

6.035

Fall 2006

Lecture 9: Introduction to Program Analysis and Optimization

Outline

- Introduction
- Basic Blocks
- Common Subexpression Elimination
- Copy Propagation
- Dead Code Elimination
- Algebraic Simplification
- Summary

Program Analysis

- Compile-time reasoning about run-time behavior of program
 - Can discover things that are always true:
 - “x is always 1 in the statement $y = x + z$ ”
 - “the pointer p always points into array a”
 - “the statement return 5 can never execute”
 - Can infer things that are likely to be true:
 - “the reference r usually refers to an object of class C”
 - “the statement $a = b + c$ appears to execute more frequently than the statement $x = y + z$ ”
 - Distinction between data and control-flow properties

Samir Aravamudan

3

6.035 ©MIT Fall 1998

Transformations

- Use analysis results to transform program
- Overall goal: improve some aspect of program
- Traditional goals:
 - Reduce number of executed instructions
 - Reduce overall code size
- Other goals emerge as space becomes more complex
 - Reduce number of cycles
 - Use vector or DSP instructions
 - Improve instruction or data cache hit rate
 - Reduce power consumption
 - Reduce memory usage

Samir Aravamudan

4

6.035 ©MIT Fall 1998

Control Flow Graph

- Nodes Represent Computation
 - Each Node is a Basic Block
 - Basic Block is a Sequence of Instructions with
 - No Branches Out Of Middle of Basic Block
 - No Branches Into Middle of Basic Block
 - Basic Blocks should be maximal
 - Execution of basic block starts with first instruction
 - Includes all instructions in basic block
- Edges Represent Control Flow

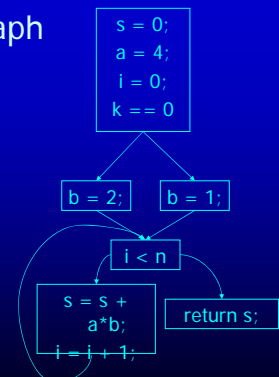
Samir Aravamudan

5

6.035 ©MIT Fall 1998

Control Flow Graph

```
into add(n, k) {
  s = 0; a = 4; i = 0;
  if (k == 0) b = 1;
  else b = 2;
  while (i < n) {
    s = s + a*b;
    i = i + 1;
  }
  return s;
}
```



Samir Aravamudan

6

6.035 ©MIT Fall 1998

Basic Block Construction

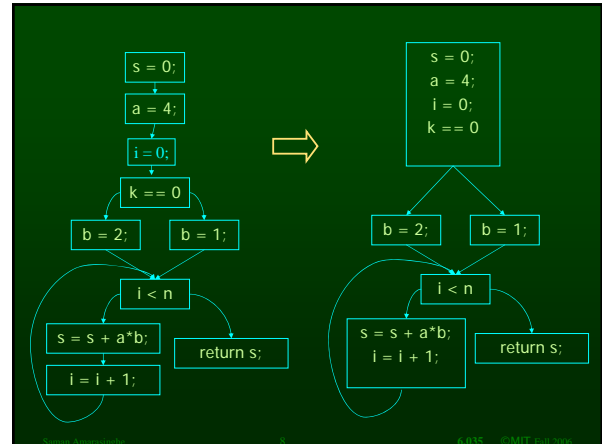
- Start with instruction control-flow graph
- Visit all edges in graph
- Merge adjacent nodes if
 - Only one edge from first node
 - Only one edge into second node



Samir Amarasinghe

7

6.035 ©MIT Fall 1998



Samir Amarasinghe

8

6.035 ©MIT Fall 1998

Program Points, Split and Join Points

- One program point before and after each statement in program
- Split point has multiple successors – conditional branch statements only split points
- Merge point has multiple predecessors
- Each basic block
 - Either starts with a merge point or its predecessor ends with a split point
 - Either ends with a split point or its successor starts with a merge point

Samir Amarasinghe

9

6.035 ©MIT Fall 1998

Basic Block Optimizations

- Common Sub-Expression Elimination
 - $a = (x+y)+z$; $b = x+y$;
 - $t = x+y$; $a = t+z$; $b = t$;
- Copy Propagation
 - $a = x+y$; $b = a$; $c = b+z$;
 - $a = x+y$; $b = a$; $c = a+z$;
- Constant Propagation
 - $x = 5$; $b = x+y$;
 - $x = 5$; $b = 5+y$;
- Dead Code Elimination
 - $a = x+y$; $b = a$; $b = a+z$;
 - $a = x+y$; $b = a+z$
- Algebraic Identities
 - $a = x*1$;
 - $a = x$;
- Strength Reduction
 - $t = i*4$;
 - $t = i < 2$;

Samir Amarasinghe

10

6.035 ©MIT Fall 1998

Basic Block Analysis Approach

- Assume normalized basic block - all statements are of the form
 - $var = var \text{ op } var$ (where op is a binary operator)
 - $var = \text{op } var$ (where op is a unary operator)
 - $var = var$
- Simulate a symbolic execution of basic block
 - Reason about values of variables (or other aspects of computation)
 - Derive property of interest

Samir Amarasinghe

11

6.035 ©MIT Fall 1998

Two Kinds of Variables

- Temporaries Introduced By Compiler
 - Transfer values only within basic block
 - Introduced as part of instruction flattening
 - Introduced by optimizations/transformations
 - Typically assigned to only once
- Program Variables
 - Declared in original program
 - May be assigned to multiple times
 - May transfer values between basic blocks

Samir Amarasinghe

12

6.035 ©MIT Fall 1998

Value Numbering

- Reason about values of variables and expressions in the program
 - Simulate execution of basic block
 - Assign virtual value to each variable and expression
- Discovered property: which variables and expressions have the same value
- Standard use:
 - Common subexpression elimination
 - Typically combined with transformation that
 - Saves computed values in temporaries
 - Replaces expressions with temporaries when value of expression previously computed

Suman Amarasinghe

13

6.035 ©MIT Fall 1998

Original Basic Block

```
a = x+y
b = a+z
b = b+y
c = a+z
```

New Basic Block

```
a = x+y
t1 = a
b = a+z
t2 = b
b = b+y
t3 = b
c = t2
```

Var to Val

```
x → v1
y → v2
a → v3
z → v4
b → v6
c → v5
```

Exp to Val

```
v1+v2 → v3
v3+v4 → v5
v5+v2 → v6
```

Exp to Tmp

```
v1+v2 → t1
v3+v4 → t2
v5+v2 → t3
```

Suman Amarasinghe

14

6.035 ©MIT Fall 1998

Value Numbering Summary

- Forward symbolic execution of basic block
- Each new value assigned to temporary
 - $a=x+y$; becomes $a=x+y$; $t=a$;
 - Temporary preserves value for use later in program even if original variable rewritten
 - $a=x+y$; $a=a+z$; $b=x+y$ becomes
 - $a=x+y$; $t=a$; $a=a+z$; $b=t$;
- Maps
 - Var to Val – specifies symbolic value for each variable
 - Exp to Val – specifies value of each evaluated expression
 - Exp to Tmp – specifies tmp that holds value of each evaluated expression

Suman Amarasinghe

15

6.035 ©MIT Fall 1998

Map Usage

- Var to Val
 - Used to compute symbolic value of y and z when processing statement of form $x = y + z$
- Exp to Tmp
 - Used to determine which tmp to use if $\text{value}(y) + \text{value}(z)$ previously computed when processing statement of form $x = y + z$
- Exp to Val
 - Used to update Var to Val when
 - processing statement of the form $x = y + z$, and
 - $\text{value}(y) + \text{value}(z)$ previously computed

Suman Amarasinghe

16

6.035 ©MIT Fall 1998

Interesting Properties

- Finds common subexpressions even if they use different variables in expressions
 - $y=a+b$; $x=b$; $z=a+x$ becomes
 - $y=a+b$; $t=y$; $x=b$; $z=t$
 - Why? Because computes with symbolic values
- Finds common subexpressions even if variable that originally held the value was overwritten
 - $y=a+b$; $y=1$; $z=a+b$ becomes
 - $y=a+b$; $t=y$; $y=1$; $z=t$
 - Why? Because saves values away in temporaries

Suman Amarasinghe

17

6.035 ©MIT Fall 1998

One More Interesting Property

- Flattening and CSE combine to capture partial and arbitrarily complex common subexpressions
 - $w=(a+b)+c$; $x=b$; $y=(a+x)+c$; $z=a+b$;
 - After flattening:
 - $t1=a+b$; $w=t1+c$; $x=b$; $t2=a+x$; $y=t2+c$; $z=a+b$;
 - CSE algorithm notices that
 - $t1+c$ and $t2+c$ compute same value
 - In the statement $z = a+b$, $a+b$ has already been computed so generated code can reuse the result
- ```
t1=a+b; w=t1+c; t3=w; x=b; t2=t1; y=t3; z=t1;
```

Suman Amarasinghe

18

6.035 ©MIT Fall 1998

## Problems I

- Algorithm has a temporary for each new value
  - $a=x+y$ ;  $t1=a$ ;
- Introduces
  - lots of temporaries
  - lots of copy statements to temporaries
- In many cases, temporaries and copy statements are unnecessary
- So we eliminate them with copy propagation and dead code elimination

Suman Amarasinghe

19

6.035 ©MIT Fall 1998

## Problems II

- Expressions has to be identical
  - $a=x+y+z$ ;  $b=y+z+x$ ;  $c=x*2+y+2*z-(x+z)$
- We use canonicalization
- We use algebraic simplification

Suman Amarasinghe

20

6.035 ©MIT Fall 1998

## Copy Propagation

- Once again, simulate execution of program
- If can, use original variable instead of temporary
  - $a=x+y$ ;  $b=x+y$ ;
  - After CSE becomes  $a=x+y$ ;  $t=a$ ;  $b=t$ ;
  - After CP becomes  $a=x+y$ ;  $t=a$ ;  $b=a$ ;
  - After DCE becomes  $a=x+y$ ;  $b=a$ ;
- Key idea:
  - determine when original variable is NOT overwritten between its assignment statement and the use of the computed value
  - If not overwritten, use original variable

Suman Amarasinghe

21

6.035 ©MIT Fall 1998

## Copy Propagation Maps

- Maintain two maps
  - tmp to var: tells which variable to use instead of a given temporary variable
  - var to set: inverse of tmp to var. tells which temps are mapped to a given variable by tmp to var

Suman Amarasinghe

22

6.035 ©MIT Fall 1998

## Copy Propagation Example

- Original
  - $a = x+y$
  - $b = a+z$
  - $c = x+y$
  - $a = b$
- After CSE
  - $a = x+y$
  - $t1 = a$
  - $b = a+z$
  - $t2 = b$
  - $c = t1$
  - $a = b$
- After CSE and Copy Propagation
  - $a = x+y$
  - $t1 = a$
  - $b = a+z$
  - $t2 = b$
  - $c = a$
  - $a = b$

Suman Amarasinghe

23

6.035 ©MIT Fall 1998

## Dead Code Elimination

- Copy propagation keeps all temps around
- May be temps that are never read
- Dead Code Elimination removes them

Basic Block After  
CSE and CP

```
a = x+y
t1 = a
b = a+z
t2 = b
c = a
a = b
```

Basic Block After  
CSE, CP and DCE

```
a = x+y
b = a+z
c = a
a = b
```

Suman Amarasinghe

24

6.035 ©MIT Fall 1998

## Dead Code Elimination

- Basic Idea
  - Process Code In Reverse Execution Order
  - Maintain a set of variables that are needed later in computation
  - If encounter an assignment to a temporary that is not needed, remove assignment

Suman Amarasinghe

25

6.035 ©MIT Fall 1998

## Algebraic Simplification

- Apply our knowledge from algebra, number theory etc. to simplify expressions

Suman Amarasinghe

26

6.035 ©MIT Fall 1998

## Algebraic Simplification

- Apply our knowledge from algebra, number theory etc. to simplify expressions
- Example
  - $a + 0 \Rightarrow a$
  - $a * 1 \Rightarrow a$
  - $a / 1 \Rightarrow a$
  - $a * 0 \Rightarrow 0$
  - $0 - a \Rightarrow -a$
  - $a + (-b) \Rightarrow a - b$
  - $-(-a) \Rightarrow a$

Suman Amarasinghe

27

6.035 ©MIT Fall 1998

## Algebraic Simplification

- Apply our knowledge from algebra, number theory etc. to simplify expressions
- Example
  - $a \wedge \text{true} \Rightarrow a$
  - $a \wedge \text{false} \Rightarrow \text{false}$
  - $a \vee \text{true} \Rightarrow \text{true}$
  - $a \vee \text{false} \Rightarrow a$

Suman Amarasinghe

28

6.035 ©MIT Fall 1998

## Algebraic Simplification

- Apply our knowledge from algebra, number theory etc. to simplify expressions
- Example
  - $a \wedge 2 \Rightarrow a * a$
  - $a \wedge 2 \Rightarrow a + a$
  - $a \wedge 8 \Rightarrow a \ll 3$

Suman Amarasinghe

29

6.035 ©MIT Fall 1998

## Opportunities for Algebraic Simplification

- In the code
  - Programmers are lazy to simplify expressions
  - Programs are more readable with full expressions
- After compiler expansion
  - Example: Array read  $A[8][12]$  will get expanded to  $*(Abase + 4*(12 + 8*256))$  which can be simplified
- After other optimizations

Suman Amarasinghe

30

6.035 ©MIT Fall 1998

## Usefulness of Algebraic Simplification

- Reduces the number of instructions
- Uses less expensive instructions
- Enable other optimizations

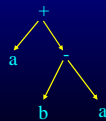
## Implementation

- Not a data-flow optimization!
- Find candidates that matches the simplification rules and simplify the expression trees
- Candidates may not be obvious

## Implementation

- Not a data-flow optimization!
- Find candidates that matches the simplification rules and simplify the expression trees
- Candidates may not be obvious

– Example  
 $a + b - a$



## Use knowledge about operators

- Commutative operators
  - $a \text{ op } b = b \text{ op } a$
  -
- Associative operators
  - $(a \text{ op } b) \text{ op } c = b \text{ op } (a \text{ op } c)$

## Canonical Format

- Put expression trees into a canonical format
  - Sum of multiplicands
  - Example  
 $(a+3)*(a+8)*4 \Rightarrow 4*a*a+44*a+96$
- Section 12.3.1 of whale book talks about this

## Effects on the Numerical Stability

- Some algebraic simplifications may produce incorrect results

## Effects on the Numerical Stability

- Some algebraic simplifications may produce incorrect results
- Example
  - $(a / b) * 0 + c$

## Effects on the Numerical Stability

- Some algebraic simplifications may produce incorrect results
- Example
  - $(a / b) * 0 + c$
  - we can simplify this to  $c$

## Effects on the Numerical Stability

- Some algebraic simplifications may produce incorrect results
- Example
  - $(a / b) * 0 + c$
  - we can simplify this to  $c$
  - But what about when  $b = 0$  should be an exception, but we'll get a result!

## Interesting Properties

- Analysis and Transformation Algorithms Symbolically Simulate Execution of Program
  - CSE and Copy Propagation go forward
  - Dead Code Elimination goes backwards
- Transformations stacked
  - Group of basic transformations work together
  - Often, one transformation creates inefficient code that is cleaned up by following transformations
  - Transformations can be useful even if original code may not benefit from transformation

## Other Basic Block Transformations

- Constant Propagation
- Strength Reduction
  - $a << 2 = a * 4$ ;  $a + a + a = 3 * a$ ;
- Do these in unified transformation framework, not in earlier or later phases

## Summary

- Basic block analyses and transformations
- Symbolically simulate execution of program
  - Forward (CSE, copy prop, constant prop)
  - Backward (Dead code elimination)
- Stacked groups of analyses and transformations that work together
  - CSE introduces excess temporaries and copy statements
  - Copy propagation often eliminates need to keep temporary variables around
  - Dead code elimination removes useless code
- Similar in spirit to many analyses and transformations that operate across basic blocks