Slide 1

Administrivia

- First homework to be on the Web soon; likely due date is a week from Monday. I will send e-mail.

Slide 2

Compiling and Executing Programs — Recap/Review

- Several ways source code can be executed:
- Interpreted directly (e.g., shell scripts).
- Compiled to intermediate form, interpreted/executed by possibly-language-specific runtime system (e.g., Scala and Java).
- Compiled to "native code" (usually producing "executable") and executed. (We will focus on this one.)

Slide 3

Running Executable Files — Recap/Review?

- What a processing element can do is fetch machine-language instructions from memory (RAM) and execute them one at a time.
- So to execute a program — somehow get machine-language instructions into memory and transfer control to a starting instruction.
- Several ways to do that, but most typical in general-purpose systems involves operating system that reads contents of "executable file" from storage device. Executable file contains machine-language instructions (a.k.a. "object code") and possibly other information (e.g., how much space to reserve for fixed data).
- Programs can be completely self-contained or can contain instructions that request operating-system services (e.g., for I/O).

Slide 4

From Programs to Executables — Recap/Review

- Source code is translated into assembly language (symbolic representation of machine language via a compiler, then converted to object code (machine language, plus other information) via an assembler. Note that all compilers/assemblers follow some of the same conventions for passing of arguments, etc. — this is part of an ABI ("application binary interface"). Another part of the ABI defines how application programs make requests of the operating system.
- Object code is linked with library code via a so-called linker, making use of that "other information" (such as references to library code) to form an "executable" file, which conforms to the part of the ABI that specifies a format specific to architecture and operating system. Typically this file contains machine language plus extra information such as size.

Slide 5

From Programs to Executables, Continued

- At runtime, operating system loads machine language from executable file and transfers control to address of starting instruction. This is also the point at which calls to dynamically-linked library code are resolved.
- (As noted last time, not all languages work this way, but this is the "compile to native code" model used by, e.g., C and C++.)

Slide 6

A Little About Integrated Circuits — Conceptual View

- *Transistor* — on/off switch controlled by electrical current.
- Combine/connect a lot of transistors to get *circuit* that does interesting things (e.g., addition).
- Put a bunch of circuits together to get a *chip* / *integrated circuit* (IC). If lots of transistors, *VLSI chip*.

Slide 7

A Little About Integrated Circuits — Logic Gates

- Many circuits (can be) built from "logic gates" — simplest are NOT, AND, and OR, pretty much same as Boolean algebra.
- A popular implementation technology is CMOS ("Complementary Metal-Oxide Semiconductor").

Slide 8

CMOS and Logic Gates, Very Simplified View

- Basic idea is that this technology offers two ways to build "switches", one that "conducts 0s well" and one that "conducts 1s well".
- Can put these together with two sources of constant voltage, one representing 1 and the other 0, to get something that can implement logic gates.
- Example — inverter (figure at start of Wikipedia article on CMOS).

Slide 9

A Little About Integrated Circuits, Continued

- Manufacturing process starts with a thin flat piece of silicon, adds metal and other stuff to make wires, insulators, transistors, etc.
- Of course, this is all automated! Low-level chip designers use CAD-type tools, which save designs in a standard format, which the chip designers simulate/test with other software, and then send off to be *fabricated*.
- Typically make many chips on a *wafer*, discard those with defects, bond each good one to something larger with *pins* to allow connections to other parts of computer.

Slide 10

Defining Performance

- What does it mean to say that computer A "has better performance than" computer B?
- Really — "it depends". Some answers:
  - Computer A has better response time / smaller execution time.
  - Computer A has higher throughput.
- Trickier than it might seem to come up with one number that means something.

Slide 11

Evaluating / Comparing Performance — Approaches

- Use the actual workload, on the actual hardware platform(s), and compare times.
- Put together a representative simulated workload — "benchmark"; run and compare times.
- Compare code size.
- Compare number of instructions per second ("MIPS" or "MFLOPS", once).

Slide 12

Evaluating / Comparing Performance, Continued

- Alas, all the methods just mentioned are flawed in some way.

  (In particular, paraphrasing someone whose name I don't remember, "peak MIPS is just the number you can't go any faster than.")
- Textbook chooses to focus in this chapter on "execution time". Might not be meaningful for comparing systems but seems like reasonable way to compare processors at least.

Slide 13

Measuring Performance

- If we use execution time as criterion, how to measure?
- Wall-clock time seems fairest, since it includes
  - Time for CPU to execute instructions.
  - Any waiting for memory access.
  - Any waiting for I/O.
  - Any waiting for operating system.
- Is that easy to measure reliably / repeatably?

Slide 14

Measuring Performance, Continued

- No — to get repeatable measure of wall clock time, need an otherwise unused system.
- So instead we could use "CPU performance" — amount of time CPU needs to run program. Easier to measure, more consistent, and at least says *something* about the processor.
- Even that, though, is not as simple as it might seem.

Slide 15

Defining Performance

- Textbook chooses to focus on CPU (processor) time, and say

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

exactly when

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

Slide 16

Measuring Performance, Continued

- CPU execution time for program X is given by

$$\text{CPU cycles} \times \text{clock cycle time}$$

and then CPU cycles in turn is the product of count of instructions and cycles per instruction.

("Cycles"? processors typically are mostly-synchronous devices, in which all the parts do some basic operation at fixed intervals called cycles.)

- And then it *might* seem like we can say something meaningful about what happens if we change one of these numbers — but only if all other things remain the same, which might or might be true!

Slide 17

Calculating Program Execution Time (CPU Only)

- CPU execution time for program X is given by

$$\text{CPU cycles} \times \text{clock cycle time}$$

- We can expand this a bit to get

$$\text{instruction count} \times \text{cycles per instruction} \times \text{clock cycle}$$

- We can then come up with many variations — e.g., one that uses clock rate rather than clock cycle time — based largely on consideration of units of measure (e.g., clock cycle time is seconds per cycle, while clock rate is cycles per second).

Slide 18

Sidebar: Dimensional Analysis

- (Or at least I think that's close to the term I want.)
- Idea here is to approach "word problems" in terms of units, treating them almost like factors in multiplication and division. (Example is converting, say, inches to cm by multiplying by 1 in the form 2.54cm/1in.)
- If the formula you propose to use produces the right units (e.g., seconds for execution time), there's at least a good chance it's the right one.

Slide 19

Calculating Execution Time — Example(s)

- Given the following about some program P:

  On computer A execution requires 2×10^9 instructions.

  Instructions take 3 cycles each.

  Clock rate is 1GHz (so cycle time is $1/10^9$.

  On computer A execution requires 1.5×10^9 instructions.

  Instructions take 5 cycles each.

  Clock rate is also 1GHz.

- Calculate execution times for P:

  On computer A, $2 \times 10^9 \times 3 \times 10^{-9}$, i.e., 6

  On computer B, $1.5 \times 10^9 \times 5 \times 10^{-9}$, i.e., 7.5

- So for P, A's performance is 1.25 times as good as B's (7.5/6).

Slide 20

Calculating Execution Time, Continued

- One factor in the basic formula is cycles per instruction. What if that isn't the same for all instructions?
- Common sense(?) may tell you . . .

Slide 21

Calculating Execution Time, Continued

- If different types of instructions need different numbers of cycles, have to do something like a weighted sum. Usually instructions fall into one of a few "classes", each with a common number of cycles per instruction.
- So, compute times for each "class" of instruction and add. Would also allow you to compute an average CPI.

Slide 22

Calculating Execution Time — Example Continued

- Suppose we change computer A so that there are two "classes" of instructions, a class 2 in which instructions take 2 cycles and a class 4 in which instructions take 4 cycles, and suppose 3/4 of all instructions are class 2 while the other 1/4 are class 4.
- Now execution time is

  $(2 \times 3/4) \times 10^9 \times 2 \times 10^{-9} +$
  $(2 \times 1/4) \times 10^9 \times 4 \times 10^{-9}$

  i.e., 5

Slide 23

Parallelism (Hardware)

- Executive-level definition of "parallelism" might be "doing more than one thing at a time". In that sense, it's been used in processors for a very long time, via *pipelining*, and (in some high-performance processors) *vector processing*.
- For a (relatively!) long time, hardware designers were able to make single processors faster using these and other techniques (e.g., reducing sizes of things). In the mid-2000s, however, they ran out of ways to do that. But they could still put larger numbers of transistors on the chip. How to use that to get better performance?

Slide 24

Parallelism (Hardware), Continued

- All that time there were people saying we would hit a limit on single-processor performance, and the only answer would be paralleism at a higher level — executing multiple instruction streams at the same time.
- So . . . use all those transistors to put multiple *cores* (processing elements) on a chip!
- Why wasn't this done even earlier? because alas the "magic parallelizing compiler" —- the one that would magically turn "sequential" programs into "parallel" versions — has proved elusive, and (re)training programmers is not trivial.

Slide 25

Parallelism (Hardware/Software)

- Multicore computers offer one kind of potential parallelism — "multithreading".
- Networks of computers offer another — "message-passing".
- Sufficiently advanced graphics processors offer yet another — limited form of multithreading.
- Exploiting any of these traditionally requires significant programmer effort. Hiding the details in libraries — research topic for many years, becoming much more mainstream now that the hardware is.

Slide 26

Parallelism — Performance

- One use of multithreading is simply to make the code simpler, at least for the programmer — as an example consider the typical GUI-based program, where it makes sense to think in terms of one thread of control for getting user input and one for drawing. Doable on a single processor via interleaving.
- But it *can* also be used to improve performance. Often a discussion of "how much" is in terms of "speedup".
- Here, "speedup" is defined as some sort of function of the number of processing elements (cores, fully independent processors, etc.), where the speedup for P processing elements is the ratio of execution time using 1 PE to execution time using P PEs.

Slide 27

Parallel Performance, Continued

- While it might seem like with P processing elements you could get a speedup of P, in fact most if not all programs have at least a few parts that have to be executed sequentially. This limits P, and if we can estimate what fraction of the program is sequential we can compute speedups for some values of P.
- Further, typically "parallelizing" programs involves adding some sort of overhead for managing and coordinating more than one stream of control.
- But even ignoring those, as long as any part must remain sequential . . .

Slide 28

One More Thing About Performance — Amdahl's Law

- (Named after Gene Amdahl, a key figure in developing some of IBM's early mainframes who left to start his own company to make hardware "plug-compatible" with IBM's. Interaction between the two companies was — interesting?)
- His observation ("Amdahl's law") can be more generally stated, but in the context of parallel programming it's this:

  If γ is the "serial fraction", speedup on P PEs is (at best, i.e., ignoring overhead)

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

  and as P increase, this approaches $\frac{1}{\gamma}$ — upper bound on speedup.

Slide 29

Minute Essay

- Suppose you are trying to decide which of two computers, call them `Foo` and `Bar`, will give you the best performance. You run two test programs on `Foo` and observe execution times of 10 seconds for one and 20 seconds for the other. If the first program takes 5 seconds on `Bar`, how long does the second program take? (Hint: This might be something of a trick question.)
- Other questions?

Slide 30

Minute Essay Answer

- It might seem like that second program would take 10 seconds on `Bar`, but in truth you probably can't be sure without doing the experiment, since the two machines, or the two test programs, could differ in ways that would make this obvious answer wrong.