MIT 6.035
Introduction to Program Analysis and Optimization

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
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
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
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
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;
}
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

```
s = 0;
a = 4;
```

```
s = 0;
a = 4;
```
s = 0;
a = 4;
i = 0;
k == 0
b = 2;
b = 1;
i < n
s = s + a*b;
i = i + 1;
return s;

s = 0;
a = 4;
s = 0;
a = 4;
i = 0;
k == 0

b = 2;
b = 1;
i < n

s = s + a*b;
i = i + 1;

return s;

s = 0;
a = 4;
i = 0;
s = 0;  
a = 4;  
i = 0;  
k == 0

b = 2;  
b = 1;  
i < n

s = s + a*b;  
return s;  
i = i + 1;

s = 0;  
a = 4;  
i = 0;  
k == 0
```c
s = 0;
a = 4;
i = 0;
k == 0
b = 2;
b = 1;
i < n
s = s + a*b;
i = i + 1;
return s;
	s = 0;
a = 4;
i = 0;
k == 0
b = 2;
```
s = 0;
a = 4;
i = 0;
k == 0

b = 2;
b = 1;
i < n

s = s + a*b;
return s;
i = i + 1;

s = 0;
a = 4;
i = 0;
k == 0

b = 2;
i < n
s = 0;
a = 4;
i = 0;
k == 0

b = 2;
b = 1;
i < n
s = s + a*b;
i = i + 1;

return s;

s = 0;
a = 4;
i = 0;
k == 0

b = 2;
i < n
s = s + a*b;
return s;
s = 0;
a = 4;
i = 0;
k == 0
b = 2;
i < n
s = s + a*b;
i = i + 1;

s = 0;
a = 4;
i = 0;
k == 0
b = 2;
i < n
s = s + a*b;
i = i + 1;
s = 0;
a = 4;
i = 0;
k == 0
b = 2;
i < n
s = s + a*b;
i = i + 1;
return s;

s = 0;
a = 4;
i = 0;
k == 0
b = 2;
i < n
s = s + a*b;
i = i + 1;
return s;
```c
s = 0;
a = 4;
i = 0;
k == 0
b = 2;

if (i < n)
    s = s + a*b;
i = i + 1;
return s;
```
```c
s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n
s = s + a*b;
i = i + 1;
return s;
```
s = 0;
a = 4;
i = 0;
k == 0

b = 2;
b = 1;
i < n

s = s + a*b;
i = i + 1;

return s;

s = 0;
a = 4;
i = 0;
k == 0

b = 2;
b = 1;
i < n

s = s + a*b;
i = i + 1;

return s;
s = 0;
a = 4;
i = 0;
k == 0

b = 1;
b = 2;
i < n

s = s + a*b;
i = i + 1;
return s;

s = 0;
a = 4;
i = 0;
k == 0

b = 2;
b = 1;
i < n

s = s + a*b;
i = i + 1;
return s;
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
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
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 Identities
  – 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;
Basic Block Analysis Approach

- Assume normalized basic block - all statements are of the form
  - \( \text{var} = \text{var} \text{ op } \text{var} \) (where \( \text{op} \) is a binary operator)
  - \( \text{var} = \text{op } \text{var} \) (where \( \text{op} \) is a unary operator)
  - \( \text{var} = \text{var} \)

- Simulate a symbolic execution of basic block
  - Reason about values of variables (or other aspects of computation)
  - Derive property of interest
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
**Original Basic Block**

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

**New Basic Block**

\[
\begin{align*}
a &= x + y \\
t1 &= a \\
b &= a + z \\
t2 &= b \\
b &= b + y \\
t3 &= b \\
c &= t2
\end{align*}
\]

**Var to Val**

\[
\begin{align*}
x &\rightarrow v1 \\
y &\rightarrow v2 \\
a &\rightarrow v3 \\
z &\rightarrow v4 \\
b &\rightarrow v6 \\
c &\rightarrow v5
\end{align*}
\]

**Exp to Val**

\[
\begin{align*}
v1 + v2 &\rightarrow v3 \\
v3 + v4 &\rightarrow v5 \\
v5 + v2 &\rightarrow v6
\end{align*}
\]

**Exp to Tmp**

\[
\begin{align*}
v1 + v2 &\rightarrow t1 \\
v3 + v4 &\rightarrow t2 \\
v5 + v2 &\rightarrow t3
\end{align*}
\]
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
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
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
  – $y = a+b; \ x = b; \ y = 1; \ z = a+x$ becomes
  – $y = a+b; \ t = y; \ x = b; \ y = 1; \ z = t$
  – Why? Because saves values away in temporaries
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:
  – \( t_1 = a+b; \) \( w = t_1+c; \) \( x = b; \) \( t_2 = a+x; \) \( y = t_2 + c; \) \( z = a+b; \)
  – CSE algorithm notices that
    • \( t_1+c \) and \( t_2+c \) compute same value
    • In the statement \( z = a+b, \) \( a+b \) has already been computed so generated code can reuse the result
  – \( t_1=a+b; \) \( w = t_1+c; \) \( t_3 = w; \) \( x = b; \) \( t_2=a+x; \) \( y = t_3; \) \( z = t_1; \)
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 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; 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
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

- Original
  
  \[
  \begin{align*}
  a &= x+y \\
  b &= a+z \\
  c &= x+y \\
  a &= b
  \end{align*}
  \]

- After CSE
  
  \[
  \begin{align*}
  a &= x+y \\
  t1 &= a \\
  b &= a+z \\
  t2 &= b \\
  c &= t1 \\
  a &= b
  \end{align*}
  \]

- After CSE and Copy Propagation
  
  \[
  \begin{align*}
  a &= x+y \\
  t1 &= a \\
  b &= a+z \\
  t2 &= b \\
  c &= a \\
  a &= b
  \end{align*}
  \]
## 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>
</tbody>
</table>

- tmp to var
  - \( t1 \rightarrow a \)

- var to set
  - \( a \rightarrow \{t1\} \)
Copy Propagation Example

Basic Block After CSE

\[
\begin{align*}
  a &= x+y \\
  t1 &= a \\
  b &= a+z \\
  t2 &= b
\end{align*}
\]

Basic Block After CSE and Copy Prop

\[
\begin{align*}
  a &= x+y \\
  t1 &= a \\
  b &= a+z \\
  t2 &= b
\end{align*}
\]

tmp to var

\[
\begin{align*}
  t1 &\rightarrow a \\
  t2 &\rightarrow b
\end{align*}
\]

var to set

\[
\begin{align*}
  a &\rightarrow \{t1\} \\
  b &\rightarrow \{t2\}
\end{align*}
\]
<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></td>
</tr>
</tbody>
</table>

**tmp to var**
- t1 → a
- t2 → b

**var to set**
- a → {t1}
- b → {t2}
**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>
</tbody>
</table>

**tmp to var**

| t1 → a |
| t2 → b |

**var to set**

| a → {t1} |
| b → {t2} |
### 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>a = b</td>
</tr>
<tr>
<td>tmp to var</td>
<td>var to set</td>
</tr>
<tr>
<td>t1 → a</td>
<td>a → {t1}</td>
</tr>
<tr>
<td>t2 → b</td>
<td>b → {t2}</td>
</tr>
</tbody>
</table>
### 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>(t_1 = a)</td>
<td>(t_1 = a)</td>
</tr>
<tr>
<td>(b = a + z)</td>
<td>(b = a + z)</td>
</tr>
<tr>
<td>(t_2 = b)</td>
<td>(t_2 = b)</td>
</tr>
<tr>
<td>(c = t_1)</td>
<td>(c = a)</td>
</tr>
<tr>
<td>(a = b)</td>
<td>(a = b)</td>
</tr>
<tr>
<td><strong>tmp to var</strong></td>
<td><strong>var to set</strong></td>
</tr>
<tr>
<td>(t_1 \rightarrow t_1)</td>
<td>(a \rightarrow \emptyset)</td>
</tr>
<tr>
<td>(t_2 \rightarrow b)</td>
<td>(b \rightarrow {t_2})</td>
</tr>
</tbody>
</table>
Dead Code Elimination

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

Basic Block Before

\[ a = x + y \]
\[ t1 = a \]
\[ b = a + z \]
\[ t2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE and Copy Prop

\[ a = x + y \]
\[ b = a + z \]
\[ c = a \]
\[ a = b \]
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
Basic Block After CSE and Copy Prop

\[
\begin{align*}
  a &= x+y \\
  t1 &= a \\
  b &= a+z \\
  t2 &= b \\
  c &= a \\
  \rightarrow \quad a &= b
\end{align*}
\]

Needed Set
\{b\}
Basic Block After CSE and Copy Prop

\[
\begin{align*}
a &= x+y \\
t1 &= a \\
b &= a+z \\
t2 &= b \\
c &= a \\
a &= b
\end{align*}
\]

Needed Set
\{a, b\}
Basic Block After CSE and Copy Prop

\[
\begin{align*}
  a &= x + y \\
  t1 &= a \\
  b &= a + z \\
  \rightarrow t2 &= b \\
  c &= a \\
  a &= b
\end{align*}
\]

Needed Set
\[\{a, b\}\]
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\}
Basic Block After CSE and Copy Prop

\[
\begin{align*}
a &= x + y \\
t1 &= a \\
\Rightarrow b &= a + z \\
c &= a \\
a &= b
\end{align*}
\]

Needed Set

\{a, b, 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, 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, z\}
Basic Block After CSE Copy Propagation, and Dead Code Elimination

\[ a = x + y \]
\[ b = a + z \]
\[ c = a \]
\[ a = b \]

Needed Set
\[ \{a, b, z\} \]
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, z\}
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
  – $a << 2 = a \times 4; a + a + a = 3 \times a$;

• Algebraic Simplification
  – $a = a \times 1; b = b + 0$;

• 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