CSE 504: Compiler Design

Code Generation

Pradipta De
pradipta.de@sunykorea.ac.kr
• Introducing basic concepts in code generation phase
The problem of generating an optimal target program for a source program is undecidable.
Goals of Code Generator

- Code Generator must produce **correct** code
  - Preserve the semantic meaning of the source code ➔ make effective use of the target resources without changing meaning

- Inputs to Code Generator
  - IR of source program
    - IR can be 3-address code, stack machine code, bytecode, linear representations
    - Syntactic and semantic errors are eliminated, type checking done, type conversions inserted
  - Information in symbol table
    - Used to determine runtime address of data objects denoted by names in IR
Target Machine

• Instruction set architecture (ISA) of target machine determines difficulty of CG

• Types of target machines
  – RISC
    • Many registers, 3-address instructions, simple addressing modes, simple ISA
    • X86 processors
  – CISC
    • Few registers, 2-address instructions, many addressing modes, several register classes, variable length instructions, and instructions with side effects
  – Stack based:
    • Use of stack to execute instructions
    • JVM is a stack based abstract machine
### Target machine code

- **Absolute vs. relocatable machine code**
  - Absolute machine language program: can be placed in fixed location in memory
  - Relocatable machine language program:
    - set of relocatable object modules are generated
    - Object modules are linked by the linker-loader

- **Output of code generation**
  - Assembly language code
  - Assembler converts to target machine code
Instruction Selection

• IR program into executable code sequence
  – Level of IR
    • High level IR $\Rightarrow$ CG may translate each IR into a sequence of machine instructions using code templates
    • Low level IR $\Rightarrow$ may produce more efficient code
  – Nature of ISA
    • Uniform ISA makes CG easier $\Rightarrow$ less exceptions to handle
  – Desired quality of generated code
    • A naïve translation of IR will produce correct but inefficient code
Example of instruction selection

Source code: \[ x = y + z \]

Target Code

\[
\begin{align*}
    & \text{LD } RO, y \quad // \ RO = y \\
    & \text{ADD } RO, RO, z \quad // \ RO = RO + z \\
    & \text{ST } x, RO \quad // \ x = RO
\end{align*}
\]

(load y into register RO)
(add z to RO)
(store RO into x)

Source code: \[ a = b + c \]
\[ d = a + e \]

\[
\begin{align*}
    & \text{LD } RO, b \quad // \ RO = b \\
    & \text{ADD } RO, RO, c \quad // \ RO = RO + c \\
    & \text{ST } a, RO \quad // \ a = RO \\
    & \text{LD } RO, a \quad // \ RO = a \\
    & \text{ADD } RO, RO, e \quad // \ RO = RO + e \\
    & \text{ST } d, RO \quad // \ d = RO
\end{align*}
\]
Register Allocation

• Register are fast to access, but there are not many of them
  – Values not in register is in memory \(\rightarrow\) takes longer to access them

• Two issues
  – Register allocation: select set of variables that will reside in registers
  – Register assignment: select the specific register in which to store the value

• Computation order can affect code efficiency
  – Some order may require less registers to store intermediate results
Simple Target Machine

• Byte addressable machine with n general purpose registers
• Operations:
  – Load: LD dst, addr,
    • LD r, x (load value in location x into r)
  – Store:
    • ST x, r (store value in register r into x)
  – Computation: OP dst, src1, src2
    • SUB r1,r2,r3 (r1 = r2 − r3)
  – Unconditional Jumps: BR L
  – Conditional Jumps: B cond r, L
    • BLTZ r,L ➔ jump to label L if value in register r is less than 0
Simple Target Machine

- **Addressing Modes**
  - Variable name x refers to location in memory
  - Location can be indexed \( a(r) \), where \( a \) is a variable and \( r \) is a register
    - LD R1, a(R2) \( \rightarrow \) R1 = contents(a + contents(R2))
  - Memory location can be an integer indexed by a register
    - LD R1, 100(R2) \( \rightarrow \) R1 = contents(100 + contents(R2))
  - Indirect addressing modes
    - LD R1, *100(R2) \( \rightarrow \) R1 = contents(contents(100 + contents(R2)))
  - Immediate addressing
    - LD R1, #100 \( \rightarrow \) load integer 100 into R1
Examples

Source: \( x = y-z \)

Target

- \( \text{LD \ R1, y} \) // R1 = y
- \( \text{LD \ R2, z} \) // R2 = z
- \( \text{SUB \ R1, R1, R2} \) // R1 = R1 - R2
- \( \text{ST x, R1} \) // x = R1

Source: \( b = a[i] \)

Target

- \( \text{LD \ R1, i} \) // R1 = i
- \( \text{MUL \ R1, R1, 8} \) // R1 = R1 * 8
- \( \text{LD \ R2, a(R1)} \) // R2 = contents(a + contents(R1))
- \( \text{ST b, R2} \) // b = R2

Source: \( x = *p \)

Target

- \( \text{LD \ R1, p} \) // R1 = p
- \( \text{LD \ R2, 0(R1)} \) // R2 = contents(0 + contents(R1))
- \( \text{ST x, R2} \) // x = R2

Source:

- \( \text{if } x < y \text{ goto L} \)

Target

- \( \text{LD \ R1, x} \) // R1 = x
- \( \text{LD \ R2, y} \) // R2 = y
- \( \text{SUB \ R1, R1, R2} \) // R1 = R1 - R2
- \( \text{BLTZ \ R1, M} \) // if R1 < 0 jump to M

We can compute instruction cost based based on the addressing modes.
Address Computation

Addresses are computed based on the program layout

Diagram:

- Code
- Static
- Heap
- Stack
Basic Blocks

• The entire code is partitioned into basic blocks before generating the assembly code

• Basic blocks are transformed into a flow graph
  – Provides context information

```plaintext
for i from 1 to 10 do
  for j from 1 to 10 do
    a[i,j] = 0.0;
  for i from 1 to 10 do
    a[i,i] = 1.0;
for i from 1 to 10 do
  if i <= 10 goto B3

B1
  i = 1

B2
  j = 1
  t1 = 10 * i
  t2 = t1 + j
  t3 = 8 + t2
  t4 = t3 - 88
  a[t4] = 0.0
  j = j + 1
  if j <= 10 goto B3

B3
  i = i + 1
  if i <= 10 goto B2

B4
  i = i + 1
  if i <= 10 goto B2

B5
  i = 1

B6
  t5 = i - 1
  t6 = 88 + t5
  a[t6] = 1.0
  i = i + 1
  if i <= 10 goto B6

EXIT
```
Code Generation

• Rules for register use
  – Some or all operands must be in registers
  – Registers are used to hold temporary values
  – Registers hold global values
    • Values used across blocks
  – Registers are used in runtime storage management
    • Runtime stack pointer

• Some registers are used to store values within a block

• Reserved registers: for global variables and stack management
Code Generation

• Data Structures to manage storage
  – Register descriptor: keeps track of variable names whose values are in the register
    • Initially all register descriptors are empty
  – Address Descriptor: keeps track of location(s) where the current value of the variable can be found
    • Can be stored in the symbol table entry
Code Generation Algorithm

- getReg(I) : select registers for each memory location for the instruction I
  - Ex: getReg( x=y+z)

- Machine Instruction for Operations
  - Load y into Ry
  - Load z into Rz
  - Issue ADD, Rx, Ry, Rz

- Machine Instruction for Copy Statements
  - Adjust the register descriptor of y to include x also

- Ending Basic Block
  - If temporary value, then nothing to do
  - If variable is live on exiting a block, then
    - ST x, R → store x in its memory location

- Managing Register and Address Descriptors
  - LD R, x → change register descriptor, change address descriptor
  - ST R, x → change address descriptor
  - Operation ADD Rx,Ry,Rz → change register descriptor Rx, change address descriptor of x so that its only location is Rx (memory has old value)
  - Remove Rx from address descriptor
\[ t = a - b \]
\[ u = a - c \]
\[ v = t + u \]
\[ a = d \]
\[ d = v + u \]

\[
\begin{array}{c|c|c|c}
R1 & R2 & R3 \\
\hline
\hline
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\hline
a & b & c & d & t & u & v \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{c|c}
a & t \\
\hline
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
a,R1 & b & c & d & R2 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
u & t & c \\
\hline
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
a & b & c & R3 & d & R2 & R1 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
u & t & v \\
\hline
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
a & b & c & d & R2 & R1 & R3 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
u & a,d & v \\
\hline
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
R2 & b & c & d,R2 & R1 & R3 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
d & a & v \\
\hline
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
R2 & b & c & R1 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
d & a & v \\
\hline
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
a,R2 & b & c & d,R1 \\
\hline
\end{array}
\]

exit

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\hline
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
ST a, R2 & ST d, R1 \\
\hline
\end{array}
\]
Implementing getReg(I)

• Source → x = y + z
• Steps:
  – If y is in register, pick Ry → no machine instruction required
  – If y is not in register, but there is an empty register, then pick that register to load y → load y into Ry
  – y is not in register, and no register empty → have to pick a register and make it available
    • Register descriptor says v is in R
    • Address descriptor says v is also somewhere else other than R
    • If v is same as x, that is we can overwrite R since x will not be used before this instruction is executed
    • v is not used later → will be recomputed within the block
    • If all conditions fail, then ST v, R (termed as spill)
• The check is done for all the registers
  – Pick the one that requires least number of store instructions
• Similar rules hold for the left hand side operand, x
Summary

• Challenges of code generation
• Simple code generation
• We will look at
  – Better heuristics for instruction selection
  – Optimization techniques