CMPE 220

Class 22 – Optimization

History

UNIVAC 1

- 1952
- ~1,000 operations per second
- 1,000 words of memory
- No mass storage

Modern High-End Server

- 2023
- 500,000,000,000 operations per second
- 16,000,000,000,000 bytes of memory

And yet…
Computers are too slow!

Loss of Efficiency

- Higher Expectations
  - Windowing systems & GUIs
  - Networking
  - Process Management, Memory Management, Disc Management
  - Artificial Intelligence & Machine Learning
- Inefficient Software Development Tools
  - Compilers
  - Object Oriented Programming
  - Garbage Collection
  - Frameworks
  - Virtualization
- “Lazy” coding

- Microsoft Word: 2.25 GB on disk!

San José State
UNIVERSITY

Economics

- Rapid advances in hardware have masked rising inefficiencies for decades
- Hardware advances have slowed
  - Reaching the limits of physics
- Computers are now ubiquitous and used for an ever increasing number and size of tasks
- Competition is putting price pressure on service providers and on hardware manufacturers

Compiler Optimization

Introduction to Code Optimization

- **Goal:** The compiler generates better object code.
- Automatically discover information about the runtime behavior of the source program.
- Use that information to generate better code.

Early Compilers

Alpha = 15;	LDA	#15
	STA	Alpha
Beta = Alpha * 27;	LDA	Alpha
	MUL	#27
	STA	Beta

San José State University

“Better” Generated Object Code

- Runs faster
  - What people usually mean when they talk about optimization.

- Uses less memory
  - Embedded chips may have limited amounts of memory.

- Consumes less power
  - A CPU chip may be in a device that needs to conserve power.
  - Some operations can require more power than others.

San José State University

Code Optimization Challenges: Safety

- The code optimizer <u>must not change the results</u> of the source program.

- During execution, the optimized object code must have the <u>same runtime effects</u> as the unoptimized object code.
    - "Same effect": The variables have the same calculated values.

- <u>Bad idea</u>: Compute the wrong values, but faster!

San José State University

Instruction Selection

- What sequence of target machine instructions should the code generator emit?
- The symbol table and parse tree are the primary sources of information for the code generator.

San José State University

Instruction Selection: JVM Examples

- Load and store instructions
  - Emit **`ldc`** ***x*** or **`iconst_`*****n*** or **`bipush`** ***n***
  - Emit **`iload`** ***n*** or **`iload_`*****n***
  - Emit **`istore`** ***n*** or **`istore_`*****n***

- Pascal **`CASE`** statement
  - Emit **`lookupswitch`** if the test values are <u>sparse</u>.
  - Emit **`tableswitch`** if the test values are <u>densely packed</u>.

San José State
UNIVERSITY

Instruction Selection: JVM Examples, *cont'd*

- Pascal assignment **`i := i + 1`**
  (assume **`i`** is local variable in slot #0)

```
iload_0
iconst_1
iadd
istore_0
```

or

```
iinc 0 1
```

Register Allocation

- Unlike the JVM, many machines (like SIC/XE) can have hardware registers that are faster than main memory.
  - General-purpose registers (A, B)
  - Floating-point registers (F)
  - Address registers (X)

- A smart code generator emits code that:
  - Loads values into registers as much as possible.
  - Keeps values in registers as long as possible.
    - But no longer than necessary!

San José State University

Avoid Extraneous Loads

- What is in each register?
- A table, mapping registers to their contents

Register	Symtab Pointer
A	*temp*
S	*inventory*
T	*(unused)*
B	stack
X	stackpoint
F	*salary*

Register Allocation, *cont'd*

- The code generator assigns registers on a per-routine basis.

- Procedure or function call:
  - Emit code to save the caller's register contents.
  - The procedure or function gets a "fresh" set of registers.

- Return:
  - Emit code to restore the caller's register contents.
  - Better: Save and restore only the registers that a routine uses.

San José State University

Register Allocation Challenges

- Limited number of registers.
- May need to *spill* a register value into memory.
  - Store a register's value into memory in order to free up the register.
  - Later reload the value back from memory into the register.
- Pointer variables
  - Cannot keep a variable's value in a register if there is a pointer to the variable's memory location.

Data Flow Analysis

- Determine which variables are live.
- A variable ***v*** is live at statement ***p1*** in a program if:
  - There is an execution path from statement ***p1*** to a statement ***p2*** that uses ***v***, and
  - Along this path, the value of ***v*** does <u>not</u> change.
- Live variables should **not** be kept in registers.

San José State University

Instruction Scheduling

- Change the order of the instructions that the code generator emits to take advantage of pipelining

- But don't change the program's results!

- A form of optimization to increase execution speed.

San José State University

Instruction Scheduling, *cont'd*

- With most machine architectures, different instructions take different amounts of time to execute.
  - Example: Floating-point instructions take longer than the corresponding integer instructions.
  - Example: Loading from memory and storing to memory each takes longer than adding two numbers in registers.

San José State UNIVERSITY

Instruction Scheduling Example

- Assume that **`load`** and **`store`** each takes 3 cycles, **`mult`** takes 2 cycles, and **`add`** takes 1 cycle.
- Simple case:
  Sequential execution only.

Cycle start	Instruction	Operation
1	`load`	w ➔ *r1*
4	`add`	*r1* + *r1* ➔ *r1*
5	`load`	x ➔ *r2*
8	`mult`	*r1* * *r2* ➔ *r1*
10	`load`	y ➔ *r2*
13	`mult`	*r1* * *r2* ➔ *r1*
15	`load`	z ➔ *r2*
18	`mult`	*r1* * *r2* ➔ *r1*
20	`store`	*r1* ➔ w

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
load r1			+	load r2			mult		load r2			mult		load r2			mult		store r1		

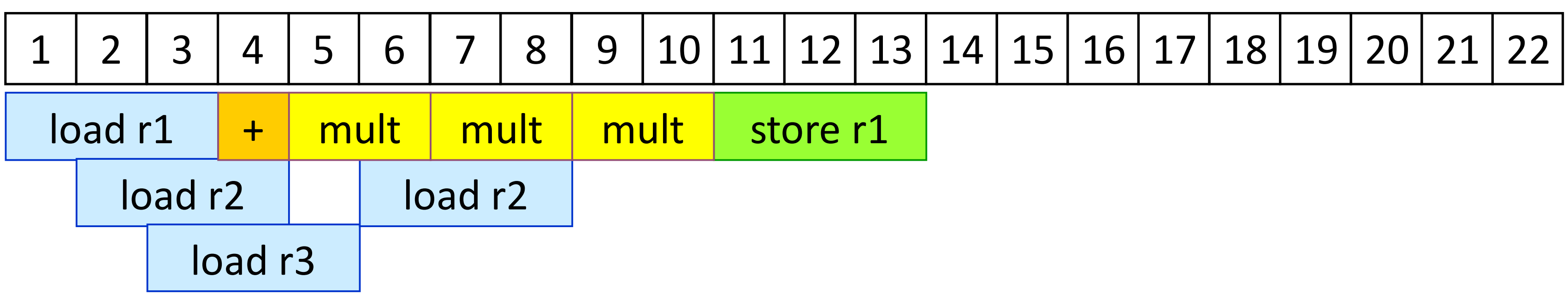
Instruction Scheduling, *cont'd*

- Assume that **`load`** and **`store`** each takes 3 cycles, **`mult`** takes 2 cycles, and **`add`** takes 1 cycle.

- Assume the machine can overlap instruction execution.
  - instruction-level parallelism

Cycle start	Instruction	Operation
1	`load`	w → *r1*
2	`load`	x → *r2*
3	`load`	y → *r3*
4	`add`	*r1* + *r1* → *r1*
5	`mult`	*r1* * *r2* → *r1*
6	`load`	z → *r2*
7	`mult`	*r1* * *r3* → *r1*
9	`mult`	*r1* * *r2* → *r1*
11	`store`	*r1* → w

Requires using another register *r3*.



Speed Optimization: Constant Folding

- Suppose we have the constant definition:

```
CONST pi = 3.14;
```

and we have the real expression `2*pi`

- Instead of emitting instructions to load 2, convert to float, load 3.14, and multiply ...
  - Simply emit a single instruction to load the value 6.28

San José State University

Speed Optimization: Constant Propagation

- Suppose parse tree analysis determines that a variable ***v*** always has the value ***c*** for a given set of statements.

- When generating code for those statements, instead of emitting an instruction to load the value of ***v*** from memory ...

- Emit an instruction to load the constant ***c***.

San José State University

Speed Optimization: Strength Reduction

- Replace an operation by a <u>faster equivalent operation</u>.

San José State University

Speed Optimization: Strength Reduction, *cont'd*

- Example: Suppose the integer expression `5*i` appears in a tight loop.

  - Given: Multiplication is more expensive than addition.

  - One solution: Generate code for `i+i+i+i+i` instead.

  - Another solution: Treat the expression as if it were written `(4*i)+i` and do the multiplication as a <u>shift left</u> of 2 bits.
    - Generate the code to <u>shift</u> the value of `i` and then <u>add</u> the original value of `i`.

San José State University

Speed Optimization: Dead Code Elimination

- Consider the following statements:
  - `WHILE  (i<>i) DO • • •`
  - `IF (1 == 2) THEN • • •`

- Don’t emit any code for these statements.

Speed Optimization: Loop Unrolling

- Loop overhead: <u>initialize</u>, <u>test</u>, and <u>increment</u>.

- Example:

```
FOR i := 1 TO 10000 DO BEGIN
    FOR j := 1 TO 3 DO BEGIN
        s[i,j] := a[i,j] + b[i,j]
    END
END
```

- Unroll the inner loop by generating code for:

```
FOR i := 1 TO 10000 DO BEGIN
    s[i,1] := a[i,1] + b[i,1];
    s[i,2] := a[i,2] + b[i,2];
    s[i,3] := a[i,3] + b[i,3];
END
```

San José State University

Common Subexpression Elimination

- Example: `x := y*(i-j*k) + (w + z/(i-j*k))`

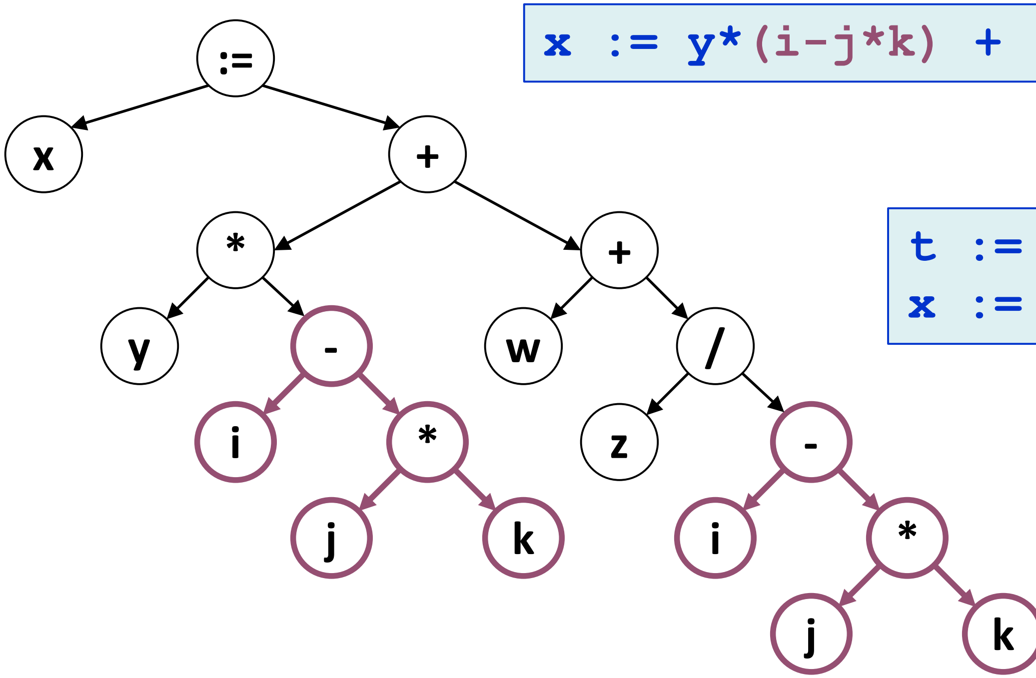
- Generate code as if the statement were instead:

```
t := i-j*k;
x := y*t + (w + z/t);
```

- This may not be so easy for the back end to do!

San José State University

Common Subexpression Elimination, *cont'd*




```
x := y*(i-j*k) + (w + z/(i-j*k))
```

```
t := i-j*k;
x := y*t + (w + z/t)
```

- How do you recognize the common subexpression in the parse tree? (Hash values)

Loop Optimization: Invariant Code *Hoisting*

- **Invariant code within the loop**

- Example:

```
FOR i := 1 TO 10000 DO BEGIN
    a[i] := i * 3.14159;
    x = y + z;
END
```

- Extract Invariants:

```
x = y + z;
FOR i := 1 TO 10000 DO BEGIN
    a[i] := i * 3.14159;
END
```

San José State University

Loop Optimization: Invariant Code *Hoisting*

- **Invariant code within the loop**

- Example:

```
FOR i := 1 TO max-1 DO BEGIN
    a[i] := i * 3.14159;
END
```

- Extract Invariants:

```
temp = max - 1;
FOR i := 1 TO temp DO BEGIN
    a[i] := i * 3.14159;
END
```

San José State University

Function Inlining

- **Replace Function Calls with Inline Code**
- Saves overhead of function call and return

- Example:

```
int add (int x, int y)
   { return x + y; }

int sub (int x, int y)
   { return add (x, -y); }
```

- Inline Code:

```
int sub (int x, int y)
   { return x + (– y); }
```

Compiling Object-Oriented Languages

- Extra challenges!

- Dynamically allocated objects
  - Allocate objects in the heap.

- Inheritance

- Method overriding and overloading
  - Liskov Substitution Principle
  - Run-time function binding

- Polymorphism and virtual methods