Partial Redundancy Elimination using Lazy Code Motion

Sandeep Dasgupta[1] Tanmay Gangwani[2]

May 11, 2014

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

1. 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. In this project, we implemented a PRE optimization pass in LLVM and measured results on a variety of applications. We chose PRE because it's a powerful technique that subsumes Common Subexpression Elimination (CSE) and Loop Invariant Code Motion (LICM), and hence has a potential to greatly improve performance.

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;
```

2. Related Work

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.

PRE in 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.

3. Algorithm Overview

Our algorithm for PRE is a slightly modified version of the iterative bit-vector data flow algorithm by LCM [8]. It uses four data flow equations to identify for each expression in the program, the optimal evaluation point. The first flow equation calculates down-safe (anticipatible) points for an expression. An expression is said to be down-safe at a point p if computing the expression at p woud be useful along any path from p. The second flow equation calculates

up-safe (available) points. An expression is up-safe at a point p is it has been computed on every path from the entry node to p and not killed after the last computatioon on each path. Using these, the algorithm calculates the *Earliest* property. An expression is said to be *Earliest* at a point p if there doesn't exist an earlier point where the computation of the expression is both down-safe and produces the correct values. Such points are known as *computationally optimal* placement points.

Evaluating the expression at computationally optimal points could negatively impact performance due to increased register pressure. Therefore, the latter half of the LCM algorithm pushes the computation of the expression close to the use of the expression. More specifically, it the third flow equation calculates the *Latest* property. An expression is said to be *Latest* at a point p if it is computationally optimal at p, and on every path from p, any later optimal point on the path would be after some use of the expression. Through the fourth and final flow equation, the algorithm determines if it is necessary to allocate a temporary at a point p for the expression. The property is known as *Isolated*. An expression is *Isolated* at a point if it is optimal, and the value of the expression is only used immediately after the point. Therefore, allocation of temporaries at *Isolated* points is avoided.

In summary, the four flow equations provide computationally optimal placement points which require the shortest lifetimes for the temporary variables introduced.

4. Implementation Details

Value Numbering

Prior research [2] 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 were not able to use the same for our code. We wrote our own value numbering pass which fed expression value numbers to the PRE stage. It should be noted, however, that we did not implement value numbering from scratch and used an old (now defunct) LLVM pass as a starting point. Most importantly, we augmented the basic value numbering in the following ways -

- Added 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

Reassociation has also been shown to make the code more amenable for PRE. It refers to using associativity, commutativity and distribitivity to divide expressions into parts that are constant, loop invariant and variable. We used an already existing LLVM pass (-reassociate) for

Global Reassociation. As per our testing, optimizations 2 and 3 (above) are also done by this pass, and hence, we disabled our version for the more robust LLVM version. Optimizations 1 and 4, however, are still our contribution.

Notion Of Leader Expression

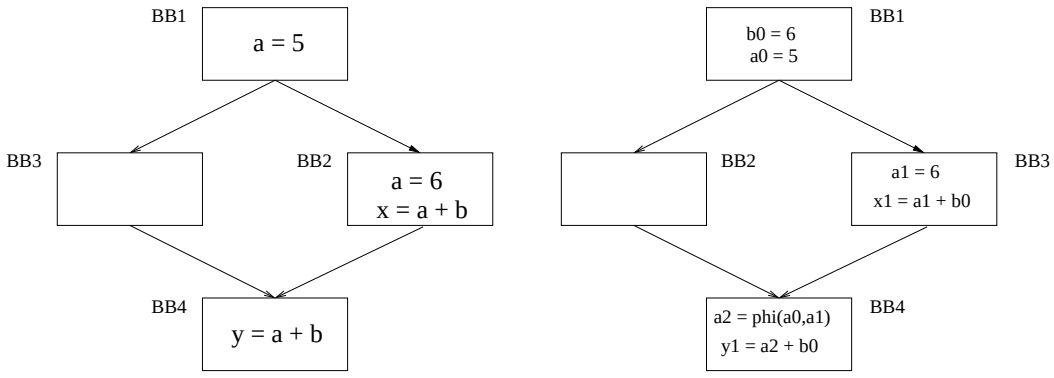
The value numbering algorithm computes the RPO solution as outlined in [5]. It goes over the basic blocks in reverse post order and adds new expressions to a hash table based on the already computed value numbers of the operands. We call an expression a 'leader' if at the time of computing its value 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 our current implementation.**



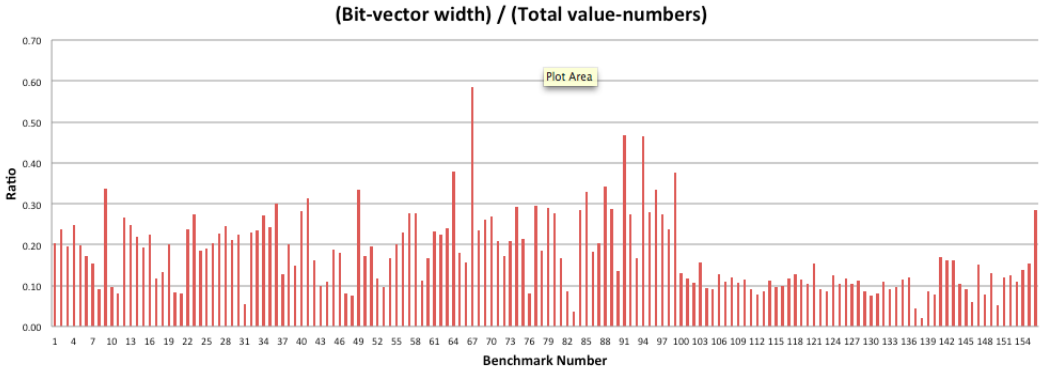
Code not in SSA Form; Two lexically equivalent expressions in Basic block 3 and 4 with different value numbers.

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

Figure 1

Value-Number driven code motion

We initially implemented the flow equations from the Lazy Code Motion paper [8]. 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 in the appendix, 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. Therefore, 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. A downside to this approach is that 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. Figure **??** quantifies the savings we observe using the LLVM multi-source package as testbed.



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 $a + b$ & $a < b$;

5. Miscellaneous

Zero-trip Loops

Our algorithm moves 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. We achieved this effect using an existing LLVM pass *-loop-rotate*.

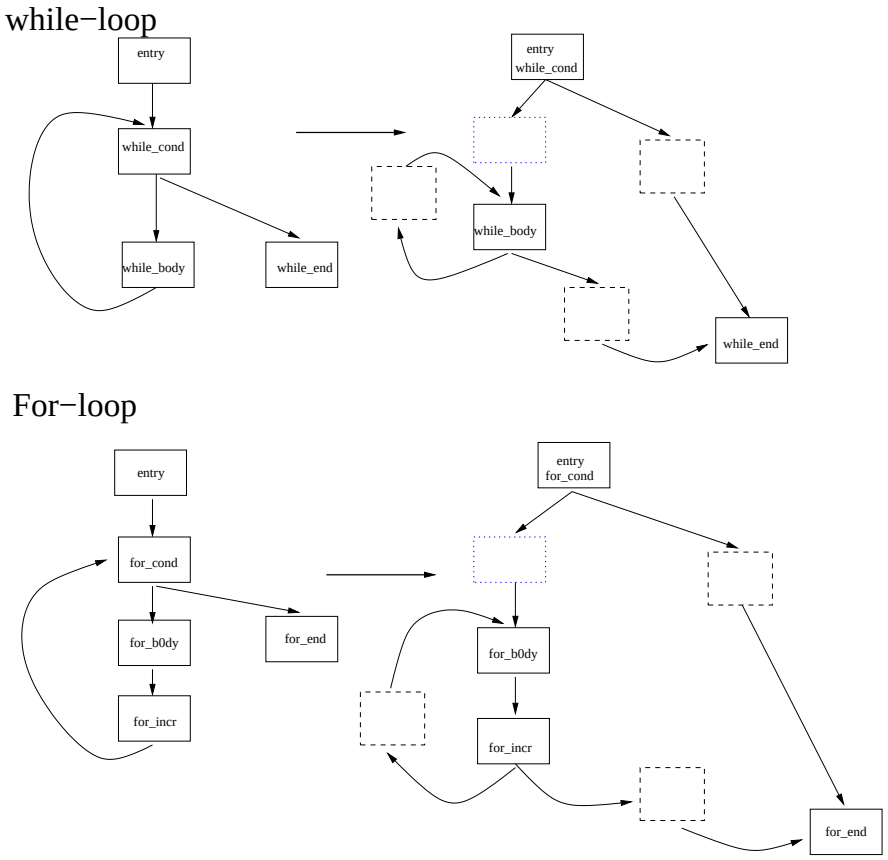


Figure 2: Loop transformations done by *-loop-rotate*. *do-while* loops remain unaffected. Blue dotted boxes are the ones inserted by loop rotate. PRE can insert the computations in these places.

Critical Edges

A critical edge in a flow graph is an edge from a node with multiple successors to a node with multiple predecessors. Splitting such edges and inserting dummy nodes aids PRE by offering more anticipatible points. We used an existing LLVM pass (BreakCriticalEdges) for the same. In many cases, however, the dummy nodes created by this pass do not hold any computation after PRE. We used *-simplifycfg* to clean up the mess created by BreakCriticalEdges.

Unresolved Issues

There were a couple of issues on which we would have liked to spend more time. The first is redundancy elimination for expressions which are lexically diffferent in SSA, and have different value numbers. The second issue pertains to the insertion step of our algorithm and needs slightly detailed explanation. 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 exhaustive testing on multiple suites suggests that this is a very rare occurrence.

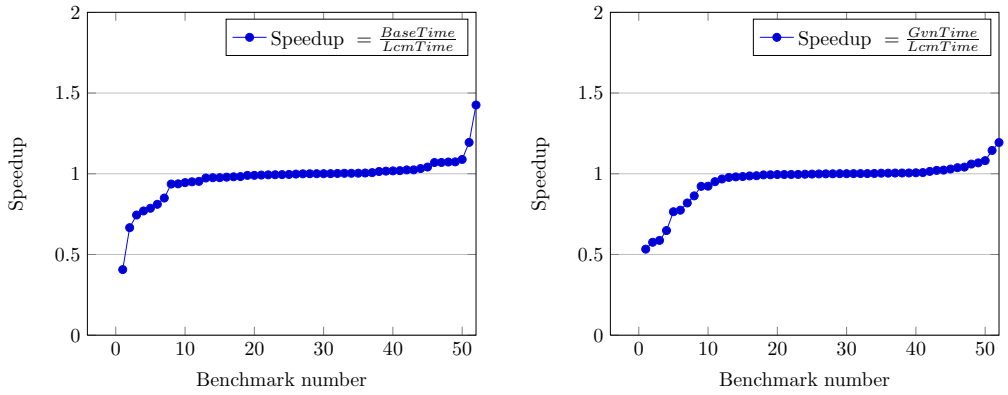


Figure 3: Performance evaluation with LLVM SingleSource Benchmark

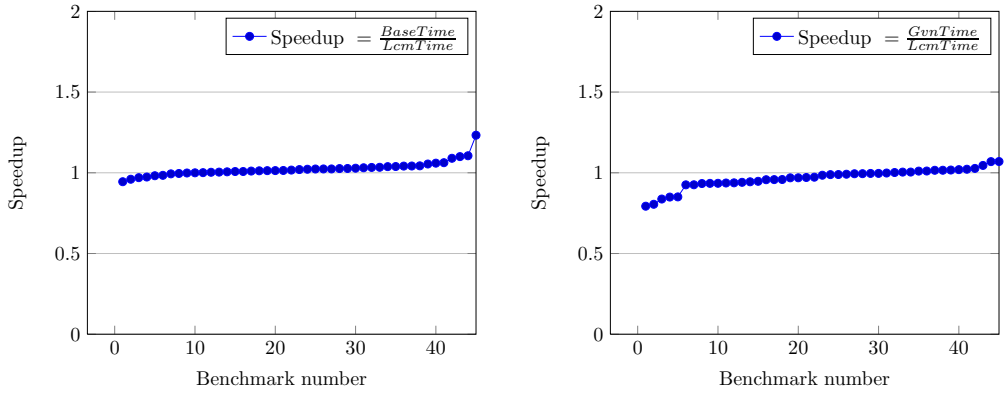


Figure 4: Performance evaluation with LLVM MultiSource Benchmark

Testing

LLVM Single source & Multi source

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 53 benchmarks from LLVM single-source and 46 benchmarks from LLVM Multisource packages 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 LLVM's existing GVN based PRE, with our version of PRE (which we will henceforth refer to as LCM PRE) and without the PRE pass and take the ratios of runtimes (Base to LCM and GVN to LCM) on the hardware. Figures **??** & **??** shows the S-curves for performance for the above benchmarks. 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%.

Spec2006 Benchmark

To test our implementation, we ported the spec2006 benchmark to the LLVM testing infrastructure and made the runs. Our runs are made of specint and specfp portions of the benchmark. Folowing is the link which lists all the inputs that we tried to run those benchmarks. `http://boegel.kejo.be/ELIS/spec_cpu2006/spec_cpu2006_command_lines.html`

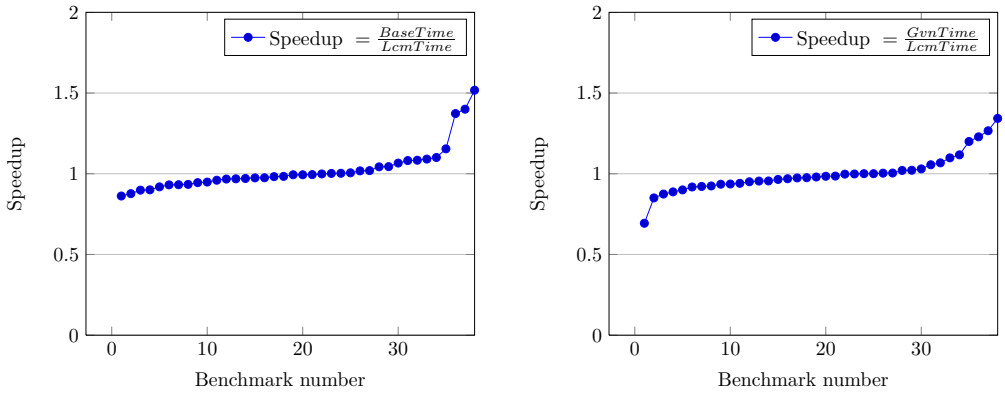


Figure 5: Performance evaluation with SPEC 2006

As before, we compile the codes with (1) LLVMs existing GVN based PRE, (2)with our LCM PRE and (3) without the PRE pass and take the ratios of runtimes (Base to LCM and GVN to LCM) on the hardware. Figures **??** shows the S-curves for performance.

We found that 15/35 benchmarks, our pass improves the run time by varying amounts (upto 52% w.r.t Base and 34% for Gvn). Performance drops for few benchmarks, but the degradation is bound by 8% for Base and 7% for Gvn.

Following Table **??** is the measurements from the top three performers from each of Single-Source, MultiSource and SPEC2006 benchmarks.

Benchmark Name	Base Pre Time (B)	LCM Pre Time(L)	GVN Pre Time(G)	B/L	G/L
SingleSource/Benchmarks/Dhrystone/fldry	5.405	4.965	5.263	1.088	1.060
SingleSource/Benchmarks/Misc/oourafft	5.112	4.281	4.286	1.194	1.001
SingleSource/Benchmarks/Misc/lowercase	40.795	28.612	28.628	1.425	1.000
MultiSource/Benchmarks/TSVC/NodeSplitting-flt	7.378	6.706	5.398	1.100	0.804
MultiSource/Benchmarks/TSVC/Expansion-flt	6.339	5.734	5.308	1.105	0.925
MultiSource/Benchmarks/TSVC/Expansion-dbl	7.134	5.787	5.355	1.232	0.925
SPECINT2006/456.hmmer	1547.77	1019.885	993.581	1.517	0.974
SPECINT2006/403.gcc	0.007	0.005	0.006	1.4	1.2
SPECINT2006/464.h264ref	185.551	168.563	163.363	1.100	0.969

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

Here we have shown that in the optimizations performed by our PRE pass. The intention here is to show how our version of PRE performed on the computations $a + b$ & $a < b$;. The BB marked red is the one where Local common subexpression elimination happened.

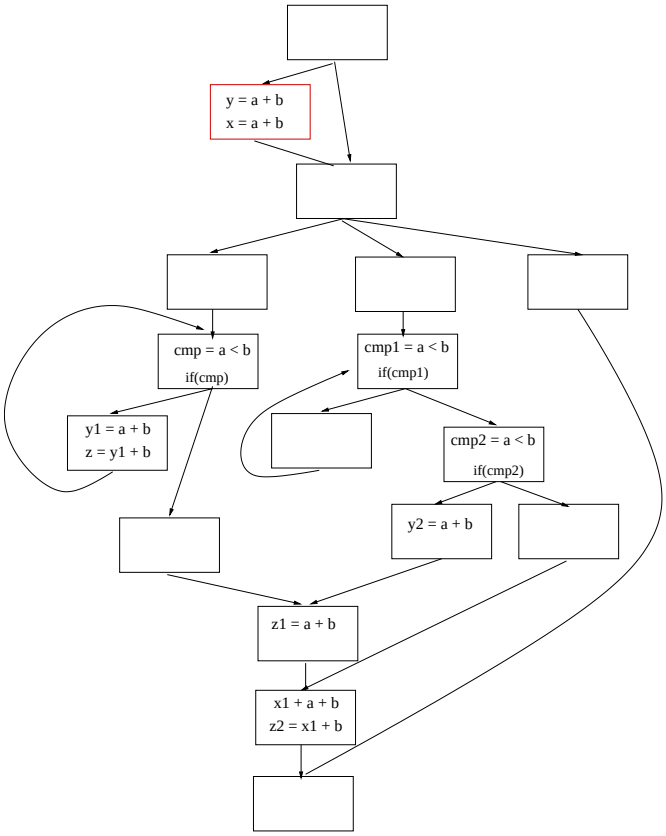


Figure D.1: A motivating example

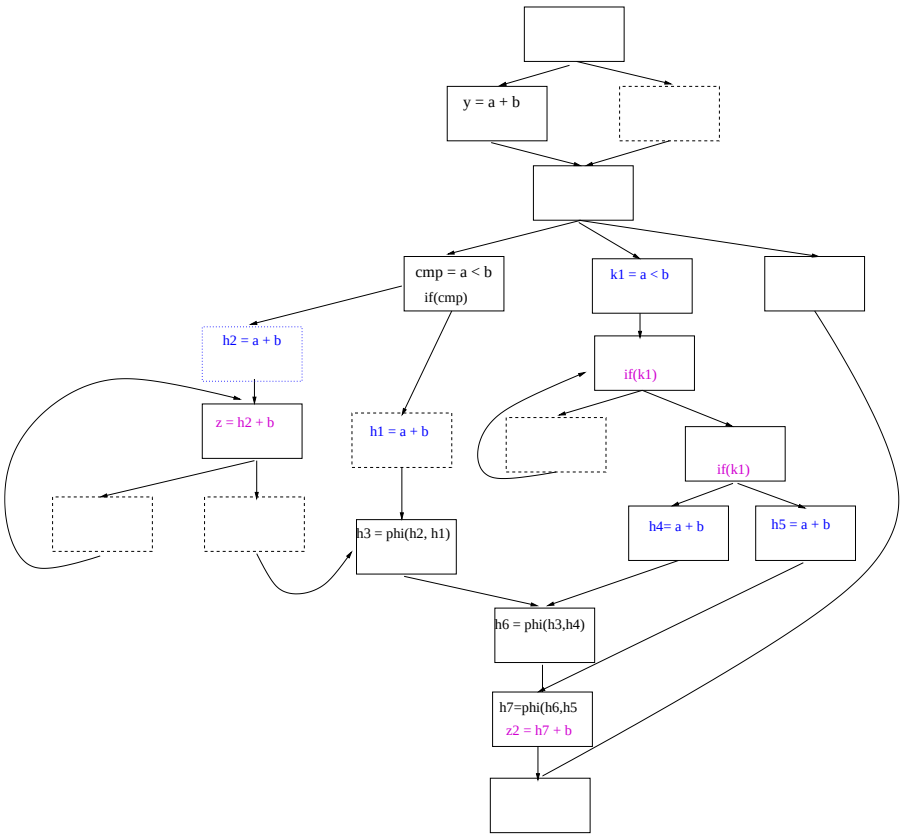


Figure D.2: Lazy code motion transformation on computations $a + b$ & $a < b$. Dotted boxed denote critical edges. Blue dotted boxes are the ones inserted by loop rotate. PRE can insert the computations in these places. Inserted statements are marked blue and replaced ones with magenta

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.