Partial Redundancy Elimination using Lazy Code Motion

Sandeep Dasgupta[1] Tanmay Gangwani[2]

May 10, 2014

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

Problem Statement & Motivation

Partial Redundancy Elimination (PRE) is a compiler optimization that eliminates expressions that are redundant on some but not necessarily all paths through a program. PRE is a powerful technique since it subsumes Common Subexpression Elimination (CSE) and Loop Invariant Code Motion (LICM).

In the example below, the computation of the expression (a + b) is partially redundant because it is redundant on the path $1 \rightarrow 2 \rightarrow 5$, but not on the path $1 \rightarrow 4 \rightarrow 5$. PRE works by first introducing operations that make the partially redundant expressions fully redundant and then deleting the redundant computations. The computation of (a + b) is added to 4 and then deleted from 5.

```
(1) if (OPAQUE)
(2)   x = a + b;
(3) else
(4)   x = 0;
(5) y=a+b;
```

TO DO: Helps in loop invariant code motion

Related Work

Below we summarize the relevant literature and outline our approach to implementing a PRE transformation pass in LLVM (`http://llvm.org/`).

Partial Redundancy Elimination

Morel et al. [10] first proposed a bit-vector algorithm for the suppression of partial redundancies. The bi-directionality of the algorithm, however, proved to be computationally challenging. Knoop et al. [8] solved this problem with their Lazy Code Motion (LCM) algorithm. It is composed of uni-directional data flow equations and provides the earliest and latest placement points for operations that should be hoisted. Drechsler et al. [6] present a variant of LCM which they claim to be more useful in practice. Briggs et al. [2] allude to two pre-passes to make PRE more effective - Global Reassociation and Value Numbering.

Value Numbering

Briggs et al. [3] compare and contrast two techniques for value numbering - hash based[4] and partition based[1]. In subsequent work they provide SCC-based Value Numbering [5] which combines the best of the previously mentioned approaches. Cooper et al. [11] show how to incorporate value information in the data flow equations of LCM to eliminate more redundancies.

LLVM

Since LLVM is a Static Single Assignment (SSA) based representation, algorithms based on identifying expressions which are lexically identical or have the same static value number may fail to capture some redundancies. Keneddy Chow et al. [7] provide a new framework for PRE on a program in SSA form. The present GVN-PRE pass in LLVM appears to be inspired by the work of Thomas et al. [12] which also focusses on SSA.

Our Aproach

- In this project, we implemented the iterative bit-vector data flow algorithm inspired by LCM [8]. It uses five data flow equations to identify optimal placement points for hoisting. The first two - anticipatable and earliest - calculate the computationally optimal placement

and the next three - latest (in 2 steps) and isolated - calculate the lifetime optimal placement. The final step of the algorithm entails inserting new code and removing redundant expressions. All critical edges[1] are removed prior to starting LCM.

- As suggested in [2], running two pre-passes produces code more amenable to PRE. We plan to use an already existing LLVM pass (-reassociate) for Global Reassociation. We have implemented our own pass for Value Numbering using the approach outlined in [5]. This is based on Tarjan's SCC algorithm. It works on the strongly connected components of the SSA graph as opposed to the CFG. The algorithm combines the ability to perform constant folding and algebraic simplifications with the ability to make optimistic assumptions (and later disprove them). Because of this, it can find at least as many equivalences as the hashing or partitioning based approach.
- To incorporate information from the Value Numbering pass into the LCM pass, we modified the flow equations to work on *values* rather than lexical names. [11] details how the flow equations change for a variant of LCM. We make the observation that only ANTLOC and TRANSP predicates would need to be suitably modified to handle *values*.

The following is the detailed explanation od our approach.

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

[1] Edges from blocks with more than one successor to blocks with more than one predecessor

The value numbering algorithm computes the RPO solution as outlined in [5]. 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. **This is not handled in the current implementation.**

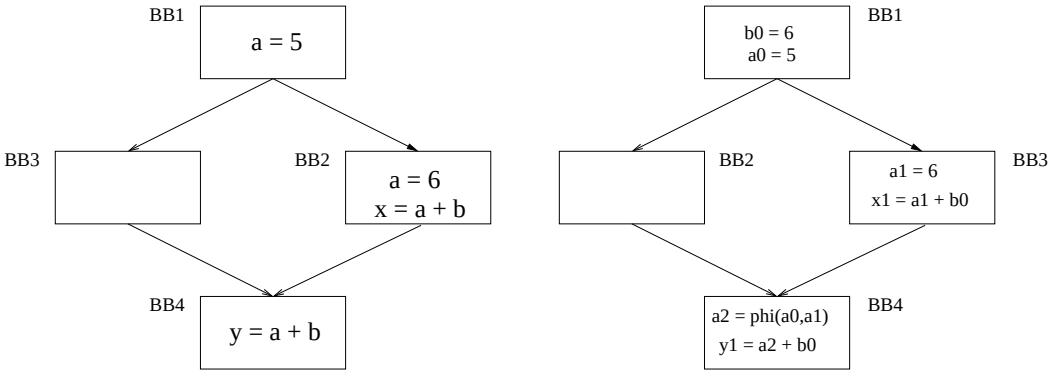

Figure 1

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[9] 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 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 [9]. We perform this step before calling our data flow framework. TO DO: Why needed.

Insert and Replace

To maintain compatibility with SSA, we perform insertion and replacement through memory and re-run the *mem2reg* pass after our PRE pass to convert the newly created load and store instructions to register operations. Following are the major points:

- Assign stack space *(allocas)* at the beginning of the function for all the expressions that need movement
- At insertion point, compute the expression and save the value to the stack slot assigned to the expression
- At replacement point, load from the correct stack slot, replace all uses of the original expression with the load instruction, and delete the original expression
- *mem2reg* converts stack operations to register operations and introduces the necessary Φ instructions

In appendix D, we have shown that in FigureD.1 and FigureD.2, the optimizations performed by our PRE pass. The intention here is to show how our version of PRE performed on the computations $a + b$ & $a < b$;

One point about the insertion step is worth mentioning. Suppose that an expression, with value number vn, is to be inserted in a basic block. Although our algorithm can handle all cases, for simplicity, assume that the insertion point is the end of the basic block. To insert the expression we scan the list of the expressions in the whole function which have the same value number vn. We then clone one of these expressions (called provider) and place at the end of the basic block. The trivial case is when the the provider is available in the same basic block. If however, the provider comes from another basic block, then we need to ensure that the operands of the provider dominate the basic block where we wish to insert the expression in. Not being able to find a suitable provider is the only case where we override the suggestion of the data flow analysis and not do PRE for that expression only. PRE for other expressions proceeds as usual. Our initial testing suggests that this is a very rare occurrence.

Loop Invariant Code Motion

We first address a question raised in the phase-1 report. We had previously mentioned that to optimize for space and time, we provide a bit vector slot only to value numbers which have more than one expression linked to them. With this, however, we could miss opportunities for loop invariant code motion. As a solution, we extend the bit vector to include value numbers which have only a single expression linked to them but only if the expression is inside a loop. Note

that we still exclude the cases where the expression is not part of a loop, and expect reasonable space and time savings.

The second issue we resolved with respect to loop invariant code motion was checking for zero-trip loops. Our PRE algorithm would move the loop invariant computations to the loop pre-header only if placement in the loop pre-header is anticipatible. Such a pre-header is always available for *do-while* loops, but not for *while* and *for* loops. Hence, a modification is required to the structure of *while* and *for* loops which peels off the first iteration of the loop, protected by the loop condition. This alteration provides PRE with a suitable loop pre-header to hoist loop independent computations to. In Figure 2 we show the CFG changes. Fortunately, we were able to achieve this effect using an existing LLVM pass *-loop-rotate* rather than having to write it ourselves.

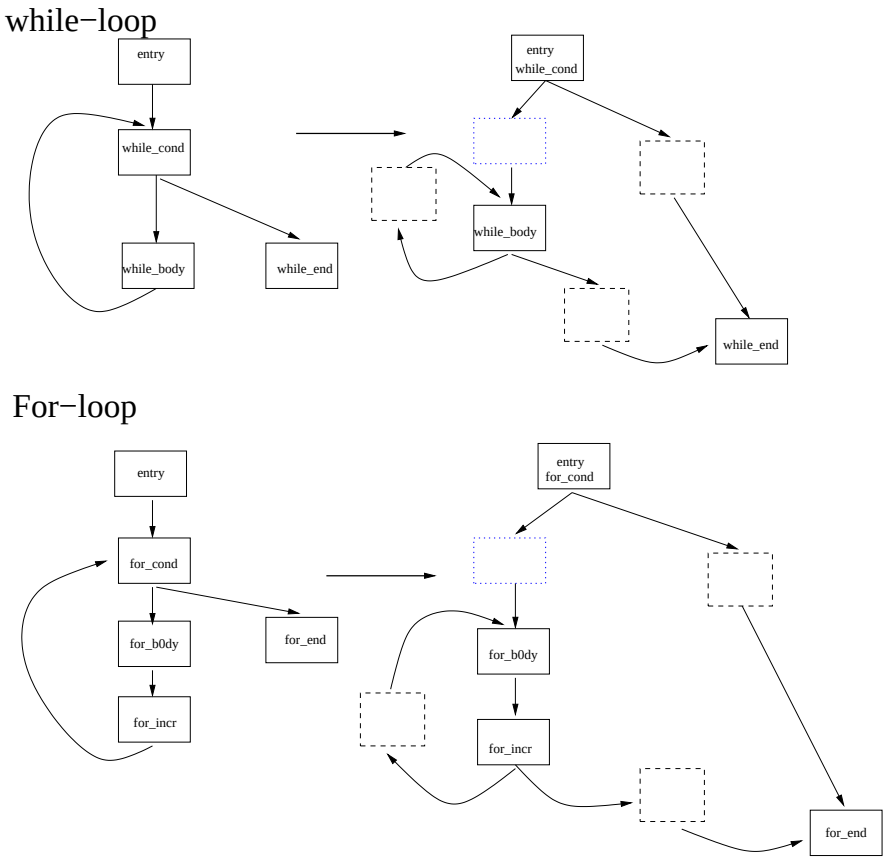

Figure 2: Loop transformations done by *-loop-rotate*. *do-while* loops remain unaffected. Blue dotted boxes are the ones inserted by loop rotate. PRE can insert the computations in these places.

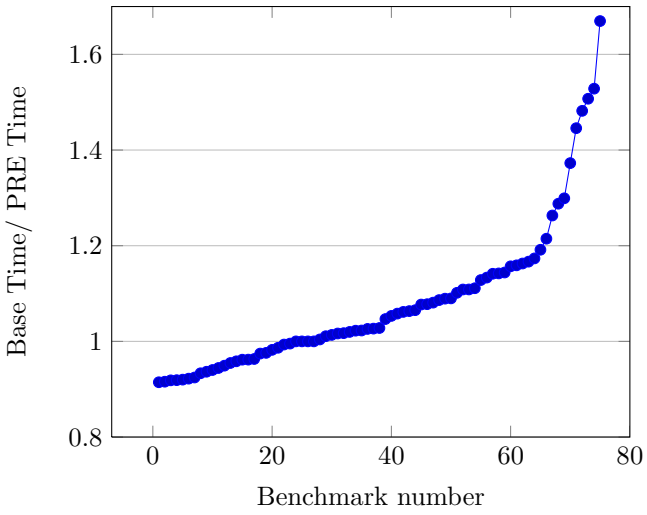

Figure 3: S-curve for performance improvements over Baseline (no PRE)

Testing

Apart from the small test cases which we created in phase-1, we have been able to successfully test our PRE pass on real codes from the LLVM test-suite. Specifically, we have completed the testing for 75 benchmarks from LLVM single-source package *"test-suite/SingleSource/Benchmarks/"* for correctness and performance. Correctness is checked by comparing the output of the binary optimized with our PRE pass with the reference output provided along with benchmarks. All benchmarks pass the correctness test. For performance, we compile the codes with and without the PRE pass and take the ratio of the run-times on the hardware. Figure 4 shows the S-curve for performance. For 55/75 benchmarks, our pass improves the run time by varying amounts (upto 67%). Performance drops for few benchmarks, but the degradation is bound by 8%.

Apart from measured runtime on the hardware, another metric to quantify the effects of our optimization pass is the dynamic instruction count. We use Pin (dynamic binary instrumentation tool) from Intel for this purpose. We have written a very simple Pintool to dump the dynamic instruction count of each type of instruction. We would present supporting data from Pin our final report.

Conclustion and Future Work

In this phase, we were able to code the missing pieces of our PRE algorithm, thereby achieving full functionality of the project. The highlights were the insert-replace algorithm, LICM improvements, and fixes for numerous bugs which surfaced during testing on the LLVM single-source package. We have written scripts to automate the testing, and this would speed up work for the final phase.

The S-curve in this report (Figure 5) presents improvements with respect to the baseline (no PRE pass) for single-source package. We would like to evaluate the cases where our pass degrades performance compared to baseline. In our final report, we plan to include similar curves for benchamarks from the LLVM multi-source package as well as the SPEC 2006 suite. Also, we would include S-curves to compare the effectiveness of our PRE pass with the LLVM GVN-PRE pass. For extreme outliers, we hope to present supporting data to reason about the performance change. Pin tool analysis and the statistics dumped by our PRE pass would be used for this.

Appendix A

Computation of localized sets

For each basic block there are 3 bit vectors dedicated to the block-specific properties, namely `Transp`, `Antloc` and `Xcomp`. As mentioned before, a bit vector is a boolean array of value numbers. Each value number is associated with at-least two expressions from the IR. Let the leader expression (as defined in the section on value numbering) associated with the value number v be called $L(v)$.

$$
\begin{aligned}
\texttt{Transp(v,B)} &= \begin{cases} false & \text{iff } \exists \text{ x} \in \text{operands of L(v) such that } \texttt{Mod(x,B)} = \text{true} \\ true & \text{Otherwise} \end{cases} \\
\texttt{Antloc(v,B)} &= \texttt{Eval(v,B)} \cap \overline{\texttt{Transp(v,B)}} \\
\texttt{Xcomp(v,B)} &= \texttt{Eval(v,B)} \cap \overline{\texttt{Transp(v,B)}}
\end{aligned}
$$

where

$$
\begin{aligned}
\texttt{Eval(v,B)} &= \{\text{v} \mid \text{value number v is computed in B}\} \\
\texttt{Mod(op,B)} &= \text{operand op modified in B}
\end{aligned}
$$

(A.1)

Appendix B

Lazy Code motion Transformations

- Down Safety Analysis (Backward data flow analysis)

$$\begin{aligned} \texttt{Antin(b)} &= \texttt{Antloc(b)} \cup (\texttt{Tranp(b)} \cap \texttt{Antout(b)}) \\ \texttt{Antout(b)} &= \texttt{Xcomp(b)} \cup \begin{cases} \phi & \text{if b = exit} \\ \bigcap\limits_{s \in succ(b)} \texttt{Antin(s)} \end{cases} \end{aligned} \tag{B.1}$$

- Up Safety Analysis (Forward data flow analysis)

$$\begin{aligned} \texttt{Availin(b)} &= \begin{cases} \phi & \text{if b = entry} \\ \bigcap\limits_{p \in pred(b)} (\texttt{Xcomp(p)} \cup \texttt{Availout(p)}) \end{cases} \\ \texttt{Availout(b)} &= \texttt{Tranp(b)} \cap (\texttt{Antloc(b)} \cup \texttt{Availin(b)}) \end{aligned} \tag{B.2}$$

- Earliest-ness (No data flow analysis)

$$\begin{aligned} \texttt{Earliestin(b)} &= \texttt{Antin(b)} \cap \bigcap_{p \in pred(b)} (\overline{\texttt{Availout(p)} \cup \texttt{Antout(p)}}) \\ \texttt{Earliestout(b)} &= \texttt{Antout(b)} \cap \overline{\texttt{Tranp(b)}} \end{aligned} \tag{B.3}$$

- Delayability (Forward data flow analysis)

$$\begin{aligned} \texttt{Delayin(b)} &= \texttt{Earliestin(b)} \cup \begin{cases} \phi & \text{if b = entry} \\ \bigcap\limits_{p \in pred(b)} (\overline{\texttt{Xcomp(p)}} \cap \texttt{Delayout(p)}) \end{cases} \\ \texttt{Delayout(b)} &= \texttt{Earliestout(b)} \cup (\texttt{Delayin(b)} \cap \overline{\texttt{Antloc(b)}}) \end{aligned} \tag{B.4}$$

- Latest-ness (No data flow analysis)

$$\begin{aligned} \texttt{Latestin(b)} &= \texttt{Delayin(b)} \cap \texttt{Antloc(b)} \\ \texttt{Latestout(b)} &= \texttt{Delayout(b)} \cap (\texttt{Xcomp(b)} \cup \bigcup_{s \in succ(b)} \overline{\texttt{Delayin(s)}}) \end{aligned} \tag{B.5}$$

- Isolation Analysis (Backward data flow analysis)

$$\begin{array}{lcl} \texttt{Isolatedin(b)} & = & \texttt{Earliestout(b)} \cup \texttt{Isolatedout(b)} \\ \texttt{Isolatedout(b)} & = & \begin{cases} U & \text{if b = exit} \\ \bigcap\limits_{s \in succ(b)} (\texttt{Earliestin(s)} \cup (\overline{\texttt{Antloc(s)}} \cap \texttt{Isolatedin(s)})) \end{cases} \end{array} \tag{B.6}$$

- Insert and Replace points

$$\begin{array}{lcl} \texttt{Insertin(b)} & = & \texttt{Latestin(b)} \cap \overline{\texttt{Isolatedin(b)}} \\ \texttt{Insertout(b)} & = & \texttt{Latestout(b)} \cap \overline{\texttt{Isolatedout(b)}} \\ \texttt{Replacein(b)} & = & \texttt{Antloc(b)} \cap \overline{\texttt{Latestin(b)} \cap \texttt{Isolatedin(b)}} \\ \texttt{Replaceout(b)} & = & \texttt{Xcomp(b)} \cap \overline{\texttt{Latestout(b)} \cap \texttt{Isolatedout(b)}} \end{array} \tag{B.7}$$

Appendix C

Generalized data flow framework

All the equations in Appendix B can be computed using the generic framework defined below.

C.1 Forward Analysis

$$\begin{aligned} \texttt{In(b)} &= \alpha(b) \cup \begin{cases} \perp & \text{if b = entry} \\ \bigwedge\limits_{p \in pred(b)} \beta(p) \end{cases} \\ \texttt{Out(b)} &= \gamma(b) \end{aligned} \tag{C.1}$$

C.2 Backward Analysis

$$\begin{aligned} \texttt{In(b)} &= \gamma(b) \\ \texttt{Out(b)} &= \alpha(b) \cup \begin{cases} \perp & \text{if b = exit} \\ \bigwedge\limits_{s \in succ(b)} \beta(s) \end{cases} \end{aligned} \tag{C.2}$$

The following is the function which we call with dataflow equation specific parameters defined subsequently.

$$\texttt{callFramework}(\texttt{Out(b)}, \texttt{In(b)}, \alpha(b), \beta(b), \gamma(b), \bigwedge, \perp, \top, \text{Direction})$$

Following is the list of values that we need to plug-in to α, β and γ for the above generic framework to work.

- Down Safety Analysis (Backward data flow analysis)

$$\begin{array}{lcl} \alpha(x) & = & \texttt{Xcomp(x)} \\ \beta(x) & = & \texttt{Antin(x)} \\ \gamma(x) & = & \texttt{Tranp(x)} \cap \texttt{Antout(x)} \cup \texttt{Antloc(x)} \\ \bigwedge & = & \cap \\ \perp & = & \phi \\ \top & = & V, \text{set of all values} \\ \text{Direction} & = & \text{Backward} \end{array} \tag{C.3}$$

- Up Safety Analysis (Forward data flow analysis)

$$\begin{array}{lcl} \beta(x) & = & \texttt{Xcomp(x)} \cup \texttt{Availout(x)} \\ \gamma(x) & = & \texttt{Antloc(x)} \cup \texttt{Availin(x)} \cap \texttt{Tranp(x)} \\ \bigwedge & = & \cap \\ \bot & = & \phi \\ \top & = & V, \text{set of all values} \\ \text{Direction} & = & \text{Forward} \end{array} \tag{C.4}$$

- Delayability (Forward data flow analysis)

$$\begin{array}{lcl} \alpha(x) & = & \texttt{Earliestin(x)} \\ \beta(x) & = & \overline{\texttt{Xcomp(x)}} \cap \texttt{Delayout(x)} \\ \gamma(x) & = & \texttt{Delayin(x)} \cap \overline{\texttt{Antloc(x)}} \cup \texttt{Earliestout(x)} \\ \bigwedge & = & \cap \\ \bot & = & \phi \\ \top & = & V, \text{set of all values} \\ \text{Direction} & = & \text{Forward} \end{array} \tag{C.5}$$

- Isolation Analysis (Backward data flow analysis)

$$\begin{array}{lcl} \beta(x) & = & \overline{\texttt{Antloc(x)}} \cap \texttt{Isolatedin(x)} \cup \texttt{Earliestin(x)} \\ \gamma(x) & = & \texttt{Earliestout(x)} \cup \texttt{Isolatedout(x)} \\ \bigwedge & = & \cap \\ \bot & = & V, \text{set of all values} \\ \top & = & V, \text{set of all values} \\ \text{Direction} & = & \text{Backward} \end{array} \tag{C.6}$$

Appendix D

An Extended Example

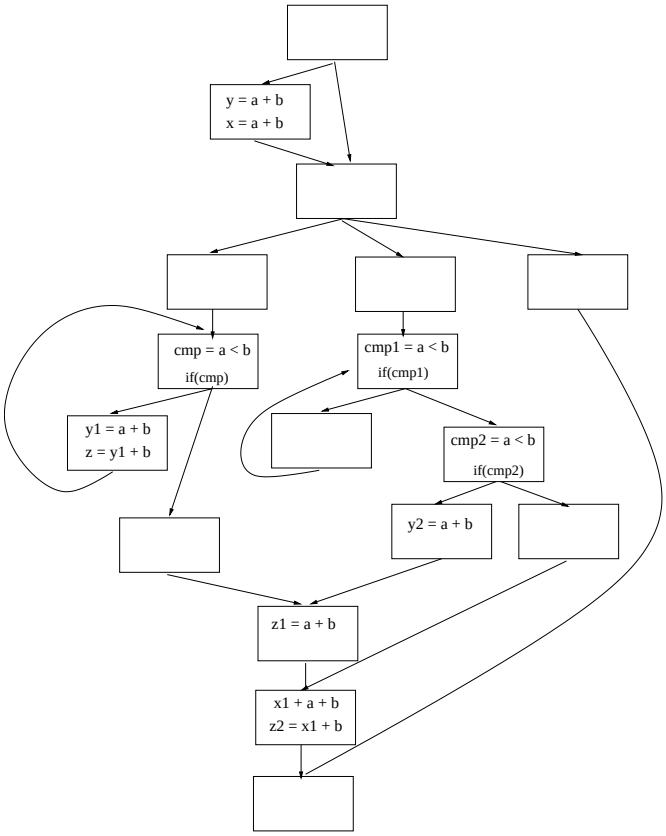

Figure D.1: A motivating example

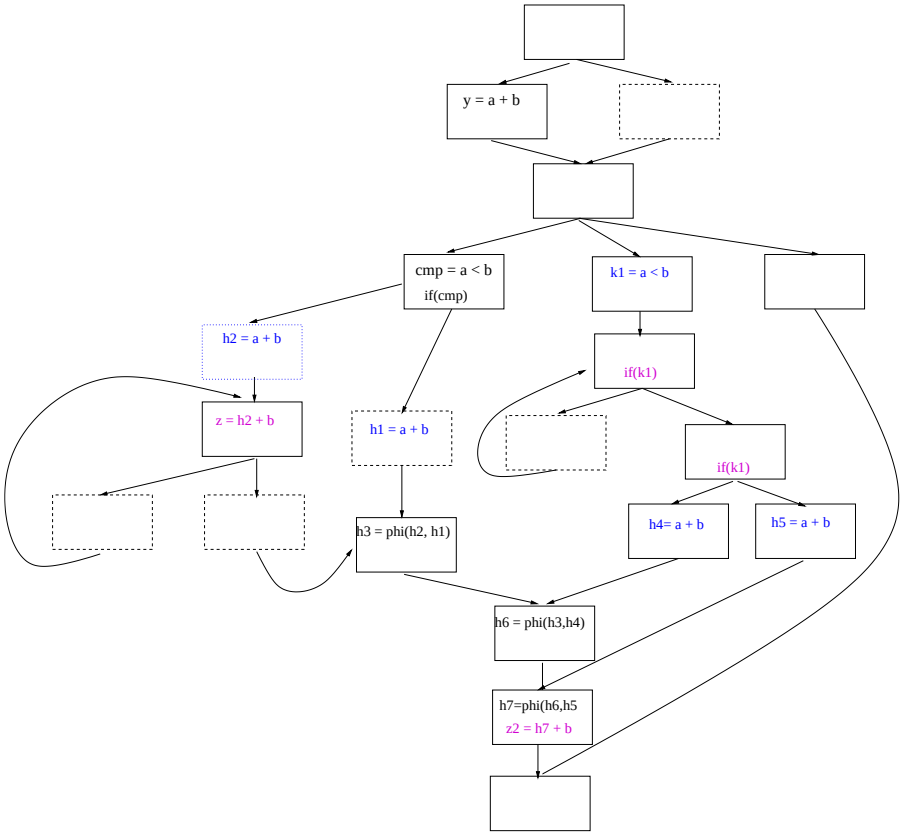

Figure D.2: Lazy code motion transformation on computations $a + b$ & $a < b$. Dotted boxed denote critical edges. Blue dotted boxes are the ones inserted by loop rotate. PRE can insert the computations in these places. Inserted statements are marked blue and replaced ones with magenta

Bibliography

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck, *Detecting equality of variables in programs*, in Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88, New York, NY, USA, 1988, ACM, pp. 1–11.

[2] P. Briggs and K. D. Cooper, *Effective partial redundancy elimination*, in Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94, New York, NY, USA, 1994, ACM, pp. 159–170.

[3] P. Briggs, K. D. Cooper, and L. T. Simpson, *Value numbering*, Softw. Pract. Exper., 27 (1997), pp. 701–724.

[4] J. Cocke, *Programming Languages and Their Compilers: Preliminary Notes*, Courant Institute of Mathematical Sciences, New York University, 1969.

[5] K. D. Cooper and L. T. Simpson, *Scc-based value numbering*, Software Practice and Experience, 27 (1995), pp. 701–724.

[6] K. heinz Drechsler and M. P. Stadel, *A variation of knoop, rothing, and steffen's lazy code motion*.

[7] R. Kennedy, S. Chan, S. ming Liu, R. LO, P. Tu, and F. Chow, *Partial redundancy elimination in ssa form*, ACM Transactions on Programming Languages and Systems, 21 (1999), pp. 627–676.

[8] J. Knoop, O. Rüthing, and B. Steffen, *Lazy code motion*, in Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, PLDI '92, New York, NY, USA, 1992, ACM, pp. 224–234.

[9] ——, *Optimal code motion: Theory and practice*, ACM Trans. Program. Lang. Syst., 16 (1994), pp. 1117–1155.

[10] E. Morel and C. Renvoise, *Global optimization by suppression of partial redundancies*, Commun. ACM, 22 (1979), pp. 96–103.

[11] L. T. Simpson and K. D. Cooper, *Value-driven code motion*, IEEE Transactions on Image Processing, (1995).

[12] T. Vandrunen and A. L. Hosking, *Value-based partial redundancy elimination*, in In CC, 2004, pp. 167–184.