CS410P/510 Programming Language
Compilation
Fall 2022
Lecture on Optimization
• Really “improvement” rather than “optimization;” results are seldom optimal.

• Remove inefficiencies in user code and (at least as importantly) in compiler-generated code.

• Can be applied at several phases in pipeline, chiefly on intermediate or assembly code.

• Goal is usually to decrease execution time; sometimes it is important to decrease code size.

• Can operate at several levels of granularity:
  - “Local” : within basic blocks
  - “Global” : entire functions
  - “Interprocedural” : entire programs (maybe even multiple source files)

• Most of a serious modern compiler is devoted to optimization.
SOME IMPORTANT CLASSIC OPTIMIZATIONS

- Constant folding (partial evaluation)
- Constant propagation
- Dead code elimination
- Useless code elimination
- Common subexpression elimination (redundancy elimination)
- Invariant hoisting from loops
- Strength reduction (replacing an expensive operation with an equivalent cheaper one)
- Function inlining

We hope for modest constant factor improvements in running time and code space.

Asymptotic improvements generally require a different algorithm, which the compiler is very unlikely to discover!
KEY CONSIDERATIONS FOR ANY OPTIMIZATION

- Safety: Transformation **must** maintain observable behavior of program (on all inputs).
- Profitability: Transformation **should** speed up execution (or shrink code size, or both).
- Opportunity: We need an efficient way to find out where we can apply transformation safely and profitably.
THEORETICAL UNDERPINNINGS

• Mostly about finding static approximations to dynamic behavior.
• But a very ad hoc subject, with relatively few unifying principles!

Some useful tools:
• Control-flow graphs
• Data-flow analysis
• Pointer analysis
• Dominators
• Static single assignment
• Polyhedral analysis
OPPORTUNITIES FOR OPTIMIZATION

• Compensating for abstractions in source language code, e.g.
  - Array access requires non-trivial address calculations
  - Object-oriented languages call many small methods, using expensive dynamic dispatch

• Utilizing resources in target code, e.g.
  - Certain processors have specialized instructions for common patterns
  - Co-processors (GPUs, etc.) – although today few compilers use these automatically

(Note: modern CPUs do lots of dynamic optimization in hardware, which may lessen the importance/impact of compiler optimization.)

• Optimization opportunities are cumulative: doing one transformation often enables others.
WHERE ARE OPTIMIZATIONS DONE?

- On assembly code
  - particularly useful when optimization depends on details of target machine architecture
- At source level, by rewriting the program.
  - e.g., like our $L_{\text{Int}}$ PartialEvaluation pass
  - can be limited by expressiveness of source language
- On an intermediate language, typically with explicit control flow structure and unlimited registers and memory
  - e.g. our $C_{\text{If}}$
  - portable across different target (and source!) languages
  - often the “sweet spot” for general-purpose optimizations
These work within a single basic block, so control flow is trivial.

Some simple examples in $C_{If}$:

- **Constant folding; static conditional evaluation**

  \[
  \begin{align*}
  &L1: a = 2 + 3 \\
  &\text{if } a > 4 \text{ goto } L2 \\
  &\text{else goto } L3
  \end{align*}
  \Rightarrow
  \begin{align*}
  &L1: a = 5 \\
  &\text{goto } L2
  \end{align*}
  \]

- **Together with constant propagation**

  \[
  \begin{align*}
  &L1: a = -9 \\
  &b = 10 + a \\
  &c = c + b
  \end{align*}
  \Rightarrow
  \begin{align*}
  &L1: a = 5 \\
  &b = 1 \\
  &c = c + 1
  \end{align*}
  \]

(Subsequent transformations might get rid of the assignment to $b$.)
“Peephole” Optimizations

Local optimizations performed on machine code using minimal context information, e.g.

- **Algebraic Simplification**
  
  \[
  \text{addq } 0, \%rsp \implies \text{(nothing)}
  \]

- **Redundant load or store removal**
  
  \[
  \text{movq } -20(\%rbp), \%r10 \implies \text{movq } -20(\%rbp), \%r10
  
  \text{movq } \%r10, -20(\%rbp)
  \]

- **Strength Reduction**
  
  \[
  \text{imulq } 8, \%r10 \implies \text{salq } 3, \%r10
  \]

- **Use of machine idioms**
  
  \[
  \text{imulq } 8, \%r10 \implies \text{leaq } 20(\%r11,\%r10,8), \%r10
  
  \text{addq } \%r11, \%r10
  
  \text{addq } 20, \%r10
  \]
**Whole-procedure Optimizations**

- Consider entire control-flow graph (CFG) of procedure instead of just one basic block at a time.

- Typically requires deeper analysis of code (e.g. data-flow analysis).

- Example: Reorder blocks; remove jumps to jumps; remove unreachable code

```
L1: cmpx x,2         ⇒         L1: cmpx x,2
       jl L2
       jmp L3
L2: jmp L4
       jmp L1
L3: addx $1,x
       jmp L1
L4: ...
```
MORE WHOLE-PROCEDURE OPTIMIZATIONS

- Example: Remove useless code (e.g. stores to non-live variables)

L1: \( x = x + 1 \)  \( \Rightarrow \)  L1: \( y = z + w \)
\( y = z + w \)
\( x = y + z \)

if \( x < 10 \) goto L1;
else goto L2

if \( x < 10 \) goto L1;
else goto L2
Loop optimizations are most important whole-procedure transformations.

- **Code motion**: “hoist” expensive calculations above the loop.

- Use **induction variables** and reduction in strength. Change only one index variable on each loop iteration, and choose one that’s cheap to change.

- Partially **unrolling** the loop can reduce per-iteration overheads and improve instruction scheduling.
Illustrating hoisting and strength reduction.

```python
x = 0
while x < 1000:
a[x] = a[y]
x = x + 1
```

L0: \( x = 0 \)
L1: if \( x < 1000 \) goto L2
   else goto L3
L2: \( j = y \times 8 \)
    \( u = a + j \)
    \( v = *u \)
    \( i = x \times 8 \)
    \( t = a + i \)
    \( *t = v \)
    \( x = x + 1 \)
    goto L1
L3: ...

\( \Rightarrow \)
L0: \( t = a \)
    \( j = y \times 8 \)
    \( u = a + j \)
    \( w = a + 8000 \)
L1: if \( t < w \) goto L2
    else goto L3
L2: \( v = *u \)
    \( *t = v \)
    \( t = t + 8 \)
    goto L1
L3: ...
```
**Procedure inlining** is most important.

- Replace a procedure call with a copy of the procedure body (including initial assignments to parameters).
- Applicable when body is not too big, or is called only once.

**Benefits:**
- Saves overhead of procedure entry/exit, argument passing, etc.
- Permits other optimizations to work over procedure boundaries.
- Particularly useful for languages that encourage use of small procedures (e.g. OO state get/set methods).

**Cost:**
- Risk of “code explosion.”
- Doesn’t work when callee is not statically known (e.g. OO dynamic dispatch or FP first-class calls).
Compiler Correctness

Optimizing compilers are complex artifacts, and they have bugs!

Some promising approaches to enhancing compiler correctness:

- Randomized testing (can find dark corners that human-written tests may miss)
- Formal verification of correctness using machine-assisted theorem proving
One important optimization opportunity is removing repeated calculations of the same value.

For remainder of lecture, we consider how this can be done at different levels of granularity.

Consider this $C_{if}$ sequence:

$$
\begin{align*}
g &= x + y \\
h &= u - v \\
w &= g + h \\
u &= x + y \quad \# \text{ redundant calculation} \\
x &= u - v \quad \# \text{ not redundant (why not?)}
\end{align*}
$$

**Value numbering** is an approach to finding and eliminating common subexpressions

- Process each instruction in order.
- Maintain a mapping from identifiers (e.g. $x$) and arithmetic expressions (e.g. $(\#1 + \#2)$) to **value numbers**.
- If an entry in the mapping already exists, rewrite the instruction to use it.
## Local Value Numbering

<table>
<thead>
<tr>
<th>Initial code</th>
<th>Final code</th>
<th>Mapping entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g = x + y )</td>
<td>( g = x + y )</td>
<td>( x \rightarrow #1 ) #1: ( x )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( y \rightarrow #2 ) #2: ( y )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((#1 + #2) \rightarrow #3 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( g \rightarrow #3 ) #3: ( g )</td>
</tr>
<tr>
<td>( h = u - v )</td>
<td>( h = u - v )</td>
<td>( u \rightarrow #4 ) #4: ( u )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( v \rightarrow #5 ) #5: ( v )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((#4 - #5) \rightarrow #6 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( h \rightarrow #6 ) #6: ( h )</td>
</tr>
<tr>
<td>( w = g + h )</td>
<td>( w = g + h )</td>
<td>((#3 + #6) \rightarrow #7 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( w \rightarrow #7 ) #7: ( w )</td>
</tr>
<tr>
<td>( u = x + y )</td>
<td>( u = g )</td>
<td>( u \rightarrow #3 )</td>
</tr>
<tr>
<td>( x = u - v )</td>
<td>( x = u - v )</td>
<td>((#3 - #5) \rightarrow #8 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( x \rightarrow #8 ) #8: ( x )</td>
</tr>
</tbody>
</table>

- Now can potentially replace uses of \( u \) by \( g \) and eliminate the assignment \( u = g \).
- This scheme works better when all names are assigned just once.
Can do better by analyzing 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).
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:

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

where $\text{pred}(n) =$ set of predecessors of $n$ in CFG.
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

```
ENTER

k ← 0
i ← 1
j ← 2

i ≤ N?

k > 0?

EXIT
0 1 34 56 7
0 1 2 34 56
```

PSU CS410P/510 FALL'22 LECTURE ON OPTIMIZATION  © 1992–2022 ANDREW TOLMACH
DOMINATOR-BASED VALUE NUMBERING

Do analysis over paths in dominator tree.

Do analysis over paths in dominator tree.

1

x1 ← a1 + b1
y1 ← a1 - b1
w1 ← a1 * b1
P ?

2

x2 ← a1 - b1

3

z1 ← a1 + b1

4

z2 ← a1 * b1
Even with dominator-based VN, we cannot find redundant expressions computed on different paths.

An alternative approach is to compute available expressions. An expression $e$ is available at node $n$ if on every path from $S$ to $n$, $e$ is evaluated and none of its constituent variables is redefined between that evaluation and $n$.

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 thus:

$$
\begin{align*}
gen(t \leftarrow b \text{ bop } c) &= \{b \text{ bop } c\} \quad \text{kill}(t \leftarrow \_ ) = \bigcup_{u,bop} \{t \text{ bop } u, u \text{ bop } t\} \\
gen(\text{other instruction}) &= \emptyset \quad \text{kill}(\text{other instruction}) = \emptyset \\
in(n) &= \bigcap_{p \in \text{pred}(n)} \text{out}(p) \\
\text{out}(n) &= (\text{in}(n) \cup \text{gen}(n)) - \text{kill}(n)
\end{align*}
$$

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

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

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

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

\[ \text{gen}[4] = \{a\times b, a/b\} \]
This is a forwards data flow problem, with initial approximation

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

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

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

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

- Keith Cooper and Linda Torczon, *Engineering a Compiler*, 2nd ed., Morgan Kaufmann, 2012, has thorough and practical coverage of many standard optimizations. (The slides on redundancy analysis are inspired by their treatment.)

