EE108B – Lecture 5

CPU Performance

Christos Kozyrakis

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

Announcements

- Upcoming deadlines
  - HW1 due on Tuesday 1/25
  - Lab1 due on Tuesday 1/30
  - PA1 due on Thursday 2/8

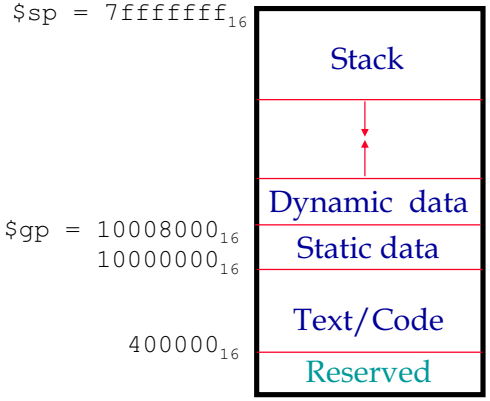
Review: Procedure Call and Return

- Procedures are required for structured programming
  - Aka: functions, methods, subroutines, …
- Implementing procedures in assembly requires several things to be done
  - Memory space must be set aside for local variables
  - Arguments must be passed in and return values passed out
  - Execution must continue after the call
- Procedure Steps
  1. Place parameters in a place where the procedure can access them
  2. Transfer control to the procedure
  3. Acquire the storage resources needed for the procedure
  4. Perform the desired task
  5. Place the result value in a place where the calling program can access it
  6. Return control to the point of origin

Stacks

- Data is pushed onto the stack to store it and popped from the stack when not longer needed
  - MIPS does not support in hardware (use loads/stores)
  - Procedure calling convention requires one
- Calling convention
  - Common rules across procedures required
  - Recent machines are set by software convention and earlier machines by hardware instructions
- Using Stacks
  - Stacks can grow up or down
  - Stack grows down in MIPS
  - Entire stack frame is pushed and popped, rather than single elements

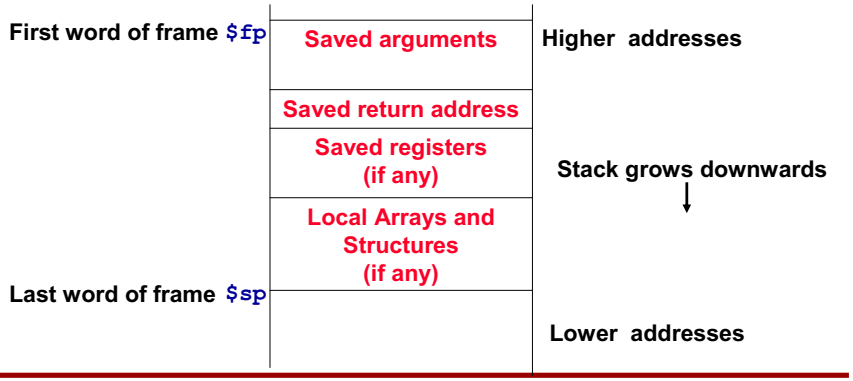
MIPS Storage Layout



Stack and dynamic area grow towards one another to maximize storage use before collision

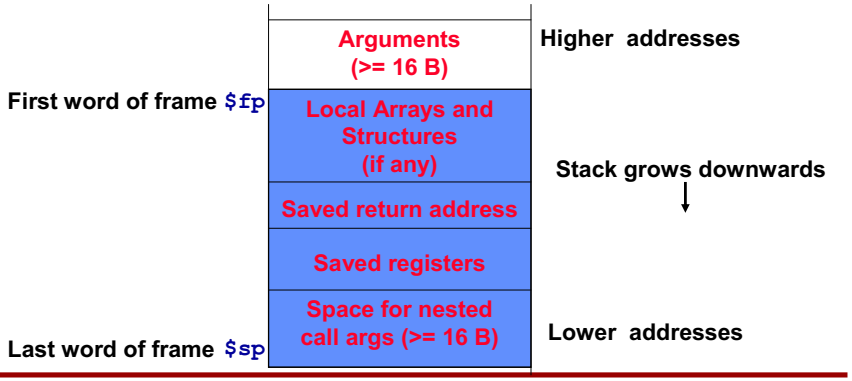
Procedure Activation Record (Frame) textbook

- Each procedure creates an activation record on the stack



Procedure Activation Record (Frame) SGI/GCC Compiler

- Each procedure creates an activation record on the stack
  - At least 32 bytes by convention to allow for $a0-$a3, $ra, $fp and be double double word aligned (16 byte multiple)



Register Assignments

- Use the following calling conventions

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Register Assignments (cont)

0	$zero Zero constant (0)	16	$s0 Callee saved
1	$at Reserved for assembler	. . .	
2	$v0 Expression results	23	$s7
3	$v1	24	$t8 Temporary (cont'd)
4	$a0 Arguments	25	$t9
5	$a1	26	$k0 Reserved for OS kernel
6	$a2	27	$k1
7	$a3	28	$gp Pointer to global area
8	$t0 Caller saved	29	$sp Stack Pointer
. . .		30	$fp Frame Pointer
15	$t7	31	$ra Return address

Caller vs. Callee Saved Registers

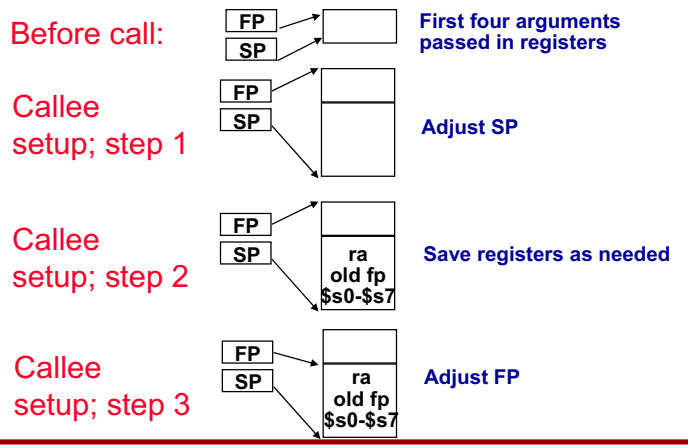
Preserved	Not Preserved
Saved registers ($s0-$s7)	Temporary registers ($t0-$t9)
Stack/frame pointer ($sp, $fp, $gp)	Argument registers ($a0-$a3)
Return address ($ra)	Return values ($v0-$v1)

- Preserved registers (Callee Save)
  - Save register values on stack prior to use
  - Restore registers before return
- Not preserved registers (Caller Save)
  - Do what you please and expect callees to do likewise
  - Should be saved by the caller if needed after procedure call

Call and Return

- Caller
  - Save caller-saved registers `$a0-$a3`, `$t0-$t9`
  - Load arguments in `$a0-$a3`, rest passed on stack
  - Execute `jal` instruction
- Callee Setup
  1. Allocate memory for new frame (`$sp` = `$sp` - frame)
  2. Save callee-saved registers `$s0-$s7`, `$fp`, `$ra`
  3. Set frame pointer (`$fp` = `$sp` + frame size - 4)
- Callee Return
  - Place return value in `$v0` and `$v1`
  - Restore any callee-saved registers
  - Pop stack (`$sp` = `$sp` + frame size)
  - Return by `jr $ra`

Calling Convention Steps



Simple Example

```
int foo(int num)
{
   return(bar(num + 1));
}

int bar(int num)
{
   return(num + 1);
}
```

```
foo:
        addiu   $sp, $sp, -32   # push frame
        sw      $ra, 28($sp)    # Store $ra
        sw      $fp, 24($sp)    # Store $fp
        addiu   $fp, $sp, 28    # Set new $fp
        addiu   $a0, $a0, 1     # num + 1
        jal     bar             # call bar
        lw      $fp, 24($sp)    # Load $fp
        lw      $ra, 28($sp)    # Load $ra
        addiu   $sp, $sp, 32    # pop frame
        jr      $ra             # return

 bar:
        addiu   $v0, $a0, 1     # leaf procedure
        jr      $ra             # with no frame
```

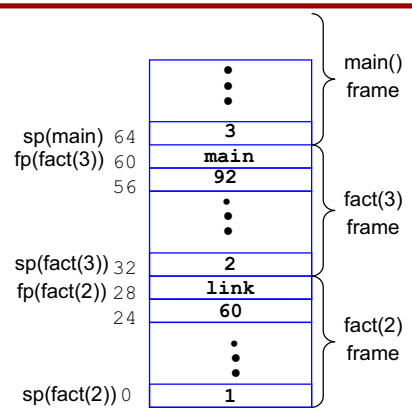

Factorial Example

```
int fact(int n)
{
   if (n <= 1)
      return(1);
   else
      return(n*fact(n-1));
}
```

```
fact:   slti    $t0, $a0, 2       # a0 < 2
        beq     $t0, $zero, skip  # goto skip
        ori     $v0, $zero, 1     # Return 1
        jr      $ra               # Return
skip:   addiu   $sp, $sp, -32     # $sp down 32
        sw      $ra, 28($sp)      # Save $ra
        sw      $fp, 24($sp)      # Save $fp
        addiu   $fp, $sp, 28      # Set up $fp
        sw      $a0, 32($sp)      # Save n
        addui   $a0, $a0, -1      # n - 1
        jal     fact              # Call recursive
link:   lw      $a0, 32($sp)      # Restore n
        mul     $v0, $v0, $a0     # n * fact(n-1)
        lw      $ra, 28($sp)      # Load $ra
        lw      $fp, 24($sp)      # Load $fp
        addiu   $sp, $sp, 32      # Pop stack
        jr      $ra               # Return
```


Running Factorial Example

```
main()
{
   fact(3);
}
```



Stack just before call of `fact(1)`

Factorial Again
Tail Recursion

Tail Recursive Procedure

```
int fact_t (int n, int val)
{
  if (n <= 1)
    return val;
  return
    fact_t(n-1, val*n);
}
```

General Form

```
fact_t(n, val)
{
   • • •
  return
   fact_t(Nexpr, Vexpr)
}
```

Top-Level Call

```
int fact(int n)
{
  return fact_t(n, 1);
}
```

- Form
  - Directly return value returned by recursive call
- Consequence
  - Can convert into loop

Removing Tail Recursion

Optimized General Form

```
fact_t(n, val)
{
 start:
   • • •
  val = Vexpr;
  n = Nexpr;
  goto start;
}
```

Resulting Code

```
int fact_t (int n, int val)
{
 start:
  if (n <= 1)
    return val;
  val = val*n;
  n = n-1;
  goto start;
}
```

- Effect of Optimization
  - Turn recursive chain into single procedure (leaf)
  - Eliminate procedure call overhead
  - No stack frame needed
  - Constant space requirement
    - Vs. linear for recursive version

Assembly Code for fact_t

```
int fact_t (int n, int val)
{
 start:
  if (n <= 1)
    return val;
  val = val*n;
  n = n-1;
  goto start;
}
```

```
fact_t: addui $v0, $a1, $zero    # $v0 = $a1
start:  slti  $t0, $a0, 2        # n < 2
        beq   $t0, $zero, skip   # if n >= 2 goto skip
        jr    $ra                # return
skip:   mult  $v0, $v0, $a0      # val = val * n
        addiu $a0, $a0, -1       # n = n - 1
        j     start              # goto start
```


Today's Topics

- Reading 2.7, 4.1–4.4
- Psuedo Instructions
- CPU performance
  - CPU == central processing unit == the processor

Pseudo-instructions

- 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
```

Computer Performance Metrics

- Response Time (latency)
  - How long does it take for my job to run?
  - How long does it take to execute a job?
  - How long must I wait for the database query?
- Throughput
  - How many jobs can the machine run at once?
  - What is the average execution rate?
  - How many queries per minute?
- If we upgrade a machine with a new processor what do we increase?
- If we add a new machine to the lab what do we increase?

Performance = Execution Time

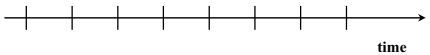
- Elapsed Time
  - Counts everything *(disk and memory accesses, I/O , etc.)*
  - A useful number, but often not good for comparison purposes
    - E.g., OS & multiprogramming time make it difficult to compare CPUs
- CPU time
  - Doesn't count I/O or time spent running other programs
  - Can be broken up into system time, and user time
- Our focus: user CPU time
  - Time spent executing the lines of code that are "in" our program
  - Includes arithmetic, memory, and control instructions…

Clock Cycles

- Instead of reporting execution time in seconds, we often use cycles

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- Clock "ticks" indicate when to start activities:



- Cycle time = time between ticks = seconds per cycle
- Clock rate (frequency) = cycles per second (1 Hz. = 1 cycle/sec)

A 2 Ghz. clock has a $\frac{1}{2\times10^{9}} \times 10^{12} = 500$ picoseconds (ps) cycle time

Now that we understand Clock Cycles

- A given program will require
  - some number of instructions (machine instructions)
  - some number of cycles
  - some number of seconds
- We have a vocabulary that relates these quantities:
  - cycle time (seconds per cycle)
  - clock rate (cycles per second)
  - CPI (cycles per instruction)
  - MIPS (millions of instructions per second)

Performance and Execution Time

- User *CPU execution time*

$$Execution\ Time = Clock\ Cycles\ for\ \mathrm{Pr}ogram \times Clock\ Cycle\ Time$$

- Since Cycle Time is 1/Clock Rate (or clock frequency)

$$Execution\ Time = \frac{Clock\ Cycles\ for\ \mathrm{Pr}ogram}{Clock\ Rate}$$

- The program should be something real people care about
  - Desktop: MS office, edit, compile
  - Server: web, e-commerce, database
  - Scientific: physics, weather forecasting

Measuring Clock Cycles

- Clock cycles/program is not an intuitive or easily determined value, so

$$Clock\ Cycles = Instructions \times Clock\ Cycles\ Per\ Instruction$$

- Cycles Per Instruction (CPI) used often
- CPI is an **average** since the number of cycles per instruction varies from instruction to instruction
  - Average depends on instruction mix, latency of each inst. type etc.
- CPIs can be used to compare two implementations of the same ISA, but is not useful alone for comparing different ISAs
  - An x86 add is different from a MIPS add

Using CPI

- Drawing on the previous equation:

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

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

- 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.

Other Benchmarks

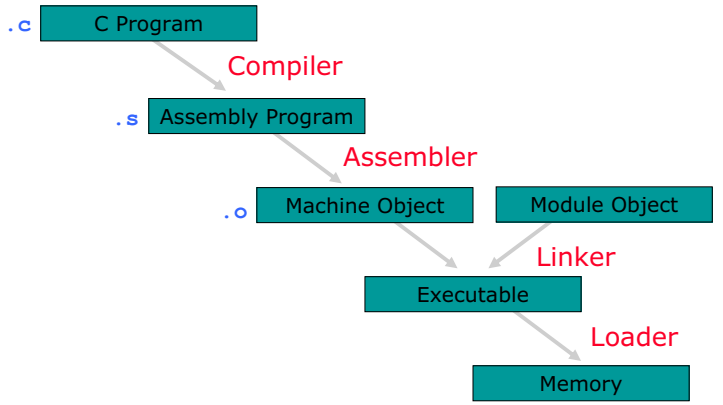
- Scientific computing: Linpack, SpecOMP, SpecHPC, …
- Embedded benchmarks: EEMBC, Dhrystone, …
- Enterprise computing
  - TCP-C, TPC-W, TPC-H
  - SpecJbb, SpecSFS, SpecMail, Streams,
- Other
  - 3Dmark, ScienceMark, Winstone, iBench, AquaMark, …
- Watch out: your results will be as good as your benchmarks
  - 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

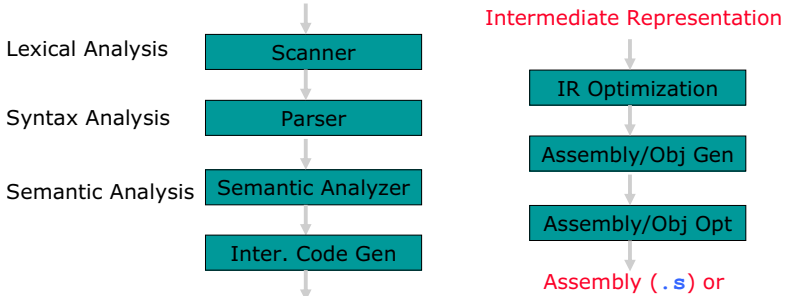
- High-level → Assembly → Machine



Compiler

Converts high-level language to machine code

- Compiler phases



Front-end (Analysis) Back-end (Synthesis)

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
- **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
  - Creating simpler operations
    - Dealing with constants in the compiler
    - Strength reduction
  - Managing registers well

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