The Hardware/Software Interface
CSE351 Winter 2011
1st Lecture, 3 January

Instructor:
John Zahorjan

Teaching Assistants:
David Cohen, Michael Ratanapintha

Overview

Course Synopsis
Course themes: big and little
Four important realities
How the course fits into the CSE curriculum
Logistics5

HW0 is out. Due end of day Wednesday.
Course Synopsis: Preliminaries

- **A program is an expression of a computation**
  - It *describes* what the output should be when given some input
- **Programs are written to some specification**
  - E.g., Java defines how to write statements and what they mean
- **How to write something is called syntax**
  - We usually think of syntax as a relatively minor issue, although it can have substantial impact on the likelihood of making mistakes
- **What it means is called semantics**
  - “if (x != 0) y = (y+z)/x;” vs. “when (x != 0) y = (y+z)/x;”
    - different syntax, same semantics

Course Synopsis: Programs and Hardware

- **A hardware architecture defines its programming specification**
  - How to write instructions and what they mean
- **That specification isn't Java!**
  - We’ll say why in a moment...
- **So, what happens?**
  - A Java compiler translates the computation as expressed in Java into a computation expressed in the language the hardware defines
  - The translation is correct if the two programs are equivalent
    - For every input, the hardware program produces the same outputs as the Java program would if executed according to the semantics defined by Java

*Note: I'm taking some liberties with full truth for the sake of clarity.*
HW/SW Interface: The Historical Perspective

- **Hardware started out quite primitive**
  - Design was expensive \(\Rightarrow\) the instruction set was very simple
    - E.g., a single instruction can add two integers
    - Forget about \(x = (2*y + 17) / (x*y*z + 3*w)\)

- **Software was also very primitive**
  - Forget about \(x = (2*y + 17) / (x*y*z + 3*w)\)

HW/SW Interface: Assemblers

- **Life was made a lot better by assemblers**
  - 1 assembler instruction = 1 machine instruction, but...
  - different syntax: assembly instructions are character strings, not bit strings
HW/SW Interface: Higher Level Languages (HLL's)

- Human was still writing 1 line of assembler for each machine instruction
- HLL's (e.g., C) provided a higher level of abstraction:
  - 1 HLL line is compiled into many (many) assembler lines

![Diagram](image)

C vs. Assembler vs. Machine Programs

```c
if ( x != 0 ) y = (y+z) / x;
```

```assembly
        cmpl     $0, -4(%ebp)
          je        .L2
        movl     -12(%ebp), %eax
        movl     -8(%ebp), %edx
        leal     (%edx,%eax), %eax
        movl     %eax, %edx
        sarl     $31, %edx
        idivb    -4(%ebp)
        movl     %eax, -8(%ebp)
```

- The three program fragments are equivalent
- You'd rather write C!
- The hardware likes bit strings!
  - The machine instructions are actually much shorter than the bits required to represent the characters of the assembler code
Near-Recent History: Java

- **Hardware is really, really fast and really, really cheap**
- **Programming is really, really hard, and programmers aren't cheap**
- **So...**
  - Help the programmer by making it harder to make (unnoticed) mistakes
  - One program runs everywhere, not one per system type
- **How?**
  - More precisely defined language semantics
  - More restrictive language semantics
  - The “Java virtual machine”
The Java Virtual Machine (JVM)

The JVM is a program that simulates the operation of a hypothetical piece of hardware.

More Translation: Compiler Optimizations

- Some compiler optimizations can be viewed as “source to source” translations

```java
for (i=0; i<10; i++) {
    a[i] = i;
}
```

| 1 scalar assignment + |
| 11 integer compares + |
| 11 integer increments + |
| 10 array element assignments |

```java
a[0] = 0;
a[1] = 1;
a[2] = 2;
a[3] = 3;
a[4] = 4;
a[5] = 5;
a[6] = 6;
a[7] = 7;
a[8] = 8;
a[9] = 9;
i = 10;
```

| 1 scalar assignment + |
| 10 array element assignments |
And more translation: The C Preprocessor

- C programs can include “preprocessor directives,” which are executed at compile time
- The directives can alter the program that is actually compiled by the C compiler

```c
#define NUMELEMENTS 10
int X[NUMELEMENTS];
for (i=0; i<NUMELEMENTS; i++) {
    ...
}
```

```c
int X[10];
for (i=0; i<10; i++) {
    ...
}
```

Now this text is compiled

One More Thing...

- Attempts have been made to build hardware that directly executes HLL's
  - That is, the hardware architecture defines instruction syntax and semantics very similar to HLL's
  ```
  Hardware
  HLL Program
  ```
  - It hasn’t worked
    - The hardware was slow
  - Generally applicable moral: Simpler is faster.
  - Hardware architectures today look a lot like architectures from decades ago.
Translation Summary

• **Pros:**
  • Translation overhead is suffered once (at compile time), not for each execution of the program
  • Raises level of abstraction for the programmer (C vs. assembler)

• **Cons:**
  • Raising level of abstraction can come at the cost of some inefficiency
    − On the other hand, the compiler is better at some sorts of optimizations than humans
  • The program that’s actually running isn’t the one you wrote
    − That can make debugging somewhat tricky...

---

Big Theme #1: The HW/SW Interface

• **THE HARDWARE VIEW**
  • What is the programming model supported by the hardware?
  • How does that influence programs you might write?
    − How does it influence programming languages?
  • How do the requirements of programs and systems software (e.g., compilers, operating systems) influence what the hardware supports?

• Understanding the HW/SW interface might make you a more effective programmer
  • It will certainly make you a more versatile and comfortable one
Big Theme #2: The HW/SW Interface

- **THE SOFTWARE VIEW**
  - A “system” is an orchestration of hw & sw
  - The sw needs hw to run, but the hw needs the sw as well
    - Compilers/translators
    - Resource allocators
    - Protection mechanisms
    - I/O systems
    - ...
  - We’ll look at some of the functionality that “systems software” provides

Little Theme 1: Representation

- At the hardware level, everything is 0s and 1s
  - numbers, characters, strings, instructions, objects, classes, ...

- We’ll look at the base representations
  - The ones the hardware “understands”
    - numbers, characters, hardware instructions
  - We'll also look up a few layers of abstraction to the ones created by software
    - procedure class, objects

- An important implication:
  - We'll better understand what a type is in a programming language
Little Theme 2: Translation

Translation is everywhere...

But, we'll look particularly at the path C programs to execution, and from Java programs to execution

- We’ll encounter Java byte-codes, C language, assembly language, and machine code (for the X86 family of CPU architectures)

Little Theme 3: Correctness + Performance

- Up to now you've mostly struggled just with getting an implementation that works
  - Optimizing performance was ignored, or...
  - Performance was assumed to be purely an (asymptotic) algorithmic issue

- In this course we'll consider the effect of implementation (rather than algorithm) on performance
  - For example:
    - Choice of language
    - How the language is used

- And, we'll explain why!
Course Outcomes

- Foundation: basics of high-level programming (Java)
- Understanding of some of the abstractions that exist between programs and the hardware they run on, why they exist, and how they build upon each other
- Knowledge of some of the details of underlying implementations
- Become more effective programmers
  - More efficient at finding and eliminating bugs
  - Understand the many factors that influence program performance
  - Facility with some of the many languages that we use to describe programs and data
- Prepare for later classes in CSE

Reality 1: Ints ≠ the Integers & Floats ≠ Reals

- Representations are finite
- Example 1: Is $x^2 \geq 0$?
  - Floats: Yes!
  - Ints:
    - $40,000 \times 40,000 \rightarrow 1,600,000,000$
    - $50000 \times 50000 \rightarrow ??$
- Example 2: Is $(x + y) + z = x + (y + z)$?
  - Unsigned & Signed Ints: Yes!
  - Floats:
    - $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
    - $1e20 + (-1e20 + 3.14) \rightarrow ??$
Reality #2: Memory Matters

- **Memory is not unbounded**
  - It must be allocated and managed
  - Many applications are memory-dominated

- **Memory referencing bugs are especially pernicious**
  - Effects are distant in both time and space

- **Memory performance is not uniform**
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements

Memory Referencing Errors

- **C (and C++) do not provide any memory protection**
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free

- **Can lead to nasty bugs**
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
  - Corrupted object logically unrelated to one being accessed
  - Effect of bug may be first observed long after it is generated

- **How can I deal with this?**
  - Program in Java (or C#, or ML, or ...)
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (valgrind)
Memory System Performance Example

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how program steps through multi-dimensional array

```c
void copyij(int src[2048][2048],
           int dst[2048][2048])
{
    int i, j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

```c
void copyji(int src[2048][2048],
           int dst[2048][2048])
{
    int i, j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

21 times slower
(Pentium 4)

---

The Memory Mountain

- Intel Core i7
- 2.87 GHz
- 3D: 64 MiB L1 (data) cache
- 256 MiB L2 cache
- 6 MB L3 cache
Reality #3: Performance isn’t counting ops

- Exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops

- Must understand system to optimize performance
  - How programs compiled and executed
  - How memory system is organized
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Example Matrix Multiplication

- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have exactly the same operations count \((2n^3)\)

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double precision)

![Graph showing performance comparison between triple loop and best code (K. Goto). The graph shows a significant improvement in performance for the best code.]
MMM Plot: Analysis

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

Gflop/s

Multiple threads: 4x

Vector instructions: 4x

Memory hierarchy and other optimizations: 20x

• Reason for 20x: blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice

• Effect: less register spills, less L1/L2 cache misses, less TLB misses

CSE351’s role in the “new CSE Curriculum”

• Pre-requisites
  § 142 and 143: Intro Programming I and II

• One of 6 core courses
  § 311: Foundations I
  § 312: Foundations II
  § 331: SW Design and Implementation
  § 332: Data Abstractions
  § 351: HW/SW Interface
  § 352: HW Design and Implementation

• 351 sets the context for many follow-on courses
CSE351’s place in new CSE Curriculum

CSE351

The HW/SW Interface
Underlying principles linking hardware and software

CS 143
Intro Prog II

Textbooks

- Randal E. Bryant and David R. O’Hallaron
- Prentice-Hall, 2010
- http://csapp.cs.cmu.edu
- This book really matters for the course!
- How to solve labs
- Practice problems typical of exam problems

- Samuel P. Harbison III and Guy L. Steele, Jr.
- Prentice-Hall, 2002
- Solid C programming language reference
- Useful book to have on your shelf
Course Components

- Lectures (~30)
  - Higher-level concepts – I’ll assume you’ve done the reading in the text
- Sections (~10)
  - Applied concepts, important tools and skills for labs, clarification of lectures, exam review and preparation
- Written assignments (~4)
  - Problems from text to solidify understanding
- Labs (4)
  - Provide in-depth understanding (via practice) of an aspect of systems
- Exams (midterm + final)
  - Motivation to stay on top of things
  - Demonstrate your understanding of concepts and principles

Resources

- Course Web Page
  - http://www.cse.washington.edu/351
  - Copies of lectures, assignments, exams
- Course Discussion Board
  - Keep in touch outside of class – help each other
  - Staff will monitor and contribute
- Course Mailing List
  - Low traffic – mostly announcements; you are already subscribed
- Staff email
  - Things that are not appropriate for discussion board or better offline
- Anonymous Feedback (linked from homepage)
  - Any comments about anything related to the course where you would feel better not attaching your name
  - By default, all anonymous feedback is posted (so you can view it)
Policies: Grading

- Exams: weighted 1/3 (midterm), 2/3 (final)
- Written assignments: weighted according to effort
  - We’ll try to make these about the same
- Labs assignments: weighted according to effort
  - These will likely increase in weight as the quarter progresses
- Late Policy
  - Two discretionary late days
  - 10%/day after that
- Grading:
  - 55% assignments
  - 45% exams

Welcome to CSE351!

- Let’s have fun
- Let’s learn – together
- Let’s communicate
- Let’s set the bar for a useful and interesting class

- Many thanks to the many instructors who have shared their lecture notes – I will be borrowing liberally through the qtr – they deserve all the credit, the errors are all mine
  - UW: Gaetano Borriello (Inaugural edition of CSE 351, Spring 2010)
  - CMU: Randy Bryant, David O’Halloran, Gregory Kesden, Markus Püschel
  - Harvard: Matt Welsh
  - UW: Tom Anderson, Luis Ceze