COMPILER OPTIMIZATION

Baishakhi Ray

Fall 2020

These slides are motivated from Prof. Alex Aiken and Prof. Calvin Lin
Optimization

- Optimization is our last compiler phase
- Most complexity in modern compilers is in the optimizer
  - Also by far the largest phase
- Optimizations are often applied to intermediate representations of code
When should we perform optimizations?

- **On AST**
  - Pro: Machine independent
  - Con: Too high level

- **On assembly language**
  - Pro: Exposes optimization opportunities
  - Con: Machine dependent
  - Con: Must reimplement optimizations when retargetting

- **On an intermediate language**
  - Pro: Machine independent
  - Pro: Exposes optimization opportunities
Intermediate Languages

- Intermediate language = high-level assembly
  - Uses register names, but has an unlimited number
  - Uses control structures like assembly language
  - Uses opcodes but some are higher level
    - E.g., push translates to several assembly instructions
  - Most opcodes correspond directly to assembly opcodes
Three-Address Intermediate Code

- Each instruction is of the form
  - $x := y \text{ op } z$ (binary operation)
  - $x := \text{ op } y$ (unary operation)
  - $y$ and $z$ are registers or constants
  - Common form of intermediate code

- The expression $x + y \ast z$ is translated
  - $t1 := y \ast z$
  - $t2 := x + t1$
  - Each subexpression has a “name”
Optimization Overview

- Optimization seeks to improve a program’s resource utilization
  - Execution time (most often)
  - Code size
  - Network messages sent, etc.

- Optimization should not alter what the program computes
  - The answer must still be the same
A Classification of Optimizations

- For languages like C there are three granularities of optimizations
  1. Local optimizations
     - Apply to a basic block in isolation
  2. Global optimizations
     - Apply to a control-flow graph (method body) in isolation
  3. Inter-procedural optimizations
     - Apply across method boundaries

- Most compilers do (1), many do (2), few do (3)
Cost of Optimizations

- In practice, a conscious decision is made not to implement the fanciest optimization known

- Why?
  - Some optimizations are hard to implement
  - Some optimizations are costly in compilation time
  - Some optimizations have low benefit
  - Many fancy optimizations are all three!

- Goal: Maximum benefit for minimum cost
Local Optimizations

- The simplest form of optimizations
- No need to analyze the whole procedure body
  - Just the basic block in question
- Example: algebraic simplification
Algebraic Simplification

- Some statements can be deleted
  \[ x := x + 0 \]
  \[ x := x \times 1 \]

- Some statements can be simplified
  \[ x := x \times 0 \Rightarrow x := 0 \]
  \[ y := y \times 2 \Rightarrow y := y \times y \]
  \[ x := x \times 8 \Rightarrow x := x \ll 3 \]
  \[ x := x \times 15 \Rightarrow t := x \ll 4; x := t - x \]

(on some machines \( \ll \) is faster than \( \times \); but not on all!)
Constant Folding

- Operations on constants can be computed at compile time
  - If there is a statement $x := y \text{ op } z$
  - And $y$ and $z$ are constants
  - Then $y \text{ op } z$ can be computed at compile time

- Example: $x := 2 + 2 \Rightarrow x := 4$

- Example: if $2 < 0$ jump L can be deleted

- When might constant folding be dangerous?
  - Floating point errors in cross-architecture compilation
Flow of Control Optimizations

- **Eliminate unreachable basic blocks:**
  - Code that is unreachable from the initial block
    - E.g., basic blocks that are not the target of any jump or “fall through” from a conditional
  
- **Removing unreachable code makes the program smaller**
  - And sometimes also faster
    - Due to memory cache effects (increased spatial locality)
Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment

- Rewrite intermediate code in single assignment form

  \[
  \begin{align*}
  x & := z + y & b & := z + y \\
  a & := x & \Rightarrow & a := b \\
  x & := 2 \times x & x & := 2 \times b \\
  \end{align*}
  \]

  (b is a fresh register)

- More complicated in general, due to loops
Static Single Assignment (SSA) Form

- **Idea**
  - Each variable has only one static definition
  - Makes it easier to reason about values instead of variables
  - The point of SSA form is to represent use-def information explicitly

- **Transformation to SSA**
  - Rename each definition
  - Rename all uses reached by that definition

- **Example:**
  
  \[
  \begin{align*}
  v & := \ldots \\
  \ldots & := \ldots v \ldots \\
  v & := \ldots \\
  \ldots & := \ldots v \ldots \\
  v & := \ldots \\
  \ldots & := \ldots v \ldots \\
  v_0 & := \ldots \\
  \ldots & := \ldots v_0 \ldots \\
  v_1 & := \ldots \\
  \ldots & := \ldots v_1 \ldots \\
  v_2 & := \ldots \\
  \ldots & := \ldots v_2 \ldots 
  \end{align*}
  \]
Problem: A use may be reached by several definitions
SSA and Control Flow (cont)

- **Merging Definitions**
  - $\varnothing$-functions merge multiple reaching definitions
Merging Definitions

- $\emptyset$-functions merge multiple reaching definitions
SSA vs. use-def chain

- SSA form is more constrained

- Advantages of SSA
  - More compact
  - Some analyses become simpler when each use has only one def
  - Value merging is explicit
  - Usually, easier to update and manipulate

- Furthermore
  - Eliminates false dependences (simplifying context)
SSA vs. use-def chain

- Worst case du-chains?

```c
switch (c1) {
    case 1: x = 1; break;
    case 2: x = 2; break;
    case 3: x = 3; break;
}
switch (c2) {
    case 1: y1 = x; break;
    case 2: y2 = x; break;
    case 3: y3 = x; break;
    case 4: y4 = x; break;
}
m defs and n uses leads to m x n du chains
```
Transformation to SSA Form

- **Two steps**
  - Insert $\emptyset$-functions
  - Rename variables

- **Basic Rule of Placing $\emptyset$-Functions?**
  - If two distinct (non-null) paths $x \rightarrow z$ and $y \rightarrow z$ converge at node $z$, and nodes $x$ and $y$ contain definitions of variable $v$, then we insert a $\emptyset$-function for $v$ at $z$
Approaches to Placing $\emptyset$-Functions

- **Minimal**
  - As few as possible subject to the basic rule

- **Briggs-Minimal**
  - Same as minimal, except $v$ must be live across some edge of the CFG
    - Briggs Minimal will not place a $\emptyset$ function in this case because $v$ is not live across any CFG edge.
    - Exploits the short lifetimes of many temporary variables
SSA: Variable Renaming

- When we see a variable on the LHS, create a new name for it.
- When we see a variable on the RHS, use appropriate subscript.
- Easy for straightforward code.

- Harder when there’s control flow:
  - For each use of x, find the definition of x that dominates it.
Common Subexpression Elimination

If

- Basic block is in single assignment form
- A definition $x :=$ is the first use of $x$ in a block

Then

- When two assignments have the same rhs, they compute the same value

Example:

$x := y + z$  \hspace{1cm}  x := y + z$

$\ldots \implies \ldots$

$w := y + z$  \hspace{1cm}  w := x$

(the values of $x$, $y$, and $z$ do not change in the $\ldots$ code)
Copy Propagation

- If \( w := x \) appears in a block, replace subsequent uses of \( w \) with uses of \( x \)
  - Assumes single assignment form

- Example:
  
  \[
  \begin{align*}
  b & := z + y \\
  a & := b \\
  x & := 2 \cdot a
  \end{align*}
  \]

  \[
  \begin{align*}
  b & := z + y \\
  a & := b \\
  x & := 2 \cdot b
  \end{align*}
  \]

- Only useful for enabling other optimizations
  - Constant folding
  - Dead code elimination
Copy Propagation and Constant Folding

- Example:

\[
\begin{align*}
  a & := 5 \\
  x & := 2 \times a \\
  y & := x + 6 \\
  t & := x \times y
\end{align*}
\quad \Rightarrow 
\begin{align*}
  a & := 5 \\
  x & := 10 \\
  y & := 16 \\
  t & := x \ll 4
\end{align*}
\]
Copy Propagation and Dead Code Elimination

- **If**
  - w := rhs appears in a basic block
  - w does not appear anywhere else in the program

- **Then the statement w := rhs is dead and can be eliminated**
  - Dead = does not contribute to the program’s result
  - Example: (a is not used anywhere else)

\[
\begin{align*}
x & := z + y & b & := z + y & b & := z + y \\
a & := x & \Rightarrow & a & := b & \Rightarrow & x & := 2 \times b \\
x & := 2 \times a & x & := 2 \times b
\end{align*}
\]
Applying Local Optimizations

- Each local optimization does little by itself
- Typically optimizations interact
  - Performing one optimization enables another
- Optimizing compilers repeat optimizations until no improvement is possible
  - The optimizer can also be stopped at any point to limit compilation time
An Example

- **Initial code:**
  
a := x ** 2  
b := 3  
c := x  
d := c * c  
e := b * 2  
f := a + d  
g := e * f
An Example

- **Algebraic optimization:**
  
  \[
  \begin{align*}
  a & := x ** 2 \\
  b & := 3 \\
  c & := x \\
  d & := c * c \\
  e & := b * 2 \\
  f & := a + d \\
  g & := e * f \\
  
  a & := x * x \\
  b & := 3 \\
  c & := x \\
  d & := c * c \\
  e & := b \ll 1 \\
  f & := a + d \\
  g & := e * f
  \end{align*}
  \]
An Example

- Copy Propagation:
  
  \[
  \begin{align*}
  a &:= x \times x \\
  b &:= 3 \\
  c &:= x \\
  d &:= c \times c \\
  e &:= b \ll 1 \\
  f &:= a + d \\
  g &:= e \times f
  \end{align*}
  \]

  \[
  \begin{align*}
  a &:= x \times x \\
  b &:= 3 \\
  c &:= x \\
  d &:= x \times x \\
  e &:= 3 \ll 1 \\
  f &:= a + d \\
  g &:= e \times f
  \end{align*}
  \]
An Example

- Constant folding:
  
  \[
  \begin{align*}
  a & := x \times x \\
  b & := 3 \\
  c & := x \\
  d & := x \times x \\
  e & := 3 \ll 1 \\
  f & := a + d \\
  g & := e \times f
  \end{align*}
  \]

  \[
  \begin{align*}
  a & := x \times x \\
  b & := 3 \\
  c & := x \\
  d & := x \times x \\
  e & := 6 \\
  f & := a + d \\
  g & := e \times f
  \end{align*}
  \]
An Example

- Common subexpression elimination:

  \[
  a := x \times x \\
  b := 3 \\
  c := x \\
  d := x \times x \\
  e := 6 \\
  f := a + d \\
  g := e \times f
  \]

  \[
  a := x \times x \\
  b := 3 \\
  c := x \\
  d := a \\
  e := 6 \\
  f := a + d \\
  g := e \times f
  \]
An Example

- **Copy propagation:**
  
  a := x * x
  b := 3
  c := x
  d := a
  e := 6
  f := a + d
  g := e * f

  a := x * x
  b := 3
  c := x
d := a
  e := 6
  f := a + a
g := 6 * f
An Example

- **Dead code elimination:**
  
  \[
  \begin{align*}
  a & := x \times x \\
  b & := 3 \\
  c & := x \\
  d & := a \\
  e & := 6 \\
  f & := a + a \\
  g & := 6 \times f
  \end{align*}
  \]
Peephole Optimizations on Assembly Code

- These optimizations work on intermediate code
  - Target independent
  - But they can be applied on assembly language also

- **Peephole optimization is effective for improving assembly code**
  - The “peephole” is a short sequence of (usually contiguous) instructions
  - The optimizer replaces the sequence with another equivalent one (but faster)
Peephole Optimizations (Cont.)

- Write peephole optimizations as replacement rules
  \[ i_1, \ldots, i_n \rightarrow j_1, \ldots, j_m \]
  where the rhs is the improved version of the lhs

- Example:
  \[ \text{move } a \text{ } b, \text{move } b \text{ } a \rightarrow \text{move } a \text{ } b \]
  - Works if \text{move } b \text{ } a \text{ is not the target of a jump}

- Another example
  \[ \text{addiu } a \text{ } a \text{ } i, \text{addiu } a \text{ } a \text{ } j \rightarrow \text{addiu } a \text{ } a \text{ } i+j \]
Many (but not all) of the basic block optimizations can be cast as peephole optimizations

- Example: addiu $a $b 0 → move $a $b
- Example: move $a $a → –
- These two together eliminate addiu $a $a 0

As for local optimizations, peephole optimizations must be applied repeatedly for maximum effect
Local Optimizations: Notes

- Intermediate code is helpful for many optimizations
- Many simple optimizations can still be applied on assembly language
- “Program optimization” is grossly misnamed
  - Code produced by “optimizers” is not optimal in any reasonable sense
  - “Program improvement” is a more appropriate term