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.

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,rx
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:

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

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

**Local Value Numbering (2)**

- 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.)
**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 &lt;- x + y</td>
<td>z &lt;- x + y</td>
<td>x -&gt; 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y -&gt; 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1,+,2) -&gt; 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>z -&gt; 3</td>
</tr>
<tr>
<td>h &lt;- u - v</td>
<td>h &lt;- u - v</td>
<td>u -&gt; 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v -&gt; 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(4,-,5) -&gt; 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h -&gt; 6</td>
</tr>
<tr>
<td>z &lt;- z + h</td>
<td>z &lt;- z + h</td>
<td>(3,+,6) -&gt; 7</td>
</tr>
<tr>
<td>u &lt;- x + y</td>
<td>u &lt;- x + y</td>
<td>z -&gt; 7</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>1:x0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y0 -&gt; 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2:y0</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>4:u0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>v0 -&gt; 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5:v0</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></td>
<td></td>
<td></td>
<td>6:h0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(3,+,6) -&gt; 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>z1 -&gt; 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7:z1</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?

<table>
<thead>
<tr>
<th>Initial code</th>
<th>Renamed code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if a &gt; 0 goto L1</td>
<td>if a0 &gt; 0 goto L1</td>
</tr>
<tr>
<td>b = x + y</td>
<td>b0 = x0 + y0</td>
</tr>
<tr>
<td>goto L2</td>
<td>goto L2</td>
</tr>
<tr>
<td>L1: b = x - y</td>
<td>L1: b1 = x0 - y0</td>
</tr>
<tr>
<td>L2: c = a + b</td>
<td>L2: c0 = a0 + ??</td>
</tr>
</tbody>
</table>

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)

```
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
```

```
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.

**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.
**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:

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

- 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.
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 $x$. (More later.)

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:

$$
gen(t <- b \ bop \ c) = \{b \ bop \ c\}
\gen(other \ instruction) = \emptyset
\in(n) = \bigcap_{p \in \text{pred}(n)} \out(p)
\out(n) = \in(n) \cup \gen(n)
$$

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

Available Expressions Example

$\gen[1] = \{a+b,a-b,a\ast b\}$

$\gen[2] = \{a-b,a\ast b\}$

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

$\gen[4] = \{a\ast b,a/b\}$
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.

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

\[
\begin{align*}
\text{in}[1] &= \{\} & \text{out}[1] &= \{b1+c1,x1+y1,x1-y1\} \\
\text{in}[2] &= \{b1+c1,x1+y1,x1-y1\} & \text{out}[2] &= \{b1+c1,x1+y1,x1-y1, x2+y1,x2-y1,x2+1,x3+y1\}
\end{align*}
\]

Using this, we can avoid recomputing \(b1+c1\) in block 2.

Standard available expressions algorithm doesn’t let us avoid recomputing \(x2+y1\), but perhaps we could be clever and notice that because \(x1+y1\) and \(x3+y1\) are available into block 2 on paths 1 and 2, respectively, \(\phi(x1,x3)+y1\) is available too.

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

\[
\begin{align*}
\text{in}[1] &= \{\} & \text{out}[1] &= \{b1+c1,x1+y1,x1-y1\} \\
\text{in}[2] &= \{b1+c1\} & \text{out}[2] &= \{x2+y1,x2-y1,b1+c1,x2+1,x3+y1\}
\end{align*}
\]

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

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

\[
\begin{align*}
\text{in}[1] &= \{\} & \text{out}[1] &= \{b1+c1,x1+y1,x1-y1\} \\
\text{in}[2] &= \{b1+c1\} & \text{out}[2] &= \{x2+y1,x2-y1,b1+c1,x2+1,x3+y1\}
\end{align*}
\]

Using this, we can avoid recomputing \(b1+c1\) in block 2.

Standard available expressions algorithm doesn’t let us avoid recomputing \(x2+y1\), but perhaps we could be clever and notice that because \(x1+y1\) and \(x3+y1\) are available into block 2 on paths 1 and 2, respectively, \(\phi(x1,x3)+y1\) is available too.

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

\[
\begin{align*}
\text{in}[1] &= \{\} & \text{out}[1] &= \{b1+c1,x1+y1,x1-y1\} \\
\text{in}[2] &= \{b1+c1\} & \text{out}[2] &= \{x2+y1,x2-y1,b1+c1,x2+1,x3+y1\}
\end{align*}
\]

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

Here’s the (unique) solution to the data flow equations.

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

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

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

\[
\begin{align*}
in[1] &= \{} & out[1] &= \{b1+c1,x1+y1,x1-y1\} \\
in[2] &= \{b1+c1,x1+y1,x1-y1\} & out[2] &= \{b1+c1,x1+y1,x1-y1, x2+y1,x2-y1,x2+1,x3+y1\}
\end{align*}
\]

Using this, we can avoid recomputing \(b1+c1\) in block 2.

Standard available expressions algorithm doesn’t let us avoid recomputing \(x2+y1\), but perhaps we could be clever and notice that because \(x1+y1\) and \(x3+y1\) are available into block 2 on paths 1 and 2, respectively, \(\phi(x1,x3)+y1\) is available too.

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

\[
\begin{align*}
in[1] &= \{} & out[1] &= \{b1+c1,x1+y1,x1-y1\} \\
in[2] &= \{b1+c1\} & out[2] &= \{x2+y1,x2-y1,b1+c1,x2+1,x3+y1\}
\end{align*}
\]

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

Here’s the (unique) solution to the data flow equations.

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

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

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

\[
\begin{align*}
in[1] &= \{} & out[1] &= \{b1+c1,x1+y1,x1-y1\} \\
in[2] &= \{b1+c1,x1+y1,x1-y1\} & out[2] &= \{b1+c1,x1+y1,x1-y1, x2+y1,x2-y1,x2+1,x3+y1\}
\end{align*}
\]

Using this, we can avoid recomputing \(b1+c1\) in block 2.

Standard available expressions algorithm doesn’t let us avoid recomputing \(x2+y1\), but perhaps we could be clever and notice that because \(x1+y1\) and \(x3+y1\) are available into block 2 on paths 1 and 2, respectively, \(\phi(x1,x3)+y1\) is available too.

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

\[
\begin{align*}
in[1] &= \{} & out[1] &= \{b1+c1,x1+y1,x1-y1\} \\
in[2] &= \{b1+c1\} & out[2] &= \{x2+y1,x2-y1,b1+c1,x2+1,x3+y1\}
\end{align*}
\]

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

Here’s the (unique) solution to the data flow equations.

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

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

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

\[
\begin{align*}
in[1] &= \{} & out[1] &= \{b1+c1,x1+y1,x1-y1\} \\
in[2] &= \{b1+c1,x1+y1,x1-y1\} & out[2] &= \{b1+c1,x1+y1,x1-y1, x2+y1,x2-y1,x2+1,x3+y1\}
\end{align*}
\]

Using this, we can avoid recomputing \(b1+c1\) in block 2.

Standard available expressions algorithm doesn’t let us avoid recomputing \(x2+y1\), but perhaps we could be clever and notice that because \(x1+y1\) and \(x3+y1\) are available into block 2 on paths 1 and 2, respectively, \(\phi(x1,x3)+y1\) is available too.

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

\[
\begin{align*}
in[1] &= \{} & out[1] &= \{b1+c1,x1+y1,x1-y1\} \\
in[2] &= \{b1+c1\} & out[2] &= \{x2+y1,x2-y1,b1+c1,x2+1,x3+y1\}
\end{align*}
\]

Note the importance of taking an “optimistic” view of \(\text{in}[2]\).
Computing GFP Solution Example

Let $A$ = set of all potentially interesting expressions, namely
\begin{align*}
&\{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*}

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 Example

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

i 1 ← 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

7

i6 = phi(i4,i5)

EXIT

i6 = phi(i4,i5)

EXIT