Optimization

ASU Textbook Chapter 9

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.

- 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 block of memory space.*
  - Pick a fastest instruction combination for a certain target machine.
Register allocation:
- C language allows the user to manage a pool of registers.
- Some languages leave the task to the compiler.
- Idea: save mostly used intermediate results in a register. However, finding an optimal solution for using a limited set of registers is NP-hard.
- Example:
  ```
  t := a + b
  load R0,a
  load R1,b
  add R0,b
  add R0,R1
  store R0,T
  store R0,T
  ```
- Heuristic solutions: similar to the ones used for the swapping problem.
Machine-independent issues

- Dependence graphs.
- Basic blocks and flow graphs.
- Structure-preserving transformations.
- Algebraic transformations.
- Peephole optimization.
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

- **Assumption:** the input is an intermediate code program.
- **Basic block:** a sequence of intermediate 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:
    - \( t_1 := a * a \)
    - \( t_2 := a * b \)
    - \( t_3 := 2 * t_2 \)
    - \( goto \ \text{outer} \)

- **Flow graph:** represent the program using a flow chart-like graph where nodes are basic blocks and edges are flow of control.

---

Compiler notes #8, Tsan-sheng Hsu, IIS
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 all 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 expressions.
  - 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 $b$.
  - $\textit{prod} := 0$
  - $i := 1$
  - $\textbf{loop: } t_1 := 4 \times i$
  - $t_2 := a[t_1]$
  - $t_3 := 4 \times i$
  - $t_4 := b[t_3]$
  - $t_5 := t_2 \times t_4$
  - $t_6 := \textit{prod} + t_5$
  - $\textit{prod} := t_6$
  - $t_7 := i + 1$
  - $i := t_7$
  - $\textit{if } i \leq 20 \textit{ goto loop}$
  - $\ldots$

- 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. \( t_6 := prod + t_5 \)
  7. \( prod := t_6 \)
  8. \( t_7 := i + 1 \)
  9. \( i := t_7 \)
  10. if \( i \leq 20 \) goto (1)
Structure-preserving transformations

- 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.
- Renaming temporary variables: better usage of registers and avoiding using unneeded temporary variables.
- Interchange of two independent adjacent statements, which might be useful in discovering the above three transformations.
  - Same expressions that are too far away to store $E_1$ into a register.
    \[
    \begin{align*}
    t_1 & := E_1 \\
    t_2 & := \text{const} \\
    \ldots \\
    t_n & := E_1 \\
    \end{align*}
    \]
  - Example: \[
  \begin{align*}
  t_2 & := a + b \\
  \ldots \\
  t_n & := t_1 + c \\
  \end{align*}
  \]
  - The order of dependence cannot be altered after the exchange.
    \[
    \begin{align*}
    t_2 & := a + b \\
    \ldots \\
    t_n & := t_1 + c \\
    \end{align*}
    \]
Algebraic transformations

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

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

- **Constant folding:**
  - $2 \times 3.14 == 6.28$

- **Standard representation for subexpression by commutativity and associativity:**
  - $n \times m == m \times n.$
  - $b < a == a > b.$
Peephole optimization (1/2)

- **Idea:**

  - 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.
  - Complexity depends on the “window size”.

- **Techniques: remove redundant codes.**

  - Redundant loads and stores.
    - $\text{MOV } R_0, a$
    - $\text{MOV } a, R_0$
  
  - Unreachable codes.
    - An unlabeled instruction immediately following an unconditional jump may be removed.
    - If statements based on constants: If debug then · · ·.
More techniques:

- Flow of control optimization:
  
  ```
  goto L1
  ...
  goto L2
  ...
  L1: goto L2
  L1: goto L2
  ```

- Algebraic simplification.
- Use of special machine idioms.
- Better usage of registers.
- Loop unwrapping.
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 simplification.
  - 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 also involved.