EE108B – Lecture 6

Performance and Compilers

Christos Kozyrakis
Stanford University
http://eeclass.stanford.edu/ee108b

Announcements

- TBD

CPU Execution Time

- CPU Execution Time

$$Execution\ Time = Instructions \times CPI \times Clock\ Cycle\ Time$$

$$Execution\ Time = \frac{Instructions \times CPI}{Clock\ Rate}$$

- To improve performance (i.e., reduce execution time)
  - Increase clock rate (decrease clock cycle time) OR
  - Decrease CPI OR
  - Reduce the number of instructions
- Designers balance cycle time against the number of cycles required
  - Improving one factor may make the other one worse…

Clock Rate ≠ Performance

- Mobile Intel Pentium 4 Vs Intel Pentium M
  - 2.4 GHz 1.6 GHz
- Performance on Mobilemark with same memory and disk
  - Word, excel, photoshop, powerpoint, etc.
  - Mobile Pentium 4 is only 15% faster
- What is the relative CPI?
  - ExecTime = IC • CPI/Clock rate
  - IC • CPI_M/1.6 = 1.15 • IC • CPI_4/2.4
  - CPI_4/CPI_M = 2.4/(1.15•1.6) = 1.3

CPI Varies

- Different instruction types require different numbers of cycles
- CPI is often reported for types of instructions

$$Clock\ Cycles = \sum_{i=1}^{n} (CPI_i \times IC_i)$$

- where CPI_i is the CPI for the type of instructions and IC_i is the count of that type of instruction

Computing CPI

- To compute the overall average CPI use

$$CPI = \sum_{i=1}^{n} \left(CPI_i \times \frac{Instruction\ Count_i}{Instruction\ Count} \right)$$

Computing CPI Example

Instruction Type	CPI	Frequency	CPI * Frequency
ALU	1	50%	0.5
Branch	2	20%	0.4
Load	2	20%	0.4
Store	2	10%	0.2

- Given this machine, the CPI is the sum of CPI × Frequency
- Average CPI is 0.5 + 0.4 + 0.4 + 0.2 = 1.5
- What fraction of the time for data transfer?

What is the Impact of Displacement Based Memory Addressing Mode?

Instruction Type	CPI	Frequency	CPI * Frequency
ALU	1	50%	0.5
Branch	2	20%	0.4
Load	2	20%	0.4
Store	2	10%	0.2

- Assume 50% of MIPS loads and stores have a zero displacement.

CPU Time Vs. MIPS (MIPS = million instructions per second)

- Two different compilers are being tested for a 1 GHz. machine with three different classes of instructions: Class A, Class B, and Class C, which require 1, 2, and 3 cycles (respectively). Both compilers are used to produce code for a large piece of software.

  The first compiler's code uses 5 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

  The second compiler's code uses 10 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

- Which sequence will be faster according to MIPS?
- Which sequence will be faster according to execution time?

Speedup

- Speedup allows us to compare different CPUs or optimizations

$$Speedup = \frac{CPUtimeOld}{CPUtimeNew}$$

- Example
  - Original CPU takes 2sec to run a program
  - New CPU takes 1.5sec to run a program
  - Speedup = 1.333 or speedup or 33%

Amdahl's Law

- If an optimization improves a fraction f of execution time by a factor of a

$$Speedup = \frac{Told}{[(1-f)+f/a]*Told} = \frac{1}{(1-f)+f/a}$$

- This formula is known as Amdahl's Law
- Lessons from
  - If f→100%, then speedup = a
  - If a→∞, the speedup = 1/(1-f)
- Summary
  - Make the common case fast
  - Watch out for the non-optimized component

Amdahl's Law Example

- Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?"

- How about making it 5 times faster?

Evaluating Performance

- Performance best determined by running a real application
  - Use programs typical of expected workload
  - Or, typical of expected class of applications
    - e.g., compilers/editors, scientific applications, graphics, etc.
- Small benchmarks
  - nice for architects and designers
  - easy to standardize
  - can be abused
- SPEC (System Performance Evaluation Cooperative)
  - Companies have agreed on a set of real program and inputs
  - Valuable indicator of performance (and compiler technology)
  - Can still be abused

Benchmark Games

- *An embarrassed Intel Corp. acknowledged Friday that a bug in a software program known as a compiler had led the company to overstate the speed of its microprocessor chips on an industry benchmark by 10 percent. However, industry analysts said the coding error…was a sad commentary on a common industry practice of "cheating" on standardized performance tests…The error was pointed out to Intel two days ago by a competitor, Motorola …came in a test known as SPECint92…Intel acknowledged that it had "optimized" its compiler to improve its test scores. The company had also said that it did not like the practice but felt to compelled to make the optimizations because its competitors were doing the same thing…At the heart of Intel's problem is the practice of "tuning" compiler programs to recognize certain computing problems in the test and then substituting special handwritten pieces of code…*

  Saturday, January 6, 1996 New York Times

SPEC CPU2000

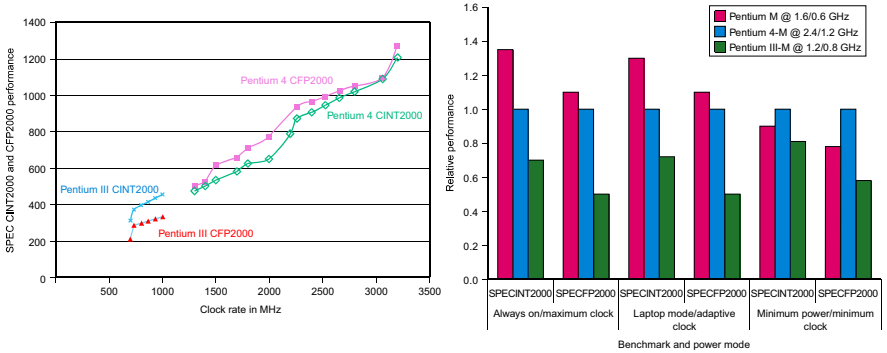
Integer benchmarks		FP benchmarks	
Name	Description	Name	Type
gzip	Compression	wupwise	Quantum chromodynamics
vpr	FPGA circuit placement and routing	swim	Shallow water model
gcc	The Gnu C compiler	mgrid	Multigrid solver in 3-D potential field
mcf	Combinatorial optimization	applu	Parabolic/elliptic partial differential equation
crafty	Chess program	mesa	Three-dimensional graphics library
parser	Word processing program	galgel	Computational fluid dynamics
eon	Computer visualization	art	Image recognition using neural networks
perlbmk	perl application	equake	Seismic wave propagation simulation
gap	Group theory, interpreter	facerec	Image recognition of faces
vortex	Object-oriented database	ammp	Computational chemistry
bzip2	Compression	lucas	Primality testing
twolf	Place and rote simulator	fma3d	Crash simulation using finite-element method
		sixtrack	High-energy nuclear physics accelerator design
		apsi	Meteorology: pollutant distribution

FIGURE 4.5 The SPEC CPU2000 benchmarks. The 12 integer benchmarks in the left half of the table are written in C and C++, while the floating-point benchmarks in the right half are written in Fortran (77 or 90) and C. For more information on SPEC and on the SPEC benchmarks, see www.spec.org. The SPEC CPU benchmarks use wall clock time as the metric, but because there is little I/O, they measure CPU performance.

SPEC CPU2000

Does doubling the clock rate double the performance?

Can a machine with a slower clock rate have better performance?



Other Benchmarks

- Scientific computing: Linpack, SpecOMP, SpecHPC, …
- Embedded benchmarks: EEMBC, Dhrystone, …
- Enterprise computing
  - TCP-C, TPC-W, TPC-H
  - SpecJbb, SpecSFS, SpecMail, Streams,
- Other
  - 3Dmark, ScienceMark, Winstone, iBench, AquaMark, …

- Watch out: your results will be as good as your benchmarks
  - Make sure you know what the benchmark is designed to measure
  - Performance is not the only metric for computing systems
    - Cost, power consumption, reliability, real-time performance, …

Performance Summary

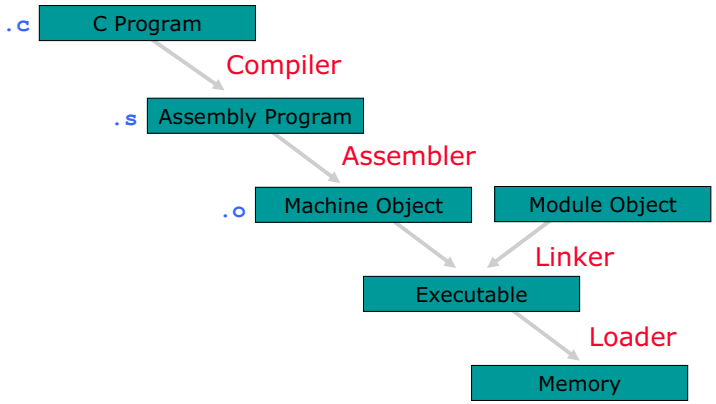
- Performance is specific to a particular program/s
  - Total execution time is a consistent summary of performance

- For a given architecture performance increases come from:
  - Increases in clock rate (without adverse CPI affects)
  - Improvements in processor organization that lower CPI
  - Compiler enhancements that lower CPI and/or instruction count
  - Algorithm/Language choices that affect instruction count

- Pitfall: expecting improvement in one aspect of a machine's performance to affect the total performance

Translation Hierarchy

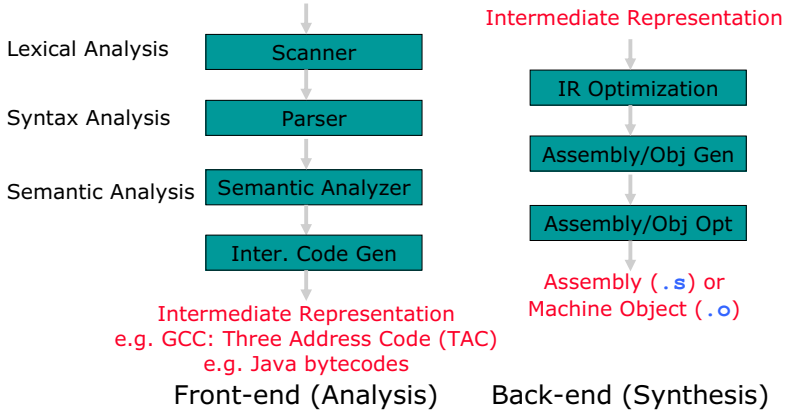
- High-level → Assembly → Machine



Compiler

Converts high-level language to machine code

- Compiler phases



Intermediate Representation (IR)

- Three-address code (TAC)

```
j = 2 * i + 1;
if (j >= n)
   j = 2 * i + 3;
return a[j];
```

```
      t1 = 2 * i
      t2 = t1 + 1
      j = t2
      t3 = j < n
      if t3 goto L0
      t4 = 2 * i
      t5 = t4 + 3
      j = t5
L0:   t6 = a[j]
      return t6
```

Single assignment

Optimizing Compilers

- Provide efficient mapping of program to machine
  - Code selection and ordering
  - eliminating minor inefficiencies
  - register allocation
- Don't (usually) improve asymptotic efficiency
  - Up to programmer to select best overall algorithm
  - Big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Optimization types
  - Local: inside basic blocks
  - Global: across basic blocks e.g. loops

Limitations of Optimizing Compilers

- Operate Under Fundamental Constraints
  1. Improved code has same output
  2. Improved code has same side effects
  3. Improved code is as correct as original code
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
  - Compiler has difficulty anticipating run-time inputs
  - Profile driven dynamic compilation becoming more prevalent
  - Dynamic compilation (e.g. Java JIT)
- **When in doubt, the compiler must be conservative**

Preview: Code Optimization

- Goal: Improve performance by:
  - Removing redundant work
    - Unreachable code
    - Common-subexpression elimination
    - Induction variable elimination
  - Using simpler and faster operations
    - Dealing with constants in the compiler
    - Strength reduction
  - Managing registers well

$$Execution\ Time = Instructions \times CPI \times Clock\ Cycle\ Time$$

Constant Folding

- Detect & combine values that will be constant.
- Respect laws of arithmetic on target machine.

```
a = 10 * 5 + 6 - b;
```

```
_t0 = 10 ;
_t1 = 5 ;
_t2 = _t0 * _t1 ;
_t3 = 6 ;
_t4 = _t2  * _t3 ;
_t5 = _t4  - b ;
a = _t5 ;
```

```
_t0 = 56 ;
_t1 = _t0 - b ;
a = _t1 ;
```


Constant Propagation

- When x gets assigned a constant c, replace uses of x with uses of c, as long as x remains unchanged.

```
_tmp4 = 0 ;
f0 = _tmp4 ;
_tmp5 = 1 ;
f1 = _tmp5 ;
_tmp6 = 2 ;
i = _tmp6 ;
```

```
f0 = 0 ;
f1 = 1 ;
i = 2 ;
```

- VERY useful for turning constants into immediates, for code generator.

Constant Propagation

```
_tmp0 = 12 ;
_tmp1 = arr + _tmp0 ;
_tmp2 = *(_tmp1) ;
```

```
addiu $t0, $zero, 12
lw $t1, -8($fp)
addu $t2, $t1, $t0
lw $t3, 0($t2)
```

- Replace explicit add instructions with offsets or immediates in the MIPS instruction.

```
_tmp0 = *(arr + 12) ;
```

```
lw $t0, -8($fp)
lw $t1, 12($t0)
```


Reduction in Strength

- Some operations are faster (lower CPI) than others:
  - Multiplication slower than Addition
  - Mult or Divide by 2^n slower than shift left or right
  - Multiplies take 12 cycles on MIPS R2000
  - ALU instructions take 1 cycle
- Can replace complex instruction with equivalent simple instructions
- Correctness:
  - Overflow behavior preserved?
  - Other side-effects of operation? (interrupts)

Reduction in Strength

- How to replace multiplication with addition?
- Often presents in *for* loops & array walks.

```
while (i<100) arr[i++] = 0;
```

```
L0:   _tmp2 = i < 100;
   IfZ _tmp2 Goto _L1 ;
   _tmp4 = 4 * i ;
   _tmp5 = arr + _tmp4 ;
   *(_tmp5) =  0 ;
   i = i + 1 ;
   Goto L0 ;
L1:
```

```
   _tmp4 = arr ;
L0:   _tmp2 = i < 100;
   IfZ _tmp2 Goto _L1 ;
   *_tmp4 =  0;
   _tmp4 = _tmp4 + 4;
   i = i + 1 ;
   Goto L0 ;
L1:
```


Common Sub-Expression Elimination

```
main() {
  int x, y, z;
  // They get
  // initial values
  x = (1+20)* -x;
  y = (x*x)+(x/y);
  y = z = (x/y)/(x*x);
}
```

```
tmp1 = 1 + 20 ;
tmp2 = -x ;
x = tmp1 * tmp2 ;
tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
tmp5 = x / y ;
tmp6 = x * x ;
z = tmp5 / tmp6 ;
y = z ;
```


Common Sub-Expression Elimination

```
t1 = 1 + 20 ;
t2 = -x ;
x = t1 * t2 ;
t3 = x * x ;
t4 = x / y ;
y = t3 + t4 ;
t5 = x / y ;
t6 = x * x ;
z = t5 / t6 ;
y = z ;
```

```
t2 = -x ;
x = 21 * t2 ;
t3 = x * x ;
t4 = x / y ;
y = t3 + t4 ;
t5 = x / y ;
z = t5 / t3 ;
y = z ;
```


Induction Variable Elimination

```
while (i<100) arr[i++] = 0;
```

- Remember replacing multiply with add in loop?
- Why not get rid of induction variable i altogether?

```
   t4 = arr ;
L0:   t2 = i < 100;
   IfZ t2 Goto _L1 ;
   *t4 =  0;
   t4 = t4 + 4;
   i = i + 1 ;
L1:
```

```
   t4 = arr ;
L0:   t2 = t4 < 400;
   IfZ t2 Goto _L1 ;
   *t4 =  0;
   t4 = t4 + 4;
L1:
```

Register Allocation & Assignment

- Registers provide fastest data access in the processor.
- Two terms:
  - What variables belong in registers? (allocation)
  - What register will hold this value? (assignment)
- Once a value is in a register, we'd like not to spill & restore it to use same value again.

Assembly Code Generation

- Example: a in $a1, i in $a0, n in $a2

```
      t1 = 2 * i

      j = t1 + 1
      t3 = j < n
      if t3 goto L0

      j = t1 + 3

L0:   t6 = a[j]
      return t6
```

• Register allocation

```
      slli  $t0, $a0, 1

      addiu $t2, $t0, 1
      slt   $t2, $t2, $a2
      bne   $t2, $zero, L0
      addiu $t2, $t0, 3

L0:   slli  $t2, $t2, 2
      addu  $t2, $t2, $a1
      lw    $v0, 0($t2)
      jr    $ra
```


Compiler Performance on Bubble Sort

Gcc Optimization	Relative Performance	Clock Cycles (billions)	Instr. Count (billions)	CPI
none	1.00	158.6	114.9	1.38
O1 (medium)	2.37	67.0	37.5	1.79
O2 (full)	2.38	66.5	40.0	1.66
O3 (proc. Inline)	2.41	65.7	45.0	1.46

100,000 word bubble sort

Assembler

- Expands macros and pseudoinstructions as well as converts constants
- Primary purpose is to produce an object file
  - Machine language instructions
  - Application data
  - Information for memory organization

Pseudoinstructions

- Assembler expands *pseudoinstructions*

```
move $t0, $t1              # Copy $t1 to $t0

addu $t0, $zero, $t1       # Actual instruction
```

- Some pseudoinstructions need a temporary register
  - Cannot use `$t`, `$s`, etc. since they may be in use
  - The `$at` register is reserved for the assembler

```
blt $t0, $t1, L1     # Goto L1 if $t0 < $t1

slt $at, $t0, $t1    # Set $at = 1 if $t0 < $t1
bne $at, $zero, L1   # Goto L1 if $at != 0
```


Object File

- Includes
  - Object header – Describes file organization
  - Text segment – Machine code
  - Data segment – Static and dynamic data
  - Relocation information – Identifies instructions/data that depend on absolute addresses when program is loaded
  - Symbol table – List of labels that are not defined (ex. external references)
  - Debugging information – Describes relationship between source code and machine instructions

Linker

- Linker combines multiple object modules
  - Identify where code/data will be placed in memory
  - Resolve code/data cross references
  - Produces executable if all references found
- Steps
  1. Place code and data modules in memory
  2. Determine the address of data and instruction labels
  3. Patch both the internal and external references
- Separation between compiler and linker makes standard libraries an efficient solution to maintaining modular code

Loader

- Loader used at run-time
  1. Reads executable file header for size of text/data segments
  2. Create address space sufficiently large
  3. Copy program from executable on disk into memory
  4. Copy arguments to main program's stack
  5. Initialize machine registers and set stack pointer
  6. Jump to start-up routine
  7. Terminate program when execution completes

SPIM System Calls

- SPIM provides a small number of OS "system calls"

Service	Code	Argument	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer in $v0
read_float	6		float if $f0
read_double	7		double in $f0
read_string	8	$a0 = buffer, $a1 = length	data in buffer
sbrk	9	$a0 = amount	address in $v0
exit	10		

- Set system call code in `$v0` and pass arguments as needed

```
li $v0, 1           # Set system call to print_int
li $a0, 12          # Load constant 12
syscall
```

- See Appendix A for details and examples

MIPS Assembler Directives

- SPIM supports a subset of the MIPS assembler directives
- Some of the directives include:
  - `.asciiz` – Store a null-terminated string in memory
  - `.data` – Start of data segment
  - `.global` – Identify an exported symbol
  - `.text` – Start of text segment
  - `.word` – Store words in memory
- See Appendix A for details and examples