Partial Redundancy Elimination using Lazy Code Motion

Sandeep Dasgupta[1]   Tanmay Gangwani[2]

April 11, 2014

[1]Electronic address: `sdasgup3@illinois.edu`
[2]Electronic address: `gangwan2@illinois.edu`

Value Numbering

Prior research has shown that value numbering can increase opportunities for PRE. LLVM presently has a GVN-PRE pass which exploits this. However, value numbering in GVN-PRE is tightly coupled with the code for removing redundancies, and hence we deem it not useful for our purposes. We have written our own value numbering pass which feeds expression value numbers to the PRE stage. It should be noted, however, that we do not implement value numbering from scratch and use an old (now defunct) LLVM pass as a starting point. Most importantly, we augment the basic value numbering in the following ways -

- Add the notion of leader expression (described below), with associated data structures and functions.

- Functionality to support value-number-based bitvectors rather than expression-name-based bitvectors.

- (Optimization 1) If the expression operator is one of these - AND, OR, CMP::EQ or CMP::NE, and the operands have the same value number, we replace all uses accordingly and then delete the expression.

- (Optimization 2) If all operands of an expression are constants, then we evaluate and propagate constants.

- (Optimization 3) If one operand of an expression is a constant (0 or 1), then we simplify the expression. e.g. a+0 = a , b*1 = b.

- (Optimization 4) If the incoming expressions to a `Phi` node have the same value number, then the `Phi` node gets that same value number

As per our testing, optimizations 2 and 3 are also done by the reassociation pass in LLVM. In our final code we omit our implementation and rely on the more robust LLMV pass. Optimizations 1 and 4, however, are still our contribution.

Notion Of Leader Expression

The value numbering algorithm computes the RPO solution as outlined in [1]. It goes over the basic blocks in reverse post order and adds new expressions to a hash table based on the already computed value numbers of the operands. We call an expression a 'leader' if at the time of computing its value number, the value number doesn't already exist in the hash table. In other words, out of a potentially large set of expressions that map to a particular value number, the leader expression was the first to be encountered while traversing the function in reverse post order. Leader expressions are vital to our algorithm as they are used to calculate the block local properties of the dataflow equations.

Types Of Redundancies

Given two expression X and Y in the source code, following are the possibilities -
   1. X and Y are lexically equivalent, and have the same value numbers
   2. X and Y are lexically equivalent, but have different value numbers
   3. X and Y are lexically different, but have the same value numbers
   4. X and Y are lexically different, and have different value numbers
In the source code, there could be opportunities for redundancy elimination in cases 1, 2 and 3 above. If the source code is converted to an intermediate representation in SSA form then case 2 becomes an impossibility (by guarantees of SSA). Therefore, our algorithm presently handles the cases when X and Y are lexically same/different, but both have the same value number (cases 1

and 3). Driven by this observation, we implement value number based code motion, the details of which are presented below. It should be noted that even though case 2 above is not possible in SSA, the source code redundancies of this type transform into that of type case 4. Figure 1 presents an illustration of the same. **One of the items for future work in the semester is to handle this case.**

Value-Number driven code motion

We implement an iterative bit vector data flow algorithm for PRE. We initially implemented the flow equations from the Lazy Code Motion paper. This set included a total of 13 bit vectors for each basic block - 2 for block local properties ANTLOC and TRANSP, and 11 for global properties. These equations, however, could only be applied to single instruction basic blocks. We therefore, derived a new set of equations which are motived by later work[2] of the same authors. This set of equations apply to maximal basic blocks and entails a total of 19 bit vectors for each basic block in our current implementation - 3 for block local properties ANTLOC, TRANSP, XCOMP and 16 for global properties. We include the equations in appendix A and B. In appendix C, we provide our generalized data flow framework, and show how each PRE equation maps to the framework. We call the algorithm value-number driven because each slot in each of the bit vectors is reserved for a particular value number rather than a particular expression. Also, we make the observation that a large number of expressions in the program only occur once, and are not useful for PRE. Hence to further optimize for space and time, we only give bit vector slots to value numbers which have more than one expression linked to them.

Local CSE

For our data flow equations to work correctly, a local CSE pass has to be run on each basic block. Basically, this path removes the redundancies inside straight line basic block code and sanitizes it for the iterative bit vector algorithm. This idea is borrowed from [2]. We perform this step before calling our data flow framework.

Status

We summarize accomplished work in this section. We completed the value numbering pass and included some optimizations as part of it. We then derived PRE equations for maximal basic block CFG and molded them to work with value numbers rather than lexical names. This was followed by construction of a generalized data flow framework which could incorporate PRE equations. More specifically, we define a single function which can be called with dataflow equation specific parameters. We also wrote a pass to perform local CSE on each basic block. We have tested our implementation of small code segments with good amount of success. We are able to identify the placements in the CFG which are computationally optimal as well as lifetime optimal. As an example, we present the CFG for a fairly involved program we wrote using goto statements(Figure 2). The figure on the right illustrates the placement suggestions provided by our PRE pass.

Ongoing Work and Future Tasks

Having identified the INSERT points (where computation should be added) and REPLACE points (from where computation should be removed), we are now working on doing the actual IR modifications. Once completed we would begin with testing on fairly large codes for 1) correctness and 2) performance improvements. This would be followed by a comparison with the LLVM GVN-PRE pass. Time permitting, we would like to provide support for eliminating redundancies where two expressions are lexically different and have different value numbers (mentioned in section **Types Of Redundancies**).
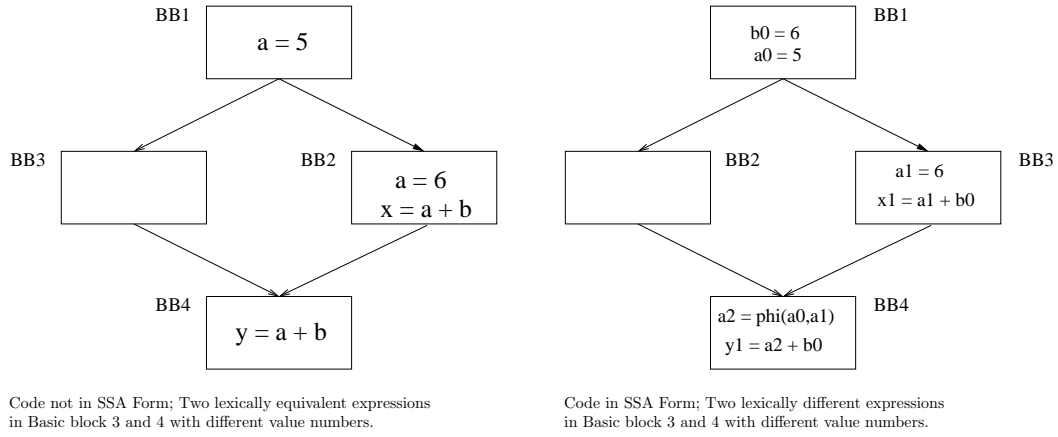
BB1

a = 5

BB3

BB2

a = 6
x = a + b

BB4

y = a + b

Code not in SSA Form; Two lexically equivalent expressions
in Basic block 3 and 4 with different value numbers.

BB1

b0 = 6
a0 = 5

BB2

BB3

a1 = 6
x1 = a1 + b0

BB4

a2 = phi(a0,a1)
y1 = a2 + b0

Code in SSA Form; Two lexically different expressions
in Basic block 3 and 4 with different value numbers.

Figure 1

c = a;

y = a + b;

y = a + b;

y = a + b;

z = a + b;

x = a + b;

The motivating example

c = a;

y = a + b;

h = a + b;

y = h;

h = a + b;
y = h;

h = a + b;

z = h;

x = h;

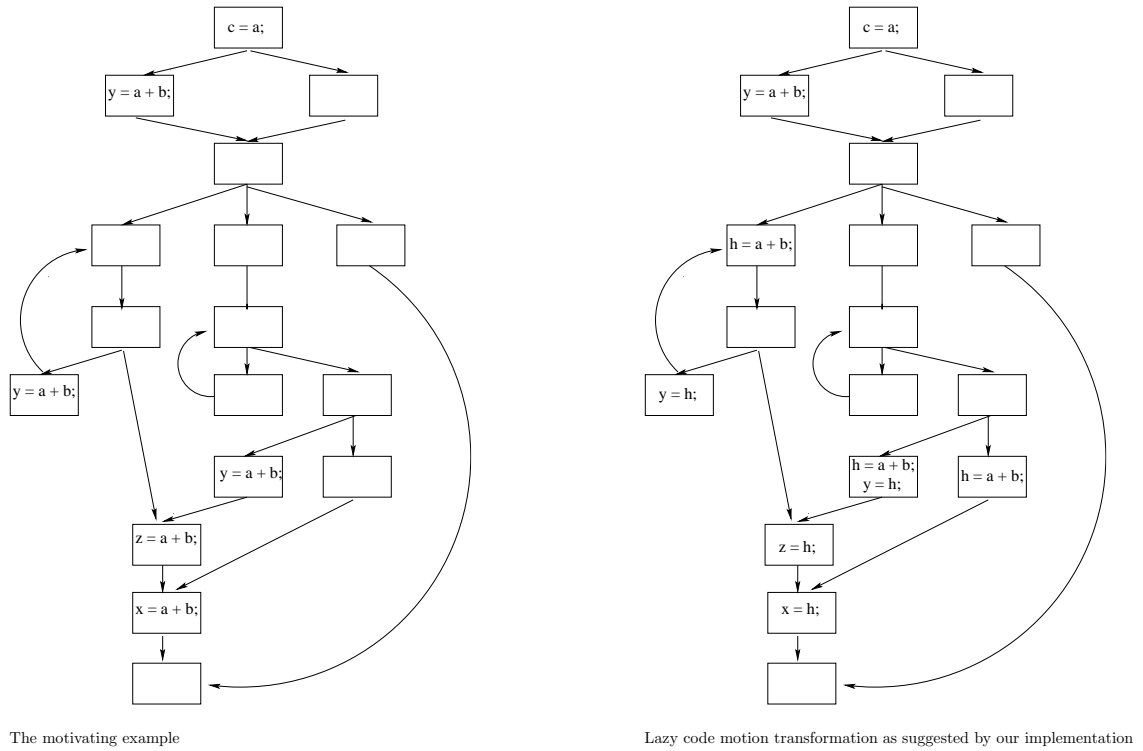Lazy code motion transformation as suggested by our implementation

Figure 2

Bibliography

[1] K. D. COOPER AND L. T. SIMPSON, *Scc-based value numbering*, Software Practice and Experience, 27 (1995), pp. 701–724.

[2] J. KNOOP, O. RÜTHING, AND B. STEFFEN, *Optimal code motion: Theory and practice*, ACM Trans. Program. Lang. Syst., 16 (1994), pp. 1117–1155.