 \times 0.40 = 6.4 \text{ cycles per instruction}$$

Slides 16 — 17 CPU Run Time

The **time required to run one instance of an instruction of type *i*** is just the number of clock cycles to run that instruction multiplied by the number of seconds in a clock cycle:

$$\text{single instruction run time} = \frac{\text{clock cycles}}{\text{instruction of type } i} \times \frac{\text{seconds}}{\text{clock cycle}} = CPI_i \times \tau$$

Multiplying by the number of instruction in the group, the **run time for all instructions of type *i***

$$t_i = \text{instructions of type } i \times \frac{\text{clock cycles}}{\text{instruction of type } i} \times \frac{\text{seconds}}{\text{clock cycle}} = IC_i \times CPI_i \times \tau$$

Adding the contributions from all groups, the **total run time for the program** is

$$T = \sum_{\textbf{all groups } i} t_i = \sum_i CPI_i \times IC_i \times \tau = \left(\sum_i CPI_i \times \frac{IC_i}{IC} \right) \times IC \times \tau$$

Since the quantity in parentheses is the average CPI (from the CPU equation), this is

$$T = CPI \times IC \times \tau = \text{clock cycles per instruction} \times \text{number of instructions} \times \text{seconds per clock cycle}$$

Slide 17 shows an **example** of the run time calculation. The instructions are divided into 5 instruction groups. For each group we are shown the percentage of instructions in that group and the number of clock cycles required to run one instruction from that group. Using the CPU equation, the average CPI is found to be 10.5 clock cycles per instruction. We are given the total number of instructions in the program is 1,000,000 and the clock speed R = 100 MHz. Since the clock cycle time is 1 / R, we can calculate the program run time as 0.105 seconds.

Slides 18 — 20 C Code to Runtime Example

Another example begins with a program written in C.

The program is **compiled** to assembly language and the assembly code is sorted into **instruction types**: load, $ALU_{\text{auto-increment}}$, ALU, and JMP. The first two instructions run once each (2 instructions) and the three instructions in the loop run 5 time each (15 instructions) for a total IC =17. In relative terms, 1/17 = 6.5% and 5/17 = 29%. So the loads are 13% and the other three types are 29% each.

On slide 19 we show a sample interpretation of an assembly instruction into a **microprogram**. Assuming an instruction queue mechanism (as in the 8086) that prefetches instructions, this instruction will run in $CPI_{\text{ALU-AI}}$ = 9 clock cycles (9 lines of microcode).

Slide 20 shows the results of similar analysis for the other instructions. With the distribution of instructions in the program and the CPI_i of each instruction type, the average CPI is found from the **CPU equation** to be 7.2. The total **number of clock cycles** is N = CPI × IC = 7.2 × 17 = 122. Multiplying by the clock period (10^{-9} sec at 1 GHz) gives the **run time** as 0.122 microseconds.

Slides 21 Applying CPU Equation

The main application of the CPU equation is to calculate speedup. The run time before enhancements is calculated as in the examples above. Following some architectural enhancement, there will be new values of IC', CPI' and τ'. These permit calculation of the new run time and the speedup as

$$S = \frac{T}{T'} = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'}$$

Slides 22 — 30 CPU Equation — Example 2

As another example, on a certain CPU:

25% of all instructions in an average program are floating point (IC_{FP} / IC = 0.25). The FP group includes 5 operations: ADD, SUB, MULT, DIV, SQRT.

Each FP instruction runs in a different number of clock cycles. The average FP instruction runs in 4 clock cycles (CPI_{FP} = 4).

The SQRT instruction is the most complex. Only 2% of all instructions in the typical program are square root (IC_{SQRT} / IC = 0.02) but each SQRT runs in 20 cycles (CPI_{SQRT} = 20).

The average CPI for all other instructions in program is 4/3 clock cycles (CPI_{other} = 4/3). Since the FP instructions are 25% all the other instructions are 75%.

Based on this information we can calculate

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = CPI_{FP} \times \frac{IC_{FP}}{IC} + CPI_{other} \times \frac{IC_{other}}{IC} = 4 \times 0.25 + \frac{4}{3} \times (1 - 0.25) = 2.00$$

Slide 23 describes two possible enhancements:

Enhancement 1 improves performance of all FP instructions

Reduce average CPI_{FP} = 4 cycles to CPI_{FP}' = 2 cycles

There is no change in the program content so $IC_i' = IC_i$ for all instruction types

There is no change to clock rate $\tau' = \tau$

Enhancement 2 improves performance of SQRT (FP) instruction

Reduce CPI_{SQRT} = 20 cycles to CPI_{SQRT}' = 2 cycles

There is no change in the program content so $IC_i' = IC_i$ for all instruction types

There is no change to clock rate $\tau' = \tau$

Slide 24 shows the straightforward **calculation for enhancement 1**. CPI_{FP} = 4 cycles is replaced by CPI_{FP}' = 2 cycles and all other factors remain unchanged. The average CPI changes from 2 CC per instruction to 1.5 CC per instruction. The speedup is then found from S = T / T' where since $T' = CPI' \times IC' \times \tau' = CPI' \times IC \times \tau$ the speedup is just the ration of CPIs = 1.33. Enhancement 1 provides a 33% speedup in performance.

Slide 25 shows that calculation of the **speedup for enhancement 2** requires splitting the FP group into SQRT and FP/SQRT which is the group of FP instructions without SQRT, {ADD, SUB, MULT, DIV}. Since FP is 25% of all instructions and SQRT is 2% of all instructions, the group FP/SQRT is 23% of all instructions. Now the CPI' is found from

$$CPI' = \sum_i CPI'_i \times \frac{IC_i'}{IC'}$$

$$= CPI'_{FP/SQRT} \times \frac{IC_{FP/SQRT}'}{IC'} + CPI'_{SQRT} \times \frac{IC_{SQRT}'}{IC'} + CPI'_{other} \times \frac{IC_{other}'}{IC'}$$

$$= CPI_{FP/SQRT} \times \frac{IC_{FP/SQRT}}{IC} + CPI'_{SQRT} \times \frac{IC_{SQRT}}{IC} + CPI_{other} \times \frac{IC_{other}}{IC}$$

where only CPI'_{SQRT} is changed from its previous value. But there is one missing piece of information that must be calculated first: We do not know the $CPI_{FP/SQRT}$ of the group FP/SQRT.

To find $CPI_{FP/SQRT}$ we must use the definition of average CPI. We were told that the average CPI_{FP} for the group of FP instructions is 4. This means that the total number of CC for FP instructions divided by the total number of FP instructions is 4.

$$4 = CPI_{FP} = \frac{\text{total } FP \text{ cycles}}{FP \text{ instructions}} = \frac{N_{FP}}{IC_{FP}} = \frac{\sum_{k \in FP} N_k}{IC_{FP}} = \sum_{k \in FP} \frac{CPI_k \times IC_k}{IC_{FP}} = \sum_{k \in FP} CPI_k \times \frac{IC_k}{IC_{FP}}$$

The parameter that we must find is $CPI_{FP/SQRT}$ which is defined as

$$CPI_{FP/SQRT} = \frac{\text{total } FP/SQRT \text{ cycles}}{FP/SQRT \text{ instructions}} = \frac{N_{FP/SQRT}}{IC_{FP/SQRT}} = \sum_{k \in FP/SQRT} \frac{N_k}{IC_{FP/SQRT}} = \sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP/SQRT}}$$

On slide 27 we combine these expressions:

1. Write the definition of CPI_{FP}

$$CPI_{FP} = \sum_{k \in FP} CPI_k \times \frac{IC_k}{IC_{FP}}$$

2. Split the sum into FP/SQRT and SQRT

$$= \sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP}} + CPI_{SQRT} \times \frac{IC_{SQRT}}{IC_{FP}}$$

3. Insert two factors of 1

$$= \left[\sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP}} \right] \times \frac{IC_{FP/SQRT} / IC}{IC_{FP/SQRT} / IC} + \left[CPI_{SQRT} \times \frac{IC_{SQRT}}{IC_{FP}} \right] \times \frac{1/IC}{1/IC}$$

4. In the first term exchange IC_{FP} with $IC_{FP/SQRT}$. In the second term move $1/IC$ inside.

$$= \left[\sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP/SQRT}} \right] \frac{IC_{FP/SQRT} / IC}{IC_{FP} / IC} + CPI_{SQRT} \times \frac{IC_{SQRT} / IC}{IC_{FP} / IC}$$

5. The factor in parentheses in the first term is precisely the definition of $CPI_{FP/SQRT}$

$$= CPI_{FP/SQRT} \times \frac{IC_{FP/SQRT} / IC}{IC_{FP} / IC} + CPI_{SQRT} \times \frac{IC_{SQRT} / IC}{IC_{FP} / IC}$$

Plugging in the numbers we know we find

$$4 = CPI_{FP/SQRT} \times \frac{0.23}{0.25} + 20 \times \frac{0.02}{0.25} \quad \Rightarrow \quad CPI_{FP/SQRT} = 2.61$$

We notice that the large $CPI_{SQRT} = 20$ raises the average CPI_{FP} from 2.61 to 4 even though SQRT is only 0.02 / 0.25 = 8% of all FP instructions.

On slide 28 we find the **speedup for enhancement 2**. Using

$$CPI' = CPI_{FP/SQRT} \times \frac{IC_{FP/SQRT}}{IC} + CPI'_{SQRT} \times \frac{IC_{SQRT}}{IC} + CPI_{\text{other}} \times \frac{IC_{\text{other}}}{IC} = 1.64$$

the speedup is S = T / T' = 2 / 1.64 = 1.22.

We see that the dramatic improvement in SQRT has less overall effect than the more moderate improvement to all the FP instructions in enhancement 1.

Slide 29 shows a trick for calculating $CPI_{FP/SQRT}$ in the case that the instruction count does not change:

Add zero = $CPI - CPI$

$$CPI' = CPI - (CPI - CPI')$$

Expand CPI and CPI'

$$= CPI - \left(\sum_i CPI_i \times \frac{IC_i}{IC} - \sum_i CPI'_i \times \frac{IC'_i}{IC'} \right)$$

Change order of addition and subtraction

$$= CPI - \sum_i \left(CPI_i \times \frac{IC_i}{IC} - CPI'_i \times \frac{IC'_i}{IC'} \right)$$

When $IC'_i = IC_i$ remove common factor of $\frac{IC_i}{IC}$

$$CPI' = CPI - \sum_i \left[(CPI_i - CPI'_i) \times \frac{IC_i}{IC} \right]$$

For the case of $SQRT$

$$= CPI - \left[(CPI_{SQRT} - CPI'_{SQRT}) \times \frac{IC_{SQRT}}{IC} \right]$$

$$= 2.00 - \left[(20 - 2) \times 0.02 \right]$$

$$= 1.64$$

This method only works if the structure of the program does not change.

It should be recalled that the enhancements in this example are similar to those in the example of the **Amdahl equation** (slide 9) and the speedups are identical. Slide 30 shows that the relative quantities given in the case of Amdahl can be derived from the parameters in this example.

From the definitions of F_e and S_e (slide 7) we can write

$$F_e^{FP} = \frac{t_{FP}}{T} = \frac{CPI_{FP} \times IC_{FP} \times \tau}{CPI \times IC \times \tau} = \frac{4 \times (0.25 \times IC) \times \tau}{2 \times IC \times \tau} = 50\%$$

$$S_e^{FP} = \frac{t_{FP}}{t'_{FP}} = \frac{CPI_{FP} \times IC_{FP} \times \tau}{CPI'_{FP} \times IC'_{FP} \times \tau'} = \frac{4}{2} = 2$$

$$F_e^{SQRT} = \frac{t_{SQRT}}{T} = \frac{CPI_{SQRT} \times IC_{SQRT} \times \tau}{CPI \times IC \times \tau} = \frac{20 \times (0.02 \times IC) \times \tau}{2 \times IC \times \tau} = 20\%$$

$$S_e^{SQRT} = \frac{t_{SQRT}}{t'_{SQRT}} = \frac{CPI_{SQRT} \times IC_{SQRT} \times \tau}{CPI'_{SQRT} \times IC'_{SQRT} \times \tau'} = \frac{CPI_{SQRT}}{CPI'_{SQRT}} = \frac{20}{2} = 10$$

which recovers the parameters given in the Amdahl equation example.

Slides 31 — 32 Changing Instruction Mix

Up to now we have discussed examples in which an architectural enhancement changes CPI_i for some group but the **instruction distribution** represented by the ratios IC_i / IC is unchanged. Because CPI is a weighted average of the CPI_i values for the various groups, changing the instruction mix will generally change CPI.

As an example, slide 31 shows a CPU that runs a **program of 10,000 instructions** for which the average CPI is calculated to be 6.4 cycles per instruction. The (unrealistic) enhancement made in this case is that **2,000 ALU instructions are erased** from the program (with no reasonable explanation).

For the new program, the total IC is now 10,000 – 2,000 = 8,000. As a result, all of the relative weights IC_i / IC are changed. For example, IC_{branch} = 1000 / 10000 = 10% becomes IC'_{branch} = 1000 / 8000 = 12.5%.

The removal of many fast ALU instructions (CPI_{ALU} = 4) means that this group has a smaller weight in the program: from 50% down to 37.5%. Calculation of the **average CPI** shows that it rises from 6.4 to 7.0 because the slower instructions (with more cycles per instruction) are a bigger part of the instruction distribution.

As shown on slide 32 the **speedup** is 14%, because the rise in CPI is compensated by the fall in the instruction count. This example shows that speedup must be calculated from comparative run times and not only from CPI.

Slides 33 — 35 The Instructions Per Second Myth

Two measures often used to describe computing power are

MIPS = million instructions per second

FLOPS = floating point operations per second

But these measures cannot provide a fair comparison because

$$\text{MIPS} = \frac{\text{instructions} / 10^6}{\text{run time}} = \frac{IC}{10^6 \times T} = \frac{IC}{10^6 \times CPI \times IC \times \tau} = \frac{R}{10^6 \times CPI} = \frac{R(\text{in MHz})}{CPI}$$

which does not include instruction count.

For example, consider similar processors **CPU-1** and **CPU-2** which both run ALU instructions in 1 cycle and all other instructions in 2 cycles. Both have clock speed of 1 GHz = 1000 MHz. The processors differ in their compiler:

CPU-1 compiler produces 50% ALU instructions and 50% other

CPU-2 compiler is more efficient and produces 25% fewer ALU instructions than CPU-1

The CPU equation provides $CPI_1 = 1 \times 0.50 + 2 \times 0.50 = 1.50$ which means that

$$\text{MIPS}_1 = \frac{1000}{1.50} \cong 667 \text{ million instructions/sec}$$

To find the CPI for CPU-2 we first find the instruction distribution (slide 34):

We know that on CPU-1, $IC_1^{ALU} / IC_1 = 50\%$ and so $IC_1^{ALU} = 0.5 \times IC_1$.

We also know that on CPU-2 the compiler produced 25% fewer ALU instructions which means that $IC_2^{ALU} = (1 - 0.25) \times IC_1 = 0.375 \times IC_1$.

The other instructions are not affected and so $IC_2^{other} = IC_1^{other} = 0.5 \times IC_1$ (the other 50% of instructions). Thus

$IC_2 = IC_2^{ALU} + IC_2^{other} = 0.375 \times IC_1 + 0.5 \times IC_1 = 0.875 \times IC_1$

meaning that CPU-2 has 87.5% as many instructions as CPU-1 (12.5% fewer instructions).

This allows us to calculate

$IC_2^{ALU} / IC_2 = 0.375 \times IC_1 / (0.875 \times IC_1) = 0.375 / 0.875 = 0.43$

$IC_2^{other} / IC_2 = 0.5 \times IC_1 / (0.875 \times IC_1) = 0.5 / 0.875 = 0.57$

Now the average $CPI_2 = 1 \times 0.43 + 2 \times 0.57 = 1.57$ which is larger than $CPI_1 = 1.5$. It is not surprising that CPU-2 has a **lower MIPS** of 1000 / 1.57 = 637 score than CPU-1.

Nevertheless, we expect that CPU-2 is faster than CPU-1 and calculating the speedup

$$S = \frac{T_1}{T_2} = \frac{CPI_1 \times IC_1 \times \tau_1}{CPI_2 \times IC_2 \times \tau_2} = \frac{CPI_1 \times IC_1}{CPI_2 \times IC_2} = \frac{1.50 \times IC_1}{1.57 \times (0.875 \times IC_1)} \cong 1.09$$

we see that this is true.

Slides 36 — 39 Replacing Instruction Types

The problem of changing the instruction distribution begins with summing the IC_i for the instruction groups. Generally,

$IC = IC_1 + IC_2 + ... + IC_n$

Where Type 1 = ALU, Type 2 = Conditional Branch, and so on.

Replacing an instruction in the ISA may affect more than one instruction type. For example, the Intel instruction **`loop target`** replaces the 3 instructions

```
DEC CX          ; CX ← CX – 1 (ALU type on register CX)
CMP CX, 0       ; compare CX == 0   (ALU type)
JNZ target      ; branch to target if CX != 0 (branch type)
```

Adding the loop instruction to the ISA will change both IC'_1 and IC'_2.

Slides 37 – 39 show another example, the **addition of a floating point unit** (FPU) to a CPU which did not have one. If no FPU is present in a CPU then FP instructions are executed by **emulation**. This means that the FP operation is converted to a sequence of integer instructions.

For example, to multiply $(2.165 \times 10^4) \times (3.247 \times 10^{-3})$ the compiler produces code that performs integer multiplication of 2165 × 3247 = 7029755, calculates exp = 4 – 3 + (–3) – 3 = –5 and shifts the result to $7.030 \times 10^6 \times 10^{-5} = 7.030 \times 10^1$. In this situation, the compiler produces a large number of ALU instructions that emulate FPU operations.

Slide 37 shows the instruction distribution for a CPU with no FPU. ALU instructions run in 1 CC per instruction are since they are 75% the average CPI is small.

Slide 38 shows the new distribution when an FPU is added to the CPU.

First we split the old ALU instructions into two groups: ALU^{INT} represents actual ALU instructions and $ALU^{EMULATION}$ represents ALU instructions for FPU emulation. We are told that 2/3 of the old ALU instructions emulate FPU instructions. Therefore

$ALU^{EMULATION}$ = (2/3) × 75% = 50% of all instructions

ALU^{INT} = (1/3) × 75% = 25% of all instructions

The ALU^{INT} instructions will still be in the program after addition of the FPU, but the $ALU^{EMULATION}$ instructions will be replaced. We are told that 1 new FPU instruction replaces 10 $ALU^{EMULATION}$ instructions. Therefore

IC'_{ALU} = 0.25 IC

IC'_{FPU} = (1/10) × 0.50 IC = 0.05 IC

The values of IC for the other groups are unchanged so that IC'_{other} = (1 – 0.75) IC = 0.25 IC and

IC' = IC'_{ALU} + IC'_{FPU} + IC'_{other} = 0.55 IC

Slide 39 shows the calculation of the new distribution and the average CPI'. For example

IC'_{ALU} / IC' = 0.25 IC / 0.55 IC = 0.25/0.55 = 0.46

The average CPI' = 0.95/0.55 is found from the CPU equation.

The speedup is found by comparing the run times

$$S = \frac{T}{T'} = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{1.25 \times IC}{\frac{0.95}{0.55} \times (0.55IC)} = \frac{1.25}{0.95} \cong 1.32$$

where we see that the factor of 0.55 cancels as expected.

Slides 40 — 43 Load-Store versus Register-Memory

As another example, we consider the **register-memory instructions**.

CPU-1 is a load-store machine — all ALU operands must come from a register, so that a memory operand must be loaded to a register before an ALU operation uses it. Slide 40 shows the **instruction distribution**, the $\mathbf{CPI_i}$ for the instruction groups and the calculation of the average **CPI**.

We are told that 25% of all ALU memory operands are used only in only 1 ALU operation. This means that 75% of memory operands are used in more than 1 operation and it is more efficient to keep them in a register for multiple accesses.

To consider the register-memory instruction for the 25% of memory operands that are used just once, we define CPU-2 as an **ideal register-memory machine** that operates as follows:

> For 75% of memory operands used in multiple ALU operations, the perfect compiler loads these **"multiple" memory operands** to registers.
>
> For 25% of ALU memory operands used in only a single ALU operation, the perfect compiler does not load these **"single" memory operands** to registers, but uses them directly from memory.

To calculate the speedup we split the ALU operations into ALU^{multi} and ALU^{single} so that we replace ALU^{single} with $ALU^{register\text{-}memory}$ and cancel 1 register load for every ALU^{single}.

Of the 40% of instructions that are ALU, 75% are ALU^{multi} so $IC^{ALU\text{-}multi}$ = 0.30 IC and $IC^{ALU\text{-}single}$ = 0.10 IC. This split is shown on slide 42.

We must also split the load instructions into $IC^{load\text{-}single} = IC^{ALU\text{-}single}$ =0.10 IC (one load instruction for each ALU^{single} instruction) and $IC^{load\text{-}multi} = IC^{load}$ – 0.10 IC = (0.25 – 0.10) IC = 0.15 IC.

Since the load and ALU take 4+5 clock cycles, the new instruction $ALU^{register\text{-}memory}$ runs in 9 clock cycles.

Slide 43 shows the new table after replacing ALU^{single} and $load^{single}$ with $ALU^{register\text{-}memory}$. The total instruction count is IC – 10% IC – 10% IC + 10% IC = 0.90 IC.

Calculating CPI' we see a small rise which exactly compensates the smaller number of instructions. All together, the speedup is 1 meaning no change in performance.

Slides 44 – 50 Analysis of 8086 Example

In chapter 1 (slides 17 – 22) we considered compiler optimization for an Intel 8086 CPU. Slide 44 shows the program before optimization, the number of clock cycles per instruction and the number of time each instruction runs. Adding the entries in the Runs column we see that the total number of instructions is IC = 7N – 3. The IC_i / IC for each row is just the entry in Runs divided by 7N – 3.

In order to compute IC_i / IC we must first find averages for the instructions that require different numbers of cycles in different conditions. On slide 45 we calculate the averages from their definitions.

For example JGE runs N – 1 times in 4 CC and 1 time in 13 CC. The total number of runs is N – 1 + 1 = N and the total number of cycles is 4(N – 1) + 13. We write CPI_{JGE} = (4(N – 1) + 13)/N. We have a similar result for the store.

Slide 46 shows the instruction distribution for the program and the CPI_i for each row. Slide 47 shows the numerical values for N = 100. Using these values we calculate CPI = 9.45 and using R= 4 MHz, find the run time T = 1.646 ms.

Slide 48 shows the optimized program (using register variables which improves CPI_i for access to operands) and the slide 49 shows the new distribution after replacement of memory access instructions by register access instructions. The updated CPI' = 4.31 is found on slide 50 and the run time T' = 0.7515 ms. The speed up is thus S = T / T' = 2.19.

Slides 51 – 52 Register Variables and Two Threads

On slide 51 we split the program into two threads using the method of **Data Decomposition**. Thread 0 performs the work of loop iterations 0 to 49 (of the original non-threaded sequential code) and Thread 1 performs the work of loop iterations 50 to 99. Now each thread will run in parallel (at the same time) on two processor cores. In the original code, all the iterations run sequentially on one core. The number of iterations is now N = 100/2 = 50 and the instruction distribution changes very slightly (values from Slide 49 with N = 50). As expected the threaded version runs almost twice as fast. We see that S < 2 because the first and last instructions — a very small part of the run time — do the same work on each core and are therefore non-parallel.

Slides 53 – 62 Software Content — Instruction Distribution

The method of CPU analysis is summarized on slides 51 – 60.

Slide 53 shows a sample instruction distribution with the CPI_i value for each group.

Slide 54 calculates the total IC

Slide 55 calculates the relative values IC_i / IC

Slide 56 calculates the relative run time per instruction group $CPI_i \times IC_i$ / IC

Slide 57 calculates the average CPI

Slide 58 calculates the total number of clock cycles N and the run time T

Slide 59 calculates the number of clock cycles N_i per instruction group

Slide 60 compares the results of slides 56 and 57

Slide 61 calculates the relative fraction of run time F_i

Slide 62 shows the relationship of CPU analysis and Amdahl's law

---

Slide 61 Instruction Distributions

CPU analysis permits performance analysis of a new machine design "on the drawing board" and allows us to evaluate a proposed design without building a CPU implementation. The procedure can be summarized as:

1. Specify Instruction Set Architecture (ISA). The ISA describes the machine language for proposed CPU and provides a human-readable assembly language. The proposed microarchitecture associated with the ISA determines CPI_i for each instruction group i, by allowing us to count the number of clock cycles required to implement a single instruction in the ISA.
2. Write C, C++, Fortran compilers for the proposed machine language
3. Compile representative programs to machine language. To insure that the programs are representative, we can use programs from SPEC CINT and CFP.
4. Sort the instructions in the compiled machine language code into groups and calculate the relative instruction count IC_i/IC for each group i.
5. Calculate average CPI and the run time T based on a predicted clock speed.
6. Compare the run time with a reference machine to decide whether the CPU design is worth implementing.