Code Generation and Optimization

ALSU Textbook Chapters 8.4, 8.5, 8.7, 8.8, 9.1

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu
Introduction

- For some compiler, the intermediate code is a pseudo code of a virtual machine.
  - Interpreter of the virtual machine is invoked to execute the intermediate code.
  - No machine-dependent code generation is needed.
  - Usually with great overhead.
  - Example:
    - Pascal: P-code for the virtual P machine.
    - JAVA: Byte code for the virtual JAVA machine.

Motivation:
- Statement by statement translation might generate redundant codes.
- Locally improve the target code performance by examine a short sequence of target instructions (called a peephole) and do optimization on this sequence.
- Note: Complexity depends on the “window size.”

Optimization.
- Machine-dependent issues.
- Machine-independent issues.
Machine-dependent issues (1/2)

- **Input and output formats:**
  - The formats of the intermediate code and the target program.

- **Memory management:**
  - Alignment, indirect addressing, paging, segment, ...
  - Those you learned from your assembly language class.

- **Instruction cost:**
  - Special machine instructions to speed up execution.
  - Example:
    - *Increment by 1.*
    - *Multiplying or dividing by 2.*
    - *Bit-wise manipulation.*
    - *Operators applied on a continuous block of memory space.*
  - Pick a fastest instruction combination for a certain target machine.
Register allocation: in-between machine dependent and independent issues.

- C language allows the user to manage a pool of registers.
- Some language leaves the task to the compiler.
- Idea: save mostly used intermediate result in a register. However, finding an optimal solution for using a limited set of registers is NP-hard.
- Example:
  
  \[
  t := a + b \\
  \text{load } R0, a \\
  \text{load } R1, b \\
  \text{add } R0, Rb \\
  \text{add } R0, R1 \\
  \text{store } R0, T \\
  \text{store } R0, T
  \]

- Heuristic solutions: similar to the ones used for the swapping problem.
Machine-independent issues

- Techniques.
  - Analysis of dependence graphs.
  - Analysis of basic blocks and flow graphs.
  - Semantics-preserving transformations.
  - Algebraic transformations.
Dependence graphs

**Issues:**
- In an expression, assume its dependence graph is given.
- We can evaluate this expression using any topological ordering.
- There are many legal topological orderings.
- Pick one to increase its efficiency.

**Example:**

<table>
<thead>
<tr>
<th>order#1</th>
<th>reg#</th>
<th>order#2</th>
<th>reg#</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2</td>
<td>1</td>
<td>E6</td>
<td>1</td>
</tr>
<tr>
<td>E3</td>
<td>2</td>
<td>E5</td>
<td>2</td>
</tr>
<tr>
<td>E5</td>
<td>3</td>
<td>E4</td>
<td>1</td>
</tr>
<tr>
<td>E6</td>
<td>4</td>
<td>E3</td>
<td>2</td>
</tr>
<tr>
<td>E4</td>
<td>3</td>
<td>E1</td>
<td>1</td>
</tr>
<tr>
<td>E1</td>
<td>2</td>
<td>E2</td>
<td>2</td>
</tr>
<tr>
<td>E0</td>
<td>1</td>
<td>E0</td>
<td>1</td>
</tr>
</tbody>
</table>

On a machine with only 2 free registers, some of the intermediate results in order#1 must be stored in the temporary space.
- STORE/LOAD takes time.
Basic blocks and flow graphs

- **Basic block**: a sequence of code such that
  - jump statements, if any, are at the end of the sequence;
  - codes in other basic block can only jump to the beginning of this sequence, but not in the middle.
  - Example:
    - \( t_1 := a \ast a \)
    - \( t_2 := a \ast b \)
    - \( t_3 := 2 \ast t_2 \)
    - `goto outer`
  - Single entry, single exit.

- **Flow graph**: Using a flow chart-like graph to represent a program where nodes are basic blocks and edges are flow of control.
How to find basic blocks

- How to find leaders, which are the first statements of basic blocks?
  - The first statement of a program is a leader.
  - For each conditional and unconditional goto,
    - *its target is a leader;*
    - *its next statement is also a leader.*

- Using leaders to partition the program into basic blocks.

- Ideas for optimization:
  - Two basic blocks are equivalent if they compute the same expression.
  - Use transformation techniques below to perform machine-independent optimization.
Finding basic blocks — examples

- Example: Three-address code for computing the dot product of two vectors $a$ and $b$.
  
  - $prod := 0$
  - $i := 1$
  - $loop$:
    - $t_1 := 4 \times i$
    - $t_2 := a[t_1]$
    - $t_3 := 4 \times i$
    - $t_4 := b[t_3]$
    - $t_5 := t_2 \times t_4$
    - $t_6 := prod + t_5$
    - $prod := t_6$
    - $t_7 := i + 1$
    - $i := t_7$
    - $if \ i \leq 20 \ goto \ loop$
    - $\ldots$

- There are three blocks in the above example.
DAG representation of a basic block

- **Inside a basic block:**
  - Expressions can be expressed using a DAG that is similar to the idea of a dependence graph.
  - Graph might not be connected.

- **Example:**
  (1) \( t_1 := 4 \times i \)
  (2) \( t_2 := a[t_1] \)
  (3) \( t_3 := 4 \times i \)
  (4) \( t_4 := b[t_3] \)
  (5) \( t_5 := t_2 \times t_4 \)
  (6) \( t_6 := \text{prod} + t_5 \)
  (7) \( \text{prod} := t_6 \)
  (8) \( t_7 := i + 1 \)
  (9) \( i := t_7 \)
  (10) *if* \( i \leq 20\) *goto* (1)
Semantics-preserving transformations (1/3)

- Techniques: using the information contained in the flow graph and DAG representation of basic blocks to do optimization.
  - Common sub-expression elimination.
  - Dead-code elimination: remove unreachable codes.
  - Remove redundant codes such as loads and stores.
    - \( MOV \ R_0, a \)
    - \( MOV \ a, R_0 \)
  - Code motion.
    - Find loop-invariants inside a loop.
    - Obtain the values of loop-invariants outside the loop.
    - Example:
      \[
      t = limit - 2 \\
      \text{while}(i \leq limit - 2) \\
      \ldots
      \]
  - Renaming temporary variables: better usage of registers and avoiding using unneeded temporary variables.
More techniques:

- Copy propagation:
  - De-reference a chain of variable copies.
  - Example:
    ```
    a = x;
    y = a;
    b = y;
    ```

- Flow of control simplification:
  - De-reference a chain of goto’s.
  - Example:
    ```
    goto L1
    ...
    L1: goto L2
    ```
Interchange of two independent adjacent statements, which might be useful in discovering the above transformations.

- Same expressions that are too far away to store $E_1$ into a register.

  $t_1 := E_1$
  $t_2 := \text{const}$
  ... // value of $t_n$ is not used
  $t_n := E_1$

  $t_1 := E_1$
  $t_n := E_1$ // swap $t_2$ and $t_n$
  $t_2 := \text{const}$

  **Example:**
  In the example above, we can swap $t_2$ and $t_n$ since there is no dependence between $t_2$ and $t_n$.
  After the swapping, we can use the register storing $E_1$ twice.

- Note: The order of dependence cannot be altered after the exchange.

  $t_1 := E_1$
  $t_2 := t_1 + t_n$ // cannot swap $t_2$ and $t_n$
  $t_n := E_1$

  **Example:**
  In the example above, we cannot swap $t_2$ and $t_n$ because $t_2$ needs to be executed before $t_n$. 
### Algebraic transformations

- **Algebraic identities:**
  - $x + 0 \equiv 0 + x \equiv x$
  - $x - 0 \equiv x$
  - $x * 1 \equiv 1 * x \equiv x$
  - $x / 1 \equiv x$

- **Reduction in strength:**
  - $x^2 \equiv x * x$
  - $2.0 * x \equiv x + x$
  - $x / 2 \equiv x * 0.5$

- **Constant folding:**
  - $2 * 3.14 \equiv 6.28$

- **Standard representation for subexpression by commutativity and associativity:**
  - $n * m \equiv m * n.$
  - $b < a \equiv a > b.$
Correctness after optimization

- When side effects are expected, different evaluation orders may produce different results for expressions.

- Assume $E_5$ is a procedure call with the side effect of changing some values in $E_6$.
- $LL$ and $LR$ parsing produce different results.

- Watch out precisions when doing algebraic transformations.
- if $(x = 321.00000123456789 - 321.00000123456788) > 0$ then ... 

- Need to make sure code before and after optimization produce the same result.
- Complications arise when debugger is involved.