Georgia Tech Computer Science

CS2200
Systems and Networks
Spring 2024

Lecture 9: Performance

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

Lecture slides adapted from Bill Leahy and Charles Lively of Georgia Tech

Announcements

- Homework 1 grades posted
- Homework 3 due Wednesday
- Lab 1 due Friday
- Starting Chapter 5

- Will be using this logo on the slide *preceding* the first PS question
- So you make sure you are already logged in
- There will be a LOT of PS questions today!

Incoming

Metrics

If want to try to make processors better, we have to take measurements

Common Metrics:

- Space → memory footprint
- Time → execution time
- Instruction frequency (or count)
  - Static
  - Dynamic
- Benchmarks

What determines execution time?

- CPI = "Cycles Per Instruction" → number of clock cycles each instruction takes
- *Execution time = ($\sum CPI_j$) * clock cycle time*,
  where $1 \leq j \leq n$
- That's a pretty tough sum to compute because modern computers can execute billions of instructions per second
- So, we approximate as
  *Execution time = n * CPI_{Avg} * clock cycle time*,
  where n is the number of instructions
  (executed—i.e., dynamic, not static—instruction count)

- This is known as the **Iron Law** of processor performance

The "Iron Law" of Processor Performance

$$Processor\ Performance = \frac{Time}{Program}$$

$$= \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

ISA & Compiler   Microarchitecture   Circuit

Incoming

Session ID: dag2200

What's the execution time?

1GHz processor, 10K instructions, $CPI_{Avg} = 3$

0% A. 3 msec

0% B. 1 GHz

0% C. 0.003 msec

0% D. 30 µsec

0% E. 30K

0% F. About 11, sir

$$10^{-9}\ \text{sec}/\cancel{\text{cy}} \times 10^{4}\ \cancel{\text{inst}} \times 3\ \cancel{\text{cy}/\text{inst}}$$

$$= 3 \times 10^{-5}\ \text{sec}$$
$$= .00003\ \text{sec}$$
$$= .03\ \text{msec}$$
$$= 30\ \mu\text{sec}$$

Instruction Frequency

- *Static* instruction frequency refers to number of times a particular instruction occurs in compiled code.
  - Impacts memory footprint
  - If a particular instruction appears a lot in a program, can try to optimize amount of space it occupies by clever instruction encoding techniques in the instruction format.

- *Dynamic* instruction frequency refers to number of times a particular instruction is executed when program is run.
  - Impacts execution time of program
  - If dynamic frequency of an instruction is high then can try to make enhancements to datapath and control to ensure that CPI taken for its execution is minimized.

Session ID: dag2200

Static instruction frequency…

A. Refers to the type of instructions in the instruction-set
B. Refers to the frequency of occurrence of instructions in compiled code
C. Refers to the frequency of occurrence of instructions that actually get executed ← Dynamic instruction frequency
D. Refers to the clock frequency of the processor
E. Is the basis for datapath design

Refers to the type of instructions in the instruction-set
Refers to the frequency of occurrence of instructions in compil...
Refers to the frequency of occurrence of instructions that actu...
Refers to the clock frequency of the processor
Is the basis for datapath design

What is the static frequency of ADD?

The ADD instruction occurs twice in a program that contains a total of 1000 instructions in the compiled code. All 1000 instructions get executed during a program run. One of the ADD instructions is in a 5-instruction loop that gets executed 1000 times.

$$2/1000 = 0.2\%$$

Session ID: dag2200

What about the dynamic frequency of ADD?

The ADD instruction occurs twice in a program that contains a total of 1000 instructions in the compiled code. All 1000 instructions get executed during a program run. One of the ADD instructions is in a 5-instruction loop that gets executed 1000 times.

A. Two
B. 0.2%
C. (1000/5995) *100%
D. (1001/5995) * 100%
E. (1/5000) *100%
F. (1001/5000) * 100%

ADDs executed:
1 + 1000 = 1001

Total executed:
(1000-5) inst + 1000 * 5 = 5995

Dynamic add frequency: 1001/5995

Two	0.2%
(1000/5995) *100%	(1001/5995) * 100%
(1/5000) *100%	(1001/5000) * 100%

I feel the need, the need for speed!

- How do we improve performance?
- Reduce execution time (of course)
- How?
- The Iron Law tells us:

$$\text{Execution time} = N{\downarrow} * \text{CPI}{\downarrow} * \text{Cycle Time}{\downarrow}$$

- How can we reduce the left-hand side?
- Reduce one or more of the right-hand factors

How can we measure improvement?

$$\text{speedup}_{\text{AoverB}} = \frac{\text{Execution Time On Processor B}}{\text{Execution Time On Processor A}}$$

$$\text{speedup}_{\text{improved}} = \frac{\text{Execution Time Before Improvement}}{\text{Execution Time After Improvment}}$$

$$\text{improvement in execution time} = \frac{\text{old execution time} - \text{new execution time}}{\text{old execution time}}$$

Example

You improve your application's algorithm so that it runs in 29 seconds instead of 38 seconds.

What is the speedup?

E_{Before}/E_{After}

$38 / 29 = 1.3103$
$= 31\%$ speedup

$29 * 1.31 \sim= 38$

What is the improvement in execution time?

$(E_{old}-E_{new})/E_{old}$

$(38 – 29) / 38 = .2368…$
$= 24\%$

$38 – (24\% \text{ of } 38) \sim= 29$

Session ID: dag2200

Improvement in execution speed

It takes me 8 minutes to walk to class from my office. I can run twice as fast as I walk. If I walk half the distance and run the remaining half, how much time will I take to reach class from my office?

$8 = d/v$

$n = \frac{1}{2}\, d/v + \frac{1}{2}\, d/2v$

$8v = d$

$n = 4v/v + 4v/2v$

$n = 4 + 2$

$n = 6$

Rank **Responses**

Amdahl's Law

Amdahl's Law

$$\text{Time}_{\text{after}} = \frac{\text{Time}_{\text{affected}}}{\text{Amount of Improvement}} + \text{Time}_{\text{unaffected}}$$

My office walk: 6 = 4 / 2 + 4

Improving an instruction

A processor spends 20% of its time on ADD instructions.
An engineer proposes to improve the speed of the ADD instruction by 4 times.
What is the speedup achieved by this modification?

The improvement only applies for the ADD instruction, so 80% of the execution time is unaffected by the improvement.

Original normalized execution time = 1
New execution time = (time spent in ADD/4) + remaining time
= 0.2 / 4 + .8
= 0.85

Speedup = execution time before /execution time after
= 1 / 0.85 = 1.18 = 18%

Session ID: dag2200

Improving an instruction

An engineer is asked to improve the processor's overall performance by 2 times by optimizing the ADD instruction.
The processor spends 20% of its time on ADD instructions.
How much faster must the ADD instruction become?

A. 2x
B. 10x
C. 100x
D. That's impossible!!

Session ID: dag2200

Execution time?

CPI of Instruction Classes	Code 1	Code 2
R-type = 2	3	10
I-type = 10	3	1
J-type = 3	5	2
S-type = 4	2	3

13 inst
6+30
+15+8 =
59 cycles

16 inst
20+10+6
+12 =
48 cycles

A. Code 1 is better than Code 2 since it has fewer instructions

B. Code 2 is better than Code 1 since it has fewer instructions

C. Code 1 is better than Code 2 since it takes fewer total clock cycles to execute

D. Code 2 is better than Code 1 since it takes fewer total clock cycles to execute

Microarchitecture change?

- We have a computer with three types of instructions that have the following CPI:

Type	CPI
A	2
B	5
C	1

- An architect determines that she can reduce the CPI for B to 3 but will need to slow the clock speed of the processor. What is the maximum permissible slowing of the clock that will make this change worthwhile?
- Assume that all the workloads for this processor use 30% of A, 10% of B, and 60% of C types of instructions

How do we answer that?

Execution time of the old machine:

$$ET_o = N * (F_{Ao} * CPI_{Ao} + F_B * CPI_{Bo} + F_C * CPI_{Co}) * C_o$$

(where Fx and CPIx are the dynamic frequencies and CPIs of each type of instruction, respectively)

$$ET_o = N * (0.3 * 2 + 0.1 * 5 + 0.6 * 1) * C_o \quad = N * 1.7 * C_o$$

Execution time for the new machine:

$$ET_n = N * (0.3 * 2 + 0.1 * 3 + 0.6 * 1) * C_n \quad = N * 1.5 * C_n$$

For the design to be viable,

$$ET_n < ET_o$$

$$N * 1.5 * C_n < N * 1.7 * C_o$$

$$C_n < 1.7/1.5 * C_o$$

$$C_n < 1.13 * C_o$$

Maximum permissible increase in clock cycle time = 13%

Type	CPI
A	2
B	5
C	1

30% of A,
10% of B,
60% of C

Combining two instructions?

Instruction	CPI
Add	2
Shift	3
Others	2
Add/Shift	4

Assume a SHIFT instruction is always preceded by an ADD.
If SHIFT instructions represent 10% of the dynamic instruction frequency of a program, what is the speedup of the program with all {ADD, SHIFT} replaced by the combined instruction?

[HINT: For every 10 instructions in the original program, 2 instructions are the ADD/SHIFT combo. Thus the number of instructions in the new program shrinks to 90% of the original program.]

A solution

$$\begin{aligned}ET_o &= N * (F_{ADD} * 2 + F_{SHIFT} * 3 + F_{others} * 2) * C \\ &= N * (0.1 * 2 + 0.1 * 3 + 0.8 * 2) * C \\ &= 2.1*N*C\end{aligned}$$

With the combo instruction replacing {ADD SHIFT}, the number of instructions in the new program shrinks to 0.9N in the new program. The frequency of the combo instruction is 1/9 and the other instructions are 8/9.

$$\begin{aligned}ET_n &= (0.9 * N) * (F_{COMBO} * 4 + F_{others} * 2) * C \\ &= (0.9* N) * (1 / 9 * 4 + 8 / 9 * 2) * C \\ &= 2*N*C\end{aligned}$$

$$\begin{aligned}\text{Speedup} &= \text{old execution time / new execution time} \\ &= 2.1NC / 2NC \\ &= 1.05\end{aligned}$$

Instruction	CPI
Add	2
Shift	3
Others	2
Add/Shift	4

Note: textbook example 5.5 on p. 168 has an error. Above is the corrected solution

Benchmarks

- **Benchmarks** are a set of programs that are **representative** of the workload for a processor.
- The key difficulty is to be sure that the benchmark program selected **really** are representative of the prospective workload.
- Standard benchmark suites (SPEC, LINPACK, Whetstone, Dhrystone, and many more) are used to try to compare "apples" with "apples" by summarizing performance *across a set of programs*
  - E.g., SPEC uses perl, gcc, AI apps, compression, imaging apps, modeling apps, etc. to represent a common workload
- A radical new hardware design is hard to benchmark because there may not yet be a compiler or much code.

Using a benchmark

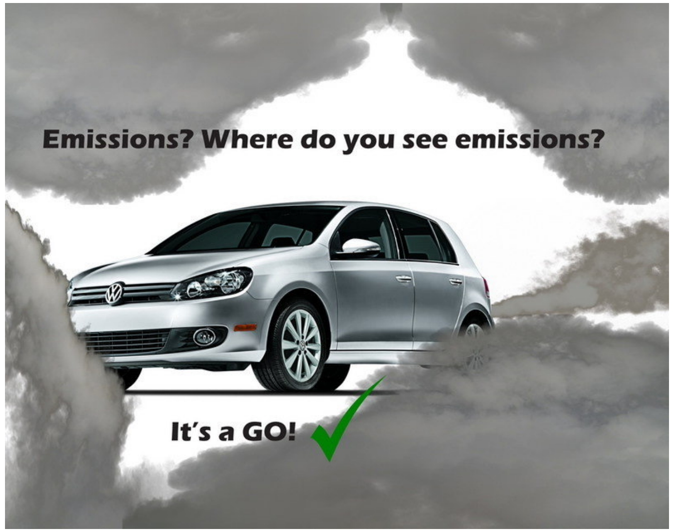
Some caveats:

- Testing a single system component (e.g., the processor) only gives a limited view: e.g., memory organization and memory—processor—bus bandwidths are also key
- Some processors do well on certain benchmark programs and other do well on other programs

- A composite index can be useful when we want to compare two processors without knowing the exact kind of workload they are going to run, but we must be very cautious
  - More on this later

Reasons to be skeptical of a benchmark

- The vendor gave you the benchmark results (in polite company, we call this a conflict of interest)
- The vendor wrote the benchmark suite
- The benchmark suite doesn't seem to have any elements that represent your workload (e.g., you run web server farms and the benchmark represents only computationally intensive scientific calculations)
- The equipment being benchmarked is different from the equipment you want to evaluate (maybe a little different, maybe a lot different)
- The benchmark uses a different compiler suite than you plan to use



The cost of mistakenly choosing the wrong equipment is very high!

Comparing Multiple Programs

	Computer A	Computer B	Computer C
Program 1 (secs)	1	10	20
Program 2 (secs)	1000	100	20
Program 3 (secs)	1001	110	40

A is 10 times faster than B for program 1
B is 10 times faster than A for program 2
A is 20 times faster than C for program 1
C is 50 times faster than A for program 2
B is 2 times faster than C for program 1
C is 5 times faster than B for program 2

Each statement above is correct…

…but I just want to know which machine is the best?

Need a composite metric

Let's Try a Simpler Example

Two machines timed on two benchmarks

	Machine A	Machine B
Program 1	2 seconds	4 seconds
Program 2	12 seconds	8 seconds

How much faster is Machine A than Machine B?

Attempt 1: ratio of runtimes, normalized to Machine A runtimes

program1: 4/2 program2 : 8/12

- Machine A ran 2 times faster on program 1, 2/3 times faster on program 2
- On average, Machine A is (2 + 2/3) / 2 = 4/3 times faster than Machine B

It turns this "averaging" stuff can fool us; watch…

Example (con't)

Two machines timed on two benchmarks

	Machine A	Machine B
Program 1	2 seconds	4 seconds
Program 2	12 seconds	8 seconds

How much faster is Machine A than B?

Attempt 2: ratio of runtimes, normalized to Machine B runtimes

program 1: 2/4 program 2 : 12/8

- Machine A ran program 1 in 1/2 the time and program 2 in 3/2 the time
- On average, (1/2 + 3/2) / 2 = 1
- Put another way, Machine A is 1.0 times faster than Machine B

Example (con't)

Two machines timed on two benchmarks

	Machine A	Machine B
Program 1	2 seconds	4 seconds
Program 2	12 seconds	8 seconds

How much faster is Machine A than B?

Attempt 3: ratio of aggregated runtimes, norm. to A

- Machine A took 14 seconds for both programs
- Machine B took 12 seconds for both programs
- Therefore, Machine A takes 14/12 of the time of Machine B
- Put another way, Machine A is 6/7 faster than Machine B

Which is Right?

Question:

- How can we get three different answers?

Answer:

- Because, while they are all reasonable calculations…

  …each answers a different question

Need to be more precise in understanding and posing these performance & metric questions

Arithmetic and Harmonic Mean

Average of the execution time that tracks total execution time is the arithmetic mean

$$\frac{1}{n}\sum_{i=1}^{n} Time_i$$

This is the definition for "average" you are most familiar with

If performance is expressed as a rate, then the average that tracks total execution time is the harmonic mean

$$\frac{n}{\sum_{i=1}^{n} \frac{1}{Rate_i}}$$

This is a different definition for "average" you are probably less familiar with

Geometric Mean

- Used for relative rate (i.e., ratio) or normalized performance

$$Relative\_Rate = \frac{Rate}{Rate_{ref}} = \frac{Time_{ref}}{Time}$$

- Geometric mean

$$\sqrt[n]{\prod_{i=1}^{n} Relative\_Rate_i} = \frac{\sqrt[n]{\prod_{i=1}^{n} Rate_i}}{Rate_{ref}}$$

Why does the choice of the mean matter?

Benchmark	**Ops (millions)**	**Computer 1**	**Computer 2**	**Speedup (C2 vs C1)**
Absolute performance (Time)				
Program 1	100	1	20	
Program 2	100	1000	20	
Total time		1001	40	25
Avg (arith mean)		500	20	25

Why does the choice of the mean matter?

Benchmark	Ops (millions)	Computer 1	Computer 2	Speedup (C2 vs C1)
Absolute performance (Time)				
Program 1	100	1	20	
Program 2	100	1000	20	
Total time		1001	40	25
Avg (arith mean)		500	20	25
Performance in MFLOPS (Rate)				
Program 1		100	5	
Program 2		0.1	5	
Arith. mean		50.1	5	0.1
Geom. mean		3.2	5	1.6
Harm. mean		0.2	5	25

For rates use Harmonic Mean!

Problems with Arithmetic Mean

- Applications do not have the same probability of being run
- Longer programs weigh more heavily in the average

For example, two machines timed on two benchmarks

	Machine A	Machine B
Program 1	2 seconds (20%)	4 seconds (20%)
Program 2	12 seconds (80%)	8 seconds (80%)

- If we do arithmetic mean, Program 2 "counts more" than Program 1

  An X% improvement in Program 2 changes the average more than an X% improvement in Program 1

- But perhaps Program 2 is 4 times more likely to run than Program 1

Weighted Execution Time

Often, one runs some programs more often than others. Therefore, we should weight the more frequently used programs' execution time

$$\sum_{i=1}^{n} \boxed{Weight_i} \times Time_i$$

Weighted Harmonic Mean

$$\frac{1}{\sum_{i=1}^{n} \frac{\boxed{Weight_i}}{Rate_i}}$$

Using a Weighted Sum (or weighted average)

	Machine A	Machine B
Program 1	2 seconds (20%)	4 seconds (20%)
Program 2	12 seconds (80%)	8 seconds (80%)
Total	10 seconds	7.2 seconds

Allows us to determine relative performance 10/7.2 = 1.39

→ Machine B is 1.39 times faster than Machine A

(instead of 14/12 = 1.16x faster without weighting)

What if we only know normalized runtimes?

Normalize runtime of each program to a reference

	Machine A (ref)	Machine B
Program 1	2 seconds	4 seconds
Program 2	12 seconds	8 seconds

	Machine A (norm to B)	Machine B (norm to A)
Program 1	0.5	2.0
Program 2	1.5	0.666
Average?	1.0	1.333

- When we normalize A to B and average, it looks like A & B are the same.
- But when we normalize B to A and average, it looks like A is better!

Using Geometric Mean

	Machine A (norm to B)	Machine B (norm to A)
Program 1	0.5	2.0
Program 2	1.5	0.666
Geometric Mean	0.866	1.154

Note that 1.154 = 1/0.866

Drawbacks:

- Does not reflect actual runtime because it normalizes
- Each application now counts equally

When is geomean useful?

Geometric mean of ratios is not proportional to total time

Use to compare machines when

- Relative performance on each program is known
- Relative runtime/weights of different programs is not known
- E.g., to aggregate speedups on set of programs

Rule of thumb: Use AM for times, HM for rates, GM for ratios

Summary of metrics

Name	Notation	Units	Comment
Memory footprint	-	Bytes	Total space occupied by the program in memory
Execution time	$(\sum CPI_j)$ * clock cycle time, where $1 \leq j \leq n$	Seconds	Running time of the program that executes n instructions
Arithmetic mean	$(E_1+E_2+\ldots+E_p)/p$	Seconds	Average of execution times of constituent p benchmark programs
Weighted Arithmetic mean	$(f_1*E_1+f_2*E_2+\ldots+f_p*E_p)$	Seconds	Weighted average of execution times of constituent p benchmark programs
Geometric mean	p^{th} root $(E_1*E_2*\ldots*E_p)$	Seconds	p^{th} root of the product of execution times of p programs that constitute the benchmark
Static instruction frequency		%	Occurrence of instruction i in compiled code
Dynamic instruction frequency		%	Occurrence of instruction i in executed code
Speedup (M_A over M_B)	E_B/E_A	Number	Speedup of Machine A over B
Speedup (improvement)	E_{Before}/E_{After}	Number	Speedup After improvement
Improvement in Exec time	$(E_{old}-E_{new})/E_{old}$	Number	New Vs. old
Iron Law	Time/Program = Insn/Prog * CPI * Time/Cycle	Seconds	Caution: optimization of one component may hurt another one, causing overall perf. loss!
Amdahl's law	$Time_{after} = Time_{unaffected} + Time_{affected}/x$	Seconds	x is amount of improvement