CS577 Modern Language Processors
Spring 2008
Lecture 4
Generating Better Code

What does a conventional compiler do to improve quality of generated code?

- Eliminate redundant computation
- Move computations so they are executed less often
- Inline small functions (a double win)
- Choose data layouts wisely
- Allocate register resources wisely
- Schedule instructions wisely

Hope to get modest constant-time improvement (x2 is excellent).

Actual improvement depends on source language and target architecture.
SOME IMPORTANT CLASSIC OPTIMIZATIONS

- Constant propagation
- Constant folding
- Strength reduction
- Dead code elimination
- Common subexpression elimination
- Invariant hoisting
- Inlining

Most of these are best done on malleable register-based IR.

Some key ideas: basic blocks, control-flow graphs, static single assignment, dominators, dataflow analysis.
Consider this example:

```java
static void foo(int x, int y, int u, int v) {
    int w = (x + y) + (u - v);
    u = x + y;
    x = u - v;
}
```
SIMPLE REDUNDANCY ELIMINATION

Corresponding bytecode and possible compiled output (using symbolic names):

```plaintext
load x
load y
add               add rx,ry,r0
load u
load v
sub               sub ru,rv,r1
add
store w           add r0,r1,rw
load x
load y
add
store u           add rx,ry,ru
load u
load v
sub
store x           sub ru,rv,rx
```

Where are the redundant computations?
Intermediate Representation (IR) for Optimization

Very common to use a somewhat abstract, register-based 3-address code.

Assume an infinite number of temporary registers (also that local variables and arguments are already in registers to start with).

Instruction set:

\[
\begin{align*}
& a \leftarrow b \ bop \ c & \text{Binary operation (for any binary operator } bop) \\
& a \leftarrow b & \text{Move} \\
& a \leftarrow M[b] & \text{Memory fetch (from address } b) \\
& M[a] \leftarrow b & \text{Memory store (to address } a) \\
& L: & \text{Label} \\
& \text{goto } L & \text{Unconditional branch} \\
& \text{if } a \ \text{relop } b \ \text{goto } L & \text{Conditional branch (for any relational operator } \text{relop}) \\
& a \leftarrow f(a1, \ldots, an) & \text{Function call (where } f \text{ is fixed or computed)}
\end{align*}
\]

How does the “level” of this language compare to JVM bytecodes?
A simple approach to common-subexpression elimination that works on straight-line code.

- Process each 3-addr instruction in order.
- Maintain a mapping from identifiers (x) and binop expressions (left,op,right) to value numbers.
- Whenever an entry exists already, rewrite the instruction to use it.

(Alternatively, could build DAG showing relationships between entries.)

First, our example in the new IR:

\[
\begin{align*}
g & \leftarrow x + y \\
h & \leftarrow u - v \\
w & \leftarrow g + h \\
u & \leftarrow g + h \\
x & \leftarrow u - v
\end{align*}
\]
### Local Value Numbering (2)

<table>
<thead>
<tr>
<th>Initial code</th>
<th>Final code</th>
<th>Mapping entries</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>g &lt;- x + y</code></td>
<td><code>g &lt;- x + y</code></td>
<td><code>x -&gt; 1</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>1:x</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>y -&gt; 2</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>2:y</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>(1,+,2) -&gt; 3</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>g -&gt; 3</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>3:g</code></td>
</tr>
<tr>
<td><code>h &lt;- u - v</code></td>
<td><code>h &lt;- u - v</code></td>
<td><code>u -&gt; 4</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>4:u</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>v -&gt; 5</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>5:v</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>(4,−,5) -&gt; 6</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>h -&gt; 6</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>6:h</code></td>
</tr>
<tr>
<td><code>w &lt;- g + h</code></td>
<td><code>w &lt;- g + h</code></td>
<td><code>(3,+ ,6) -&gt; 7</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>w -&gt; 7</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>7:w</code></td>
</tr>
<tr>
<td><code>u &lt;- x + y</code></td>
<td><code>u &lt;- g</code></td>
<td><code>u -&gt; 3</code></td>
</tr>
<tr>
<td><code>x &lt;- u - v</code></td>
<td><code>x &lt;- u - v</code></td>
<td><code>(3,−,5) -&gt; 8</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>x -&gt; 8</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>8:x</code></td>
</tr>
</tbody>
</table>

Now do copy propagation to get rid of `u <- g`. 
ISSUES WITH NAMING

- If there are (re-)assignments, a name is not the same thing as a value!

- Value numbering successfully distinguishes between different values with the same name (e.g., \( u \) in the example).

- But can still lose access to a value if its name gets overwritten.

Modified Example:

<table>
<thead>
<tr>
<th>Initial code</th>
<th>Final code</th>
<th>Mapping entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z \leftarrow x + y )</td>
<td>( z \leftarrow x + y )</td>
<td>( x \rightarrow 1 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( y \rightarrow 2 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( (1,+,2) \rightarrow 3 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( z \rightarrow 3 )</td>
</tr>
<tr>
<td>( h \leftarrow u - v )</td>
<td>( h \leftarrow u - v )</td>
<td>( u \rightarrow 4 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( v \rightarrow 5 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( (4,-,5) \rightarrow 6 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( h \rightarrow 6 )</td>
</tr>
<tr>
<td>( z \leftarrow z + h )</td>
<td>( z \leftarrow z + h )</td>
<td>( (3,+,6) \rightarrow 7 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( z \rightarrow 7 )</td>
</tr>
<tr>
<td>( u \leftarrow x + y )</td>
<td>( u \leftarrow ?? )</td>
<td>3:?? 7:z</td>
</tr>
</tbody>
</table>
# Using Unique Names

Idea: rename variables so that every assignment gets a unique name.

<table>
<thead>
<tr>
<th>Initial code</th>
<th>Renamed code</th>
<th>Final code</th>
<th>Mapping entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>z &lt;- x + y</td>
<td>z0 &lt;- x0 + y0</td>
<td>z0 &lt;- x0 + y0</td>
<td>x0 -&gt; 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y0 -&gt; 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(1,+,2) -&gt; 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>z0 -&gt; 3</td>
</tr>
<tr>
<td>h &lt;- u - v</td>
<td>h0 &lt;- u0 - v0</td>
<td>h0 &lt;- u0 - v0</td>
<td>u0 -&gt; 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>v0 -&gt; 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(4,-,5) -&gt; 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>h0 -&gt; 6</td>
</tr>
<tr>
<td>z &lt;- z + h</td>
<td>z1 &lt;- z0 + h0</td>
<td>z1 &lt;- z0 + h0</td>
<td>(3,+6) -&gt; 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>z1 -&gt; 7</td>
</tr>
<tr>
<td>u &lt;- x + y</td>
<td>u1 &lt;- x0 + y0</td>
<td>u1 &lt;- z0</td>
<td>u1 -&gt; 3</td>
</tr>
</tbody>
</table>
What about Control Flow Joins?

How can we generate unique names when control can reach uses in multiple ways?

Initial code

\[
\begin{align*}
&\text{if } a > 0 \text{ goto } L1 \\
&b = x + y \\
&\text{goto } L2 \\
\text{L1: } &b = x - y \\
\text{L2: } &c = a + b
\end{align*}
\]

Renamed code

\[
\begin{align*}
&\text{if } a0 > 0 \text{ goto } L1 \\
&b0 = x0 + y0 \\
&\text{goto } L2 \\
\text{L1: } &b1 = x0 - y0 \\
\text{L2: } &c0 = a0 + ??
\end{align*}
\]

Will consider soon in context of SSA form.
**Simple Control Flow Graphs (CFG’s)**

- One node per instruction (and perhaps for procedure entry and exit)
- Edge from A to B if control **might** flow directly from A to B.

Example (Appel, “Modern Compiler Implementation,” Ex. 17.3)

```plaintext
1  a <- 5
2  c <- 1
3  L1:   if c > a goto L2
4  c <- c + c
5  goto L1
6  L2:   a <- c - a
7  c <- 0
```

![CFG Diagram](image_url)
Often useful to factor a program into its **basic blocks**, which are sequences of consecutive instructions in which control always enters at the top and exits from the bottom.

```
1   a <- 5
2   c <- 1
3   L1: if c > a goto L2
4   c <- c + c
5   goto L1
6   L2: a <- c - a
7   c <- 0
```

Often use basic blocks as nodes in CFG.
Do analysis over paths in **extended** basic blocks.

(An EBB has one entry, but can have multiple exits. It forms a subtree of the CFG; all the blocks in the EBB except perhaps the root have a unique predecessor inside the EBB).
To define dominators, assume that CFG has a distinguished start node $S$, and has no disconnected subgraphs (nodes unreachable from $S$).

Then we say node $d$ dominates node $n$ if all paths from $S$ to $n$ include $d$. (In particular, every node dominates itself.)

Fact: $d$ dominates $n$ iff $d = n$ or $d$ dominates all predecessors of $n$.

So can define the set $D(n)$ of nodes that dominate $n$ as follows:

- $D(S) = \{S\}$
- $D(n) = \{n\} \cup (\bigcap_{p \in \text{pred}(n)} D(p))$

where $\text{pred}(n) = \text{set of predecessors of } n$ in CFG.
The **immediate dominator** of \( n \), \( idom(n) \), is defined thus:

- \( idom(n) \) dominates \( n \)
- \( idom(n) \) is not \( n \)
- \( idom(n) \) does not dominate any other dominator of \( n \) (except \( n \) itself)

Fact: every node (except \( S \)) has a unique immediate dominator

Hence the immediate dominator relation defined a tree, called the **dominator tree**, whose nodes are the nodes of the CFG, where the parent of a node is its immediate dominator.

Have \( D(n) = \{n\} \cup \) (ancestors of \( n \) in dominator tree)

(Nontrivial) Fact: The dominator tree of a CFG can be computed in almost-linear time.
DOMINATOR TREE EXAMPLE

```
ENTER

k <- 0
i <- 1
j <- 2

k > 0?
i <= N?

i <- 0
i <- i + 1
i <- i + 1

EXIT
```
Do analysis over paths in dominator tree.

\[ w_1 \leftarrow a_1 \cdot b_1 \]
\[ x_2 \leftarrow a_1 - b_1 \]
\[ y_1 \leftarrow a_1 - b_1 \]
\[ w_1 \leftarrow a_1 \cdot b_1 \]
\[ z_1 \leftarrow a_1 + b_1 \]
\[ x_1 \leftarrow a_1 + b_1 \]
\[ y_1 \leftarrow a_1 - b_1 \]
\[ z_2 \leftarrow a_1 \cdot b_1 \]
STATIC SINGLE ASSIGNMENT (SSA) FORM

- Every variable has just one (static) definition (though defining instruction may be executed many times)

- For straightline code, this is just what we did for value numbering:

  Original code  
  \[\begin{align*}
  v & \leftarrow 4 \\
  w & \leftarrow v + 5 \\
  v & \leftarrow 6 \\
  w & \leftarrow v + 7
  \end{align*}\]

  SSA Code  
  \[\begin{align*}
  v1 & \leftarrow 4 \\
  w1 & \leftarrow v1 + 5 \\
  v2 & \leftarrow 6 \\
  w2 & \leftarrow v2 + 6
  \end{align*}\]

- For general flow, must introduce \(\phi\)-nodes ("phi"-nodes). These are fictitious operations, (usually) not intended to have execution significance. To interpret them, must view code as CFG, with the in-edges to each node having a well-defined order.
SSA Example 1

Original CFG

SSA CFG

P?

v ← 4
v ← 5

w ← v + v

v1 ← 4
v2 ← 5

v3 ← phi(v1, v2)
w ← v3 + v3
SSA Example 2

Original CFG

- $i \leftarrow 0$
- $j \leftarrow 0$
- $i > N$ ?
- $i \leftarrow j + 1$
- $j \leftarrow j + i$

SSA CFG

- $i_1 \leftarrow 0$
- $j_1 \leftarrow 0$
- $i_2 = \text{phi}(i_1, i_3)$
- $j_2 = \text{phi}(j_1, j_3)$
- $i_2 > N$ ?
- $i_3 \leftarrow j_2 + 1$
- $j_3 \leftarrow j_2 + i_2$
- EXIT
Where should we put φ assignments, and for which variables?

Answer: there are many options, so long as single-assignment property is obeyed and each use of a variable in the original program has a corresponding uniquely-defined SSA variable.

Simplistic approach: put φ assignments in every join node, for every variable in scope. Much too expensive!

Suffices to put a φ assignment for x in join nodes that are not dominated by a single definition of x. (More later.)
Even with dominator-based VN, we cannot find redundant expressions computed on different paths.

An alternative approach is to compute available expressions.

For SSA graphs, an expression is available at node \( n \) if it is computed at least once on every path from \( S \) to \( n \).

If an expression is available at a node where it is being recomputed, it is possible to replace the recomputation by a variable representing the result of the previous computation.

This is a classic data flow analysis problem, specified by the following equations:

\[
\text{gen}(t \leftarrow \text{b \ bop c}) = \{\text{b \ bop c}\}
\]
\[
\text{gen}(\text{other instruction}) = \emptyset
\]
\[
in(n) = \bigcap_{p \in \text{pred}(n)} \text{out}(p)
\]
\[
\text{out}(n) = in(n) \cup \text{gen}(n)
\]

Here we want \( in(n) \), the set of expressions available on entry to \( n \).
AVAILABLE EXPRESSIONS EXAMPLE

\[ \text{gen}[1] = \{a+b, a-b, a \times b\} \]

\[ \text{gen}[2] = \{a-b, a/b\} \]

\[ \text{gen}[3] = \{a+b, a/b\} \]

\[ \text{gen}[4] = \{a \times b, a/b\} \]
Here’s the (unique) solution to the data flow equations.

\[
\begin{align*}
\text{in}[1] &= \{\} & \text{out}[1] &= \{a+b, a-b, a*b\} \\
\text{in}[2] &= \{a+b, a-b, a*b\} & \text{out}[2] &= \{a+b, a-b, a*b, a/b\} \\
\text{in}[3] &= \{a+b, a-b, a*b\} & \text{out}[3] &= \{a+b, a-b, a*b, a/b\} \\
\text{in}[4] &= \{a+b, a-b, a*b, a/b\} & \text{out}[4] &= \{a+b, a-b, a*b, a/b\}
\end{align*}
\]

So nothing needs to be recomputed in nodes 2, 3, or 4.
Another Available Expressions Example

Original code:

\[
\begin{align*}
    a & \leftarrow b + c \\
    g & \leftarrow x + y \\
    i & \leftarrow x - y \\
    L: \quad & r \leftarrow x + y \\
    s & \leftarrow x - y \\
    d & \leftarrow b + c \\
    x & \leftarrow x + 1 \\
    h & \leftarrow x + y \\
    \text{if } x < 10 \text{ goto } L
\end{align*}
\]
Here's a solution (the maximal one, which is what we want):

\[
\begin{align*}
in[1] &= \{\} & \text{out}[1] &= \{b_1+c_1, x_1+y_1, x_1-y_1\} \\
in[2] &= \{b_1+c_1, x_1+y_1, x_1-y_1\} & \text{out}[2] &= \{b_1+c_1, x_1+y_1, x_1-y_1, x_2+y_1, x_2-y_1, x_2+1, x_3+y_1\}
\end{align*}
\]

Using this, we can avoid recomputing \(b_1+c_1\) in block 2.

Standard available expressions algorithm doesn't let us avoid recomputing \(x_2+y_1\), but perhaps we could be clever and notice that because \(x_1+y_1\) and \(x_3+y_1\) are available into block 2 on paths 1 and 2, respectively, \(\phi(x_1, x_3)+y_1\) is available too.

By the way, here's another solution to the dataflow equations (a less useful one):

\[
\begin{align*}
in[1] &= \{\} & \text{out}[1] &= \{b_1+c_1, x_1+y_1, x_1-y_1\} \\
in[2] &= \{b_1+c_1\} & \text{out}[2] &= \{x_2+y_1, x_2-y_1, b_1+c_1, x_2+1, x_3+y_1\}
\end{align*}
\]

Note the importance of taking an “optimistic” view of \(\text{in}[2]\).
SOLVING DATAFLOW EQUATIONS

Completely general method: Iteration to a fixed point.

For Available Expressions problem:

- can precompute gen set for each node
- start with the optimistic approximation that all the in and out sets are full (contain all possible expressions),
- on each iteration, recompute in and out using the most recent approximations we have for them (and gen)
- iterate until computed sets don’t change

This gives us a greatest fixed point, i.e., the largest sets that solve the equations. Note that if we started with empty sets, the in sets would not contain expressions that remain available after a loop iteration, due to the \( \cap \) operation in in []; this would be the least fixed point solution.
Let $A =$ set of all potentially interesting expressions, namely
\{b_1+c_1,x_1+y_1,x_1-y_1,x_2+y_1,x_2-y_1,x_2+1,x_3+y_1\}.

\begin{center}
\begin{tabular}{llllllll}
$n$ & pred$[n]$ & gen$[n]$ & & iteration 0 & iteration 1 & iteration 2 \\
\hline
1 & - & \{b_1+c_1,x_1+y_1,x_1-y_1\} & & & & \\
2 & \{1,2\} & \{x_2+y_1,x_2-y_1,b_1+c_1,x_2+1,x_3+y_1\} & & & & \\
\hline
\hline
\end{tabular}
\end{center}

Will see more interesting examples another time.
Can use dominator information to construct a “minimal” SSA graph.

Note the following dominance properties, which follow from the requirement that each variable is necessarily defined before it is used.

1. If \( x \) is used in a non-\( \phi \) statement in block \( n \), then the definition of \( x \) dominates \( n \).

2. If \( x \) is the \( i \)th argument of a \( \phi \)-function in CFG block \( n \), then the definition of \( x \) dominates the \( i \)th predecessor of \( n \).

We say \( x \) **strictly dominates** \( w \) if \( x \) dominates \( w \) but \( x \neq w \).

The **dominance frontier** of a definition \( x \), \( DF(x) \), is the set of nodes \( w \) such that \( x \) dominates an (immediate) predecessor of \( w \), but \( x \) does not strictly dominate \( w \).

Intuition: Any node in \( DF(x) \) is the join point of two disjoint paths from \( x \) and from the ENTRY node.

Can easily compute \( DF(x) \) from the dominator tree.
DOMINANCE FRONTIER EXAMPLE

k ← 0
i ← 1
j ← 2

k > 0?

EXIT

DF(n)
0  {}
1  {}
2  {2}
3  {2}
4  {}
5  {7}
6  {7}
7  {}

n

k ← 1
i ← i + 1

i ≤ N?

i ← 0

i ← i + 1

0

1

2

3

4

5

6

7

3

5

6

7
**Dominance Frontier Criterion**

If node $x$ defines $a$, any node in $DF(x)$ requires a $\phi$-function for $a$.

- Since such a $\phi$-function is itself a definition for $a$, we must (in general) iterate until there are no more $\phi$-functions to place.

In our example, must place $\phi$-nodes for $i$ and $k$ in node 2 and for $i$ in node 7.
EXAMPLE IN SSA FORM

0

ENTER

1

k1 ← 0
i1 ← 1
j1 ← 2

2

i2 = phi(i3,i1)
k2 = phi(k3,k1)
i2 ≤ N?

3

k3 ← 1
i3 ← i2 + 1

4

k2 > 0?

5

i4 ← 0

6

i5 ← i2 + 1

i6 = phi(i4,i5)

EXIT