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-factor 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.

Redundancy Elimination

Consider this example:

```c
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):

- 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
- 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:

- a <- b bop c: Binary operation (for any binary operator bop)
- a <- b: Move
- a <- M[b]: Memory fetch (from address b)
- M[a] <- b: Memory store (to address a)
- L: Label
- goto L: Unconditional branch
- if a relop b goto L: Conditional branch (for any relational operator relop)
- a <- f(a1,...,an): Function call (where f is fixed or computed)

How does the “level” of this language compare to JVM bytecodes?

**LOCAL VALUE NUMBERING**

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:

- g <- x + y
- h <- u - v
- w <- g + h
- u <- x + y
- x <- u - v

Initial code | Final code | Mapping entries
--- | --- | ---
```
g <- x + y
g <- x + y
```
```
x -> 1
```
```
y -> 2
```
```
(1,+),2 -> 3
```
```
g -> 3
```
```
3:g
```
```
h <- u - v
h <- u - v
```
```
u -> 4
```
```
v -> 5
```
```
(4,-),5 -> 6
```
```
h -> 6
```
```
6:h
```
```
w <- g + h
w <- g + h
```
```
(3,+),6 -> 7
```
```
w -> 7
```
```
7:w
```
```
u <- x + y
u <- g
```
```
u -> 3
```
```
x <- u - v
x <- u - v
```
```
(3,-),5 -> 8
```
```
x -> 8
```
```
8:x
```

Now do copy propagation & dead code elimination 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><code>z &lt;- x + y</code></td>
<td><code>z &lt;- x + y</code></td>
<td><code>x -&gt; 1 1:x</code></td>
</tr>
<tr>
<td><code>y -&gt; 2</code></td>
<td><code>2:y</code></td>
<td></td>
</tr>
<tr>
<td><code>(1,+,2) -&gt; 3</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>z -&gt; 3</code></td>
<td><code>3:z</code></td>
<td></td>
</tr>
<tr>
<td><code>h &lt;- u - v</code></td>
<td><code>h &lt;- u - v</code></td>
<td><code>u -&gt; 4 4:u</code></td>
</tr>
<tr>
<td><code>v -&gt; 5</code></td>
<td><code>5:v</code></td>
<td></td>
</tr>
<tr>
<td><code>(4,-,5) -&gt; 6</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>h -&gt; 6</code></td>
<td><code>6:h</code></td>
<td></td>
</tr>
<tr>
<td><code>z &lt;- z + h</code></td>
<td><code>z &lt;- z + h</code></td>
<td><code>(3,+,6) -&gt; 7</code></td>
</tr>
<tr>
<td><code>z -&gt; 7</code></td>
<td><code>7:z</code></td>
<td><code>3:??</code></td>
</tr>
<tr>
<td><code>u &lt;- x + y</code></td>
<td><code>u &lt;- ??</code></td>
<td></td>
</tr>
</tbody>
</table>

## Using Unique Names

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

### Initial code Renamed code Final code Mapping entries

- `z <- x + y` → `z0 <- x0 + y0` → `z0 <- x0 + y0` → `x0 -> 1 1:x0` → `y0 -> 2 2:y0` → `(1,+,2) -> 3 z0 -> 3 3:z0` → `h <- u - v` → `h0 <- u0 - v0` → `h0 <- u0 - v0` → `u0 -> 4 4:u0` → `v0 -> 5 5:v0` → `(4,-,5) -> 6 h0 -> 6 6:h0` → `z <- z + h` → `z1 <- z0 + h0` → `z1 <- z0 + h0` → `(3,+,6) -> 7 z1 -> 7 7:z1` → `u <- x + y` → `u1 <- x0 + y0` → `u1 <- z0` → `u1 -> 3` → `z1 -> 7 7:z1` → `3:??`

## What about Control Flow Joins?

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

### Initial code Renamed code

- `if a > 0 goto L1` → `if a0 > 0 goto L1`
- `b = x + y` → `b0 = x0 + y0`
- `goto L2` → `goto L2`
- `L1: b = x - y` → `L1: b1 = x0 - y0`
- `L2: c = a + b` → `L2: c0 = a0 + ??`

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)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a &lt;- 5</td>
</tr>
<tr>
<td>2</td>
<td>c &lt;- 1</td>
</tr>
<tr>
<td>3</td>
<td>L1: if c &gt; a goto L2</td>
</tr>
<tr>
<td>4</td>
<td>c &lt;- c + c</td>
</tr>
<tr>
<td>5</td>
<td>goto L1</td>
</tr>
<tr>
<td>6</td>
<td>L2: a &lt;- c - a</td>
</tr>
<tr>
<td>7</td>
<td>c &lt;- 0</td>
</tr>
</tbody>
</table>

`1 2 3 4 5 6 7`
BASIC BLOCKS

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.

SUPERLOCAL VALUE NUMBERING

Can do better by analyzing over paths in extended basic blocks.

```
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
```

DOMINATORS

We still aren’t taking full advantage of facts of the form “this instruction is certain to be executed before this other instruction.” Capture this idea using dominators.

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.

DOMINATOR TREE

The immediate dominator of \( n \), \( \text{idom}(n) \), is defined thus:

- \( \text{idom}(n) \) dominates \( n \)
- \( \text{idom}(n) \) is not \( n \)
- \( \text{idom}(n) \) 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 \( D(n) = \{ n \} \cup (\text{ancestors of } n \text{ in dominator tree}) \)

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

Do analysis over paths in dominator tree.

STATICTERNARY
BASED

DOMINATOR

V

ALUE

NUMBERING

Do analysis over paths in dominator tree.

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
  v <- 4
  w <- v + 5
  v <- 6
  w <- v + 7

  SSA Code
  v1 <- 4
  w1 <- v1 + 5
  v2 <- 6
  w2 <- v2 + 6

  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

v1 <- 4

v <- 5

v1 <- 5

w <- v + v

w <- v3 + v3

P?

v3 <- \phi(v1,v2)

v2 <- 5
Where should we put \( \phi \) 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 \( \phi \) assignments in every join node, for every variable in scope. Much too expensive!

Suffices to put a \( \phi \) assignment for \( x \) in join nodes that are not dominated by a single definition of \( x \).

SSA Graph Construction

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 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.

**Available Expressions**

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:

$$\text{gen}(t \leftarrow b \text{ bop } c) = \{b \text{ bop } c\}$$

$$\text{gen}(\text{other instruction}) = \emptyset$$

$$\text{in}(n) = \bigcap_{p \in \text{pred}(n)} \text{out}(p)$$

$$\text{out}(n) = \text{in}(n) \cup \text{gen}(n)$$

Here we want $\text{in}(n)$, the set of expressions available on entry to $n$. 

**Example in SSA Form**

```
 ENTER
0

1
k1 <- 0
i1 <- 1
j1 <- 2

2
i2 = phi(i3,i1)
k2 = phi(k3,k1)
i2 <= N? 4
k2 > 0? 5
```

```
3
k3 < 1
i3 < i2 + 1

4
i2 <= N? 4
k2 > 0? 5
```

```
5
4 <= 0
5 <= i2 + 1
```

```
6
i6 = phi(i4,i5)
```

```
7
EXIT
```

**Available Expressions Example**

```
 gen[1] = \{a\cdot b,a-b,a\cdot b\}
```

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

```
 gen[3] = \{a\cdot b,a/b\}
```

```
 gen[4] = \{a\cdot b,a/b\}
```

```
 z1 <- a + b
z2 <- a \cdot b
w2 <- a/b
```

```
 x1 <- a + b
y1 <- a - b
w1 <- a \cdot b
```

```
 r1 <- a/b
```

```
 x2 <- a - b
```
Here’s the (unique) solution to the data flow equations.

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

So nothing needs to be recomputed in nodes 2, 3, or 4.

Here’s another solution (the maximal one, which is what we want):

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

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):

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

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 \(\text{gen}\) set for each node
- start with the optimistic approximation that all the \(\text{in}\) and \(\text{out}\) sets are full (contain all possible expressions),
- on each iteration, recompute \(\text{in}\) and \(\text{out}\) using the most recent approximations we have for them (and \(\text{gen}\))
- 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 \(\text{in}\) sets would not contain expressions that remain available after a loop iteration, due to the \(\cap\) operation in \(\text{in}[\cdot]\); this would be the least fixed point solution.

### Another Available Expressions Example

Original code:

\[
\begin{align*}
a &\leftarrow b + c \\
g &\leftarrow x + y \\
i &\leftarrow x - y \\
L: \ r &\leftarrow x + y \\
s &\leftarrow x - y \\
d &\leftarrow b + c \\
&x &\leftarrow x + 1 \\
h &\leftarrow x + y \\
\text{if } x < 10 \text{ goto } L
\end{align*}
\]

\[
\begin{align*}
a_1 &\not\leftarrow b_1 + c_1 \\
g_1 &\not\leftarrow x_1 + y_1 \\
i_1 &\not\leftarrow x_1 - y_1 \\
\phi(x_1,x_3) &\not\leftarrow x_2 + 1 \\
r_1 &\not\leftarrow x_2 + y_1 \\
s_1 &\not\leftarrow x_2 - y_1 \\
d_1 &\not\leftarrow b_1 + c_1 \\
x_3 &\not\leftarrow x_2 + 1 \\
h_1 &\not\leftarrow x_3 + y_1 \\
x_3 &\not\geq 10 ?
\end{align*}
\]
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{align*}
n & \quad \text{pred}[n] \quad \text{gen}[n] \\
1 & \quad \emptyset \quad \{b_1+c_1,x_1+y_1,x_1-y_1\} \\
2 & \quad \{1,2\} \quad \{x_2+y_1,x_2-y_1,b_1+c_1,x_2+1,x_3+y_1\}
\end{align*}

Will see more interesting examples another time.