Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel
Code Optimization
Code Optimization

- What should we optimize?
  - improve running time
  - decrease space requirements
  - decrease power consumption

- Why does optimization work?
  - remove redundancies
  - no need for full generality
    - more specific instances of abstract constructs
  - leverage knowledge of target machine
    - pipelining, runtime of instructions, …
Program Analysis

• Scope of program analysis
  – within a basic block (local)
  – within a method (global or intra-procedural)
  – across methods (whole-program or inter-procedural)

• Analysis
  – control flow graph
    • dominators, loops, etc.
  – dataflow analysis
    • flow of values
  – static-single-assignment
    • transform programs such that each variable has a unique definition
  – alias analysis
    • pointer memory usage
Optimization Overview

• Classes of optimizations
  – machine independent or dependent

• Produce faster code
  – eliminate redundant (or useless) computation
    • common (sub)-expression elimination
    • constant folding
    • dead code elimination
  – move code
    • loop transformations
  – specialize code
  – instruction selection and scheduling
  – register allocation
Eliminate Redundant Computation

Original Block

\[
\begin{align*}
    a & \leftarrow b + c \\
    b & \leftarrow a - d \\
    c & \leftarrow b + c \\
    d & \leftarrow a - d
\end{align*}
\]

Rewritten Block

\[
\begin{align*}
    a & \leftarrow b + c \\
    b & \leftarrow a - d \\
    c & \leftarrow b + c \\
    d & \leftarrow b
\end{align*}
\]
Local Value Numbering

• Basic idea
  – assigns a distinct number to each value that the block computes
  – choose the numbers so that two expressions, e1 and e2, have the same value number if and only if e1 and e2 have provably equal values for all possible operands of the expressions

\[
\begin{align*}
a^2 &\leftarrow b^0 + c^1 \\
b^4 &\leftarrow a^2 - d^3 \\
c^5 &\leftarrow b^4 + c^1 \\
d^4 &\leftarrow a^2 - d^3
\end{align*}
\]

• Is value “4” still in the hash table?
• Yes, and it is associated with “b”
• Thus, can replace last operation with copy from “b”
for \( i \leftarrow 0 \) to \( n - 1 \), where the block has \( n \) operations \( "T_i \leftarrow L_i \; Op_i \; R_i" \)

1. get the value numbers for \( L_i \) and \( R_i \)

2. construct a hash key from \( Op_i \) and the value numbers for \( L_i \) and \( R_i \)

3. if the hash key is already present in the table then
   replace operation \( i \) with a copy of the value into \( T_i \) and
   associate the value number with \( T_i \)
else
   insert a new value number into the table at the hash key location
   record that new value number for \( T_i \)
Local Value Numbering

- **Extended LVN algorithm**
  - add support for commutative operations
  - add support for constant folding
  - add support for algebraic identities

- **Algebraic identities**
  - multiply variable with 0 or 1
  - add or subtract 0 from a variable
  - xor variable with itself
  - more possibilities …
Finding Uninitialized Variables

- Simple example of global data flow analysis
  - similar techniques used for other applications (e.g., finding unused/dead code)

- Approach based on liveness analysis
  - variable $v$ is live at point $p$ if and only if there exists a path in the CFG from $p$ to a use of $v$ along which $v$ is not redefined

- LiveOut($B$)
  - set that contains all the variables that are live on exit from block $B$
  - Given a LiveOut set for the CFG entry node $n_0$, each variable in LiveOut($n_0$) has a potentially uninitialized use
Finding Uninitialized Variables

- Computing LiveOut set for block B
  - use the LiveOut sets of $B$’s successors in the CFG
  - use two sets $UEVar(B)$ and $VarKill(B)$ that encode facts how the code in $B$ manipulates variables

- $UEVar(B)$ - Upward Exposed Variable
  - this set contains all the variables that are used in $B$ (without being defined before their uses)

- $VarKill(B)$
  - this set contains all the variables that are defined in $B$

- Since $LiveOut(B)$ depends on $LiveOut$ of other blocks that it is connected to, we can use an iterative fixed-point method
Finding Uninitialized Variables

LiveOut(n) for a block n based on successor nodes m

\[
\text{LiveOut}(n) = \bigcup_{m \in \text{succ}(n)} (\text{UEVAR}(m) \cup (\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}))
\]

Variable v is live on entry to m under one of two conditions:
  – it can be referenced in m before it is redefined in m
  – it can be live on exit from m and pass unscathed through m because m does not redefine it
Finding Uninitialized Variables

• First, compute UEVar and VarKill for each block

• Second, apply iterative dataflow analysis

```c
// assume CFG has N blocks
// numbered 0 to N-1
for i ← 0 to N-1
    LiveOut(i) ← Ø

changed ← true
while (changed)
    changed ← false
    for i ← 0 to N-1
        recompute LiveOut(i)
        if LiveOut(i) changed then
            changed ← true
```
Finding Uninitialized Variables

\begin{align*}
B_0 & \quad i \leftarrow 1 \\
B_1 & \quad \text{(test on i)} \\
B_2 & \quad s \leftarrow 0 \\
B_3 & \quad s \leftarrow s + i \\
& \quad i \leftarrow i + 1 \\
& \quad \text{(test on i)} \\
B_4 & \quad \text{print } s
\end{align*}

\begin{tabular}{|c|c|c|}
\hline
\textbf{UEVAR} & \textbf{VARKILL} \\
\hline
$B_0$ & $\emptyset$ & $\{i\}$ \\
$B_1$ & $\{i\}$ & $\emptyset$ \\
$B_2$ & $\emptyset$ & $\{s\}$ \\
$B_3$ & $\{s, i\}$ & $\{s, i\}$ \\
$B_4$ & $\{s\}$ & $\emptyset$ \\
\hline
\end{tabular}
Finding Uninitialized Variables

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$B_0$</th>
<th>$B_1$</th>
<th>$B_2$</th>
<th>$B_3$</th>
<th>$B_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>1</td>
<td>${i}$</td>
<td>${s,i}$</td>
<td>${s,i}$</td>
<td>${s,i}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>${s,i}$</td>
<td>${s,i}$</td>
<td>${s,i}$</td>
<td>${s,i}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>3</td>
<td>${s,i}$</td>
<td>${s,i}$</td>
<td>${s,i}$</td>
<td>${s,i}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Moving Code

• Loop unrolling
  – take the body of a loop and make N consecutive copies
  – saves overhead of jumping back to loop head and evaluate loop condition
  – need to be careful to make sure that N copies of loop body need to be executed
  – short prologue loop that peels off enough iterations to ensure that the unrolled loop processes an integral multiple of N iterations

• Invariant code moving
  – invariant computations do not change with each loop iteration
  – compute value once outside of the loop, then use result inside loop body
Moving Code

- Code placement
  - change code layout to reduce number of jumps
  - convert frequently-taken edges into fall through operations
Moving Code

• Function inlining
  – procedure linkage creates overhead
  – function body might be very small (e.g., string copy)
  – copy function body into caller, save overhead
Instruction Selection

- **Peephole optimization**
  - use a small sliding window over sequence of instructions
  - replace individual instructions with faster alternatives
  - replace common sequences with faster alternatives

- **Individual instructions**
  - use shift instead of multiply (by power of 2)
  - use address computation logic instead of arithmetic
    
    lea (%rdi,%rdi), %eax
    instead of
    
    shl $0x1, %edi
    mov %edi, %eax
Instruction Selection

• **Store followed by load**

\[
\begin{align*}
\text{storeAI} & \quad r_1 \quad \Rightarrow \quad r_{\text{arp}}, 8 \\
\text{loadAI} & \quad r_{\text{arp}}, 8 \quad \Rightarrow \quad r_{15}
\end{align*}
\]

\[
\Rightarrow
\begin{align*}
\text{storeAI} & \quad r_1 \quad \Rightarrow \quad r_{\text{arp}}, 8 \\
\text{i2i} & \quad r_1 \quad \Rightarrow \quad r_{15}
\end{align*}
\]

• **Double jump**

\[
\begin{align*}
\text{jumpI} & \quad \rightarrow \quad l_{10} \\
l_{10}: \quad \text{jumpI} & \quad \rightarrow \quad l_{11}
\end{align*}
\]

\[
\Rightarrow
\begin{align*}
\text{jumpI} & \quad \rightarrow \quad l_{11} \\
l_{10}: \quad \text{jumpI} & \quad \rightarrow \quad l_{11}
\end{align*}
\]

• **More complex algorithms possible, which work on AST tree patterns**
Instruction Scheduling

- Exploit multiple functional units of CPU
  - make sure that all units are busy at the same time
    - integer and floating point units
    - pipeline units

- Move *independent* instructions around

- Example
  - load/store = 3 cycles, multiply = 2 cycles, rest = 1 cycle
    \[ a \leftarrow a \times 2 \times b \times c \times d \]
Instruction Scheduling

(a) Example Code

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>loadAI</td>
<td>r_{arp},@a</td>
<td>r_1</td>
</tr>
<tr>
<td>b</td>
<td>add</td>
<td>r_1, r_1</td>
<td>r_1</td>
</tr>
<tr>
<td>c</td>
<td>loadAI</td>
<td>r_{arp},@b</td>
<td>r_2</td>
</tr>
<tr>
<td>d</td>
<td>mult</td>
<td>r_1, r_2</td>
<td>r_1</td>
</tr>
<tr>
<td>e</td>
<td>loadAI</td>
<td>r_{arp},@c</td>
<td>r_3</td>
</tr>
<tr>
<td>f</td>
<td>mult</td>
<td>r_1, r_2</td>
<td>r_1</td>
</tr>
<tr>
<td>g</td>
<td>loadAI</td>
<td>r_{arp},@d</td>
<td>r_2</td>
</tr>
<tr>
<td>h</td>
<td>mult</td>
<td>r_1, r_2</td>
<td>r_1</td>
</tr>
<tr>
<td>i</td>
<td>storeAI</td>
<td>r_1</td>
<td>r_{arp},@a</td>
</tr>
</tbody>
</table>

(b) Its Dependence Graph

\[ a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i \]
### Instruction Scheduling

#### (a) Original Code

<table>
<thead>
<tr>
<th>Start</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loadAI rarp, @a ⇒ r₁</td>
</tr>
<tr>
<td>4</td>
<td>add r₁, r₁ ⇒ r₁</td>
</tr>
<tr>
<td>5</td>
<td>loadAI rarp, @b ⇒ r₂</td>
</tr>
<tr>
<td>8</td>
<td>mult r₁, r₂ ⇒ r₁</td>
</tr>
<tr>
<td>10</td>
<td>loadAI rarp, @c ⇒ r₂</td>
</tr>
<tr>
<td>13</td>
<td>mult r₁, r₂ ⇒ r₁</td>
</tr>
<tr>
<td>15</td>
<td>loadAI rarp, @d ⇒ r₂</td>
</tr>
<tr>
<td>18</td>
<td>mult r₁, r₂ ⇒ r₁</td>
</tr>
<tr>
<td>20</td>
<td>storeAI r₁ ⇒ rarp, @a</td>
</tr>
</tbody>
</table>

#### (b) Scheduled Code

<table>
<thead>
<tr>
<th>Start</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loadAI rarp, @a ⇒ r₁</td>
</tr>
<tr>
<td>2</td>
<td>loadAI rarp, @b ⇒ r₂</td>
</tr>
<tr>
<td>3</td>
<td>loadAI rarp, @c ⇒ r₃</td>
</tr>
<tr>
<td>4</td>
<td>add r₁, r₁ ⇒ r₁</td>
</tr>
<tr>
<td>5</td>
<td>mult r₁, r₂ ⇒ r₁</td>
</tr>
<tr>
<td>6</td>
<td>loadAI rarp, @d ⇒ r₂</td>
</tr>
<tr>
<td>7</td>
<td>mult r₁, r₃ ⇒ r₁</td>
</tr>
<tr>
<td>9</td>
<td>mult r₁, r₂ ⇒ r₁</td>
</tr>
<tr>
<td>11</td>
<td>storeAI r₁ ⇒ rarp, @a</td>
</tr>
</tbody>
</table>