Regular Paper

Effective Demand-driven Partial Redundancy Elimination

Yasunobu Sumikawa[1,a)] Munehiro Takimoto[1,b)]

Received: November 12, 2012, Accepted: April 25, 2013

Abstract: Partial Redundancy Elimination (PRE) is a technique that not only removes redundant expressions but also moves loop-invariant expressions out of a loop based on the lexical equality among expressions. Traditional PRE analyzes the entire program exhaustively to remove any redundancy, whereas demand-driven PRE, which propagates a query about whether the expression is redundant, can be applied to each expression with lower costs so that it, can remove the redundancy efficiently, including that which is not exposed initially by using copy propagation in the topological sort order. Furthermore, demand-driven PRE allows loop-invariant expressions to be moved out of a loop speculatively by tracing the query propagations, which allows more redundant expressions to be eliminated through algebraic transformations. However, the demand-driven approach with copy propagation is sometimes more costly than the exhaustive approach because it may entail unnecessary analyses. Thus, we propose a technique that suppresses unnecessary query propagations, which does not require any copy propagation. This is achieved by applying global value numbering and recording the value numbers reached at each program point before the query propagation. We implemented our technique using a real compiler and evaluated it with SPEC benchmarks. The experimental results showed that our technique can improve the analysis efficiency by about 56.8% in the best case.

Keywords: compiler, code optimization, partial redundancy elimination, global value numbering

1. Introduction

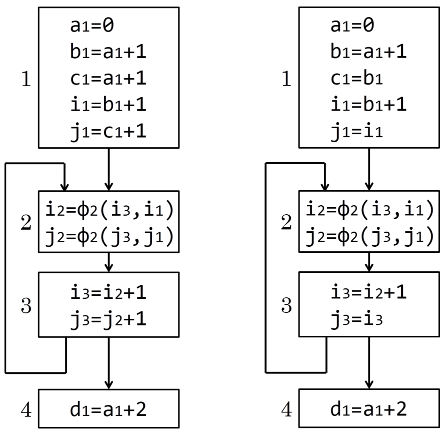
Partial redundancy elimination (PRE) is a code optimization technique that removes redundant expressions but it also moves loop-invariant expressions out of a loop based on the lexical equality among expressions [14], [21]. Traditional PRE analyzes the entire program exhaustively based on data flow equations, before transforming the program to remove all of the redundant expressions found during the analysis. The removal of the expressions leads to some copy assignments, but the application of copy propagation has the effect of exposing new redundancies, which are known as *second-order effects*. To remove more redundant expressions that capture these effects, it is necessary to apply PRE and copy propagation repeatedly.

By contrast, demand-driven PRE (DDPRE) [19] checks the redundancy of each expression by propagating a query about whether the expression is lexically redundant, so it can be eliminated if the expression is found to be redundant. The query is propagated to the program points that are reached by the expression so even if the DDPRE is applied to all expressions, the total cost is close to that incurred with the application of traditional PRE in some cases. The properties of DDPRE allow expressions to be handled sequentially, so its application to all expressions with copy propagation in a topological sort order helps to capture many second-order effects. Furthermore, DDPRE allows loop-invariant expressions to be moved out of a loop speculatively by tracing the query propagations, which allows more redundant expressions to be eliminated via algebraic transformations. However, the requirements for copy propagation other than PRE for all expressions sometimes incur greater costs than exhaustive PRE. In addition, the query may sometimes be propagated unnecessarily in order to check the redundancy of an expression, even if the expression is not redundant on some execution paths.

In this paper, we propose an effective DDPRE (EDDPRE) technique that suppresses unnecessary query propagations and that does not require copy propagation. This technique uses global value numbering (GVN) before query propagation, which records the reachable value numbers at each program point. GVN facilitates the detection of expressions that generate the same values based on their value numbers, rather than their lexical forms. The reachable value number recorded at each node indicates the earlier occurrence of the corresponding expression, so it is sufficient for the query to determine whether the expression has the same value number and it is only propagated until the program points where the value number of the expression is recorded.

The application of EDDPRE to the program is shown in **Fig. 1** (a). EDDPRE generates the value number for each expression that traverse the *control flow graph* (CFG) in topological sort order. The variables $\mathtt{b}_1$ and $\mathtt{c}_1$ have the same value number because the right-hand sides of these definitions are the same. Furthermore, the value numbers of $\mathtt{i}_1$ and $\mathtt{j}_1$ are the same because expressions $\mathtt{b}_1\mathtt{+1}$ and $\mathtt{c}_1\mathtt{+1}$ calculate the same value, which are on the left-hand side. These variables are loop induction variables, which are increased by the same value at each iteration, so $\mathtt{i}_3$ and $\mathtt{j}_3$ are assigned the same value numbers. These redundancies

1 Tokyo University of Science, Noda, Chiba 278–8510, Japan
a) yas@cs.is.noda.tus.ac.jp
b) mune@cs.is.noda.tus.ac.jp



(a) Original code. (b) Code after applying our technique.

Fig. 1 Example program which is used in this paper.

are identified using the value numbers of the operands and arguments of the ϕ function. We assume that the value numbers 1, 2, and 3 are assigned to variable a_1, natural number 1, and variables b_1 and c_1, respectively. When EDDPRE assigns a value number to b_1+1, all of the operands of the expression are translated into their value number, which produces result in a tuple (3,+,2). After b_1+1 has been translated into the tuple, the tuple and i_1 are assigned to a new value number 4. EDDPRE also translates c_1+1 into a tuple (3,+,2), so its value number is the same as that of b_1+1. EDDPRE deals with the redundancy of expressions that depend on the induction variables, such as j_2+1, using a special manner. We explain the details of this manner in Section 4.

As a result of the GVN, each CFG node records the value numbers that are reachable at the exit of the node. For example, Node 1 records reachable value numbers 1–4. These value numbers are checked during query propagation. If we consider expression a_1+2 at Node 4, the query about the expression can be stopped immediately because there is no reachable value number for the expression at the exit of Node 3.

By contrast, traditional DDPRE needs to apply copy propagation after eliminating a_1+1 at Node 1 to expose the redundancy of c_1+1. In Fig. 1 (a), copy propagation needs to analyze the entire program. Thus, c_1+1 is transformed into b_1+1, which is eliminated as a redundant expression. This elimination incurs a new copy assignment, so copy propagation is applied further to identify more redundancies. DDPRE cannot eliminate the redundancy of j_2+1 with induction variable j_2 because the analysis based on query propagation provides a pessimistic solution to the equivalences among expressions. Furthermore, the DDPRE has to propagate a query from Node 4 to Node 1 to determine whether a_1+2 is not redundant.

We implemented EDDPRE using a real compiler and evaluated it with SPEC benchmarks. The experimental results showed that EDDPRE can improve the analysis efficiency by about 56.8% in the best case.

The advantages of our technique can be summarized as follows:

(1) EDDPRE can be applied to any programs.
(2) EDDPRE can capture many second-order effects without copy propagation.
(3) EDDPRE suppresses unnecessary query propagations by recording the reachable value numbers at each node.
(4) EDDPRE eliminates redundancies that traditional DDPRE cannot eliminate lexically.
(5) EDDPRE can deal with some redundant expressions using induction variables by the partial introduction of an optimistic analyzing manner.

The rest of this paper is organized as follows. Section 2 provides the preliminary definitions needed to explain our technique. Section 3 outlines the previous demand-driven PRE technique. Section 4 gives the details of the algorithm for our technique. Section 5 presents the experimental results, which demonstrate the effectiveness of our technique. Section 6 summarizes related work and Section 7 contains our concluding remarks.

2. Preliminaries

We assume that EDDPRE is applied to a program that has been converted into an intermediate representation, which is a sequence of statements that contains at most an operator or a function. EDDPRE also assumes that a CFG has been built for each program. The CFG is a graph structure, which is represented as a quadruple (**N**,**E**,**s**,**e**), where **N** is a set of basic blocks, **E** denotes a set of edges $\mathbf{N} \times \mathbf{N}$, **s** is a start node with an empty statement, and **e** is an end node with an empty statement. A given edge is expressed as $(m, n) \in \mathbf{E}$, where m is referred to as a predecessor of n and n is known as a successor of m. In general, a node has several predecessors and successors because of the nondeterministic branching structure of a CFG. Thus, the sets of predecessors and successors of node n are denoted as $pred(n)$ and $succ(n)$, respectively.

When all paths from the start node to node n include node m, it is said that m *dominates* n [3], which is represented as $m \geq_{dom} n$. If m dominates n and m is not n, it is said that m *strictly dominates* n. An edge $a \rightarrow b$ where the head b dominates the tail a is called a *back edge* [1].

We assume that the programs have been converted into the *static single assignment* (SSA) form [3], [8]. In the SSA form, each variable is defined exactly once by assigning a unique version. This definition dominates all uses so, it is necessary to insert a special function ϕ that merge several definitions into one in a node set defined by a dominance frontier. The dominance frontier of node n is a set of nodes that n cannot dominate initially. The nodes where ϕ functions for an original variable are inserted need to be defined recursively as a dominance frontier for the definitions of the variable, which is referred to as an *iterated dominance frontier*. Our technique assumes that each of the ϕ functions is distinguished by a suffix that identifies its node, such as ϕ_n for node n.

In the CFG, it is assumed that the *critical edges*, which are edges leading from a node with more than one successor to a node with more than one predecessors, have been removed by inserting synthesized nodes, because the critical edges can block effective code motion [12]. Indeed, this assumption has been adopted for the same reason in many traditional techniques based on code motion.

3. Partial Redundancy Elimination

In this section, we describe traditional DDPRE first and provide the definitions of the availability and anticipatability, which are basic properties of PRE, before a brief description of traditional DDPRE.

3.1 Availability and Anticipatability

Expression e is *available* on node n if all of the execution paths from the start node to n include node m which has an e without any definition of all operands of e between m and n. If e is available at some nodes of $pred(n)$, e is *partially available* at node n. When e is available at n, e is *totally redundant* and is replaced by the variable with the preceding execution result. When e is partially available at n, e is *partially redundant*. The partially redundant expression is eliminated by inserting new expressions so it is totally redundant.

Expression e is *anticipatable* at node n if all of the execution paths from m to the end node include node n which has an e without any definition of all operands of e between m and n. When e is anticipatable for n, n is *down-safe* with respect to e. Down-safe is represented by the predicate *DownSafe*. PRE inserts new expressions at the down-safe nodes without extending the lengths of any paths. If new expressions are inserted at non down-safe nodes, this type of insertion is *speculative*. Speculative insertion allows loop-invariant expressions inside a 0-trip loop, such as a while loop, to be moved out of the loop.

3.2 Demand Driven Partial Redundancy Elimination

Partial redundancy elimination based on query propagation (PREQP) is a form of demand-driven PRE [19]. PREQP visits each CFG node in topological sort order, and performs query propagation [17] to check whether each expression e can be eliminated at the node. The query propagation propagates a query backwardly to determine whether the same expressions as e are present at the current node. If all of the propagated queries have *true* as their answers, e is considered to be redundant. Otherwise, PREQP checks whether new expressions can be inserted safely to make all of the queries *true*. If this is possible, e becomes redundant after the insertion, but otherwise e is not redundant. In addition, after the query returns *true* at a node, the sub-query *isSelf* is checked, which determines whether the detected expression is the same occurrence as the source expression of the query. If *isSelf* is *true*, the propagation paths include some circles, which can enforce the speculate movement of loop-invariant expressions from loops.

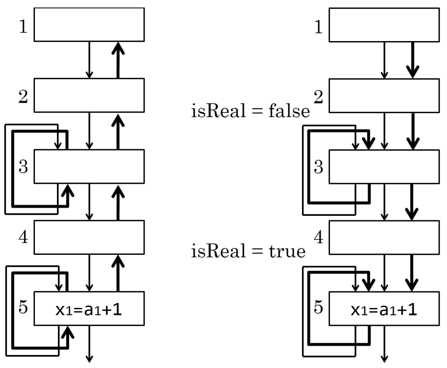
If the query is propagated to node n, the answer at n is determined as follows.

(1) If n is the start node **s**, the answer is *false*.

(2) If n is a node where the same query has been already propagated, the answer is *true*.

(3) If n is a node where the different query has been already propagated, the answer is *false*.

(4) If n is the original node of the query and the original query is also same as the query, the answers to the query and *isSelf* are both *true*.



(a) Original code. (b) Query propagation.

Fig. 2 An example of unnecessary code motion.

(5) If n has the same expression as the query, the answer is *true*.

(6) If n has the definitions of the operands for the expression of the query other than ϕ function, the answer is *false*.

(7) If all of the above rules are not applied, the query is propagated to all of the predecessors of n. Thus, if the query $query(e(x))$ is propagated from node n, which includes the ϕ function $x = \phi_n(a_0, a_1, \dots, a_i)$, the operands are changed to the corresponding arguments of the ϕ function such as $query(e(a_0))$, $query(e(a_1)), \dots, query(e(a_i))$, and these queries are then propagated to their corresponding predecessors.

The query obtains the local answer if rules **(1)** to **(6)** are applied. If the query cannot obtain the local answer, it is propagated to the predecessors according to rule **(7)**. In rule **(2)**, the answer to the query is *true* because result of the query propagation corresponds to is the maximum fixed point of the data-flow equations. However, the rule may lead to unnecessary code motion, as shown **Fig. 2**.

Consider the query propagation for expression `a1+1` at Node 5 in Fig. 2 (a). This query is first propagated to Nodes 4 and 3. Because the query propagated to Node 3 cannot obtain a local answer at the node, it is propagated to Node 2 and Node 3 via the back edge. The former query obtains the answer *false* because rule **(1)** is applied after propagating to Node 1. The latter query will obtain the answer *true* because rule **(2)** is applied. The two answers, *true* and *false*, are obtained from predecessors, so a new expression is inserted into Node 2 that satisfies down-safety. This insertion can move `a1+1` outside the loop, but a more appropriate insertion node is Node 4 because the motion from Node 4 to Node 2 is not effective. Thus, an ineffective motion increases the temporary liverange introduced for the motion, which can increase the *register pressure*.

Unnecessary code motion occurs when the answers are *true* without reaching a real expression. To avoid unnecessary code motion, PREQP defines a predicate *isReal*, which represents the existence of the real expression, and inserts new expressions only if the query propagation that occurs with the answer *true* satisfies *isReal*. In Fig. 2, the propagation to Node 3 does not satisfy this requirement, whereas that to Node 5 satisfies *isReal*. As a result, PREQP inserts a new expression into Node 4.

Rule **(2)** may also produce an incorrect *true* answer rather than *false* during propagation where *isReal* is *false*, so an inaccurate

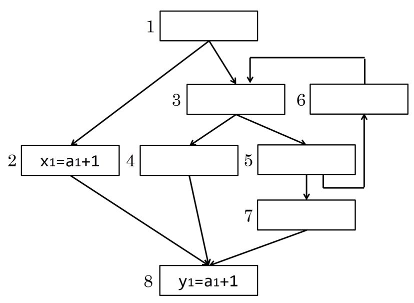


Fig. 3 Inconsistency answers.

code motion may be performed. Consider the application of PREQP to expression `a`$_1$`+1` at Node 8 in **Fig. 3**. We use a tuple (*isAvail*, *isReal*) to focus on explaining the predicate *isAvail* and the predicate *isReal*, where *isAvail* represents the answer to the query.

The answers obtained by propagating to Nodes 2 and 4 are (*true*, *true*) and (*false*, *false*), respectively. Consider the propagation to Node 4 in detail. The query propagated to Node 4 may be propagated further to Nodes 3, 6, 5, and 3. The answers are (*true*, *false*) at Nodes 3 and 5 because rule **(2)** is applied. The query propagated to Nodes 7 and 5 obtains the answer (*true*, *false*) at these nodes because rule **(2)** is applied at Node 5.

As a result of these query propagations, a new expression is inserted only at Node 4. However, the code motion is not accurate because the expression is not inserted on the paths through Node 7. The inconsistency in the answers to the queries is caused by the propagation manner where the answer (*false*, *false*) at Node 3 cannot be reflected in the answer at Node 7. PREQP checks the consistency of the answers by checking whether *isAvail* is not *false* at nodes where no new expression is inserted. PREQP cancels all insertions and removals as soon as the inconsistency of answers is detected.

To determine that an expression can be inserted at a node, the down-safety at the node has to be checked in addition to the insertion condition given by the answers of queries. The down-safety check can also be performed during the query propagation to determine anticipatability. The query propagation to determine the anticipatability $ant(e(x))$ can be achieved by changing the direction of the query propagation, deleting rule **(4)**, reordering rules **(5)** and **(6)**, and changing rule **(7)** as follows.

(7) If $ant(e(x))$ is propagated to node n_s, which is a successor of n, and there is a ϕ function $y = \phi_{n_s}(\ldots, x, \ldots)$ where x corresponds to n, the query is changed to $ant(e(y))$.

Consider the application of PREQP to Fig. 1 (a). First, the expression `a`$_1$`+1`, which is assigned to `c`$_1$, is replaced by `b`$_1$. Next, after copy propagation has been applied, the copy propagation procedure analyzes all of the nodes dominated by Node 1 and it iterates the dominance frontier of the node to replace all of the uses of `c`$_1$ with `b`$_1$. All of the nodes in this program are analyzed. As a result, `c`$_1$`+1` is transformed into `b`$_1$`+1`. Second, PREQP replaces `b`$_1$`+1` with `i`$_1$ and the copy propagation procedure is applied again.

Consider expression `j`$_2$`+1` at Node 3, which is redundant because the expression computes the same value as `i`$_2$`+1`. However, PREQP cannot eliminate the expression because the query of PREQP analyzes each expression pessimistically. After the query related to `j`$_2$`+1` is propagated to Nodes 1 and 2 via Node 2, `j`$_2$`+1` is transformed into `j`$_1$`+1` and `j`$_3$`+1`, respectively, based on rule **(7)**. These expressions cannot be satisfied by the further propagation, so `i`$_2$`+1` is no found to be redundant.

The expression `a`$_1$`+2` is not redundant at Node 4. However, PREQP propagates the query to all of the nodes because the query cannot obtain answers until it is propagated to Node 1.

Compared with the general PRE technique based on data-flow analysis, PREQP can improve the analysis efficiency with many programs because it reflects many second-order effects by repeating the redundancy elimination and copy propagation procedure efficiently without analyzing the entire program. In some programs, however, the analysis efficiency of PREQP will be worse than the general PRE because the program points analyzed during the query propagation and copy propagation with PREQP still include unnecessary points before obtaining the final result.

4. Effective Demand-driven Partial Redundancy Elimination

In this section, we provide the details of our novel algorithm, which we refer to as EDDPRE. EDDPRE involves GVN and query propagation. GVN generates the value numbers for all of the expressions. The query is back propagated from each expression e to check whether there are expressions with the same value number as e at each node.

In the following sections, we provide the details of GVN and the extension points of the query propagation procedure.

4.1 Global Value Numbering

GVN visits CFG nodes in topological sort order and assigns a value number to each expression. EDDPRE represents the value number as a natural number. The value number 0 is used for *optimistic value numbering*, which we will explain in Section 4.1.2. Whenever GVN assigns a new value number, the maximum value number is incremented by 1. GVN uses a hash table, *valueTable*, to record the value number of each variable or expression.

In the following subsections, we explain how the value numbers are assigned to scalar expressions, ϕ functions, and function calls, and how we record the reachable value numbers on each node.

4.1.1 Value Numbering of Scalar Expressions

First, for each expression, all of the operands are replaced with their corresponding value number in the *valueTable*. If there is a value number for an operand in the *valueTable*, it can be obtained by searching using operand as the key. Otherwise, a new value number is assigned to the operand and the new entry for the value number with the operand as the key is added to the *valueTable*. Second, a tuple that comprises of an operator and the value numbers of all operands is checked to determine whether there is an entry in the *valueTable*. As well as the first step, a value number is obtained if there is an entry for the tuple in the *valueTable*. Otherwise, a new value number is assigned to the tuple and a new entry of the value number with the tuple as the key is added to the *valueTable*.

The memory operations given by the load/store instruction can be treated as scalar expressions by making a tuple that contains the load/store operator and the value number of the address specified.

4.1.2 Value Numbering of ϕ Functions

The value number of the ϕ function is determined according to the following rules.

(1) If the same number has been assigned to all of the arguments of the ϕ function, the value number of the arguments is assigned to the ϕ function.
(2) If there is another ϕ function p with arguments where the value numbers are the same as those of the ϕ function at the same node, the value number of p is assigned to the ϕ function.
(3) If none of the above rules are applicable, a new value number is assigned to the ϕ function.

At the header node of the loop, there may be some ϕ functions with arguments where no value number has been assigned. For assigning value numbers to such ϕ functions, all the arguments of it must have value numbers. We assume that the virtual value number 0 is assigned to the variables that reach n through the back edges. After the value numbers have been assigned to all of the expressions in all of the loop nodes that are dominated by n, define value numbers are assigned to the ϕ functions. We refer to this value numbering manner as *optimistic value numbering*. Optimistic value numbering may assign the same virtual value number to ϕ functions that return different values, so their value numbers have to be reassigned. If the reassigned value numbers are different, the value numbers of all the expressions in the loop nodes dominated by n also have to be reassigned. Thus, the old value number is updated to a new value number in the hash table. These updating processes are repeated until the value numbers are unchanged. It is sufficient for the reassigned value number to be applied to expressions that depend on ϕ functions where incorrect value numbers are assigned. EDDPRE reassigns value numbers using dependency relations in an efficient manner. However, we omit the details of this process to simplify the explanation.

The reassignment step is guaranteed to terminate because of the following two reasons: 1) after different value numbers have been assigned to expressions, they will never have the same value number; and 2) the number of ϕ functions with optimistic value numbers decreases monotonously.

4.1.3 Value Numbering of Function Calls

We assume that EDDPRE is an intra-procedural technique, so EDDPRE considers that function calls always returns different values. Therefore, the variable defined by the function call is always assigned a new value.

4.1.4 Recording Reachable Value Numbers

To avoid unnecessary query propagation, EDDPRE records the value numbers that reach each node in the node. The value numbers recorded in node n comprise the value numbers defined in n and the value numbers that are reachable from all of the predecessors of n.

The GVN algorithm is shown in Algorithm 1. GVN initializes the value number to 0 and then calls the function *traverseDomTree* to traverse each node in topological sort order. *traverseDomTree* calls the function *numbering* to assign a value number to each expression that visits node n. To visit each node in topological sort order in the CFG, EDDPRE traverses the dominance tree using a depth-first search with left-first. EDDPRE assumes that the child nodes of each node in the dominator tree are sorted in the topological sort order. The visiting order in the dominance tree can be proven to correspond to the topological sort order in the CFG using lemmas 1 and 2, where we assume that n_i and n_j are two arbitrary children of node n in the dominance tree, and that N_i and N_j are the sub-tree node sets that have them as roots, respectively.

Algorithm 1 Global value numbering

```
Function: globalValueNumbering()
 1: let value := 0
 2: traverseDomTree(root, false)

Function: traverseDomTree(n, optimistic)
 3: numbering(n, optimistic)
 4: for all kid ∈ domTree(n) do
 5:     if (checkPhiArg(kid)) then
 6:         // optimistic value numbering
 7:         traverseDomTree(kid, true)
 8:         checkPhiVal(kid)
 9:     end if
10:     traverseDomTree(kid, optimistic)
11: end for

Function: numbering(n, optimistic)
12: for all st ∈ instrList(n) do
13:     if (isPhi(st) ∨ isFunc(st) ∨ isExp(rhs(st))) then
14:         let val := value(st)
15:         if (optimistic ∧ isPhi(st)) then
16:             let variables := nodeVariableMap.get(n)
17:             if (variables.containsKey(val)) then
18:                 let var := variables.get(val)
19:                 if (samePhiMap.containsKey(var)) then
20:                     var := samePhiMap.get(var)
21:                 end if
22:                 samePhiMap.put(lhs(st), var)
23:             end if
24:         end if
25:         setValue(val, lhs(st), n)
26:     end if
27: end for
28: recordReachableValues(n)

Function: recordReachableValues(n)
29: let reachValue := getReachValues(n)
30: for all p ∈ pred(n) do
31:     if (reachValueMap.containsKey(p)) then
32:         let predRVMap := reachValueMap.get(p)
33:         reachValue.add(predRVMap)
34:     end if
35: end for
```

Lemma 1. n_{id} is n_i if an arbitrary node n_{js} in N_j has an edge with n_{id} in N_i in the CFG.

Proof . If we assume that n_{id} is not n_i, n_{id} is included in the subtree that has a child of n_i as its root. n_{js} is not dominated by n_i, so some paths do not include n_i from the start node to n_{id} through edge (n_{js}, n_{id}) in the CFG. This contradicts the fact that n_{id} is dominated by n_i. ∎

Lemma 2. The preference relation between node sets N_i and N_j in the topological sort order of CFG corresponds to the preference relation between n_i and n_j in the topological sort order of CFG.

Proof . Assuming that N_j precedes N_i in the topological sort order of CFG, there is an edge from node $n_{js} \in N_j$ to $n_{id} \in N_i$ in CFG. According to lemma 1, n_{id} corresponds to n_i. In addition, because n_i is the earliest order node in N_i based on the definition of the dominance relation, n_j precedes all nodes in N_i. Therefore, the preference relation between N_i and N_j corresponds to the preference relation between n_i and n_j. ∎

Given that node n precedes node n' when n dominates n', lemma 2 suggests that the pre-order of the left-first and depth-first search in the dominator tree corresponds to the topological sort order of CFG, where we assume that the children of each node in the dominator tree has been sorted in the topological sort order of CFG.

The function *traverseDomTree* calls the function *numbering* first in order to perform value numbering. After the value numbering, *traverseDomTree* checks to determine whether optimistic value numbering is performed. If the function *checkPhiArg* returns *true*, optimistic value numbering is performed by *numbering*, which has *true* for the parameter *optimistic*. Optimistic value numbering assigns value numbers to all of the expressions in the sub-tree that has n as its root. After completing this optimistic value numbering, each of the value numbers in the sub-tree is checked to determine whether they are correct by calling the function *checkPhiVal*.

The function *numbering* assigns value numbers to the scalar expressions, ϕ functions, and function calls. If ϕ function p_2 is assigned the same value number as just previously detected p_1, the information of p_1 cannot be simply overwritten because the value number of variables may be partially changed in optimistic value numbering. Therefore, previously detected ϕ functions are recorded through their assigned variables in the hash table *samePhiMap*. In the case as mentioned above, the *samePhiMap* holds the assigned variable of just previously detected p_1 with the assigned variable of p_2 as a key. That is, starting with the assigned variable of the current ϕ function, all assigned variables of ϕ functions with the same value number can be retrieved by repeating the search for the *samePhiMap*.

Every time numbering is called, the assigned variable with corresponding value number as a key is added to a hash table *variables*, and the value number is inserted to a list *reachValue* as a reachable value number by calling the function *setValue*. The function *lhs* returns the left-hand side of given statement. After finishing the value numbering, all of the reachable value numbers of the predecessors are added to the *reachValue* of *n* by calling the function *recordReachableValues*. The function *getReachValues* returns a list of reachable value numbers at given node.

The value numbering algorithm is shown in Algorithm 2. First, the function *value* is called, which calls the function *getValue* if given statement is trivial assignment; otherwise, translating each operand of expression or argument of ϕ function of the statement into its value number using function *makeVExp*. *getValue* returns a value number for given expression searching *valueTable* if it includes an entry for the expression; otherwise, calling *newValue*, *newValue* generates a new value number, adds the value number with the expression as a key to *valueTable*, and then, return the value number.

When scalar expressions are assigned a value number, *makeVExp* uses a function *rhs* that returns the right-hand side

Algorithm 2 Value numbering for each expression

```
Function: value(st)
 1: if (trivialAssignment(st)) then
 2:     return getValue(rhs(st))
 3: end if
 4: let ve := makeVExp(st)
 5: if (ve =⊥) then
 6:     return newValue(lhs(st))
 7: else if (isPhi(st) ∧ all of the argument value number are same) then
 8:     return the argument value number
 9: else
10:     return getValue(ve)
11: end if

Function: makeVExp(st)
12: let valExp := rhs(st)
13: if (isPhi(st)) then
14:     for all arg_i ∈ st.argument do
15:         valExp.setKid(i, getValue(arg_i))
16:     end for
17: else if (isExp(rhs(st))) then
18:     for all op_i ∈ valExp.operand do
19:         valExp.setKid(i, getValue(op_i))
20:     end for
21: else
22:     valExp :=⊥
23: end if
24: return valExp

Function: newValue(exp)
25: value := value + 1
26: valueTable.put(exp, value)
27: return value

Function: setValue(val, exp, n)
28: if (valueTable.containsKey(exp)) then
29:     valueTable.remove(exp)
30: end if
31: valueTable.put(exp, val)
32: let variables := nodeVariableMap.get(n)
33: variables.put(val, exp)
34: let reachValue = getReachValues(n)
35: reachValue.add(val)

Function: getValue(ve)
36: if (¬valueTable.containsKey(ve)) then
37:     return newValue(ve)
38: else
39:     return valueTable.get(ve)
40: end if

Function: getVariable(val, n)
41: let variables := nodeVariableMap.get(n)
42: return variables.get(val)
```

Algorithm 3 Optimistic value numbering

```
Function: checkPhiArg($n$)
 1: let $ans := false$
 2: for all $st \in instrList(n)$ do
 3:   if ($isPhi(st)$) then
 4:     for all $arg_i \in st.argument$ do
 5:       if ($\neg\, valueTable.containsKey(arg_i)$) then
 6:         if ($n \geq_{dom} pred(n, i)$) then
 7:           $valueTable.put(arg_i, 0)$
 8:           $ans := true$
 9:         else
10:           for all $arg_i \in st.argument$ do
11:             if ($valueTable.get(arg_i) = 0$) then
12:               $valueTable.remove(arg_i)$
13:             end if
14:           end for
15:           return false
16:         end if
17:       end if
18:     end for
19:   end if
20: end for
21: return $ans$
```

Algorithm 4 Checking accuracy of ϕ function's value number

```
Function: checkPhiVal($n$)
 1: let $change := true$
 2: while $change$ do
 3:   $change := false$
 4:   for all $st \in instrList(n)$ do
 5:     let $val := value(st)$
 6:     $setValue(val, lhs(st), n)$
 7:     if ($\neg\, checkAssum(val, lhs(st))$) then
 8:       $traverseDomTree(n, true)$
 9:       $change := true$
10:     end if
11:   end for
12: end while

Function: checkAssum($val, var$)
13: if ($\neg\, samePhiMap.containsKey(var)$) then
14:   return true
15: else if ($val = getValue(samePhiMap.get(var))$) then
16:   return true
17: else
18:   $samePhiMap.remove(var)$
19:   return false
20: end if
```

of given statement. *makeVExp* is also used to assign value numbers to the ϕ functions and function calls. If ϕ function is given to *makeVExp*, the arguments of ϕ function are changed to corresponding value numbers. The operands or arguments are changed to the value numbers by *valExp.setKid*(i, val), which assigns a value number *val* to the i th operand or argument of *valExp*. If a function call is given, it returns $\perp$. Once *makeVExp* returns $\perp$, *getValue* returns a new value number calling *newValue*.

The function *getVariable* returns the variable that corresponds to the value number *val* if the variable is defined at the node n. This function is used during the redundancy elimination phase.

Algorithm 3 shows the optimistic value numbering algorithm. The function *checkPhiArg* checks whether each argument of the ϕ function has already been assigned a value number. If some arguments have no corresponding value number and n dominates the predecessors that corresponding to the arguments, the arguments are assigned a value number 0. If n does not dominate the predecessor, *checkPhiArg* returns *false* and assigns a new value number to the ϕ function in a conservative manner.

Algorithm 4 shows the algorithm used to check whether the optimistic value numbers are correct or not and to reassign a new value numbers instead of the incorrect value number. The function *checkPhiVal* calls a function *checkAssum*, which check whether the optimistic value numbering has been performed correctly. If a ϕ function has been assigned an incorrect optimistic value number, the function *checkAssum* returns *false* to enhance the reassignments of value numbers for all expressions in the nodes that are visited during optimistic value numbering until the value numbers of the expressions are fixed.

checkAssum returns *true* if given variable is not included in the *samePhiMap*, or the value number for the variable in *samePhiMap* is same as given value number. Otherwise, it returns *false* after deleting the entry of *var* in *samePhiMap* because the ϕ function needs to be assigned to a different value number.

Consider $\mathtt{i}_2 = \phi_2(\mathtt{i}_3, \mathtt{i}_1)$ at Node 2 in Fig. 1 (a). The value number 0 is assigned to $\mathtt{i}_3$ because no value number has been assigned to $\mathtt{i}_3$ and Node 2 dominates the corresponding predecessor Node 3. As a result, the ϕ function is translated into $\phi_2(0, 4)$ and the tuple and $\mathtt{i}_2$ are assigned to a new value number 5. $\mathtt{j}_2 = \phi_2(\mathtt{j}_3, \mathtt{j}_1)$ is also translated into $\phi_2(0, 4)$, so that its value number is same as that of $\phi_2(\mathtt{i}_3, \mathtt{i}_1)$. After optimistic value numbering, all of the expressions in Nodes 3 and 4 are assigned to optimistic value numbers. Because $\mathtt{i}_2$ and $\mathtt{j}_2$ were assigned to a value number 5, the tuples of $\mathtt{i}_2$+1 and $\mathtt{j}_2$+1 are changed to (5,+,2) at Node 3. As a result, a new value number 6 is assigned to the tuple, $\mathtt{i}_3$, and $\mathtt{j}_3$.

After assigning a value number 7 to constant 2 at Node 4 and a value number 8 to the expression $\mathtt{a}_1$+2, a new value number 9 is assigned to $\phi_2(\mathtt{i}_3, \mathtt{i}_1)$ because the ϕ function is translated into a tuple $\phi_2(6, 4)$. $\phi_2(\mathtt{j}_3, \mathtt{j}_1)$ is also translated into a tuple $\phi_2(6, 4)$. As a result, $\mathtt{i}_2$+1 and $\mathtt{j}_2$+1 are assigned to the same value number.

Table 1 and **Table 2** represent the *valueTable* and *reachValue*, respectively. In Table 1 and Table 2, the value numbers are enclosed in parentheses to distinguish them from constants.

4.2 Query Propagation

In this section, we explain how PREQP query propagation is extended to EDDPRE. In EDDPRE, a query checks whether some expressions have the same value number as e at each node. As well as PREQP, EDDPRE defines the availability and anticipatability as follows.

Availability : e is available at node n if all the availability queries obtain *true* from all of the predecessors of n.

Anticipatability : e is anticipatable at node n if all the anticipatablity queries obtain *true* from all of the successors of n.

The query propagation rules are extended as follows.

Table 1 Hash table *valueTable* of Fig. 1.

Key	Value number
0	[1]
a_1	[1]
1	[2]
[1] + [2]	[3]
b_1	[3]
c_1	[3]
[3] + [2]	[4]
i_1	[4]
j_1	[4]
2	[7]
[1] + [7]	[8]
d_1	[8]
ϕ_2([6], [4])	[9]
i_2	[9]
j_2	[9]
[9]+[2]	[10]
i_3	[10]
j_3	[10]

Table 2 List *reachValue* of Fig. 1.

Node	Value number
1	[1],[2],[3],[4]
2	[1],[2],[3],[4],[5],[6],[9],[10]
3	[1],[2],[3],[4],[5],[6],[9],[10]
4	[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]

Algorithm 5 Eliminating redundancy

```
Function: eliminate()
 1: for all $n \in toporogicalSortOrder$ do
 2:   let $localMap := getNewHashMap()$
 3:   for all $st \in instrList(n)$ do
 4:     if ($isExp(rhs(st))$) then
 5:       let $val := value(st)$
 6:       if ($trivialAssignment(st)$) then
 7:         $localMap.put(val, lhs(st))$
 8:       else if ($localMap.containsKey(val)$) then
 9:         let $predVar := localMap.get(val)$
10:         $replace(st, predVar)$
11:       else
12:         $initialize()$
13:         let $originalN := n$
14:         let $ve := makeVExp(st)$
15:         if ($propagate(val, ve, n)$) then
16:           for all $n_p \in link[n]$ do
17:             if ($insert(n_p) \neq \perp$) then
18:               $replace(st, link[n])$
19:               $setValue(getValue(rhs(st)), lhs(st), n)$
20:             else
21:               for all $n_p \in link[n]$ do
22:                 $cancel(n_p)$
23:               end for
24:               break
25:             end if
26:           end for
27:         end if
28:       end if
29:     end if
30:   end for
31:   $recordReachableValues(n)$
32: end for

Function: isSameVal(val, n)
33: let $variables := nodeVariableMap.get(n)$
34: let $isAvail := false$
35: if ($variables.containsKey(val)$) then
36:   $isAvail := true$
37: end if
38: let $isSelf := false$
39: if ($n = originalN$) then
40:   $isSelf := true$
41: end if
42: return ($isAvail, isReal$)
```

(2) If n is a node where the query has been already propagated, the answer is *true*.

(3) If n is a node where no value number of the query is reachable and the value number is not dependent on the ϕ function, the answer is *false*.

(5) If n is a node that includes the same value number as the query, the answer is *true*.

(6) This rule is deleted.

(7) If all of the above rules are not applied, the queries are propagated to all of the predecessors of n. In this case, if the query $query(ve(val))$ is propagated from node n that includes a ϕ function $x = \phi_n(a_0, a_1, \ldots, a_2)$, where x is assigned value number *val*, each operand is changed to the value number of the corresponding argument of the ϕ function.

Because PREQP is based on the lexical equality among expressions, it is necessary to lexically check whether a propagated query is same as some expressions at the current node or not [19]. By contrast, EDDPRE does not depend on the lexical equality, so that it is unnecessary to lexically check of the propagated query. In addition, it is not necessary to consider the definitions of operands as well as other techniques that utilize GVN [16]. The property is derived from the single assignment property of SSA form for expressions with not induction variable. For expressions that depend on loop induction variables, queries will not satisfy the query propagation by rule **(7)**. Therefore, rule **(6)** of PREQP is deleted in EDDPRE.

When expression e is partially redundant for preceding expression e' in normal form, in the SSA form, e and e' may be assigned to different value numbers respectively if e depends on the ϕ function. Once rule **(7)** is applied, the value number is changed to the argument of ϕ function corresponding to the predecessor where the query is propagated, and therefore the equality among these expressions can be detected. Therefore, queries that depend on ϕ functions are propagated to the predecessors even if there is no reachable value number at the current node, according to rule **(3)**.

The EDDPRE redundancy checking algorithm is shown in Algorithm 5. The function *eliminate* analyzes redundancies by calling the function *propagate* after initializing the global lists and arrays, such as *answer*, by calling the function *initialize*. If the expression is redundant, it is replaced by the new variable by calling the function *replace* and the value number is updated by one on the right-hand side.

The query propagation algorithm that calls the functions *propagate* and *local* is shown in Algorithm 6. The parameters of function *propagate* are value number *val*, value expression *ve* of which operands are changed to its value number, and the visiting node n. To determine the answer at each node, *propagate*

Algorithm 6 Query propagation

```
Function: $propagate(val, ve, n)$
 1: let $n' := n$ and $isDownSafe := antqp(val, n)$
 2: for all $p \in pred(n)$ do
 3:     $val_p := val$
 4:     $ve_p := transPhi(ve, n, p)$
 5:     if ($ve_p = \perp$) then
 6:         return ($false, false, false, \perp$)
 7:     else if ($ve \neq ve_p$) then
 8:         $val_p := getValue(ve_p, p)$
 9:     end if
10:     let $(isAvail_p, isReal_p, isSelf_p, n_p) := local(val_p, ve_p, p)$
11:     if ($isAvail_p$) then
12:         add $n_p$ to $link[n]$
13:     else
14:         $insertCand[p] := translate(ve_p)$
15:         add $p$ to $link[n]$
16:     end if
17:     let $isReal := \sum_{p \in pred(n)} isReal_p$
18:     and $isSelf := \sum_{p \in pred(n)} isSelf_p$
19:     if ($\Pi_{p \in pred(n)} isAvail_p \vee isReal \wedge (isDownSafe \vee isSelf)$) then
20:         if ($\exists n_p \in link[n].n_p \geq_{dom} n \wedge ve_p = ve$) then
21:             $link[n] := \{n_p\}$
22:             $n' := n_p$
23:         end if
24:         return ($true, isReal, isSelf, n$)
25:     else
26:         return ($false, false, false, \perp$)
27:     end if
28: end for

Function: $local(val, ve, n)$
29: if ($n = s$) then
30:     return ($false, false, false, \perp$)
31: else if ($visited[n]$) then
32:     if ($answer[n] \neq \perp$) then
33:         return $answer[n]$
34:     else
35:         return ($true, false, false, n$)
36:     end if
37: end if
38: $visited[n] := true$
39: let $rlt := \perp$ and$(isAvail, isSelf) := isSameVal(val, n)$
40: if ($isAvail$) then
41:     $var[n] := getVariable(val, n)$
42:     $rlt := (true, true, isSelf, n)$
43: else if ($\neg reacheValue(val, n) \wedge \neg dependPhi(ve)$) then
44:     $rlt := (false, false, false, \perp)$
45: else
46:     $rlt := propagate(val, ve, n)$
47: end if
48: $answer[n] := rlt$
49: return $rlt$
```

calls the function *local*, which determines the answer using the query propagation rule as mentioned above. The value returned by these functions is a tuple of the predicate *isAvail* that represents the availability, the predicate *isReal* that represents the actual appearance of the expression, the predicate *isSelf* that represents whether the node is the initial node of the query propagation, which includes available expressions or not. The node is used to introduce new variables when the new expressions are inserted. Lines 29, 31, and 35 correspond to rules **(1)**, **(2)**, and **(3)**, respectively.

The array *visited* $[n]$ records whether a query is propagated to n. The function *reacheValue*(val, n) checks whether the value number *val* is reachable at node n. The function *dependPhi*(ve) checks whether each operand of *ve* depends on ϕ functions. Lines 31–36 correspond to rule **(2)**. If the answer has been already determined, *local* returns the answer. Otherwise, it returns the visiting node as the temporal available node. This temporal node will be linked to the actual available expression during the translation program phase. Line 39 corresponds to rules **(4)** and **(5)**. The function *isSameVal*(val, n) checks whether node n includes the value number *val* and it also checks whether the query is generated from n. The value returned by *isSameVal* is a tuple of the predicate *isAvail* and the predicate *isSelf*. The function *transPhi* corresponds to rule **(7)**. If a ϕ function is assigned the same value number as the operand of *ve* at n, *transPhi*(ve, n, p) translates the value numbers of the operand to the argument of the ϕ function, which corresponds to the predecessor p. If the operand variable cannot reach the exit of its predecessor, *transPhi* returns $\perp$.

After the query is propagated to predecessors by calling the function *local*, if the answers is *false* at a node, the node is recorded in the list *insertCand* as an insertion node. Otherwise, the node with some expressions is recorded in the list *link* as an available node. This list is used in order to insert ϕ functions, where to suppress the insertion of unnecessary ϕ functions, node n_p is recorded in the *link* rather than n if n_p dominates n.

Each inserted expression is generated based on the value expression of the query. Since operands or arguments of the value expression are value numbers, the value numbers have to be substituted with appropriate variables. The function *translate* performs the substitution. In the substitution, the variables that are used as operands or arguments of the inserted expressions are checked whether they are dominated by their definitions to hold the static single property of SSA form. In addition, as shown in line 41, the ϕ functions with arguments that are substituted with appropriate variables are also inserted to hold the SSA property. Once the variable is selected for the value number at node n, the variable is recorded in *var*$[n]$ to avoid redundant selections of appropriate variables.

4.3 Translation of the Program

The insertion algorithm is shown in Algorithm 7. EDDPRE inserts new expressions at nodes n, which are the expressions recorded in *insertCand*$[n]$. The function *createNewVar* generates new variables for the new expressions. Line 9 checks the consistency of answers, as explained in Section 3.2. EDDPRE does not insert new ϕ functions if the size of list *link* is 1.

Consider the query propagation related to expression `a`$_1$`+2`, which is assigned a value number 10 at Node 4 in Fig. 1 (a). After the assignment, the query is propagated to the predecessor Node 3. As shown in Table 2, a value number 10 is not included in *reachValue* at Node 3, so the query obtains a *false* answer immediately. Because the predecessor of Node 4 is only Node 3, it is possible to determine that `a`$_1$`+2` is not redundant.

Table 3 Execution time of objective code.

programs	A.PRE*2	B.PREQP	C.EDDPRE	(A-C)/A	(B-C)/B
equake	69.1 sec	65.2 sec	65.5 sec	5.2%	−0.5%
art	35.7 sec	36.1 sec	33.6 sec	5.9%	6.9%
mcf	34.3 sec	33.7 sec	33.7 sec	1.7%	0.0%
bzip2	73.3 sec	77.3 sec	75.4 sec	−2.9%	2.5%
gzip	103 sec	100 sec	99 sec	3.9%	1.0%
ammp	119 sec	118 sec	120 sec	−0.8%	−1.7%
vpr	68.4 sec	72.2 sec	65.9 sec	3.7%	8.7%
parser	102 sec	105 sec	104 sec	−2.0%	1.0%
twolf	110 sec	110 sec	108 sec	1.8%	1.8%

Algorithm 7 Translating program

```
Function: insert(n)
 1: if (var[n] ≠⊥) then
 2:     return var[n]
 3: else if (insertCand[n] ≠⊥) then
 4:     var[n] := createNewVar()
 5:     let newNode := [var[n]" = "insertCand[n]]
 6:     let val := value(newNode)
 7:     setValue(val, lhs(newNode), n)
 8:     add newNode to the exit of n
 9: else if (#1(answer[n]) = false) then
10:     return ⊥
11: else if (| link[n] |= 1) then
12:     var[n] := insert(link[n])
13: else
14:     var[n] := createNewVar()
15:     let args := ∅
16:     for all n'_i ∈ link[n] do
17:         let var_i := insert(n'_i)
18:         if (var_i =⊥) then
19:             return ⊥
20:         end if
21:         add var_i to args
22:     end for
23:     let newPhi := [var[n]" = ""φ""("args")"]
24:     let val := value(newPhi)
25:     setValue(val, lhs(newPhi), n)
26:     add newPhi to the entry of n
27: end if
28: return var[n]

Function: cancel(n)
29: if (#1(answer[n]) = true) then
30:     if (insertCand[n] ≠⊥) then
31:         removeNewNode(n)
32:     else if (| link[n] |= 1) then
33:         cancel(link[n])
34:     else
35:         for all n'_i ∈ link[n] do
36:             cancel(n'_i)
37:         end for
38:         removeNewNode(n)
39:     end if
40: end if
```

5. Experimental Results

We implemented our technique as a low-level intermediate representation converter using a COINS compiler [6]. To evaluate the benefits of our technique as accurately as possible, we compared EDDPRE with PREQP and PRE*2, which applies PRE twice and that also applies copy propagation between them. The machine used in the evaluations had an Intel Core i5-2320 3.00 GHz CPU and Ubuntu 12.04 LTS was the OS.

We evaluated the effects of our technique using three programs (equake, art, and ammp) from CFP2000 and six programs (mcf, bzip2, gzip, vpr, parser, and twolf) from CINT2000 in the SPEC benchmarks.

Table 3 shows the execution time results with PRE*2, PREQP, and EDDPRE. For PREQP and EDDPRE, most of the programs were improved or matched when using EDDPRE. In particular, the efficiency of art and vpr were improved by about 6.9% and 8.7%, respectively. EDDPRE can eliminate more redundancies than PREQP because EDDPRE uses optimistic value numbering. However, compared with PRE*2, moving loop-invariant expression speculatively may decrease the execution efficiency as well as PREQP.

Table 4 shows the analysis time results for PRE*2, PREQP, and EDDPRE, where all of the programs were improved by applying EDDPRE. In particular, the efficiency of twolf was improved by about 70.7% compared with PRE*2. The efficiency of mcf was also improved by about 56.8% compared with PREQP.

Furthermore, **Table 5** shows the query propagation time results of PREQP and EDDPRE, where all of the programs were improved by applying EDDPRE. In particular, the efficiency of mcf was improved by about 88.2%. The number of nodes propagated by the queries using the two techniques is shown in **Table 6**, where EDDPRE visited fewer nodes than PREQP other than equake. PREQP does not generate queries for the expressions with operands that are defined in the node, whereas EDDPRE generates queries for the expressions because EDDPRE does not consider the definition. Thus, it is possible to that EDDPRE generates more queries than PREQP. However, the analytical efficiency of EDDPRE was better than PREQP, as shown in Table 5, because the answers to queries are obtained immediately for non-redundant expressions and EDDPRE does not need to apply copy propagation.

6. Related Work

In this section, we describe some techniques that use PRE and GVN.

The original PRE technique was proposed by Morel and Renvoise [14], which uses bi-directional data-flow analysis. This technique can eliminate some redundancies and move loop-invariant expressions out of loops, but some redundancies are not removed because the technique does not insert new expressions

Table 4 Analysis time.

programs	A.PRE*2	B.PREQP	C.EDDPRE	(A-C)/A	(B-C)/B
equake	1,949 msec	879 msec	677 msec	65.3%	23.0%
art	384 msec	423 msec	271 msec	29.4%	35.9%
mcf	998 msec	866 msec	374 msec	62.5%	56.8%
bzip2	1,018 msec	1,104 msec	745 msec	26.8%	32.5%
gzip	2,669 msec	1,952 msec	1,087 msec	59.3%	44.3%
ammp	10,959 msec	6,035 msec	3,532 msec	67.8%	41.5%
vpr	5,047 msec	4,574 msec	2,498 msec	50.5%	45.4%
parser	3,744 msec	3,945 msec	2,265 msec	39.5%	42.6%
twolf	36,012 msec	14,484 msec	10,546 msec	70.7%	27.2%

Table 5 The time of query propagation.

programs	A.PREQP	B.EDDPRE	(A-B)/A
equake	661 msec	265 msec	59.9%
art	311 msec	74 msec	76.2%
mcf	667 msec	79 msec	88.2%
bzip2	837 msec	279 msec	66.7%
gzip	1,447 msec	260 msec	82.0%
ammp	4,516 msec	1,102 msec	75.6%
vpr	3,369 msec	773 msec	77.1%
parser	2,964 msec	753 msec	74.6%
twolf	10,662 msec	2,635 msec	75.3%

Table 6 The number of nodes which query propagated.

programs	A.PREQP	B.EDDPRE	A-B
equake	31,884	37,328	−5,444
art	10,149	5,949	4,200
mcf	10,271	3,503	6,768
bzip2	37,719	21,626	16,093
gzip	43,167	18,844	24,323
ammp	169,749	90,815	78,934
vpr	108,917	64,483	44,434
parser	96,056	53,484	42,572
twolf	497,177	343,105	154,072

at non down-safe nodes.

Dhamdhere et al. extended this technique to insert expressions on edges [9] and they also proposed another technique that removes redundancies based on uni-directional data-flow analysis [10].

In general, most PRE techniques increase the register pressure because the live-range is extended by inserting new expressions.

Lazy code motion (LCM) aims to suppress the register pressure [12], [13]. Moving expressions by LCM involves performing the first code motion as early as possible and the second code motion as late as possible. The first code motion helps to eliminate all of the removable expressions, while the second one helps to minimize the live ranges of the variables.

Bodik, Gupta, and Soffa proposed the removal of all redundancies by copying certain parts of the program [4]. However, the copying process can change the reducible loops into irreducible loops. It is not possible to use this technique with other optimization techniques, which can only apply programs without irreducible loops. Techniques have been proposed for changing irreducible loops into reducible loops by copying certain parts of the program but the code size was increased.

These techniques need to apply copy propagation repeatedly to reflect second-order effects.

Kennedy et al. proposed SSAPRE, which exploits the properties of the SSA form [11]. SSAPRE produces a factored redundancy graph (FRG) for each expression, before determining the insertion points based on a sparse analysis of the FRG. However, some redundancies are not eliminated because SSAPRE is based on the lexical features of the program.

GVN was extended from the local approach by Rosen et al. [17]. Their approach utilizes a hash table to record the value numbers of each node. Expressions are moved up nodes to check whether the expression is recorded in the hash table of the node and redundant expressions were eliminated. This technique uses query propagation and the loop information to analyze totally redundancy, so it depends on the control flow structure. Furthermore, the technique needs to apply copy propagation repeatedly, like PREQP.

Our technique does not depend on the structure and it performs the analysis effectively because it does not need to use the loop information and copy propagation.

Alpern, Wegman, and Zadeck proposed a GVN technique that uses partitioning to eliminate the totally redundant expressions, which depends on the induction variable [2]. However, their technique does not eliminate all of the redundancies. Ruthing, Knoop, and Steffen proposed a technique that eliminates these redundancies [18], but its analysis was not effective. Nie and Cheng proposed a technique based on the SSA form for sparse analysis, which eliminates as many redundancies as much as Ruthing's technique [15]. Click proposed a technique that extended Alpern's technique by moving loop-invariant expressions out of the loops [5]. This technique requires the loop information because it moves expressions downward without moving into the loop, after moving upward speculatively.

Cooper and Xu proposed a technique that eliminated all of the totally redundant load instructions, which combined GVN with common sub-expression elimination [7]. This technique analyzed the redundancies by assigning value numbers to a tuple that contained the operator of the store/load instruction and the value number of the address.

VanDrunen and Hosking proposed a technique that eliminates partially redundancies, which was similar to our technique [20]. Their technique defines the availability and anticipatability based on value number, but their technique needs to be applied repeatedly to eliminate all of the redundant expressions because it is not based on the data-flow equation of pure PRE.

Odaira and Hiraki proposed the PVNRE technique, which combines GVN and PRE [16]. PVNRE maps the value numbers of the ϕ function and its arguments to another value number to eliminate lexically different partial redundancy. PVNRE also defines the transparency of the back edge to prevent the movement of expressions, which depend on induction variables, outside of the loop. The transparency of back edge means PVNRE is only applicable to programs without irreducible loops.

By contrast, our technique does not depend on the control flow structure and the movement of loop-invariant expressions outside of the loop speculatively.

7. Conclusions

In this paper, we proposed a new effective DDPRE technique (EDDPRE) that analyzes redundancies, which eliminates more redundancies than previous techniques by using optimistic value numbering and recording the reachable value numbers at each CFG node. To demonstrate its effectiveness, we applied our technique to several benchmark programs, which showed that it improved the analytical efficiency in all cases and the execution efficiency of programs in most cases. In future work, we will consider: 1) suppressing the register pressure by only applying our technique to costly expressions; and 2) comparing our technique with other techniques that combine GVN and PRE.

Acknowledgments We would like to thank the reviewers for their valuable comments and suggestions, which helped improve the paper. This work is supported in part by Grants-in-Aid for Scientific Research No.25330089.

References

[1] Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986).

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

[3] Appel, A.W.: *Modern Compiler Implementation in ML: Basic Techniques*, Cambridge University Press, New York, NY, USA (1997).

[4] Bodik, R., Gupta, R. and Soffa, M.L.: Complete removal of redundant expressions, *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI '98*, New York, NY, USA, pp.1–14, ACM (1998).

[5] Click, C.: Global code motion/global value numbering, *Proc. ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, PLDI '95*, New York, NY, USA, pp.246–257, ACM (1995).

[6] COINS, available from ⟨http://coins-compiler.sourceforge.jp/⟩.

[7] Cooper, K.D. and Xu, L.: An efficient static analysis algorithm to detect redundant memory operations, *Proc. 2002 Workshop on Memory System Performance, MSP '02*, New York, NY, USA, pp.97–107, ACM (2002).

[8] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, Technical report, Providence, RI, USA (1991).

[9] Dhamdhere, D.M.: A fast algorithm for code movement optimisation, *SIGPLAN Not.*, Vol.23, No.10, pp.172–180 (1988).

[10] Dhamdhere, D.M. and Patil, H.: An elimination algorithm for bidirectional data flow problems using edge placement, *ACM Trans. Program. Lang. Syst.*, Vol.15, No.2, pp.312–336 (1993).

[11] Kennedy, R., Chan, S., Liu, S.-M., Lo, R., Tu, P. and Chow, F.: Partial redundancy elimination in SSA form, *ACM Trans. Program. Lang. Syst.*, Vol.21, No.3, pp.627–676 (1999).

[12] Knoop, J., Ruthing, O. and Steffen, B.: Lazy code motion, *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, PLDI '92*, New York, NY, USA, pp.224–234, ACM (1992).

[13] Knoop, J., Ruthing, O. and Steffen, B.: Optimal code motion: Theory and practice, *ACM Trans. Program. Lang. Syst.*, Vol.16, No.4, pp.1117–1155 (1994).

[14] Morel, E. and Renvoise, C.: Global optimization by suppression of partial redundancies, *Commun. ACM*, Vol.22, No.2, pp.96–103 (1979).

[15] Nie, J.T. and Cheng, X.: An efficient SSA-based algorithm for complete global value numbering, *Proc. 5th Asian Conference on Programming Languages and Systems, APLAS'07*, Berlin, Heidelberg, pp.319–334, Springer-Verlag (2007).

[16] Odaira, R. and Hiraki, K.: Partial Value Number Redundancy Elimination, *IPSJ Trans. Programming*, Vol.45, No.SIG09 (PRO22), pp.59–79 (2004) (in Japanese).

[17] Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Global value numbers and redundant computations, *Proc. 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88*, New York, NY, USA, pp.12–27, ACM (1988).

[18] Ruthing, O., Knoop, J. and Steffen, B.: Detecting Equalities of Variables: Combining Efficiency with Precision, *Proc. 6th International Symposium on Static Analysis, SAS '99*, London, UK, UK, pp.232–247, Springer-Verlag (1999).

[19] Takimoto, M.: Speculative Partial Redundancy Elimination Based on Question Propagation, *IPSJ Trans. Programming*, Vol.2, No.5, pp.15–27 (2009) (in Japanese).

[20] VanDrunen, T. and Hosking, A.L.: Value-based partial redundancy elimination, *CC*, pp.167–184 (2004).

[21] Zhou, H., Chen, W. and Chow, F.: An SSA-based algorithm for optimal speculative code motion under an execution profile, *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, New York, NY, USA, pp.98–108, ACM (2011).

Yasunobu Sumikawa received his B.S. in mathematics, M.S. in information science from Tokyo University of Science in 2010 and 2012, respectively. His research interests include the compilers. He is a member of ACM and the Japan Society for Software Science and Technology.

Munehiro Takimoto is a professor in the Department of Information Sciences at Tokyo University of Science. His research interests include the design and implementation of programming languages. He received his undergraduate, postgraduate, and doctoral degrees in engineering from Keio University.