

6.035

Fall 2006

Project 4: Dataflow Optimization

Michael Gordon

Segment III

- Grading now

Michael Gordon

2

6.035 ©MIT Fall 2006

Segment IV Roadmap

Extension!

– Now due Sunday @ 11:59pm

Michael Gordon

3

6.035 ©MIT Fall 2006

Segment IV Roadmap

- Design Document
 - Now due Friday 11/3 @ 5pm
 - Should cover implementation issues:
 - Design of your dataflow analysis framework
 - Your IR (format, CFG, basic blocks/instructions?, etc.)
 - Details of global CSE (canonicalization, gen/kill, code trans.)
 - What other analyses/transformations you are implementing
 - The order of your transformations
- Group meeting
 - not mandatory, meet with me if you want.
- Implementation and Report
 - Due on 11/9

Michael Gordon

4

6.035 ©MIT Fall 2006

An “Optimizing” Compiler

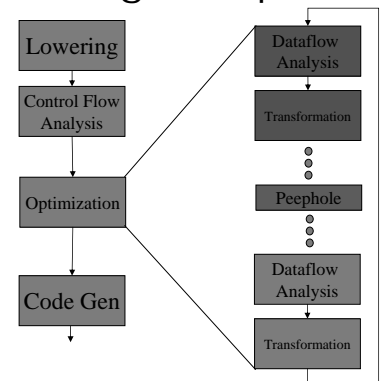
- Somehow make the code better (on average):
 - Faster
 - Smaller memory footprint of code
 - Less memory used during run
- How to prove this:
 - Experimentation on benchmark suite!
- Must preserve the meaning of the original program!

Michael Gordon

5

6.035 ©MIT Fall 2006

An Optimizing Compiler



Michael Gordon

6

6.035 ©MIT Fall 2006

Lowering (Recommendation)

- Introduce a middle-level IR:
 - Simple quadruple: $a = b + c$
 - gotos
 - labels
 - moves
 - calls
- See Tiger chp. 17 or Whale chp. 4

Lowering Cont.

- Perform dataflow optimizations on MIR and LIR
- MIR:
 - Global CSE
 - Loop invariant code motion
 - Copy propagation
- LIR
 - DCE

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
- Hint:
 - Create special entry and exit nodes

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
 - $x * 3 \rightarrow x + x + x$
 - 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);
  }
}
```

Example

```
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: Reaching Defs

- 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
 - KILL - set of definitions killed in block

Example: Reaching Defs

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

- Forward analysis
- Confluence operator: \cup
- Transfer function of form: $f(X) = A \cup (X - B)$
 - $A = GEN, B = KILL$

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
- Loop Invariant Code Motion
 - Reaching definitions
- Dead Code Elimination
 - Liveness Analysis

Constant Propagation

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

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

- Applying constant propagation once yields:

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

Constant Prop

- Reaching defs on constants

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 .

```
y = x;
z = 3 + y
```

- Copy propagation would yield:

```
z = 3 + x
```

Copy Propagation

- $OUT[b] = c_gen[b] \cup (IN[b] - c_kill[b])$
- $IN[b] = IN[b] = OUT[b_1] \cap \dots \cap OUT[b_n]$
 - where b_1, \dots, b_n are predecessors of b in CFG and b_i is not initial
- $IN[b_entry] = 0 \dots 0$
- Forward analysis
- Confluence operator \cap
- Transfer function: $f(X) = A \cup (X - B)$

Dead Code Elimination

- Removing code that has no effect or is never reached.

```
int foo(int b) {
    int x;
    x = 3 + b
    b = 7
    return x;
}
```

Dead Code Elimination

Liveness analysis:

- $IN[b] = USE[b] \cup (out[b] - kill[b])$
- $OUT[B] = IN[s_1] \cup \dots \cup IN[s_n]$
where $s_1 \dots s_n$ are successors of b

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

Loop Invariant Code Motion

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

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

- Difficult to get correct: see Dragon 10.7

Loop Invariant Code Motion

- Reaching definitions...

General Dataflow Analysis Framework

- Build parameterized dataflow analyzer once, use for all dataflow problems
 - should work on MIR and LIR
- Commonalities:
 - Transfer function form
 - Confluence operators \cup 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
 - Globals:
 - How are function calls handled?
- Examples!

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

- $IN[b] = OUT[b_1] \cap \dots \cap OUT[b_n]$
 - where b_1, \dots, b_n are predecessors of b in CFG
- $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- Initialize:
 - $IN[i] = 1 \dots 1$ (all expressions)
 - $IN[entry] = 0 \dots 0$ (or $1 \dots 1$ if we have special entry node)
- Forward analysis
- Confluence operator: \cap
- Transfer function of familiar form

Solving Equations

- Use fixed point algorithm
- $IN[entry] = 0 \dots 0$
- Initialize $OUT[b] = 1 \dots 1$
- Repeatedly apply equations
 - $IN[b] = OUT[b_1] \cap \dots \cap OUT[b_n]$
 - $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- Use a worklist algorithm to reach fixed point

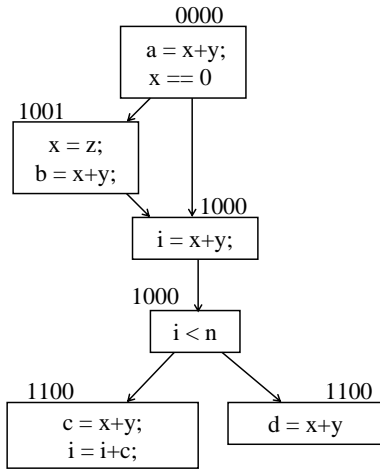
Now What?

For all blocks b and expressions exp in $IN[b]$ and evaluated in b

1. Locate occurrences in b of 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 exp in predecessor blocks
4. Select a new temp t
 - Replace exp by t for all occurrences in block that are CSE (step 2)
 - For each instruction found in (3), $a = exp$ replace with:
 $a = exp$
 $t = a$

Expressions

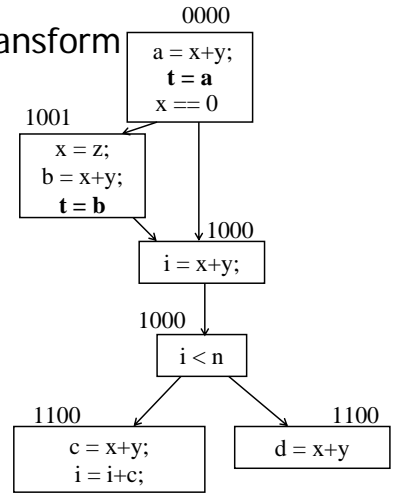
- 1: x+y
- 2: i<n
- 3: i+c
- 4: x==0



Global CSE Transform

Expressions

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



Global CSE Transform

Expressions

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

