An SSA-based Algorithm for Optimal Speculative Code Motion under an Execution Profile

Hucheng Zhou
Tsinghua University
June 2011

Joint work with: Wenguang Chen (Tsinghua University), Fred Chow (ICube Technology Corp.)
Contents

Basic Concepts
  PRE
  SSA
  SSAPRE
  Speculative Code Motion
MC-SSAPRE
  Algorithm
  Complexity
Experiments
Conclusion
Partial Redundancy Elimination (PRE)

- Eliminates expressions redundant on some (not necessarily all) paths
- One of the most important and widely applied target-independent global optimization
- Subsumes global common subexpression and loop invariant code motion

![Diagram of PRE]
PRE Facts

• Applied to each lexically identified expression independently
  – e.g $(a+b)$, $(a-b)$, $(a*c)$

• Formulated as a Placement problem:
  Step 1 – Determine where to perform insertions
    – Render more computations fully redundant
  Step 2 – Delete fully redundant computations

• Main challenge is in Step 1
The Most Popular PRE Algorithms

Lazy Code Motion (Knoop et. al)
- Computationally and Life-time Optimal
- Ordinary program representation
- Bit-vector-based iterative data flow analyses

SSAPRE
- Computationally and Life-time Optimal
- SSA form of program representation
- Sparse solution of data flow properties
- Subsumes local common subexpression
  - Insensitive to basic block boundaries
Static Single Assignment (SSA)

- Program representation with *built-in* use-def information
- Use-def edges factored at join points in CFG
- Use-def implicitly represented via unique names
- Each renamed variable has only one definition
Factored Redundancy Graph (FRG)

- Used in SSAPRE to represent redundant relationships among occurrences of the same expression via edges.
- The redundancy edges are factored as in SSA.
- Can view as SSA applied to expressions.
  - Effectively put the \( t \) storing the expression after PRE in SSA form.
Speculative Code Motion

Classical PRE only inserts at places where the expression is anticipated (down-safe)
– Many redundant computations cannot be eliminated

Speculative code motion *ignores* safety constraint
– Can remove more redundancies
– Not applicable to computations that may trigger runtime exceptions

![CFG diagram with speculative code motion example]
While Loop Example

Invariant code motion involves speculation
While Loop Restructuring

- The common solution
- Speculation no longer necessary
- But code size increases
Speculation not always beneficial

- Useless computations introduced for some paths
- Beneficial only if removed computations executed more frequently than inserted computations
- Requires execution frequency information

Non-beneficial because freq(B2) > freq(B4)
Problem Statement

How to minimize the dynamic execution count of an expression under an execution profile

• A more aggressive form of PRE
  – Classical PRE beneficial regardless of execution frequencies

• Cai and Xue (2003, 2006) first to apply min-cut to solve this problem optimally
  – Algorithm called MC-PRE
  – Uses bit-vector-based data flow analyses
  – Min-cut applied to CFG

• No SSA-based technique exists yet
Topic of this Paper

MC-SSAPRE – a new algorithm that yields optimal code placement under the SSAPRE framework

Overview:
• Form a essential flow graph (EFG) out of the FRG
• Map the BB execution frequencies to the EFG nodes
• Apply min-cut to the EFG
Algorithm Steps

SSAPRE Steps
• Construct FRG
  - Φ insertion
  - Rename
• Data Flow Attributes
  - DownSafety
  - WillBeAvail
• Book-keeping
  - Finalize
  - CodeMotion

MC-SSAPRE Steps
• Construct FRG
  o Φ insertion
  o Rename
• Form EFG and perform min-cut
  o Data flow
  o Graph reduction
  o Single source
  o Single sink
  o Minimum cut
  o WillBeAvail
• Book-keeping
  o Finalize
  o CodeMotion
Running example in SSA Form

Input Program

```
  a1+b1
  /    \
 B1 50  B2 20
  |      |
 B3 70  |  a1+b1 |
  |      |      |
 B4 50  B5 10  a1+b1
  |      |      |
 B8 60  B9 10  |  a1+b1 |
  |      |      |      |  exit |
 B12 60  B12 5  B10 5  |
  |      |      |      |      |
  a1+b1  exit  exit  exit
```

June 2011

MC-SSAPRE PLDI
FRG for Running Example

Introduce $h$ so the FRG can be viewed from an SSA perspective

Input Program

```
<table>
<thead>
<tr>
<th>B1</th>
<th>a1+b1</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B2</th>
<th>exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B3</th>
<th>a1+b1</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B6</th>
<th>a1+b1</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B7</th>
<th>exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B8</th>
<th>a1+b1</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B9</th>
<th>exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B12</th>
<th>exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B12</th>
<th>exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

FRG

```
<table>
<thead>
<tr>
<th>B1</th>
<th>h1</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B3</th>
<th>h2= Φ(h1,⊥)</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B4</th>
<th>h3</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B8</th>
<th>h4= Φ(h3,h2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B9</th>
<th>h4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

Input Program:
- $a1+b1$ at B1, B3, B5, B8, B12
- exit at B2, B6, B7, B9, B10, B12

FRG:
- $h1$ at B1
- $h2= Φ(h1,⊥)$ at B3
- $h3$ at B4
- $h4= Φ(h3,h2)$ at B8
- $h4$ at B9
Roles of Factored Redundancy Graph

- Insertions need to be considered only at $\Phi$’s
  - associated with the $\Phi$ operands
- Medium to compute data flow properties to disqualify more $\Phi$’s from being insertion candidates
- SSA form for $t$ (temporary to store the computed value) will be carved out of the FRG
- Three kinds of nodes:
  1. Real occurrences in original program
     - Def – always non-redundant
     - Use – partially redundant (including fully redundant)
  2. $\Phi$ (def)
  3. $\Phi$ operand (use) – can be $\perp$
Data Flow Properties for MC-SSAPRE

Fully available
• Insertions at these $\Phi$’s always unnecessary because the computed values are available

Partially anticipated
• Insertions should only be at these $\Phi$’s
• otherwise, the inserted computation would have no use
Graph Reduction

Use computed data flow properties to further narrow down the $\Phi$ candidates for insertion

Delete:
- $\Phi$’s that are fully available
- $\Phi$’s that are not partial anticipated
- Use nodes (real occurrences or $\Phi$ operands) that are fully redundant
- Edges from/to above nodes
Graph Reduction for Running Example

\[ h_1 \]

\[ h_2 = \Phi(h_1, \bot) \]

\[ h_3 \]

\[ h_4 = \Phi(h_3, h_2) \]

\[ h_4 \]

\[ h_2 = \Phi(h_1, \bot) \]

\[ h_2 \]

\[ h_2 \]

\[ h_4 = \Phi(h_3, h_2) \]

\[ h_4 \]

\[ h_2 \]

\[ h_2 \]

rg_excluded – fully redundant occurrences determined during Renaming
Form Essential Flow Graph (EFG)

- Introduce a virtual source node
  - Add an edge from it to each $\perp \Phi$ operand
- Introduce a virtual sink node
  - Add an edge from each real occurrence to it
- Result is a complete flow network
Edges in EFG

Edges to the sink are never insertion candidate
  – Mark with $\infty$ frequency

Other edges are:
  Type 1 edge – Edges ending at a $\Phi$ operand
  Type 2 edge – Edges from a $\Phi$ to a real occurrence
Mapping Frequencies to EFG Edges

- Model insertion at a Type 1 edge by inserting at exit of the predecessor BB corresponding to the Φ operand
  - Annotate the Type 1 edge by the node frequency of that predecessor BB
- Insertion at a Type 2 edge means performing the computation *in place*
  - Annotate the Type 2 edge by the frequency of the real occurrence
EFG annotated with Frequencies

Original Program

Final EFG

Type 1

Type 2

June 2011
Performing Minimum Cut

A minimum cut

- separates the flow network into two halves, such that
- the sum of the weights of the cut edges is minimized

By performing insertions at the cut edges, the number of execution of the computation is minimized
  - Implies computational optimality

If min-cut not unique, choose the cut nearest the sink
  - Induces life-time optimality
Our Example

- Two possible min-cuts
- Pick later red one
Final Result

**final transformed program**

```
min-cut  

h2 = \Phi(h1, \bot)  

h2  

h4 = \Phi(h3, h2)  

h4  

h2  

sink  

source  

a1 + b1  

a1 + b1  

t2 = a1 + b1  

t2  

t2  

t1 = a1 + b1  

t1  

t1  

exit  

exit  

exit  

sink  

exit
```
Complexity of MC-SSAPRE

MC-SSAPRE Steps

- Construct FRG
  - $\Phi$ insertion
  - Rename
- Form EFG and perform min-cut
  - Data flow
  - Graph reduction
  - Single source
  - Single sink
  - Minimum cut
  - WillBeAvail
- Book-keeping
  - Finalize
  - CodeMotion

$V$ – number of FRG nodes

$E$ – number of FRG edges

- Except the minimum cut step, all the steps are $O(V+E)$
- Performing minimum cut is $O(V^2 \sqrt{E})$
- In general,
  - $V_{cfg} > V_{frg} > V_{efg}$
Our Implementation

• Implemented MC-SSAPRE in the open source Path64 compiler, a descendent of the compiler with the original SSAPRE

• Leveraged existing SSAPRE infrastructure

• Resulting compiler will perform:
  – SSAPRE when no profile available
    • Perform speculation for loop-invariant computations
  – MC-SSAPRE with profile data

• Compiler always restructures while loops
Setup of Experiment 1

• Target is Intel Core™ i7-970 at 2.67GHz with 8MB cache
• Ubuntu 9.10
• With 6GB on board memory
• Compare run-time performances of all of SPEC CPU2006 (29 benchmarks)
• The 3 runs:
  SSAPRE – no speculation, no profile data
  SSAPRE_{sp} – loop-based speculation, no profile data
  MC-SSAPRE – speculation based on profile data
Experimental Results – CINT2006

- Average speedup of 2.13% over SSAPRE
- Average speedup of 2.25% over SSAPRE_{sp}
Experimental Results – CFP2006

- Average speedup of 2.76% over SSAPRE
- Average speedup of 1.96% over SSAPRE$_{sp}$
Setup of Experiment 2

- Calculate size of EFGs formed during MC-SSAPRE
- Same 29 SPEC CPU2006 benchmarks
- Target-independent
- Show
  - Optimization overhead in MC-SSAPRE
  - Impact of sparse approach
- Exclude empty EFGs
- Smallest EFG is 4 nodes:
  - Source, sink, $\Phi$, real occurrence
Sizes of EFGs

- 183,152 EFGs in the 29 SPEC CPU2006 benchmarks
- Near 50% of EFGs are only 4 nodes
- 86.5% of EFGs are less than 10 nodes
- 99.0% of EFGs are less than 50 nodes
- 24 EFGs larger than 300 nodes (largest size is 805)
Conclusion

• The minimum-cut technique for flow networks can effectively be applied to SSA graphs
• SSA-based compilers can apply MC-SSAPRE to achieve optimal speculative code motion under an execution profile
• The sparse approach is effective in reducing the problem sizes
• The polynomial time complexity of Min-cut only has limited effect on MC-SSAPRE’s optimization efficiency
• MC-SSAPRE always improves program performance over SSAPRE
Questions?