A Practical Improvement to the Partial Redundancy Elimination in SSA Form

Jongsoo Park
Department of Electrical Engineering, Stanford University, Stanford, CA 94305
jongsoo@stanford.edu

Jaejin Lee
School of Computer Science and Engineering, Seoul National University, Seoul 151-744, Korea
jlee@cse.snu.ac.kr

Received 15 February 2008; Accepted 11 July 2008

Partial redundancy elimination (PRE) is an interesting compiler optimization because of its effectiveness and generality. Among many PRE algorithms, the one in static single assignment form (SSAPRE) has benefits over other bit-vector-based PRE algorithms. It preserves the properties of the SSA form after PRE and exploits the sparsity of the SSA form, resulting in reduced analysis and optimization time. This paper presents a practical improvement of the SSAPRE algorithm that further reduces the analysis and optimization time. The underlying idea is removing unnecessary Φ's during the Φ-Insertion phase that is the first step of SSAPRE. We classify the expressions into three categories: confined expressions, local expressions, and the others. We show that unnecessary Φ's for confined and local expressions can be easily detected and removed. We implement our locality-based SSAPRE algorithm in a C compiler and evaluate its effectiveness with 20 applications from SPEC benchmark suites. In our measurements, on average 91% of Φ's identified by the original demand-driven SSAPRE algorithm are unnecessary for PRE. Pruning these unnecessary Φ's in the Φ-Insertion phase makes our locality-based SSAPRE algorithm 1.8 times faster, on average, than the original SSAPRE algorithm.

Categories and Subject Descriptors: Database Management [**Heterogeneous Databases**]

General Terms: Algorithm and Experiment

Additional Key Words and Phrases: Static Single Assignment Form, Partial Redundancy Elimination

1. INTRODUCTION

Partial redundancy elimination (PRE) [Bodík et al. 1998; Briggs and Cooper 1994;

Cai and Xue 2003; Chow 1997; Dhamdhere 1988; Drechsler and Stadel 1993; Kennedy et al. 1999; Kennedy et al. 1998; Knoop et al. 1992; Knoop et al. 1994; Lin et al. 2003; Lo et al. 1998; Morel and Renvoise 1979; Odaira and Hiraki 2004; Odaira and Hiraki 2005; Paleri et al. 1998; Scholz et al. 2004; VanDrunen and Hosking 2004; VanDrunen and Hosking 2004] is a powerful redundancy elimination algorithm that performs both of global common subexpression elimination and loop-invariant code motion. Due to its generality and the importance of redundancy elimination in compiler optimizations, PRE has become an important component in optimizing compilers.

The SSAPRE algorithm proposed by [Chow et al. 1997; Kennedy et al. 1999] is the first PRE algorithm preserving the properties of the static single assignment (SSA) form [Cytron 1991]. SSAPRE's originality is that it handles expressions, while other previous optimizations in the SSA form handle variables for analyses and optimizations.

SSAPRE has benefits over the traditional bit-vector-based PRE: First, SSAPRE exploits the *sparsity* of the representation in the SSA form. In a sparse form, information is associated with only the place where it is needed and is propagated through smaller number of steps. Although the sparse scheme has to give up the parallelism exploited by the bitvector algorithms, the analysis time can be reduced especially for large procedures due to the much smaller amount of information processed than the bit-vector algorithms [Choi et al. 1991]. Second, we do not need to transform the resulting program into an SSA form when subsequent SSA-based optimizations are required after SSAPRE. For the bit-vector PRE algorithms, the time-consuming SSA conversion has to be done again because they do not preserve the properties of the SSA form [Choi et al. 1996].

The number of Φ-assignments inserted in the SSA form is a major factor that determines the analysis time of SSAPRE steps. [Chow et al. 1997; Kennedy et al. 1999] proposed a demand-driven Φ insertion algorithm for SSAPRE to reduce the number of Φ-assignments. However, the performance comparison between SSAPRE and bit-vector PRE for SPEC95 benchmark suites in [Chow et al. 1997; Kennedy et al. 1999] reports that SSAPRE is up to 2.8 times slower than bit-vector PRE. In many programs, the majorities are small procedures, so the advantage of SSAPRE for large procedures [Chow et al. 1997; Kennedy et al. 1999] may not be applicable to the ordinary cases.

This paper presents a practical method to improve the analysis time of the original SSAPRE algorithm. With our locality-based SSAPRE, we can make further significant reduction of Φ-assignments resulting in reduced analysis time. The original SSAPRE algorithm identifies Φ-assignments that are *unsafe* to insert new computation for redundancy elimination at a later stage of SSAPRE. On the contrary, our locality-based SSAPRE algorithm filters out most of unnecessary Φ-assignments at the first stage, thus reduces the analysis time in the following stages.

We identify the expressions in the program into three categories: *confined* expressions, *local* expressions, and the others. Φ-assignments for the confined expressions are unnecessary to perform SSAPRE. A Φ-assignment for a local expression is unnecessary if the expression does not post-dominate the Φ-assignment. The remaining expressions are processed in the same way as the original SSAPRE algorithm. In our approach, the complicated criteria in the original SSAPRE algorithm to identify unsafe Φ-

assignments become a simple check of an expression's locality and post-dominance relationship.

We have implemented our method in a C compiler prototype and evaluated its performance across 20 applications from SPEC95 and SPEC2K benchmark suites. In our measurements, 91% of Φ-assignments are reduced, on average, compared to the demand-driven Φ insertion algorithm in the original SSAPRE algorithm. As a result, our locality-based SSAPRE algorithm is 1.8 times faster, on average, than the original SSAPRE algorithm.

The rest of this paper is organized as follows: Section 2 explains preliminaries required to discuss PRE and the SSA form, and the SSAPRE algorithm is outlined. Section 3 describes our observation behind locality-based SSAPRE, theoretical facts, and locality-based SSAPRE algorithm. Section 4 presents measurement results to evaluate our algorithm against the original SSAPRE algorithm. Section 5 describes related work. Finally, we conclude in Section 6.

2. BACKGROUND

In this section, we provide preliminaries required to discuss PRE and SSA form. Then, we outline the original SSAPRE algorithm.

2.1 Preliminaries

Dominance Relation. A node n dominates a node m, denoted by n **dom** m, if every path from the entry to m must go through n. A node n post-dominates a node m, denoted by n **pdom** m, if every path from m to the exit must go through n. Every node dominates and post-dominates itself. A node n strictly dominates a node m, denoted by n **sdom** m, if n **dom** m and $n \neq m$.

Dominance Frontier. The dominance frontier $DF(n)$ of a node n is the set of all nodes m such that n dominates a predecessor of m but does not strictly dominate m [Cytron 1991].

The dominance frontier $DF(S)$ of a set S, is defined by,

$$DF(S) = \bigcup_{n \in S} DF(n)$$

The iterated dominance frontier $DF^+(S)$ of a set S, is given by the limit of the increasing sequence of sets of nodes,

$$DF_1(S) = DF(S)$$
$$DF_{i+1}(S) = DF(S \cup DF_i(S))$$
$$DF^+(S) = \lim_{i \to \infty} DF_i(S)$$

Availability and Anticipability. An expression is *available* at some point q in the program *along path* P, if P is a path leading to the point q and the expression occurs at some point r on P with no changes to its operands between r and q on P. An expression is *partially available* at q if there exists a path along which the expression is available at q.

An expression is *fully available* at q if it is available at q along every path from the program entry to q.

An expression is *anticipated* at some point q in the program *along path* P, if P is a path beginning at q and the expression occurs at some point r on P with no changes to its operands between q and r on P. An expression is *partially anticipated* at q if there exists a path along which the expression is anticipated at q. An expression is *fully anticipated* (*down-safe*) at q if it is anticipated at q along every path from q to the program exit. An expression is *locally anticipated* in a basic block if the block contains at least one occurrence of the expression and before the first occurrence of the expression in the block, none of the expression's operands are modified.

2.2 SSAPRE

Similar to ϕ-assignments in SSA form [Cytron 1991], Φ-assignments play a key role in SSAPRE. As ϕ-assignments are viewed as pseudo assignments to variables in SSA form, a Φ-statement is an assignment to the hypothetical temporary h that summarizes value changes of lexically identical expressions in the original program. After SSAPRE has been performed, a Φ-assignment can be translated into a ϕ-assignment to h. The temporary variable h is used to store the result of the corresponding expression after PRE in SSA form. The use points of h are the places where a computation of the expression is replaced by the use of h, and the definition points are the places where a computation of the expression is saved to the temporary h. A Φ-assignment is needed whenever different values of the lexically identical expression may reach a control-flow join point.

Before applying SSAPRE, all critical edges in the control flow graph have been removed by inserting an empty basic block at each critical edge. If there is an edge from a basic block with multiple successors to a basic block with multiple predecessors, we call it a critical edge [Knoop 1992]. In addition, construction of dominator tree [Muchnick 1997] and iterated dominance frontiers (DF^+) [Cytron 1991] have been done. Figure 1(a) shows a sample program we use to illustrate SSAPRE. Figure 1(b) is the program in SSA form. SSAPRE is performed on a program in SSA form by following six separate steps: F-*Insertion*, *Rename*, *DownSafety*, *WillBeAvail*, *Finalize*, and *CodeMotion* [Chow et al. 1997; Kennedy et al. 1999].

In the first step, Φ-*Insertion*, there are two cases when Φ-assignments for an expression need to be inserted. One is the case when a Φ-assignment is inserted at the expression's iterated dominance frontier, $DF^+(S)$, where S is the set of occurrences of the expression.

This is because multiple occurrences of the expression may reach the nodes in $DF^+(S)$ and contribute to the definition of the temporary h. Figure 1(c) shows that a Φ-assignment is inserted in basic blocks 2 and 5 because blocks 2 and 5 are in the iterated dominance frontier of blocks 3, 4, and 5 where the expression x+y appears.

The other case is when there is a ϕ-assignment for a variable contained in the expression. This ϕ-assignment indicates that modification of the value of the expression reaches the merge point where the ϕ-assignment for the variable resides. In our example in Figure 1(c), block 2 and block 5 are the nodes where the ϕ-assignments

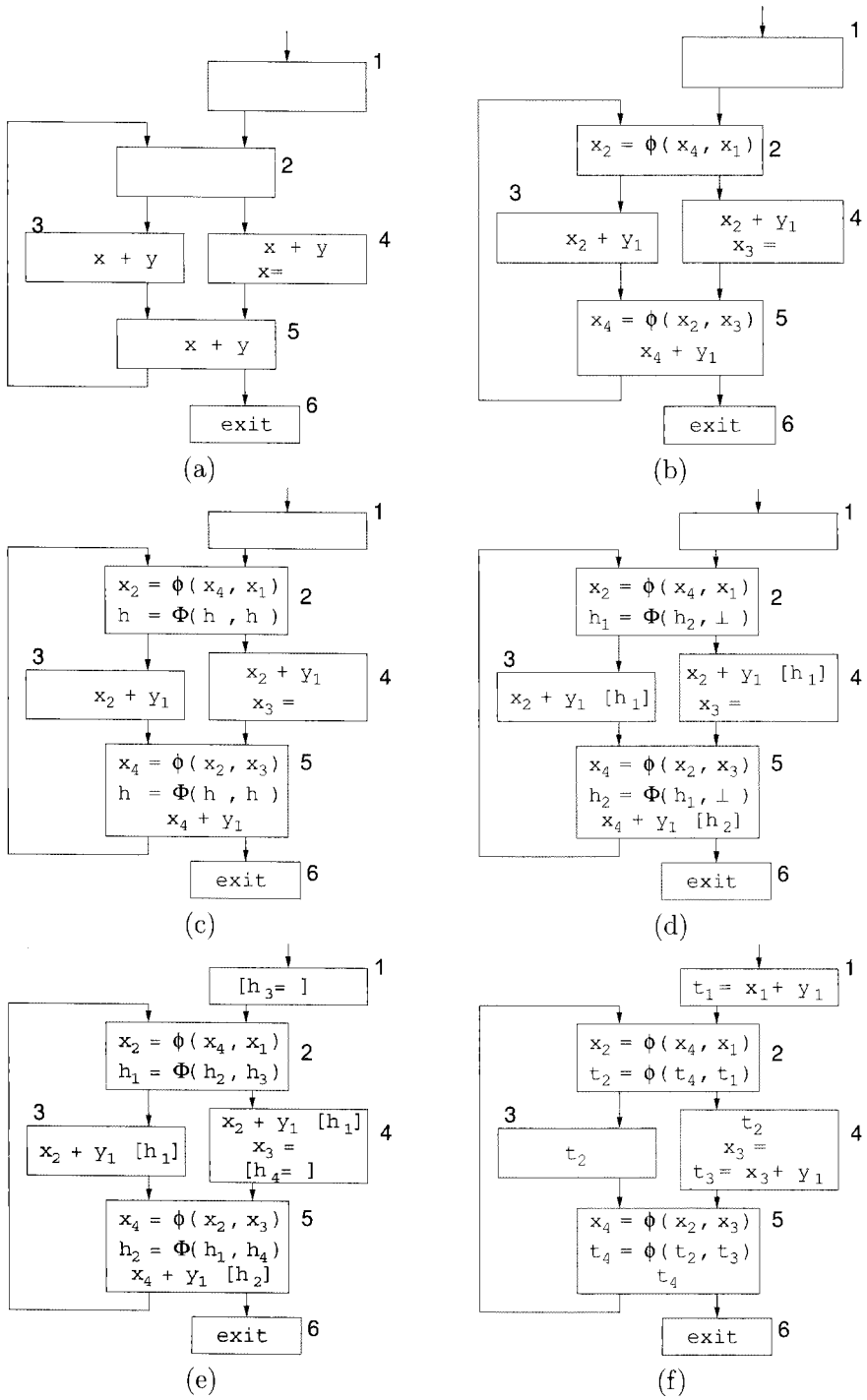


Figure 1. (a) A sample program. (b) The sample program in SSA form. (c) After Φ-*Insertion*. (d) After *Rename*. (e) After *Finalize*. (f) After *CodeMotion*.

for variable x (contained in the expression x+y) are located. Since a Φ-assignment has been already inserted due to the first case, no Φ-assignment is inserted again at the same basic block.

However, if a Φ-assignment due to a variable assignment (i.e., a ϕ-assignment) does not reach a later occurrence of the corresponding expression, the variable assignment will not contribute to the value of the expression. Consequently, the Φ-assignment will not contribute to any optimization in PRE, and will not correspond to any ϕ-assignment for the temporary h. The merge point (where a ϕ-assignment for a variable in the expression resides) needs a Φ-assignment when the Φ-assignment reaches a later occurrence of the expression. In other words, the merge point needs a Φ-assignment when the expression is partially anticipated. Therefore, the Φ insertion for ϕ-assignments is performed in a demand-driven way [Chow et al. 1997; Kennedy et al. 1999]. Both types of Φ insertion are performed in the same pass.

The next step, *Rename*, assigns SSA version numbers to h's. New version numbers are assigned to h's that correspond to three kinds of occurrences of the expression: real occurrences, Φ-assignments, and the operands of the Φ's. Some operands of a Φ-assignment are determined to be undefined and denoted by $\perp$. Insertions of new real occurrences of the expression for PRE are caused by the $\perp$ operands. Figure 1(d) shows the example after *Rename*. The variable h with its version number that corresponds to the real occurrence of the expression is placed right to the real occurrence in "[]".

The next two steps, *DownSafety* and *WillBeAvail*, perform sparse computation of global data flow values based on the SSA graph for h. *DownSafety* checks whether each Φ for the expression is fully anticipated (i.e., *down-safe* [Knoop et al. 1992; Knoop et al. 1994]). When we insert a computation in PRE, the computation must be down-safe at the point of insertion [Kennedy 1972; Knoop et al. 1992; Knoop et al. 1994; Morel and Renvoise 1979]. In our example (Figure 1(d)), both Φ-assignments in block 2 and 5 are down-safe. The next step, *WillBeAvail*, determines the Φ-assignments where the computation of the expression will be available assuming that insertions of the new computation for the expression have been performed at the appropriate incoming edges of the Φ-assignments. It consists of two forward propagation passes. The first pass computes *can_be_available* predicate for each Φ-assignment. The flag is initialized to true for all Φ-assignments at the beginning. After this pass, *can_be_available* is false for a Φ-assignment if and only if no down-safe insertion of computations can make the expression available at the Φ-assignment. In our example (Figure 1(d)), both Φ-assignments are *can_be_available*. The second pass determines the Φ-assignments where the insertion of the computation is the latest to minimize the live range of the introduced temporary for insertion. The pass uses a *later* predicate that is similar to the notion of the LATERIN predicate in [Drechsler and Stadel 1993].

The fifth step, *Finalize*, inserts computations of the expression using the results from *WillBeAvail*. It determines the use-def relationship between SSA versions of the real temporary. Figure 1(e) shows our example after *Finalize*. Insertions are performed in block 1 and 4 in "[]". Note that, in this step, a Φ-assignment whose *can_be_available* predicate is false does not produce any temporaries. Although they

are unnecessary Φ-assignments, they have slowed down SSAPRE up to this *Finalize* step.

The last step, *CodeMotion*, transforms the code to optimized SSA form with the results obtained in *Finalize*. Redundant computations of the expression are replaced by the temporary t and the Φ-assignments for h's are translated into ϕ-assignments for t's. Figure 1(f) shows our example after *CodeMotion*.

3. OUR APPROACH

In this section, we describe our approach to remove unnecessary Φ-assignments in SSAPRE.

3.1 Observation

In addition to their minimal SSA form, Cytron et al. [1991] proposed pruned SSA form where liveness analysis is performed to insert ϕ-functions for a variable V to a node $n \in DF^+(V)$ only when the variable is live at the node n. Similarly, in the demand-driven Φ insertion algorithm [Chow et al. 1997; Kennedy et al. 1999], only live Φ-assignments are considered to be inserted in the SSA form, resulting in reduced compilation time. However, we can further reduce the number of Φ-assignments by considering locality of the expression. Our idea is similar to the semi-pruned SSA form proposed by Briggs et al. [1998]. The semi-pruned SSA form relies on the observation that many variable names in a function are both defined and used entirely within a single basic block. These variables are local to the basic block and do not need any ϕ-function in the SSA form because there is no later use of the variable. The semi-pruned SSA form can reduce almost as many number of ϕ-functions as pruned SSA form with modest cost by just analyzing easily computable locality of variables.

We observe that in many cases, occurrences of an expression appear only in one basic block (About 95% of the lexically identified expressions in a program on average belong to this category, see Table 8 in Section 4).

Definition 1 (Local Expression) *If all occurrences of an expression E appear only in one basic block b in the program, E is said to be local to block b. In other words, E is a local expression at b.*

Moreover, in many cases, all the variables contained in a local expression are defined in the same basic block.

Definition 2 (Confined Expression) *If all occurrences of an expression E appear only in one basic block b in the program, and all variables contained in E are defined in b, E is said to be confined to block b. In other words, E is a confined expression at b.*

For a confined expression E in a basic block b, note that the definition of E's operand appears before its use in the occurrences of E because of the properties of SSA form.

Unlike the case of local variables in semi-pruned SSA form, we cannot guarantee that a local expression does not need any Φ-assignments in SSAPRE.

Consider the code in Figure 2(b). The code in Figure 2(a) results in the code in

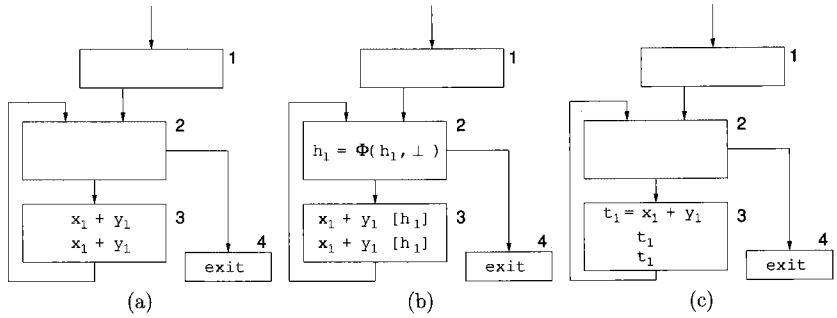


Figure 2. (a) A sample program for spurious Φ-assignment insertion. (b) After *Rename*. (c) After *CodeMotion*.

Figure 2(b) after the demand-driven Φ-*Insertion* and *Rename* stages in SSAPRE (Section 2). Suppose that expression x + y is *local* to block 3 in the original program. There are two occurrences of the expression x + y in block 3. A Φ-assignment has been inserted at block 2 in Φ-*Insertion* stage because block 2 is in the iterated dominance frontier of block 3. Note that block 3 does not post-dominate block 2. The expression x + y is neither available nor fully anticipated at the Φ-assignment in block 2. This means that the Φ-assignment in block 2 will be identified not to be *can_be_available* in *WillBeAvail* stage of SSAPRE (i.e., it will not contribute to any SSAPRE optimizations). Thus, we can safely remove the Φ-assignment inserted at block 2 resulting in reduction of SSAPRE analysis time. Figure 2(c) shows the code after SSAPRE.

Now, consider the code in Figure 3(b), which is the result of applying Φ-*Insertion* and *Rename* steps to the code in Figure 3(a). In contrast to the above case, note that block 3 does post-dominate block 2 in this case. Even though the expression x + y is *local* to block 3, we cannot remove the Φ-assignment at block 2. Since the Φ-assignment at block 2 is *down-safe* and insertion of a new computation of x + y at block 1 can make the expression available at block 2, *can_be_available* flag is true for this Φ-assignment. In addition, the Φ-assignment is the point where the insertion of the computation cannot be further postponed downward without introducing unnecessary new redundancy. This will be identified in *WillBeAvail* stage. Therefore,

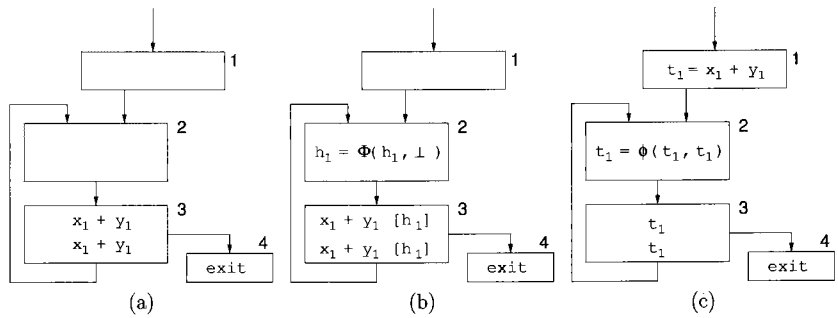


Figure 3. (a) A sample program for necessary Φ-assignment insertion. (b) After *Rename*. (c) After *CodeMotion*.

we cannot remove the Φ-assignment at block 2. Figure 3(c) shows the code after SSAPRE. The ϕ-assignment for t in block 2 can be removed because its arguments are all identical.

In the following, we describe how our method filters many of unnecessary Φ-assignments in the earlier stage, Φ-*Insertion*, by exploiting the properties of local and confined expressions. Checking down-safety and setting the flag *can_be_available* of the Φ-assignments for the local expressions in *DownSafety* and *WillBeAvail* stages are reduced to checking post-dominance relationship between the occurrences and Φ-assignments of the expression in Φ-*Insertion* stage of SSAPRE. We describe our reasoning behind this criterion in the next section.

3.2 Theorems

In SSAPRE, the first pass in the *WillBeAvail* step is a forward propagation pass to find the Φ-assignments where *can_be_available* predicate is false. *Can_be_available* is false for a Φ-assignment if and only if no down-safe placement of computations can make the expression available [Chow et al. 1997; Kennedy et al. 1999]. This is essentially the same as finding insertion points that are *not safe* for the expression. A point is *safe* for an expression if the expression is available or anticipated at that point [Kennedy 1972; Paleri et al. 1998], i.e., a point is not safe for an expression if the expression is neither available nor anticipated at that point.

Lemma 1 *If* $v \in DF^+(u)$, *then* u *does not strictly dominate* v.

Proof of Lemma 1. For induction basis, obviously $\neg(u \textbf{ sdom } v)$ when $v \in DF(u)$ by the definition of DF.

For inductive step and inductive hypothesis, suppose that $\neg(u \textbf{ sdom } w)$ when $w \in DF_i(u)$.

If $v \in DF_{i+1}(u)$, there are two cases: $v \in DF(u)$ or $v \in DF(DF_i(u))$.

When $v \in DF(u)$, obviously $\neg(u \textbf{ sdom } v)$. The other case, $v \in DF(DF_i(u))$, implies that there exist a node w such that $w \in DF_i(u)$ and $v \in DF(w)$. If $v = w$, then $v \in DF_i(u)$.

By the inductive hypothesis, $\neg(u \textbf{ sdom } v)$. If $u = w$, $\neg(u \textbf{ sdom } v)$ because $v \in DF(u)$.

Otherwise (i.e., $v \neq w$ and $u \neq w$), we assume $u \textbf{ dom } v$ and $u \neq v$ (i.e., $u \textbf{ sdom } v$), and draw a contradiction. Since $v \in DF(w)$, $w \textbf{ dom } p$ and $\neg(w \textbf{ sdom } v)$ where p is a predecessor of v. Since p is a predecessor of v and $u \textbf{ dom } v$, $u \textbf{ dom } p$. Since $w \textbf{ dom } p$ and $u \textbf{ dom } p$, either $u \textbf{ dom } w$ or $w \textbf{ dom } u$. Since $w \in DF_i(u)$ and by the inductive hypothesis, $\neg(u \textbf{ sdom } w)$.

This means that $\neg(u \textbf{ dom } w)$ because $u \neq w$. Thus, $w \textbf{ dom } u$. This and $u \textbf{ dom } v$ imply $w \textbf{ dom } v$. This contradicts to $v \in DF(w)$. Therefore, $\neg(u \textbf{ dom } v)$ or $u = v$, i.e., $\neg(u \textbf{ sdom } v)$.

Theorem 1 *If an expression* E *in the program is confined to a basic block* b, *all* Φ-*assignments inserted in* Φ-*Insertion step are unnecessary (i.e., not safe) for SSAPRE.*

Proof of Theorem 1. Let O be the set of E's operand definitions in b. E's operand definitions other than those in O do not reach the occurrences of E in b due to the

property of SSA form, i.e., they do not contribute to the value of the occurrences of E in b. Thus, the Φ-assignments that correspond to ϕ-assignments for the operand definition other than those in O are unnecessary for SSAPRE. All other Φ-assignments will be inserted in $DF^+(b)$ because $DF^+(b) = DF^+(O)$. We will show that these Φ-assignments are not safe (i.e., neither fully available nor fully anticipated).

Suppose that E is fully anticipated at a node $v \in DF^+(b)$. There must exist at least one path beginning at v's exit and the expression occurs at some point r on the path with no changes to its operands between v's exit and r on the path. However, the occurrence of the expression in b cannot be such a point r because the expression's operands are modified between r and v's exit. There must exist at least one occurrence of the expression in a basic block other than b. This contradicts to the fact that E is a confined expression. Thus, E is not fully anticipated (not down-safe) at a node $v \in DF^+(b)$.

We show that E is not fully available at the entry of a node $v \in DF^+(b)$. By Lemma 1, $\neg(b \textbf{ sdom } v)$, i.e., $\neg(b \textbf{ dom } v)$ or $b = v$. If $\neg(b \textbf{ dom } v)$, there exists a path from the program entry to v that does not go through b. Along this path, E is not available at the entry of v.

Thus, E is not fully available at the entry of v. Otherwise ($b = v$), there exists at least one path P from the program entry to the entry of v that does not go through v. Because E is local to v, E does not occur in P. Thus, E is not fully available at the entry of v.

Therefore, those Φ-assignments are neither fully available nor fully anticipated.

Lemma 2 *If E is a local expression at a basic block u, $\neg(u \textbf{ dom } v)$ implies that E is not fully available at any point in v.*

Proof of Lemma 2. By the definition of dominance relation, there is a path P from the program entry to v that does not go through u. Since E is a local expression at u, P does not contain any occurrences of E. Therefore, E is not fully available at any point in v.

Lemma 3 *If E is a local expression at a basic block u, $\neg(u \textbf{ pdom } v)$ implies that E is not fully anticipated at any point in v.*

Proof of Lemma 3. By the definition of post-dominance, there is a path P from v to the program exit, which does not go through u. Moreover, $u \neq v$. Since E is a local expression at u, P does not contain any occurrences of E. Therefore, E is not fully anticipated at any point in v.

Theorem 2 *If E is a local expression at u and $\neg(u \textbf{ pdom } v)$ such that $v \in DF^+(u)$, the Φ-assignment for E at v are unnecessary to perform SSAPRE.*

Proof of Theorem 2. First, we show that the Φ-assignment is not fully available. Since $v \in DF^+(u)$, $\neg(u \textbf{ dom } v)$ or $u = v$ by Lemma 1. If $\neg(u \textbf{ dom } v)$, E is not fully available at v's entry by Lemma 2. Otherwise (i.e., $u = v$), there exists at least one path P from the program entry to v that does not go through v. Because E is local to u, E does not occur in P. Thus, E is not fully available at v. By Lemma 3, E is not fully anticipated at v's exit.

```
01 procedure New-Φ-Insertion
02   for each expression E_i do {
03     DF_phis[i] = ∅
04     Occs[i] = ∅
05     Blocks[i] = ∅
06     isConfined[i] = true
07     bypass[i] = false
08     for each variable j in E_i do
09       Var_phis[i][j] = ∅
10     for each occurrence X of E_i do {
11       DF_phis[i] = DF_phis[i] ∪ DF+(X)
12       Occs[i] = Occs[i] ∪ {X}
13       Blocks[i] = Blocks[i]
14             ∪{block containingX}
15       for each variable j in E_i do {
16         let V be the SSA variable
17             in the jth position in X
18         if ( block containing X
19             ≠ block containing def(V))
20           isConfined[i] = false
21       }
22     }
23   }
24   for each expression E_i do {
25     isLocal[i] = (|Blocks[i] == 1)
26     isConfined[i] = isConfined[i]&&isLocal[i]
27     if (isConfined[i]) continue
28     for each occurrence X of E_i do {
29       for each block v ∈ DF+(X) do {
30         if ((isLocal[i] == false)
31             ||(X pdom v))
32           DF_phis[i] = DF_phis[i] ∪ {v}
33       }
34       for each variable j in E_i do {
35         let V be the SSA variable
36             in the jth position in X
37         if (V is defined by φ)
38           Set_var_phis(X, def(V), i, j)
39       }
40     }
41   }
42   for each expression E_i do {
43     for each variable j in E_i do
44       DF_phis[i] = DF_phis[i] ∪ Var_phis[i][j]
45     Insert Φ's for E_i into
46           the nodes in DF_phis[i]
47     if ((|Occs[i]| == 1) && (DF_phis == ∅))
48       bypass[i] = true
49   }
50 end procedure New-Φ-Insertion

51 procedure Set_var_phis(X, phi, i, j)
52   if ((phi ∉ Var_phis[i][j]) &&
53       ((isLocal[i] == false) || (X pdom phi))) {
54     Var_phis[i][j] = Var_phis[i][j]
55         ∪{block containing phi}
56     for each operand V in phi do {
57       if (V is defined by φ)
58         Set_var_phis(X, def(V), i, j)
59     }
60   }
61 end procedure Set_var_phis
```

Figure 4. New algorithm for Φ-Insertion.

Therefore, E is neither fully available nor fully anticipated at the Φ-assignment.

Theorem 3 *If E is a local expression at u and $\neg(u$ **pdom** $v)$ such that v contains a ϕ-assignment to an operand of the occurrences of E, the Φ-assignment for E at v are unnecessary to perform SSAPRE.*

Proof of Theorem 3. Let p be the point where the Φ-assignment is located. By Lemma 3, E is not fully anticipated at the Φ-assignment. We show that the Φ-assignment is not fully available at the Φ-assignment. Suppose that E is fully available at p. Since E is local at u and E is fully available at p, u **dom** p (i.e., u **dom** v) and every path from u to p does not contain any expression that modifies a variable contained in E. By the property of SSA form, the definition d_i of the ith operand a_i of the ϕ-assignment dominates the ith predecessor of v. However, d_i cannot be on any path from u to p because, if so, d_i modifies a variable contained in E. Thus, for d_i's to reach v, the paths from d_i's to v must go through u because u **dom** v. This means that multiple versions of a variable (a_i's) reach u contradicting to the property of SSA form.

Therefore, E is neither fully available nor fully anticipated at the Φ-assignment.

3.3 Algorithms

Among the Φ-assignments for an expression E inserted by the original SSAPRE Φ insertion algorithm [Chow et al. 1997; Kennedy et al. 1999], the Φ-assignments that satisfies one of the following conditions will not be inserted by our new Φ insertion algorithm:

- When expression E is a confined expression.
- When expression E is a local expression at a basic block b, and b does not post-dominate the Φ-assignment.

Figure 4 shows our new Φ-*Insertion* algorithm. Similar to the Φ-*Insertion* algorithm proposed by [Chow et al. 1997; Kennedy et al. 1999], the set $DF_{phis}[i]$ keeps track of the Φ-assignments inserted on the iterated dominance frontier of the occurrences of expression E_i. The set $Var_{phis}[i][j]$ keeps track of the Φ-assignments inserted due to the occurrences of ϕ-assignments for the jth variable in E_i for demand-driven Φ insertion.

In our algorithm, the set $Occs[i]$ contains the occurrences of expression E_i. The set $Blocks[i]$ contains the basic blocks that contain the occurrences of expression E_i. The predicate $isConfined[i]$ tells us whether expression E_i is a confined expression or not. Similarly, the predicate $isLocal[i]$ tells us whether expression E_i is a local expression. The function $def(V)$ returns the defining assignment of a variable V. The predicate *bypass* tells later steps of SSAPRE that they do not consider expression E_i for optimization because only one occurrences of the expression appears in the program, and there are no Φ-assignments inserted for the expression.

To check post-dominance relationship, a post-dominator tree needs to be built before SSAPRE, but building the tree does not introduce much overhead. There are efficient algorithms building dominator trees [Lengauer and Tarjan 1979]. The post-dominator tree is also needed for other optimizations, such as dead code elimination

[Cytron 1991], which is usually accompanied with PRE [Muchnick 1997]. In fact, the time taken to build the post-dominator tree is amortized by the time savings due to less Φ-assignments in the later SSAPRE steps.

4. EVALUATION

We have implemented the original SSAPRE algorithm and our locality-based SSAPRE algorithm in a research C compiler written in C++. The C compiler uses Edison Design Group's C/C++ front-end [Edison Design Group 2005]. The abstract syntax tree from the front-end is converted to the compiler's own abstract syntax tree (i.e., high-level IR). This high-level IR is converted to the low-level IR for code generation and register allocation, which is similar to the three address code [Muchnick 1997]. Then, the low-level IR is transformed into the SSA form on which the base SSAPRE and locality-based SSAPRE algorithms are implemented. Our base implementation of SSAPRE includes the practical improvements mentioned in [Chow et al. 1997; Kennedy et al. 1998; 1999], such as demand-driven Φ insertion and delayed renaming. Since there are no other optimizations than SSAPRE implemented in the compiler, our evaluation results show only the effects of SSAPRE.

Our measurement runs on a machine with a 3.20 GHz Intel Xeon CPU running Linux operating system. The applications evaluated are from SPECint2K and SPECfp2K benchmark suites. In addition, SPECint95 applications are also evaluated for a comparison to the experiment in [Kennedy et al. 1999].

Figure 5 shows the number of Φ-assignments inserted by our locality-based SSAPRE algorithm. The numbers are normalized to the number of Φ-assignments from the original SSAPRE algorithm (base SSAPRE). The bar labeled confined represents the number of remaining Φ-assignments after identifying confined expressions in our locality-based SSAPRE. The other bar labeled local represents the number of remaining Φ-assignments after identifying local expressions with the post-dominance

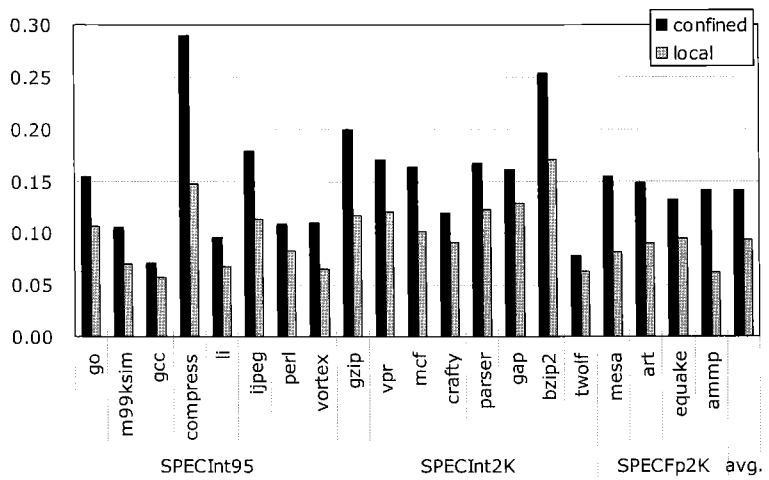


Figure 5. The number of Φ-assignments in our locality-based SSAPRE after exploiting confined expressions and local expressions. The numbers are normalized to the original SSAPRE.

analysis in addition to identifying confined expressions. On average, we reduce about 86% of the Φ-assignments by identifying confined expressions, and further reduce about 5% of the Φ-assignments by identifying local expressions that satisfies Theorem 2 or Theorem 3.

We compare the analysis time of base SSAPRE to that of our locality-based SSAPRE.

We measure the execution time of all the 6 steps of SSAPRE: Φ-*Insertion, Rename, DownSafety, WillBeAvail, Finalize*, and *CodeMotion*. The result is shown in Figure 7. The analysis time of *DownSafety* and *WillBeAvail* phases are not shown in the figure because they have negligible analysis time. On average, our locality-based SSAPRE is 1.8 times faster than original SSAPRE. The actual analysis time of our SSAPRE is from 11 ms (compress) up to 14690 ms (gcc).

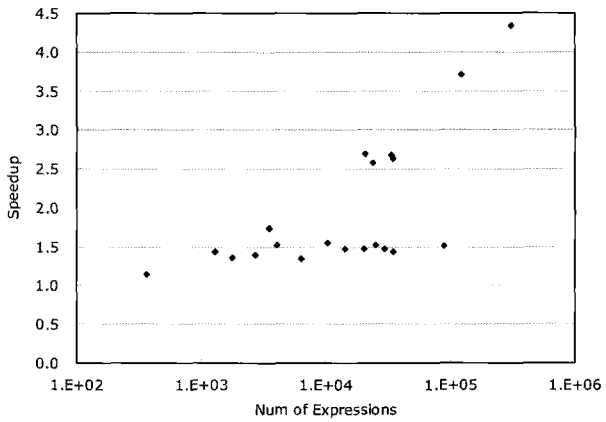


Figure 6. Correlation between the speedup of our locality-based SSAPRE over the original SSAPRE and the number of lexically identical expressions.

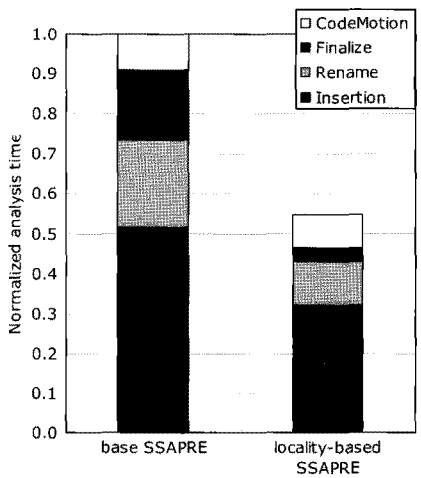


Figure 7. Breakdown of the analysis time taken by SSAPRE.

Our locality-based SSAPRE has a notable effect on the analysis time of *Rename* and *Finalize* steps due to the reduction in the number of Φ-assignments. Note that the time taken by Φ-*Insertion* step in our algorithm is also reduced even though it includes the time to build the post-dominator tree for identifying local expressions. However, the speedup of Φ-*Insertion* step is not that significant compared to *Rename* and *Finalize* steps because it has to collect all of the real occurrences of expressions and check the properties of the expressions. In addition, since there are the same number of inserts, saves and reloads of expressions after *Finalize* in base SSAPRE and locality-based SSAPRE, our method does not have much effect on *CodeMotion* either.

In the original SSAPRE algorithm, 51% and 9% of the analysis time are spent on Φ-*Insertion* and *CodeMotion*, respectively. Consequently, the analysis time has not been reduced as much as the number of Φ-assignments.

Figure 6 shows the correlation between the speedup and the number of lexically identified expressions. The speed up of our locality-based SSAPRE over original SSAPRE is from 1.14 (compress) up to 4.34 (gcc). Applications having more lexically identified expressions have larger speedup. This is because the time spent on Φ-*Insertion* is more effectively amortized by the time spent on the later steps of SSAPRE due to the reduction in the number of Φ-assignments.

Figure 8 shows the characteristics of the applications with regards to the lexically identified expressions in SSAPRE. Column A shows the number of program units in each application.

For each application, column B gives the total number of lexically identified different expressions in the whole program. Column C gives the number of local expressions among the total lexically identified expressions. On average, about 96% are local expressions and they are candidates for removal in our locality-based SSAPRE. This high percentage of local expressions is due to temporary variables generated by the compiler. Expressions having an arbitrary number of operands are broken into simple intermediate representation (IR) statements with the aid of temporary variables. Many middle-level compiler IRs, including the three address codes (this is our case), have statements with limited number of operands. Such a simple middle-level IR reduces the complexity of later compiler optimization phases, and IR statements with limited number of operands is more close to the instructions in the target machine.

Column D shows the number of confined expressions in each application. About 95% of the local expressions are confined expressions. The large number of local expressions and confined expressions accounts for the reduction of Φ-assignments in our locality-based SSAPRE. As mentioned in Section 7.2 of [Kennedy et al. 1999], if an expression does not have any Φ-assignments, and not only occurs in one basic block but also occurs once, we can bypass the rest of SSAPRE steps for the expression. Column E gives the number of bypassed expressions in each application. In our locality-based SSAPRE, about 94% of the expressions are bypassed on average.

Overall, our locality-based SSAPRE algorithm effectively removes unnecessary Φ-assignments and is 1.8 times faster, on average, than the original SSAPRE algorithm. The speedup of our locality-based SSAPRE algorithm over the traditional bit-vector

Apps.	A program units	B total exprs.	C local exprs.	C/B	D confined exprs.	D/C	E bypassed exprs.	E/B
SPECInt95								
go	372	25349	24020	94.8%	22895	95.3%	23887	94.2%
m99ksim	252	20338	19771	97.2%	19084	96.5%	19648	96.6%
gcc	1999	309444	302344	97.7%	298335	98.7%	301103	97.3%
compress	24	369	343	93.0%	283	82.5%	337	91.3%
li	357	6431	6212	96.6%	6077	97.8%	6178	96.1%
ijpeg	472	29744	28351	95.3%	26793	94.5%	27862	93.7%
perl	276	33619	32396	96.4%	31558	97.4%	32237	95.9%
vortex	923	34861	33832	97.0%	32716	96.7%	33326	95.6%
average				96.0%		94.8%		95.1%
SPECInt2K								
gzip	89	4016	3812	94.9%	3479	91.3%	3736	93.0%
vpr	272	14293	13430	94.0%	12657	94.2%	13266	92.8%
mcf	26	1788	1706	95.4%	1618	94.8%	1671	93.5%
crafty	108	23956	23176	96.7%	22686	97.9%	23066	96.3%
parser	324	10298	9582	93.0%	9101	95.0%	9448	91.7%
gap	828	89270	84290	94.4%	81574	96.8%	83708	93.8%
bzip2	74	2723	2501	91.8%	2280	91.2%	2450	90.0%
twolf	191	34799	33715	96.9%	33001	97.9%	33492	96.2%
average				94.6%		94.8%		93.4%
SPECFp2K								
mesa	1105	122519	119086	97.2%	112640	94.6%	117972	96.3%
art	26	1297	1244	95.9%	1143	91.9%	1217	93.8%
equake	27	3520	3424	97.3%	3318	96.9%	3365	95.6%
ammp	179	20637	19854	96,2%	18158	91.5%	19614	95.0%
average				95.7%		94.1%		94.3%
average				95.6%		94.6%		94.4%

Figure 8. Characteristics of lexically identified expressions in the applications.

PRE algorithm is notable too. When we extrapolate the subset of our result on SPECint95 to the experimental in [Kennedy et al. 1999], the speedup is 2.65 on average.

5. RELATED WORK

PRE was introduced by Morel and Renvoise [1979]. PRE is a powerful optimization technique that integrates loop invariant movement and common subexpression elimination together. Because of its generality, PRE has become an important component in many global optimizers and many efforts have been devoted to PRE.

Theoretical refinement and simplification of PRE has been discussed in the literature. Morel and Renvoise's algorithm is not optimal in the sense that it does not eliminate all partial redundancies that exist in a program. Taking the safety issue into account, Dhamdhere [1988] provided a solution that eliminates all partial

redundancies, using the concept of edge placement. Knoop et al. [1992] proposed a descendant of PRE, called lazy code motion that avoids the unnecessary code motion inherent in the original PRE. This feature is important because it can reduce register pressure. In addition, the original PRE algorithm is complicated because it needs bidirectional data-flow analysis. Paleri et al. [1998] proposed a simple PRE algori thm. Their algorithm uses unidirectional data-flow analysis and is based on well known concepts, such as availability, anticipability, partial availability, and partial anticipability.

PRE is more conservative than loop invariant code motion because it is able to move out of the loop only those invariants that are anticipated. Bodík et al. [1988] mentioned 73% of loop-invariant statements cannot be eliminated by PRE. Bodík et al. introduced complete PRE that performs not only code motion but also code restructuring. Even more redundancies can be eliminated by speculative PRE using profile or run-time information [Lin 2003; Cai and Xue, 2003; Scholz et al. 2004; Odaira and Hiraki 2005].

Another inherent problem of PRE is that PRE handles only lexically identical expressions as redundancy elimination candidates. Briggs and Cooper [1994] pointed out this problem. PRE and global value numbering (GVN) are applied into different redundancy cases, and none is a superset of another [Odaira and Hiraki 2004]. Value Driven Redundancy Elimination [Simpson, 1996] and Partial Value number Redundancy Elimination [Odaira and Hiraki 2004] are proposed to combine PRE and GVN.

The difficulty of combining PRE and SSA form comes from the fact that PRE's optimization focus is on expressions instead of variables. After PRE, we need to convert the program into SSA form again if subsequent SSA-based optimizations are required. SSAPRE proposed by [Chow et al. 1997; Kennedy et al. 1999] is the first algorithm that removes this complication and handles expressions in SSA form. they extended their approach to other expression related optimizations [Kennedy et al. 1998; Lo et al. 1998].

VanDunen et al. [2004] proposed the first PRE + GVN algorithm that preserves SSA property. They pointed out that SSAPRE is based on a strict assumption about SSA form that can be broken by other SSA-based algorithms, such as constant propagation. They proposed a solution that works without the assumption [VanDrunen and Hosking 2004]. However, their algorithm does not cover all redundancies described by Odaira and Hiraki [2004], and is not sparse compared to SSAPRE where computations occur only at the Φ merge points.

Our approach does not widen the coverage of SSAPRE, but it is a way to reduce the analysis time of SSAPRE significantly. Our approach can be extended to other expression-based data-flow problems in SSA form.

6. CONCLUSION

We presented a locality-based SSAPRE algorithm in this paper. It identifies local expressions whose occurrences appear only in one basic block. Among those local expressions, confined expressions, whose operand definitions also appear in the same basic block, do not need any Φ-assignments at all. For the remaining local expressions,

the algorithm checks post-dominance relationship between the expression and corresponding Φ-assignments to identify unnecessary Φ-assignments. In general, full safety analysis will be required in the original SSAPRE to identify unnecessary Φ-assignments even for the local expressions. Our locality-based SSAPRE algorithm identifies most of unnecessary Φ-assignments for local expressions with simple occurrence and post-dominance checks. By removing unnecessary Φ-assignments in the first step of SSAPRE, the analysis time of the remaining SSAPRE steps is significantly reduced resulting in the improvement of the whole SSAPRE analysis time.

We observed that more than 96% of the lexically identified expressions for PRE in original SSAPRE are local expressions, and about 91% of the Φ-assignments inserted by the original SSAPRE algorithm are unnecessary. In our measurements with the applications from SPEC benchmark suites, our locality-based SSAPRE algorithm is 1.8 times, on average, faster than the original SSAPRE algorithm.

ACKNOWLEDGEMENT

This work was partly supported by the IT R&D program of MIC/IITA [2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software] and by the Ministry of Education, Science and Technology under the BK21 Project. ICT at Seoul National University provided research facilities for this study.

REFERENCES

BODÍK, R., GUPTA, R., AND SOFFA, M. L. 1998. Complete Removal of Redundant Expressions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 1–14.

BRIGGS, P. AND COOPER, K. D. 1994. Effective Partial Redundancy Elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 159–170.

BRIGGS, P., COOPER, K. D., HARVEY, T. J., AND SIMPSON, L. T. 1998. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software Practice and Experience*, 28(8):859–881.

CAI, Q. AND XUE, J. 2003. Optimal and Efficient Speculation-Based Partial Redundancy Elimination. In *Proceedings of the International Symposium on Code Generation and Optimization*, 91–102.

CHOI, J.-D., CYTRON, R., AND FERRANTE, J. 1991. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 55–66.

CHOI, J.-D., SARKAR, V., AND SCHONBERG, E. 1996. Incremental computation of static single assignment form. In *Proceedings of the 6th International Conference on Compiler Construction*, 223–237.

CHOW, F., CHAN, S., KENNEDY, R., LIU, S.-M., LO, R., AND TU, P. 1997. A New Algorithm for Partial Redundancy Elimination Based on SSA Form. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming Language Design and Implementation*, 273–286.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language and Systems*, 13(4):451–490.

DHAMDHERE, D. M. 1988. A Fast Algorithm for Code Movement Optimization. *ACM SIGPLAN Notices*, 23(10):172–180.

DRECHSLER, K.-H. AND STADEL, M. P. 1993. A variation of Knoop, Ruthing, and Steffen's Lazy Code Motion. *SIGPLAN Notices*, 28(5):29–38.

Inc. Edison Design Group. EDG C/C++ Compiler Front End. Website, 2005. http://www.edg.com.

KENNEDY, K. 1972. Safety of Code Motion. *International Journal of Computer Mathematics*, 3(2 and 3):117–130.

KENNEDY, R., CHAN, S., LIU, S.-M., LO, R., TU, P., AND CHOW, F. 1999. Partial Redundancy Elimination in SSA Form. *ACM Transactions on Programming Language and Systems*, 21(3):627–676.

KENNEDY, R., CHOW, F. C., DAHL, P., LIU, S.-M., LO, R., AND STREICH, M. 1998. Strength Reduction via SSAPRE. In *Proceedings of the 7th International Conference on Compiler Construction*, 144–158.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1992. Lazy Code Motion. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming Language Design and Implementation*, 224–234.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1994. Optimal Code Motion: Theory and Practice. *ACM Transactions on Programming Language and Systems*, 16(4):1117–1155.

LENGAUER, T. AND TARJAN, R. E. 1979. A fast algorithm for finding dominators in a owgraph. *ACM Transactions on Programming Language and Systems*, 1(1):121–141.

LIN, J., CHEN, T., HSU, W.-C., YEW, P.-C., JU, R. D.-C., NGAI, T.-F., AND CHAN, S. 2003. A Compiler Framework for Speculative Analysis and Optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 289–299.

LO, R., CHOW, F., KENNEDY, R., LIU, S.-M., AND TU, P. 1998. Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 26–37.

MOREL, E. AND RENVOISE, C. 1979. Global Optimization by Suppression of Partial Redundancies. *Communications of ACM*, 22(2):96–103.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco.

ODAIRA, R. AND HIRAKI, K. 2004. Partial Value Number Redundancy Elimination. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'04)*, 409–423.

ODAIRA, R. AND HIRAKI, K. 2005. Sentinel PRE: Hoisting beyond Exception Dependency with Dynamic Deoptimization. In *Proceedings of the International Symposium on Code generation and Optimization*, 328–338.

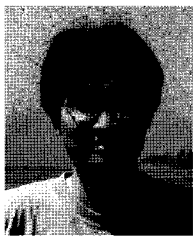
PALERI, V. K., SRIKANT, Y. N., AND SHANKAR, P. 1998. A Simple Algorithm for Partial Redundancy Elimination. *SIGPLAN Notices*, 33(12):35–43.

SCHOLZ, B., HORSPOOL, N., AND KNOOP, J. 2004. Optimizing for Space and Time Usage with Speculative Partial Redundancy Elimination. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 221–230.

SIMPSON, L. T. 1996. *Value-Driven Redundancy Elimination.* PhD thesis, Rice University.

VANDRUNEN, T. AND HOSKING, A. 2004. Value-Based Partial Redundancy Elimination. In *Proceedings of International Conference on Compiler Construction*, 167–184.

VANDRUNEN, T. AND HOSKING, A. L. 2004. Anticipation-based Partial Redundancy Elimination for Static Single Assignment Form. *Software: Practice and Experience*, 34(15):1413–1439.



Jongsoo Park is a PhD candidate at Stanford University. His research interests include compilers and energy-efficient computer architectures. Park received an MS in electrical engineering from Stanford University. Contact him at jongsoo@stanford.edu.

Jaejin Lee is an associate professor in the school of Computer Science and Engineering at Seoul National University. He received his PhD degree in Computer Science from the University of Illinois at Urbana-Champaign (UIUC) in 1999.
He received an MS degree in Computer Science from Stanford University in 1995 and a BS degree in Physics from Seoul National University in 1991. After obtaining his PhD degree, he spent a half year at the UIUC as a visiting lecturer and postdoctoral research associate. He was an assistant professor in the department of Computer Science and Engineering at Michigan State University from January 2000 to August 2002 before joining Seoul National University. His research interests include compilers, computer architectures, parallel processing, and embedded systems.