Using Program Analysis for Optimization

Analysis and Optimizations

- Program Analysis
  - Discovers properties of a program

- Optimizations
  - Use analysis results to transform program
  - Goal: improve some aspect of program
    - number of executed instructions, number of cycles
    - cache hit rate
    - memory space (code or data)
    - power consumption
  - Has to be safe: Keep the semantics of the program

Control Flow Graph

Two Kinds of Variables

- Temporaries introduced by the compiler
  - Transfer values only within basic block
  - Introduced as part of instruction flattening
  - Introduced by optimizations/transformations

- Program variables
  - Declared in original program
  - May transfer values between basic blocks

Basic Block Optimizations

- Common Sub-Expression Elimination
  - \( a = (x+y)+z; b = x+y; \)
  - \( t = x+y; a = t+z; b = t; \)

- Constant Propagation
  - \( x = 5; b = x+y; \)
  - \( b = 5+y; \)

- Algebraic Simplification
  - \( a = x * 1; \)
  - \( a = x; \)

- Copy Propagation
  - \( a = x+y; b = a; c = b+z; \)
  - \( a = x+y; b = a; c = a+z; \)

- Dead Code Elimination
  - \( a = x+y; b = a; c = a+z; \)
  - \( a = x+y; c = a+z; \)

- Strength Reduction
  - \( t = i * 4; \)
  - \( t = i << 2; \)
Value Numbering

• Normalize basic block so that all statements are of the form
  – var = var op var (where op is a binary operator)
  – var = op var (where op is a unary operator)
  – var = var

• Simulate execution of basic block
  – Assign a virtual value to each variable
  – Assign a virtual value to each expression
  – Assign a temporary variable to hold value of each computed expression

Value Numbering for CSE

• As we simulate execution of program
• Generate a new version of program
  – 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;

CSE Example

• Original
  a = x+y
  b = a+z
  b = b+y
  c = a+z

• After CSE
  a = x+y
  b = a+z
  t = b
  c = b+y

• Issues
  – Temporaries store values for use later
  – CSE with different names
    • a = x; b = x+y; c = a+y;
  – Excessive Temp Generation and Use

Problems

• 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

Copy Propagation

• Once again, simulate execution of program
• If possible, use the original variable instead of a temporary
  – a = x+y; b = x+y;
  – After CSE becomes a = x+y; t = a; b = t;
  – After CP becomes a = x+y; b = a;

• Key idea: determine when original variables are NOT overwritten between computation of stored value and use of stored value
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

Copy Propagation Example

<table>
<thead>
<tr>
<th>Original</th>
<th>After CSE</th>
<th>After CSE and Copy Propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = x+y</td>
<td>a = x+y</td>
<td>a = x+y</td>
</tr>
<tr>
<td>t1 = a</td>
<td>t1 = a</td>
<td>t1 = a</td>
</tr>
<tr>
<td>b = a+z</td>
<td>b = a+z</td>
<td>b = a+z</td>
</tr>
<tr>
<td>c = x+y</td>
<td>t2 = b</td>
<td>t2 = b</td>
</tr>
<tr>
<td>a = b</td>
<td>c = t1</td>
<td>c = a</td>
</tr>
</tbody>
</table>

Basic Block

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

Basic Block After CSE and Copy Prop

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

Basic Block After CSE

a = x+y
b = a+z
t2 = b

c = t1

Basic Block After CSE and Copy Prop

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

Basic Block

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

Basic Block After CSE

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

Basic Block After CSE and Copy Prop

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

Basic Block

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

Basic Block After CSE

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

Basic Block After CSE and Copy Prop

a = x+y
b = a+z
t2 = b
c = a
Copy Propagation Example

<table>
<thead>
<tr>
<th>Basic Block After CSE</th>
<th>Basic Block After CSE and Copy Prop</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = x+y</td>
<td>a = x+y</td>
</tr>
<tr>
<td>t1 = a</td>
<td>t1 = a</td>
</tr>
<tr>
<td>b = a+z</td>
<td>b = a+z</td>
</tr>
<tr>
<td>t2 = b</td>
<td>t2 = b</td>
</tr>
<tr>
<td>c = t1</td>
<td>c = a</td>
</tr>
<tr>
<td>a = b</td>
<td>b = a</td>
</tr>
<tr>
<td>tmp to var</td>
<td>var to set</td>
</tr>
<tr>
<td>t1 \rightarrow a</td>
<td>a \rightarrow {t1}</td>
</tr>
<tr>
<td>t2 \rightarrow b</td>
<td>b \rightarrow {t2}</td>
</tr>
</tbody>
</table>

Dead Code Elimination

- Copy propagation keeps all temps around
- There may be temps that are never read
- Dead Code Elimination (DCE) removes them

<table>
<thead>
<tr>
<th>Basic Block After CSE and Copy Prop</th>
<th>Basic Block After CSE + Copy Prop + DCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = x+y</td>
<td>a = x+y</td>
</tr>
<tr>
<td>t1 = a</td>
<td>t1 = a</td>
</tr>
<tr>
<td>b = a+z</td>
<td>b = a+z</td>
</tr>
<tr>
<td>t2 = b</td>
<td>t2 = b</td>
</tr>
<tr>
<td>c = a</td>
<td>c = a</td>
</tr>
<tr>
<td>a = b</td>
<td>a = b</td>
</tr>
</tbody>
</table>

Assume that initially 

\[ \text{Needed Set} = \{a, c\} \]
Basic Block After CSE and Copy Prop
\[
\begin{align*}
a &= x + y \\
t_1 &= a \\
b &= a + z \\
t_2 &= b \\
c &= a \\
a &= b
\end{align*}
\]
Needed Set \(\{a, b, c\}\)

Basic Block After CSE and Copy Prop
\[
\begin{align*}
a &= x + y \\
t_1 &= a \\
b &= a + z \\
c &= a \\
a &= b
\end{align*}
\]
Needed Set \(\{a, b, c\}\)

Basic Block After CSE and Copy Prop
\[
\begin{align*}
a &= x + y \\
t_1 &= a \\
b &= a + z \\
c &= a \\
a &= b
\end{align*}
\]
Needed Set \(\{a, b, c, z\}\)

Basic Block After CSE and Copy Prop
\[
\begin{align*}
a &= x + y \\
t_1 &= a \\
b &= a + z \\
c &= a \\
a &= b
\end{align*}
\]
Needed Set \(\{a, b, c, z\}\)

Basic Block After CSE and Copy Prop
\[
\begin{align*}
a &= x + y \\
b &= a + z \\
c &= a \\
a &= b
\end{align*}
\]
Needed Set \(\{a, b, c, z\}\)

Basic Block After CSE and Copy Prop
\[
\begin{align*}
a &= x + y \\
b &= a + z \\
c &= a \\
a &= b
\end{align*}
\]
Needed Set \(\{a, b, c, z\}\)

Basic Block after CSE Copy Propagation and Dead Code Elimination
\[
\begin{align*}
a &= x + y \\
b &= a + z \\
c &= a \\
a &= b
\end{align*}
\]
Needed Set \(\{a, b, c, z\}\)
Basic Block after CSE + Copy Propagation + Dead Code Elimination
\[ a = x+y \]
\[ b = a+z \]
\[ c = a \]
\[ a = b \]

Needed Set \{a, b, c, z\}

Interesting Properties
- Analysis and Optimization Algorithms Simulate Execution of Program
  - CSE and Copy Propagation go forward
  - Dead Code Elimination goes backwards
- Optimizations are stacked
  - Group of basic transformations
  - Work together to get good result
  - Often, one transformation creates inefficient code that is cleaned up by subsequent transformations

Other Basic Block Transformations
- Constant Propagation
- Strength Reduction
  - \( a \ll 2 = a \times 4 \)
  - \( a + a + a = 3 \times a \)
- Algebraic Simplification
  - \( a = a \times 1 \)
  - \( b = b + 0 \)
- Unified transformation framework

Dataflow Analysis
- Used to determine properties of programs that involve multiple basic blocks
- Typically used to enable transformations
  - common sub-expression elimination
  - constant and copy propagation
  - dead code elimination
- Analysis and transformation often come in pairs

Reaching Definitions
- Concept of definition and use
  - \( z = x+y \)
  - is a definition of \( z \)
  - is a use of \( x \) and \( y \)
- A definition reaches a use if
  - value written by definition
  - may be read by use
Reaching Definitions and Constant Propagation

- Is a use of a variable a constant?
  - Check all reaching definitions
  - If all assign variable to same constant
  - Then use is in fact a constant
- Can replace variable with constant

Is a constant in $s = s+a*b$?

Yes!
On all reaching definitions

$\text{is } a = 4$

Constant Propagation Transform

Yes!
On all reaching definitions

$\text{is } a = 4$

Is b constant in $s = s+a*b$?

No!
One reaching definition with

$\text{is } b = 1$
One reaching definition with

$\text{is } b = 2$

Computing Reaching Definitions

- Compute with sets of definitions
  - represent sets using bit vectors
  - each definition has a position in bit vector
- At each basic block, compute
  - definitions that reach start of block
  - definitions that reach end of block
- Do computation by simulating execution of program until the fixed point is reached
Formalizing Analysis

• 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 the block
• GEN\[s = s + a*b; i = i + 1;\] = 0000011
• KILL\[s = s + a*b; i = i + 1;\] = 1010000
• Compiler scans each basic block to derive GEN and KILL sets

Dataflow Equations

• IN[b] = OUT[b1] ∪ ... ∪ OUT[bn]
  – where b1, ..., bn are predecessors of b in CFG
• OUT[b] = (IN[b] - KILL[b]) ∪ GEN[b]
• IN[entry] = 0000000
• Result: system of equations

Solving Equations

• Use fixed point algorithm
• Initialize with solution of OUT[b] = 0000000
• Repeatedly apply equations
  – IN[b] = OUT[b1] ∪ ... ∪ OUT[bn]
  – OUT[b] = (IN[b] - KILL[b]) ∪ GEN[b]
• Until reaching fixed point
  – I.e., until equation application has no further effect
• Use a worklist to track which equation applications may have a further effect

Questions

• Does the algorithm halt?
  – yes, because transfer function is monotonic
  – if increase IN, increase OUT
  – in limit, all bits are 1
• If bit is 1, is there always an execution in which corresponding definition reaches basic block?
• If bit is 0, does the corresponding definition ever reach basic block?
• Concept of conservative analysis
Available Expressions

• An expression $x+y$ is available at a point $p$ if
  – every path from the initial node to $p$ evaluates $x+y$ before reaching $p$.
  – and there are no assignments to $x$ or $y$ after the evaluation but before $p$.
• Available Expression information can be used to do global (across basic blocks) CSE.
• If an expression is available at use, there is no need to re-evaluate it.

Computing Available Expressions

• Represent sets of expressions using bit vectors
• Each expression corresponds to a bit
• Run dataflow algorithm similar to reaching definitions
• Big difference:
  – A definition reaches a basic block if it comes from ANY predecessor in CFG
  – An expression is available at a basic block only if it is available from ALL predecessors in CFG

Available Expressions Example

<table>
<thead>
<tr>
<th>Expressions</th>
<th>IN</th>
<th>OUT</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $x+y$</td>
<td></td>
<td>0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: $i &lt; n$</td>
<td></td>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: $i + c$</td>
<td></td>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: $x == 0$</td>
<td></td>
<td>1000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Global CSE Transform

<table>
<thead>
<tr>
<th>Expressions</th>
<th>IN</th>
<th>OUT</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $x+y$</td>
<td></td>
<td>0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: $i &lt; n$</td>
<td></td>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: $i + c$</td>
<td></td>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: $x == 0$</td>
<td></td>
<td>1000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
  – KILL - set of expressions killed in the block
• GEN[$x = z; b = x+y$] = 1000
• KILL[$x = z; b = x+y$] = 1001
• Compiler scans each basic block to derive GEN and KILL sets

Dataflow Equations

• $IN[b] = OUT[b1] \land ... \land OUT[bn]$  
  – where $b1, ..., bn$ are predecessors of $b$ in CFG
• $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$  
• $IN[entry] = 0000$  
• Result: system of equations
Solving Equations

- Use fixed point algorithm
- \( \text{IN}[\text{entry}] = 0000 \)
- Initialize \( \text{OUT}[b] = 1111 \)
- 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 track which equation applications may have further effect

Available Expressions Algorithm

for all nodes \( n \) in \( N \)
\[ \text{OUT}[n] = E; \quad // \text{OUT}[n] = E - \text{KILL}[n]; \]
\[ \text{IN}[\text{Entry}] = \emptyset; \quad \text{OUT}[\text{Entry}] = \text{GEN}[\text{Entry}]; \]
\[ \text{Worklist} = N - \{ \text{Entry} \}; \quad // N = \text{all nodes in graph} \]
while (Worklist \( \neq \emptyset \))
  choose a node \( n \) in Worklist;
  Worklist = Worklist - \{ n \};
\[ \text{IN}[n] = E; \quad // E = \text{set of all expressions} \]
for all nodes \( p \) in predecessors(\( n \))
\[ \text{IN}[n] = \text{IN}[n] \cap \text{OUT}[p]; \]
\[ \text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]; \]
if (\( \text{OUT}[n] \) changed)
  for all nodes \( s \) in successors(\( n \)) Worklist = Worklist \cup \{ s \};

Questions

- Does algorithm always halt?
- If expression is available in some execution, is it always marked as available in analysis?
- If expression is not available in some execution, can it be marked as available in analysis?
- In what sense is the algorithm conservative?

Duality In Two Algorithms

- Reaching definitions
  - Confluence operation is set union
  - \( \text{OUT}[b] \) initialized to empty set
- Available expressions
  - Confluence operation is set intersection
  - \( \text{OUT}[b] \) initialized to set of available expressions
- General framework for dataflow algorithms.
- Build parameterized dataflow analyzer once, use for all dataflow problems

Liveness Analysis

- A variable \( v \) is live at point \( p \) if
  - \( v \) is used along some path starting at \( p \), and
  - no definition of \( v \) along the path before the use.
- When is a variable \( v \) dead at point \( p \)?
  - No use of \( v \) on any path from \( p \) to exit node, or
  - If all paths from \( p \), redefine \( v \) before using \( v \).

What Use is Liveness Information?

- Register allocation.
  - If a variable is dead, we can reassign its register
- Dead code elimination.
  - Eliminate assignments to variables not read later.
  - But must not eliminate last assignment to variable (such as instance variable) visible outside CFG.
  - Can eliminate other dead assignments.
  - Handle by making all externally visible variables live on exit from CFG
Conceptual Idea of Analysis

• Simulate execution
• But start from exit and go backwards in CFG
• Compute liveness information from end to beginning of basic blocks

Liveness Example

• Assume a,b,c visible outside function
• So are live on exit
• Assume x,y,z,t are not visible
• Represent liveness using a bit vector
  – order is abcxyzt

Using Liveness Information for Dead Code Elimination

• Assume a,b,c visible outside function
• So are live on exit
• Assume x,y,z,t are not visible
• Represent liveness using a bit vector
  – order is abcxyzt

Formalizing Analysis

• Each basic block has
  – IN - set of variables live at start of block
  – OUT - set of variables live at end of block
  – USE - set of variables with upwards exposed uses in block
  – DEF - set of variables defined in block
• USE[x = z; x = x+1;] = { z } (x not in USE)
• DEF[x = z; x = x+1; y = 1;] = { x, y }
• Compiler scans each basic block to derive USE and DEF sets

Algorithm

OUT[Exit] = ∅;
IN[Exit] = USE[Exit];
for all nodes n in N - { Exit } IN[n] = ∅;
Worklist = N - { Exit };
while (Worklist ≠ ∅)
  choose a node n in Worklist;
  Worklist = Worklist - { n };
  OUT[n] = ∅;
  for all nodes s in successors(n) OUT[n] = OUT[n] ∪ IN[p];
  IN[n] = USE[n] ∪ (OUT[n] - DEF[n]);
  if (IN[n] changed)
    for all nodes p in predecessors(n) Worklist = Worklist ∪ { p };

Similar to Other Dataflow Algorithms

• Backwards analysis, not forwards
• Still have transfer functions
• Still have confluence operators
• Can generalize framework to work for both forwards and backwards analyses
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

Summary

- Basic blocks and basic block optimizations
  - Copy and constant propagation
  - Common sub-expression elimination
  - Dead code elimination
- Dataflow Analysis
  - Control flow graph
  - IN[b], OUT[b], transfer functions, join points
- Paired of analyses and transformations
  - Reaching definitions/constant propagation
  - Available expressions/common sub-expression elimination
  - Liveness analysis/Dead code elimination