Qualitative Computer Design

Design guided by measured performance.

Covered:

- Design Principles (1.6)
- Principles Applied to Processor Design (1.6)
- Benchmarks (1.5)

(Numbers refer to book sections.)

Principles of Computer Design

Principles computer designers apply widely.

- Make the common case fast.
  Obviously.
- Amdahl's Law: Don't make common case too fast.
  As speed of one part increases. . .
  . . .impact on total performance drops.
- Locality of Reference.
  Temporal: It might happen again soon.
  Spatial: It might happen to your neighbors soon too.

Make the common case fast.

Consider a system with many parts.

Consider design options for a part:

Simple Design:

Can be quickly explained, compactly represented.

Design/validation completed quickly.

Resulting design may operate slowly.

Sophisticated Design:

Design must consider multiple situations, has lengthy description.

Design time longer.

Validation time (if done properly) much longer.

When design time is limited by a deadline:

- Use simple, but correct designs for all parts.
- Then make the most common cases fast.

Amdahl's Law: don't make common case too fast.

Speedup as a Performance Measure

Many systems evaluated by execution time.

New design compared to old one by how much faster it is, *speedup.*

Let t_{T} denote the speed of the old system and $t_{\mathrm{T}'}$ the speed of the new one.

Then speedup is defined to be $S_{\mathrm{T}} = \dfrac{t_{\mathrm{T}}}{t_{\mathrm{T}'}}$.

For example, if $S_{\mathrm{T}} = 3$ then new system takes one third the time.

Consider a system with multiple parts . . .
. . . one of which is being considered for improvement.

Amdahl's law relates speedup of part to speedup of system.

Let t_{B} and $t_{\mathrm{B}'}$ denote speed of part B before and after improvement.

Let S_{B} denote speedup of part B.

Define $f = \dfrac{t_{\mathrm{B}}}{t_{\mathrm{T}}}$, the fraction of time contributed by B.

Speedup can be written: $S_{\mathrm{T}} = \dfrac{t_{\mathrm{T}}}{(1-f)t_{\mathrm{T}} + t_{\mathrm{B}'}}$

Applying $t_{\mathrm{B}'} = \frac{t_{\mathrm{B}}}{S_{\mathrm{B}}} = \frac{f t_{\mathrm{T}}}{S_{\mathrm{B}}}$ yields $S_{\mathrm{T}} = \dfrac{t_{\mathrm{T}}}{(1-f)t_{\mathrm{T}} + f t_{\mathrm{T}}/S_{\mathrm{B}}}$

Cancelling, $S_{\mathrm{T}} = \dfrac{1}{(1-f) + f\frac{1}{S_{\mathrm{B}}}}$

When f is close to zero, impact of S_{B} is small.

Locality

The first two principles are "common sense".

However, locality is a <u>characteristic</u> of executing programs ...

... which has held and is expected to continue to hold.

Because many designs work only when locality is present ...

... if it all of a sudden programs did not exhibit locality ...

... computers would run them many, many times slower!

Locality usually applied to memory addresses issued by processor.

Temporal: there's a good chance that an address used will be used again soon.

Spatial: once an address is used there's a good chance a nearby address will be used.

For examples, analyze execution of almost any program.

Components of CPU Performance and Performance Equation

CPU Performance Decomposed into Three Components:

- Clock Frequency (ϕ)
  Determined by technology and influenced by organization.
- Clocks per Instruction (CPI)
  Determined by organization and instruction mix.
- Instruction Count (IC)
  Determined by program and ISA.

These combined to form *CPU Performance Equation*

$$\boxed{t_{\mathrm{T}} = \frac{1}{\phi} \times \mathrm{CPI} \times \mathrm{IC}},$$

where t_{T} denotes the execution time.

Component of CPU Performance: Instruction Count

Given a program there are two ways instructions could be tallied:

- *Static Instruction Count*
  The number of instructions making up the program.
- *Dynamic Instruction Count* (IC)
  The number of instructions executed in a run of the program.

For estimating performance, dynamic instruction count is used.

Example, C program that computes $a = \sum_{i=0}^{9} i$.

(For simplicity, treat each line as an instruction.)

```
IC Line Code
 1  1   a = 0;
 1  2   for( i = 0;
11  3        i < 10;
10  4      i++ )
10  5    a = a + i;
```

Static count: 5 (number of lines).

Dynamic count: 33.

Component of CPU Performance: Clock Frequency

CPUs implemented using synchronous clocked logic.

Typical Clock Cycle

- When clock switches from low to high work starts.
- While clock is high work proceeds.
- When clock goes from high to low work should be complete.

Clock frequency determined by *critical path.*

Critical Path:
Logic doing most time consuming work (in a cycle).

If clock frequency is too high work will not be completed ...
... and so system will not perform properly.

For high clock frequencies, keep critical paths short.

Component of CPU Performance: CPI

Cycles (clocks) per Instruction (CPI)

Oversimplified definition: (CPI)
Number of cycles needed to execute an instruction.

Better definition: (CPI)
Number of cycles to execute some code divided by number of instructions.

Difference:

Interested in rate at which instructions executed in program ...

... not time time for any one instruction.

Analogy: A restaurant chef preparing meals

Chef may be simultaneously preparing several orders ...

... some take more time than others ...

... some combinations are more time consuming, etc.

Knowing crawfish étouffée takes 30 minutes to prepare ...

... cannot alone predict how long it would take to serve 20 diners.

Interaction of Execution Time Components

Tradeoffs between Clock Frequency, CPI, and Instruction Count

Increasing Clock Frequency ...

... reduces the work that can be done in a clock cycle ...

... forcing designers to choose higher-CPI designs.

Reducing IC (by adding "powerful" instructions to ISA) ...

... may force implementors to increase CPI or lower clock frequency.

Balancing these is an important skill in computer design.

Since the ISA is usually fixed, IC is less of a factor.

Example: Trading off Execution Time Components

Company X is considering two clock frequencies for its next processor, 500 MHz *or* 300 MHz. *A* 500 MHz *implementation would execute instructions at a rate of 1.7 CPI, the 400 MHz part at 1.1 CPI. Which would be faster?.*

500 MHz execution rate: $500 \times 10^6 / 1.7 = 294.1 \times 10^6$

400 MHz execution rate: $400 \times 10^6 / 1.1 = 363.3 \times 10^6$.

The lower clock rate would nevertheless execute faster.

Perhaps because at 500 MHz too much work had to be split into multiple cycles.

Instruction Mix and Execution Time

An ISA contains many instructions ...

... execution characteristics differ.

E.g., division takes longer than add.

To account for this instruction count can be partitioned.

For example, $\mathrm{IC} = \mathrm{IC}_1 + \mathrm{IC}_2 + \mathrm{IC}_3$,

where IC_x is the number of instructions of class x in the execution of some program.

Choosing Instruction Classes

Option: a class for every instruction in the ISA.
This would be tedious to work with.

Option: classes for instructions sharing execution characteristics.
Example: a class for all memory instructions, class for all integer instructions, etc.
Since instructions in class similar, no need to consider separately.

Similarly, CPI can be partitioned by class.

The CPU Performance Equation can then be written:

$$\boxed{t_{\mathrm{T}} = \frac{1}{\phi} \sum_i \mathrm{IC}_i \times \mathrm{CPI}_i.}$$

Design Tradeoffs Using Instruction Classes

Change may affect one class but not another.

CPU Performance Equation, $\frac{1}{\phi}\sum_i \mathrm{IC}_i \times \mathrm{CPI}_i$, shows impact.

Impact of changes of different instructions can be estimated.

Note: unlike case without instruction classes ...

... impact computed depends on program being analyzed.

E.g., you're out of luck ...

... if $\mathrm{IC}_1 = 0$ and change reduced only CPI_1.

(The classless CPI is really an average for a typical program.)

Instruction counts by class can guide designer's effort:

First consider instructions in class i, where $\mathrm{IC}_i \geq \mathrm{IC}_x$.

Design changes can even influence IC (without changing ISA):

Suppose CPI_1 was reduced to a really low value.

Then programmer might re-write code so $\mathrm{IC}_1 \gg 0$.

Result might be faster execution with modified program.

But performance benefit must be high enough to justify re-coding.

Benchmarks

Benchmark:
program used to evaluate performance.

Uses

- Guide computer design.
- Guide purchasing decisions.
- Marketing tool.

Guiding Computer Design

Measure overall performance.

Determine characteristics of programs.
E.g., frequency of floating-point operations.

Determine effect of design options.

Choosing Benchmark Programs

Important: Choice of programs for evaluation.

Optimal but unrealistic:

The exact set of programs customer will run.

Problem: computers used for different applications.

Therefore, must model typical users' workload.

Options:

Real Programs
Programs chosen using surveys, for example.

+ Measured performance improvements apply to customer.
– Large programs hard to run on simulator. (Before system built.)

Kernels
Use part of program responsible for most execution time.

+ Easier to study.
– Not all program have small kernels.

Toy Benchmarks
Program performs simplified version of common task.

+ Easier to study.
– May not be realistic.

Synthetic Benchmarks
Program "looks like" typical program, but does nothing useful.

+ Easier to study.
– May not be realistic.

Commonly Used Option

Overall performance: real programs

Test specific features: synthetic benchmarks.

Benchmark Suites

Definition: a named set of programs used to evaluate a system.

Typically:

- Developed and managed by a publication or non-profit organization.
  E.g., Standard Performance Evaluation Corp., PC Magazine.
- Tests clearly delineated aspects of system.
  E.g., CPU, graphics, I/O, application.
- Specifies a set of programs and inputs for those programs.
- Specifies reporting requirements for results.

What Suites Might Measure

- Application Performance
  E.g., productivity (office) applications, database programs.
  Usually tests entire system.
- CPU and Memory Performance
  Ignores effect of I/O.
- Graphics Performance

Example, SPEC 95 Suites

Respected measure of CPU performance.

Managed by Standard Performance Evaluation Corporation,. . .
. . .a non-profit organization funded by computer companies.

Measures CPU and memory performance on integer and FP code.

Uses common Unix programs such as perl, gcc, compress.

Requires that results on each program be reported.

Programs compiled with publicly available compilers and libraries.

Programs compiled with and without expert tuning.

SPEC 95 Suites and Measures

CINT95 suite of integer programs run to determine:

- SPECint95, execution time of tuned code.
- SPECint_base95, execution time of untuned code.
- SPECint_rate95, throughput of tuned code.

CFP95 suite of floating programs run to determine:

- SPECfp95, execution time of tuned code.
- SPECfp_base95, execution time of untuned code.
- SPECfp_rate95, throughput of tuned code.

Other Examples

BAPCO Suites, measure productivity app. performance on Windows 95.

TPC, measure “transaction processing” system performance.

WinMARK, graphics performance.