Project 4: Dataflow Optimization
Michael Gordon
Thoughts on Code Gen

• ??

• Hidden tests have been revealed.
  – /mit/6.035/provided/codegen/tests/hidden

• Add them to your regression suite.
An “Optimizing” Compiler

• Somehow make the code better (on average):
  – Faster
  – Smaller memory footprint of code
  – Less memory used during run

• How to prove this:
  – Experimentation on benchmark suite!

• Must preserve the meaning of the original program!
  – Including errors!
An Optimizing Compiler

- Lowering
- Control Flow Analysis
- Optimization
- Code Gen
- Dataflow Analysis
- Transformation
- Peephole
- Dataflow Analysis
- Transformation
Optimizing Compiler

- Optimizations are hard
  - Have to always be correct
  - Better leave program unchanged than break it
- Have to do lots of analysis
  - Control flow
  - Data flow
Low IR (or Mid IR)

• Do analysis on low-level IR (does this fit what you had for code gen?)
  – Simple computations: a = b + c
  – explicit array accesses
  – gotos
  – labels
  – moves
  – calls

• See Tiger chp. 17 or Whale chp. 4
• Perform transformations on your IR:
  – Global CSE
  – Loop invariant code motion
  – Copy propagation
  – DCE

• Some optimizations may work better if you have info from high level IR
  – Parallelization
  – Maybe easier to do in High-level IR?
Control-Flow Analysis

• Convert the intermediate code into graph of basic blocks
  • Basic block:
    – sequence of instructions with a single entry and a single exit
    – Control must enter at beginning and leave at end
  • Simple to convert to a CFG
    – find heads of basic block:
      • after jump
      • target of jump
Peephole Optimizations

• Examine a short sequence of instructions
• Try to replace with a better sequence
• Examples:
  – Flow of controls
    • jumps to jumps
  – Algebraic Simplification
    • $x + 0 \rightarrow x$
  – Strength Reduction
    • $x \times 3 \rightarrow x + x + x$
    • Look at AMD64 documentation
Inline Function Expansion (Procedure Integration)

• Replace a function call with the body of the function
• Usually done on high-level IR (AST)
• Careful:
  – Performance?
  – Recursion?!
  – Names…
Example

Program {
    int x;
    void foo() {
        x = 2;
    }
}

void main() {
    {
        int x;
        foo();
    }
    print(x);
}
Program {
    int x;
    void foo() {
        x = 2;
    }

    void main() {
        {
            int x;
            x = 2;
        }
        print(x);
    }
}
“Global” Optimizations

- Global mean inter-basic block and intra-procedural
- You can inline functions
- Operate on control flow graph of basic blocks
  - You can use a CFG of MIR or LIR
- Usually:
  - Perform some dataflow analysis to find candidates
  - Validate the correct of candidates using other tests
Iterative Dataflow Analysis

- Use bit vectors to represent the information
  - instructions, expressions, variables, etc.
- Set of dataflow equations
- Iterate until a fixed point is reached
- For each basic block, b:
  - IN[b] – information that flows into block
  - OUT[b] – information that flows out of block
  - What happens inside the block
Example: Reaching Defs

• Concept of definition and use
  – $a = x+y$
  – is a definition of $a$
  – is a use of $x$ and $y$

• Given a program point $p$, a definition $d$ reaches $p$
  – there exists a path from $p$ to $d$ where
    • there is not a redefinition of the var of $d$
  – In other words, $d$ is not killed before it reaches $p$
Example: ReachingDefs

• Each basic block has
  – IN - set of definitions that reach beginning of block
  – OUT - set of definitions that reach end of block
  – GEN - set of definitions generated in block
    • Be careful about redefinitions in block
  – KILL - set of definitions killed in block
    • A statement does not kill itself!
Example: ReachingDefs

- $\text{IN}[b] = \text{OUT}[b_1] \cup \ldots \cup \text{OUT}[b_n]$
  - where $b_1, \ldots, b_n$ are predecessors of $b$ in CFG
- $\text{OUT}[b] = \text{GEN}[b] \cup (\text{IN}[b] - \text{KILL}[b])$
  - Transfer function!
- $\text{IN}[\text{entry}] = 0\ldots0$

- Forward analysis
- Confluence operator: $\cup$
- Transfer function of form: $f(X) = A \cup (X - B)$
  - $A = \text{GEN}$, $B = \text{KILL}$
Analysis Information Inside Basic Blocks

• One detail:
  – Given dataflow information at IN and OUT of node
  – Also need to compute information at each statement of basic block
  – Simple propagation algorithm usually works fine
  – Can be viewed as restricted case of dataflow analysis

• Generates gen[b] and kill[b] sets for each basic blocks for reaching defs

• Might have to specialize for each analysis
Transformation Examples with Dataflow Analysis

- Global Constant Propagation and Folding
  - Reaching definitions
- Global Copy Propagation
  - Reaching definitions + More
- Loop Invariant Code Motion
  - Reaching definitions
- Liveness Analysis
  - Useful for register allocation
Constant Propagation

- **Constant propagation** is the process of substituting the values of known constants in expressions at compile time.

```c
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

- Applying constant propagation once yields:

```c
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

- Can apply again after folding!
- Works on your 3-address low IR.
Useful Way to Store ReachingDefs

- Use-def and Def-use chains
  - Use-Def (UD) chain lists all definitions flowing to a use of a variable
  - Def-Use (DU) chain lists all uses which can be reached by a definition
- Ex: Global Constant Propagation
  - For each use of a variable, find all definitions
  - If all definitions of the variable are constant and same value, replace the use with the constant
Copy Propagation

• **copy propagation** is the process of replacing the occurrences of targets of direct assignments with their values.

• A direct assignment is an instruction of the form \( x = y \), which simply assigns the value of \( y \) to \( x \).

\[
\begin{align*}
  x &= y; \\
  z &= 3 + x \\
\end{align*}
\]

• Copy propagation would yield:

\[
\begin{align*}
  x &= y \\
  z &= 3 + y \\
\end{align*}
\]
Copy Propagation

• For s: x = y, we can substitute y for x in all places, u, where this definition of x is used.
  1. s must be only def of x reaching u
  2. On every path from s to u, there are no assignments to y.

• 1 and 2 can be checked with u/d chains but with additional work.

• Can check 1 and 2 with a new dataflow analysis
Copy Propagation Analysis

- Bit-vector of all copy statements (could have multiple $x = y$)
- $c_{\text{gen}}[B]$ is the copy statements generated in $B$
  - for $x = y$, $x$ and $y$ cannot be assigned later in the block
- $c_{\text{kill}}[B]$ are the copy statements killed by $B$
  - $x = \text{exp}$
    - kills copy statements
    - $\text{var} = x$ and $x = \text{var}$ in different blocks!
Copy Propagation Analysis

- **OUT\[b\] = c\_gen[b] U (IN\[b\] – c\_kill[b])**
- **IN\[b\] = OUT[b1] ∩ ... ∩ OUT[bn]**
  - where b1, ..., bn are predecessors of b in CFG and bi is not initial
- **IN[b\_entry] = 0…0**

- Forward analysis
- Confluence operator ∩
- Transfer function: f(X) = A U (X – B)
Copy Propagation

• After this analysis we know that if the bit for S is 1 at entry to a block B, only this copy can “reach” B.
• We can replace y with x in B.

• Whale Book 12.5.
Liveness Analysis

- For block B, let $\text{DEF}[B]$ be the set of vars definitely assigned values in B prior to any use of that variable in B.
  - $x$ not in $\text{DEF}[\{y = x + 5; x = q;\}]$

- Let $\text{USE}[B]$ be the set of vars whose values may be used in B prior to any def of the var.
  - $x$ not in $\text{USE}[\{x = 6; y = x + 5;\}]$
Liveness Analysis

Liveness analysis:
• $IN[b] = USE[b] \cup (out[b] - DEF[b])$
• $OUT[B] = IN[s1] \cup \ldots \cup IN[sn]$
  where $s1\ldots sn$ are successors of $b$

• Backward analysis
• Confluence operator: $\cup$
• Transfer function: $f(X) = A \cup (X - B)$
Dead Code Elimination

- Do not use liveness analysis for DCE
- It operates on program variables not on statements!
- Consult Whale Book 18.10.
  - Requires DU and UD chains
Shortcoming of Liveness-Based DCE Example
Loop Invariant Code Motion

• Statements which could be moved before the loop or after the loop, without affecting the semantics of the program.

```c
void foo(int x, int z) {
    int y;
    for a = 0, x {
        y = (x + 3) + y + bar(z);
    }
    return y;
}
```

• Difficult to get correct: see Dragon 10.7
Loop Invariant Code Motion

• UD chains (where does a value come from?)
• Control flow analysis (to figure out which definition is or is not invariant for a loop)
  – Old Dragon Book Section 10.3
General Dataflow Analysis Framework

• Build parameterized dataflow analyzer once, use for all dataflow problems
  – should work on all your IRs
• Commonalities:
  – Transfer function form
  – Confluence operators \( U \) and \( \cap \)
• Differences:
  – Dataflow equations \( A \) and \( B \) of transfer function
  – The exact confluence operator
  – Forward or backward
General Dataflow Analysis Framework

• Questions:
  – How are arrays handled?
    • Handle elements individually for more information (when you know the information)
  – Globals:
    • How are function calls handled?
    • What can a function call do to global variables?
Common Sub-Expression Elimination

- if $x \circ y$ is computed more than once, can we eliminate one of the computations
- Might not always be profitable
  - increases register pressure
  - more memory accesses (versus ALU ops)
- For local transformation (within a basic block), we can use value numbering
  - See lecture
- For global (intra-procedural) CSE, we leverage dataflow analysis
  - Available expressions
Available Expressions

• Expression $x \circ y$ is available at point $p$ if
  – on every path to $p$, $x \circ y$ is computed and
  – neither $x$ nor $y$ are redefined since the most recent $x \circ y$ on a path

• Scan function for all expressions and create a bit vector to represent them
  – Should be simple if using quadruples
Formalizing Analysis

• Each basic block has
  – **IN** - set of expressions available at start of block
  – **OUT** - set of expressions available at end of block
  – **GEN** - set of expressions computed in block
    • generated in block and operands not redefined after
    • Scan block from beginning to end:
      – add expressions evaluated
      – delete expressions whose operands are assigned
      – be careful with \( a = a + b \)
  – **KILL** - set of expressions killed in in block
    • generated in other block but operands redefined in this block
    • look for assignments and kill expressions that have an
      operand that is assigned
Dataflow Equations

- \( \text{IN}[b] = \text{OUT}[b1] \cap \ldots \cap \text{OUT}[bn] \)
  - where \( b1, \ldots, bn \) are predecessors of \( b \) in CFG
- \( \text{OUT}[b] = (\text{IN}[b] - \text{KILL}[b]) \cup \text{GEN}[b] \)
- Initialize:
  - \( \text{IN}[i] = 1\ldots1 \) (all expressions)
  - \( \text{IN}[\text{entry}] = 0\ldots0 \) (or 1\ldots1 if we have special entry node)

- Forward analysis
- Confluence operator: \( \cap \)
- Transfer function of familiar form
Solving Equations

- Use fixed point algorithm
- \( \text{IN[entry]} = 0\ldots0 \)
- Initialize \( \text{OUT[b]} = 1\ldots1 \)
- Repeatedly apply equations
  - \( \text{IN[b]} = \text{OUT[b1]} \cap \ldots \cap \text{OUT[bn]} \)
  - \( \text{OUT[b]} = (\text{IN[b]} - \text{KILL[b]}) \cup \text{GEN[b]} \)
- Use a worklist algorithm to reach fixed point
Now What?

For all blocks \( b \) and expressions \( \text{exp} \) in \( \text{IN}[b] \) and evaluated in \( b \)

1. Locate occurrences in \( b \) of \( \text{exp} \)
2. make sure that none of the operands were re-defined in \( b \) previously, if so it is not a CSE
3. Find all the reaching occurrences of \( \text{exp} \) in predecessor blocks
   - Follow flow edges backwards from \( b \)
   - Don’t go through a block that evaluates \( \text{exp} \)
   - The last evaluation of \( \text{exp} \) in each block reaches \( b \)

4. Select a new temp \( t \)
   - Replace \( \text{exp} \) by \( t \) for all occurrences in \( b \) that are CSE (step 2)
   - For each instruction found in (3), \( a = \text{exp} \) replace with:
     
     \[
     \begin{align*}
     a &= \text{exp} \\
     t &= a
     \end{align*}
     \]
Expressions
1: x+y
2: i<n
3: i+c
4: x==0
Expressions
1: x+y
2: i<n
3: i+c
4: x==0

Global CSE Transform

a = x+y;
t = a
x == 0

x = z;
b = x+y;
t = b

i = x+y;

i < n

i = i+c;
c = x+y;
d = x+y
Global CSE Transform

Expressions
1: x+y
2: i<n
3: i+c
4: x==0

0000
a = x+y;
t = a
x == 0

1001
x = z;
b = x+y;
t = b

1000
i = t;

1000
i < n

1100
c = t;
i = i+c;

1100
d = t