CSCI 565 Compiler Design

Spring 2016

Homework 4 - Solution

Problem 1: Control-Flow, Loops and Loop Optimizations [40 points]

Consider the code depicted below for a function with integer local variables i, a, b, c and d, and parameters of the enclosing procedure p.

```
00:    i = 0
01:    a = 0
02:    b = 1
03:    d = 1
04:    if (p = 0) goto L2
05:    L1 i = 0
06:    a = 1
07:    b = 1
08:    d = 1
09: L2: c = 0
10: L3: a = 0
11:    c = 0
12:    i = i + 1
13:    if (p = 0) goto L4
14:    A[c] = i
15:    c = c + 1
16:    goto L5
17:    L4: b = b + 1
18:    A[i] = a
19:    a = a + 1
20:    L5: i = i + 1
21:    d = A[i]
22:    B[i] = d
23:    if (i < 32) goto L3
24:    d = d + 1
25:    if (i < 32) goto L6
26: b = b + 1
27: c = c + 1
28:    i = 0
29:    goto L1
30: L6 A[i] = 0
31:    d = 1
32:    if (i < 16) goto L7
33:    goto L3
34: L7: return c
```

For this code fragment determine:

a. [05 points] The Control Flow Graph (CFG).
b. [10 points] The dominator tree and the loop(s) (identify the back edges).
c. [05 points] Use algebraic properties, copy propagation and dead code elimination to improve the execution of the body of the loop(s) identified in (b).
d. [10 points] Based on the code obtained in (c) recognize loop induction variables and transform the code to eliminate them as much as possible. Indicate which variables are basic induction variables and which are derived induction variables.
e. [10 points] Discuss if there are opportunities for loop invariant code motion in the original code versus the transformed code obtained in (d).

Solution:

a. [05 points] The Control Flow Graph (CFG) is as shown below (left).
b. [10 points] The dominator tree and the loops induced by the back edges (12,4) and (8,4) as shown below (right).
c. [05 points] There are several possible constant propagation and dead code elimination opportunities in this code. First, the assignment of "c = 0" on line 09 can be removed as it is post-dominated by another assignment to the same variable with the same value on line 11. This way the statement on line 14 can be simplified to "A[i] = i". Similarly, the assignment "a = 0" on line 10 can be propagated to the assignment on line 18. Lastly, the variable b is really never used for any purpose and so all references to it can be eliminated. The figure below depicts the simplified code with these transformations.
d. [10 points] The variable "i" is an induction variable in the shorter loop with an increment value of 2. There are only opportunities to use this induction variable in the calculation of the addresses used to access the array A[]. But it is tricky as there are two assignments to the variable "i" and only odd indices are used.

e. [10 points] In the transformed code there are really no opportunities for loop invariant computations.
Problem 2: Code Duplication Transformation [20 points]

A technique used to increase the size of the basic blocks so as to improve the effectiveness of instruction scheduling is basic block replication. In the illustrative example below, the basic block BB6 could be replicated thus creating two larger basic blocks {BB4;BB6a} and {BB5;BB6b} respectively.

In this problem you are asked to outline a code transformation algorithm that looks for opportunities for this code replication by inspecting the control-flow-graph and detecting situations in which basic blocks like C in this example can be replicated. Notice that you need to exploit the notion of dominance and redo the control-flow-graph to reflect the changes of the code. Also develop an algorithm that is recursive, i.e, it can apply the same analysis and transformation repeatedly over the transformed code. For the purpose of termination, you can assume that you have a function over the control-flow-graph which evaluates the profitability of replication of the code. Once this function returns a negative result, your algorithm should stop applying the replication transformation. Argue about the correctness of your transformation.

**Solution:**

The key observation regarding the control flow of both basic blocks BB4 and BB5 is that they are post-dominated by BB6. As such one should construct the post-dominator tree and examine if there is a node n that immediately post dominates two other nodes x and y such that in the control flow graph you have edges (x,n) and (y,n). If found, we can duplicate node n and merge the duplicates of n, say n_x and n_y, with the combined nodes {x,n_x} and {y,n_y} respectively, duplicating the edges of n as the outgoing edges of these two newly created nodes. Notice that the same notion can also trivially applied to two consecutive nodes in the CFG such as nodes 3 and 2 in the example above (should they not already be merged as part of the basic block construction algorithm).

For the CFG example above we would have the following post-dominator tree:
Given this insight the algorithm could be structured as follows:

1. build the post-dominator tree $T$ of the procedure’s CFG $G$;
2. compute the set of back-edges of $G$ as the set of edges $\{h,t\} \in \text{BE}$;
3. for every node $n$ in the post-order traversal of the post-dominator tree $T$ s.t. $n \neq h$ of any edge $\{h,t\}$ in $\text{BE}$ do
4. set $M = \{\text{set of all descendants } m \text{ of } n \text{ in } T \text{ such that only edges of } m \text{ are } (m,n) \in G\}$;
5. if $\text{MergeProfitable}(M, n)$ then
6. merge all $m \in M$ with replicas of $n$ to make combined nodes $(n,m)$ of $G$;
7. reset edges $(m,n)$ in $T$ to $(\text{parent}(n), (n,m))$;

For the example given above the algorithm would first pick the node 3 of the post-dominator tree $T$ and collapse the nodes $\{3 \text{ and } 2\}$ together. Then it would recognize the two nodes 4 and 5 that are post-dominated by 6 and collapse then as $\{4,6\}$ and $\{5,6\}$.

There are critical conditions in this algorithm in addition to the trivial conditions in lines 4. The need to exclude the back-edges is that we do not replicate the exit nodes of the loops as the nodes would be merged to the entry node of the loop which are the target of a goto instructions. This would violate the basic block principle. The condition in line 4 also prevents the needless duplication of nodes in control-flow conditions arising from if-then (without the else) section. Lastly, the traversal in post-order of the post-dominator tree is simple to ensure we apply the transformation backwards again the control flow and can thus update in a single pass the post-dominator tree.
Problem 3: Iterative Data Flow Analysis [40 points]

In this problem you will develop and show the application of the Live Variable analysis problem to the code of a given procedure. The live variable analysis problem seeks to determine for each scalar variable (or temporary register) if its contents is live. This has the fundamental application in the context of register allocation since if a given variable is no longer live at a given point then there is no need to keep it in register any longer.

Formally, a variable $v$ is live at an execution point $p$ if either its value is used at $p$ or there exists an execution path from $p$ to $q$ along which the value of $v$ is not written and is used in $q$.

a. [10 points] Formalize the live variable problem as an iterative data-flow analysis problem showing the equations for the $gen$ and $kill$ abstractions as well as the initialization of the values for each basic block and statement. In this section you need to determine if this is a forward or backward data flow problem.

b. [15 points] Apply your data flow problem formalization to the procedure code shown below (do not apply any transformations) showing the final result of the IN and OUT set of live variables for each basic block in the code.

c. [10 points] Review your work if you first apply constant propagation and dead-code elimination.

Solution:

a. [10 points]

<table>
<thead>
<tr>
<th>Domain:</th>
<th>Set of all variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction:</td>
<td>Backwards</td>
</tr>
<tr>
<td>Initial values:</td>
<td>Empty set as Out sets (at the output of each node since the problem flows backwards)</td>
</tr>
<tr>
<td>Function:</td>
<td>$Gen = { v \mid v \text{ is used on the RHS before being redefined in the block} }$</td>
</tr>
<tr>
<td></td>
<td>$Kill = { v \mid v \text{ is defined on the LHS of a statement in the block} }$</td>
</tr>
<tr>
<td>$Out(b) = \bigcup_{s \in succ(b)} In(s)$, $In(b) = Gen(b) \cup (Out(b) - Kill(b))$</td>
<td></td>
</tr>
</tbody>
</table>
b. [15 points]

Assuming the output of BB8 has no live variables, i.e., the empty set at the first iteration we compute the In sets for all basic blocks. In the subsequent iterations we compute the values for the In and Out sets for each basic block for the various iterations of the analysis assuming a post-topological sorting order traversal pattern of {8, 7, 6, 5, 4, 3, 2, 1}. For simplicity we assume that N is a compile-time constant and not a variable.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>BB1</th>
<th>BB2</th>
<th>BB3</th>
<th>BB4</th>
<th>BB5</th>
<th>BB6</th>
<th>BB7</th>
<th>BB8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 In</td>
<td>{}</td>
<td>{i}</td>
<td>{i}</td>
<td>{t}</td>
<td>{i}</td>
<td>{a, i}</td>
<td>{i}</td>
<td>{t}</td>
</tr>
<tr>
<td></td>
<td>Out</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>1 In</td>
<td>{}</td>
<td>{t, i}</td>
<td>{t, i}</td>
<td>{t, i}</td>
<td>{i}</td>
<td>{a, i}</td>
<td>{i}</td>
<td>{t}</td>
</tr>
<tr>
<td></td>
<td>Out</td>
<td>{t, i}</td>
<td>{t, i}</td>
<td>{t, i}</td>
<td>{i}</td>
<td>{a, i}</td>
<td>{i}</td>
<td>{}</td>
</tr>
<tr>
<td>2 In</td>
<td>{}</td>
<td>{t, i}</td>
<td>{t, i}</td>
<td>{t, i}</td>
<td>{i}</td>
<td>{a, i}</td>
<td>{i}</td>
<td>{t}</td>
</tr>
<tr>
<td></td>
<td>Out</td>
<td>{t, i}</td>
<td>{t, i}</td>
<td>{t, i}</td>
<td>{i}</td>
<td>{a, i}</td>
<td>{i}</td>
<td>{}</td>
</tr>
</tbody>
</table>

c. [10 points]

If we were to apply constant propagation and dead-code elimination, then assignment a = 0 in BB1 and a = 1 in BB2 would be removed and the use of a in BB6 would be replaced with the only definition reaching that use which has the value 1. As a result the solution to the live variable analysis would be the same with the variable a removed from the set of live variables.