Journal of Information Processing Vol.33 1–12 (Mar. 2025)

[DOI: 10.2197/ipsjjip.33.1]

Regular Paper

Demand-driven PRE using Profile Information

Takuna Uemura[1] Yasunobu Sumikawa[1,a)]

Received: March 7, 2024, Accepted: December 10, 2024

Abstract: Partial redundancy elimination (PRE) eliminates redundant expressions that repeatedly compute the same values. Following redundancy elimination, the application of copy propagation reveals additional redundancy, known as second-order effects. Eliminating this type of redundancy requires the iterative application of PRE and copy propagation. To eliminate many second-order effects within a short analysis time, demand-driven PRE (DDPRE) has been proposed. However, existing DDPREs assume that all expressions are executed an equal number of times, potentially resulting in spending analysis time on expressions with limited impact even after redundancy elimination and generating spills in register allocation that reduce the effects of redundancy elimination. This study proposes a novel type of DDPRE called profile-guided DDPRE (PDPRE) that utilizes runtime information to selectively apply DDPRE to areas where redundancy elimination is effective. Additionally, to eliminate second-order effects without executing a combination of redundancy elimination and copy propagation, PDPRE initially applies global value numbering. Subsequently, it visits expressions in the order of high execution counts, and then analyzes the redundancy of each expression. To evaluate the effectiveness of PDPRE, we applied PDPRE and existing DDPREs to the programs of the SPEC CPU2000 benchmark. We found that PDPRE both achieves shorter analysis times compared to the existing DDPREs and yields better execution times in many programs compared to existing DDPREs.

Keywords: Compiler, code optimization, partial redundancy elimination, demand-driven analysis, profile information

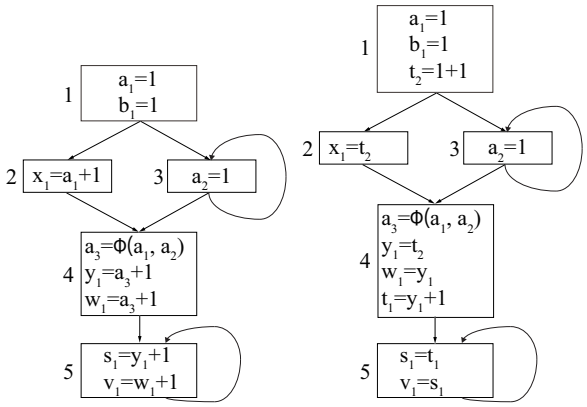
1. Introduction

Partial redundancy elimination (PRE) [7], [10] is a code optimization algorithm employed by compilers to eliminate redundant expressions along specific execution paths. The traditional PREs analyze the entire program using a data-flow analysis to identify redundancies. Subsequently, redundancy is eliminated by transforming the expressions to reference the results of previous executions. The application of copy propagation leads to subsequent modifications in the appearance of subsequent expressions, revealing newly analyzable redundancies. Such redundancies are known as second-order effects.

Demand-driven PRE (DDPRE) [15] has been proposed to efficiently eliminate numerous second-order effects within a short analysis time. Existing DDPREs traverse the control flow graph (CFG) in topological order. During traversal, DDPRE generates queries each time an expression appears in a visited node, and then propagates them toward the start node of the CFG for analyzing redundancy. When the same expression is found as that for the query, the query returns true, whereas if it is difficult to identify whether the analyzing expression is redundant, it returns false. In cases where both true and false are returned in a node, it is concluded that the expression generating the query is partially redundant at that node. To eliminate the redundancy, expressions are inserted into the predecessors that returned false, rendering it fully redundant. It then replaces the expression that generated the query with the result of inserted expression. As query propagation analyzes a portion of the program, it is known that the analysis time is shorter than that of traditional PRE, which analyzes the entire program. However, existing DDPREs assume that all expressions are executed the same number of times. This premise not only leads to the unnecessary analysis of expressions that do not significantly contribute to reducing the execution time of the objective code, but also introduces the issue of increased spills, which are known to reduce the effectiveness of PRE [7]. The redundancy elimination performed by PRE tends to extend the lifetime of variables because it transforms expressions to reference previously computed results. As a result, the usage period of registers occupied by variables is prolonged. The more redundancy is eliminated, the greater the number of variables that remain live simultaneously, which tends to generate spills that transfer data from registers to memory due to insufficient register space. When spills occur, the values of variables that should be recorded in registers are instead stored in the main memory. That is, store instructions for storing the values to the main memory and load instructions for using the variables are inserted. This not only increases the number of executed instructions but also deteriorates execution time because memory references take longer than register references. Thus, to maintain the benefits of reduced execution time gained through redundancy elimination while minimizing the occurrence of spills, it is crucial to prioritize the elimination of redundancies in frequently executed expressions.

In this study, we propose a novel DDPRE algorithm, named profile-guided DDPRE (PDPRE), that acknowledges the fact that the number of executions per expression varies in practice. To prioritize the elimination of redundancies in frequently executed expressions, PDPRE visits only the top-ranking CFG nodes based on execution count, generating queries to analyze redundancies

[1] Takushoku University, Tokyo, 193–0985, Japan
[a)] ysumikaw@cs.takushoku-u.ac.jp



(a) Original CFG (b) Result

Fig. 1 Example of PDPRE

Node	Freq.
1	50
2	20
3	1,500
4	50
5	1,000

(a)

Exp.	Val. num.
1	{1}
a_1	{1}
b_1	{1}
{1}+{1}	{2}
x_1	{2}
a_2	{1}
a_3	{1}
y_1	{2}
w_1	{2}
{2}+{1}	{3}
s_1	{3}
v_1	{3}

(b)

Fig. 2 Tables of Fig. 1

each time an expression appears. However, visiting the CFG in this order may fail to eliminate second-order effects if the query analyzes redundancy by lexical matching. To enable redundancy analysis without relying on lexical consistency, PDPRE preprocesses a program by transforming it into the static single assignment (SSA) form and applies global value numbering (GVN) [12]. GVN assigns the same value number to expressions that compute the same value, thereby eliminating all fully redundant expressions among expressions with different lexical representations. PDPRE eliminates partial redundancy by analyzing whether an expression generates the same value number as a query. During the process of expression insertions, PDPRE endeavors to minimize the number of execution for the expressions to be inserted.

Figure 1 illustrates how PDPRE eliminates redundancies using runtime information. Before applying the query propagation of PDPRE, we assume that the tables (a) and (b) shown in Fig. 2 are available. These tables list the execution counts of all nodes and the value numbers for all expressions, respectively. Notably, in this paper, we use {} to represent value number. In this example, attention is focused on Nodes 2, 4, and 5 of Fig. 1 (a), which contain expressions involving arithmetic operations. Comparing the number of times those nodes are executed, PDPRE first visits Node 5. It then analyzes redundancy of the expression y_1+1. Referring to the table (b) presented in Fig. 2, the value number for y_1+1 is {3}. This is because the value of s_1 is defined by the result of the calculation of that equation. In other words, they have the same value; therefore, the value number of expression matches the one of s_1. PDPRE generates two queries to analyze the redundancy of the expression with value number {3} and propagates them to the predecessor Nodes 4 and 5. We first look at the query propagation for the Node 4. As Node 4 does not contain an expression with the same value number, the query further visits predecessor Nodes 2 and 3. These queries finally visit Node 1; however, no expression with the same value number appears in the visited nodes. Consequently, all queries from Nodes 1 to 4 return false. We next look at the other query propagation from Node 5 to itself. This query returns true because the query finds the original expression. These two queries indicate that the expression y_1+1 is partially redundant at Node 5. To eliminate this redundancy, the path reaching Node 4, where the query returns false, is selected for the insertion of y_1+1. Among the nodes where the query returns false, Node 2 has the minimum execution count. However, the defined variable at Node 2 is not usable at Node 4 because the variable is not defined if the control flow reaches Node 4 through Nodes 1 and 3; in other words, Node 2 does not dominate Node 4. Thus, Node 2 is deemed inappropriate as a node to perform the insertion. We simply call the node where we perform the expression insertion insertion node. PDPRE selects Node 4 as the insertion node because the execution count is the minimum among the suitable insertion nodes. PDPRE inserts t_1=y_1+1 to Node 4. It then replaces the right-hand side of s_1=y_1+1 with t_1. Next, we analyze the redundancy of w_1+1. Its lexical representation is different from y_1+1; however, GVN identifies this redundancy by assigning the same value to the two variables y_1 and w_1 because the two variables are defined by the same expressions, meaning that these expressions compute the same value. Thus, we replace w_1+1 with s_1.

After eliminating the redundancy, PDPRE proceeds to visit Node 4 where this is the node with the second highest execution count among the focusing three Nodes 2, 4, and 5. For the right-hand side of y_1=a_3+1 in Node 4, PDPRE propagates queries to predecessor Nodes 2 and 3. Node 2 contains an expression with the same value number, whereas Nodes 3 and 1 do not. Consequently, a_3+1 is identified as partially redundant at Node 4. A new expression is inserted into the path that reaches Node 3, where the query returns false. In this case, Node 1 is a suitable insertion node because it has a lower execution count than Node 3 and dominates Node 3. However, because the definition of a_3 does not reach Node 3, PDPRE uses other variables or values that share the same value number. In this example, the inserted statement involves t_2=1+1. After inserting this statement, the right-hand side of y_1=a_3+1 is replaced by t_2. As the right-hand side of w_1=a_3+1 computes the same value as the left-hand side of y_1=t_2, it is further simplified to w_1=y_1. Figure 1(b) shows the results of applying PDPRE to all expressions.

As shown in the example above, PDPRE reduces the number of visiting nodes and of eliminating expressions, thereby shortening the analysis time for DDPRE and improving the execution time of the objective code by minimizing the occurrence of spills. To assess the effectiveness of the proposed algorithm, we implemented PDPRE in the COINS compiler. We performed comparative evaluations using programs from the SPEC CPU2000 benchmark and existing DDPREs. First, by decreasing the number of nodes that are analyzed, we verified that the actual analysis time would become shorter. Next, in order to determine the optimal percentage of nodes with high execution frequency for which redundancy elimination should be applied, we examined the number of programs that showed improved execution time. The analysis revealed that targeting the top 10% was most effective. Based on this result, we compared all existing DDPRE algorithms with PDPRE. We evaluated how much analysis time was reduced by limiting the scope to the top 10%. The results confirmed that PDPRE significantly reduced the analysis time for all programs. While we also observed a reduction in the number of redundant expressions eliminated due to the narrowed scope, PDPRE had the highest number of programs with the shortest execution time. To investigate the reason for this, we counted the number of spills generated by applying each DDPRE and found that, for many programs, PDPRE resulted in the fewest spills. The conclusion drawn from the experiments is that PDPRE performs faster analysis than the existing DDPREs, without compromising the quality of the objective code.

2. Preliminaries

2.1 Program Representation

When applying PDPRE, it is assumed that each expression has at most one operator known as the three-address code. For example, if an expression is of the form $z = a + b + c$, which contains two operators, it is split into separate statements to ensure that each right-hand side has only one operator. In this case, the transformation results in statements such as $t = a + b; z = t + c$.

When applying PDPRE, the target program is assumed to be represented by a CFG consisting of a set of basic blocks **N**, a set of edges **E**, a start node **start**, and an end node **end**. Note that both **start** and **end** are empty statements. We represent the sets of predecessors and successors of node n as $pred(n)$ and $succ(n)$, respectively.

When node m exists in all the execution paths leading from **start** to node n, it is said that m dominates n [1]. In this paper, the predicate $dom(m, n)$ denotes that m *dominates* n. If there exists a path from m to **end** through n such that n is the first node not dominated by m, n is in the *dominance frontier* of m. We represent the dominance frontier as the predicate $dfront(m, n)$.

PDPRE is assumed to take a program that has been transformed into the SSA form as input. This implies that the program is in a form in which each variable has only one definition. In SSA form, when multiple definitions reach a single usage point, a new variable is introduced at the point of confluence using a ϕ function. This definition allows the program to handle cases in which multiple definitions converge without compromising the semantics. The node where the ϕ function is inserted is the dominance frontier of the node that defines the argument variables.

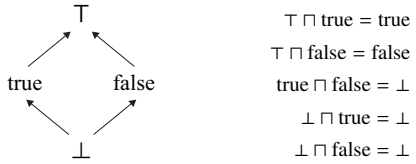


Fig. 3 Semi-lattice of answer space

2.2 Availability and Anticipability

We define four terms: *available*, *up-safe*, *anticipable*, and *down-safe*. These definitions are generally based on an analysis of the lexical consistency of the expressions being analyzed [2]. However, we extend them to be based on value numbers as our analyses use value number consistency rather than lexical consistency.

A value number v of expression e is *available* at n iff v is computed on any path p from **start** to n. We use $comp(v_e, n)$ to denote the expression e whose value number v is computed at n. When v is available at n, n is *up-safe* with respect to v. v is *partially available* at n iff there is at least one path from **start** to n where v is computed. When v is available at n, it is fully redundant. When v is partially available at n, it is partially redundant. If v is partially redundant, it is made fully redundant by inserting expressions whose value numbers are v.

A value number v of e is *anticipable* at node n iff e is computed along any path r from n to **end**. When v is anticipable at n, n is *down-safe* with respect to v. PRE inserts expressions at the down-safe nodes without extending the lengths of any path.

2.3 Answer Space

Our query propagation determines answers defined on semi-lattice $(\mathcal{A}, \sqcap, \top, \bot)$ that is shown in Fig. 3, where $\mathcal{A}$ is a set of answers, $\sqcap : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is a meet operator on $\mathcal{A}$, $\top \in \mathcal{A}$ is a top element, and $\bot \in \mathcal{A}$ is a bottom element. For all $a \in \mathcal{A}$, the top and bottom elements are defined as $\top \sqcap a = a$ and $\bot \sqcap a = \bot$, respectively.

3. Related Work

PRE analyzes both directions from each expression to **start** and to **end**, to detect partially redundant expressions. Early PRE simultaneously performs these analyses using bidirectional data-flow equations [10]. However, this approach has limitations as it cannot detect certain redundancies and sometimes leads to unnecessary code motion. Lazy code motion (LCM) addresses these issues by combining unidirectional data-flow equations to perform analyses in both directions [7], [8]. However, LCM does not eliminate all redundancies. The redundancies that LCM does not eliminate fall into two categories: those that cannot be eliminated without inserting expressions into nodes that are not down-safe, and those that differ in their lexical form.

Several algorithms have been proposed to eliminate the two types of redundancies mentioned above. To eliminate the first type of redundancies, an algorithm has been proposed that either duplicates the program to preserve down-safety while eliminat-

ing all redundancies [2], or algorithms perform speculative insertions that ignore down-safety. To eliminate the second type of redundancies, there are algorithms that combine PRE with GVN [11], [16]. These algorithms utilize value numbers obtained through GVN rather than lexical matching in the data-flow equations used for redundancy analysis. Additionally, several DDPRE algorithms have been proposed to combine redundancy analysis by query propagation and copy propagation or GVN. The algorithms particularly related to PDPRE are those involving speculative code motion, the combination of PRE and GVN, and DDPRE. Thus, in Section 3.1, we review speculative code motion algorithms, and in Section 3.2, we explore the combination of GVN and PRE. We then focus on broad studies of DDPRE in Section 3.3.

3.1 Speculative Code Motion

PRE typically prohibits modification of program semantic before and after applying code optimizations using down-safety. However, for programs in which exception handling is unnecessary or instructions without exceptions occur, moving instructions to non-down-safe execution paths may merely increase the execution time of the program without introducing exceptions. In such cases, if eliminating redundancies leads to a reduction in the number of executed expressions, thereby outweighing the drawbacks of inserting expressions, such elimination may be beneficial. Following this principle, min-cut PRE (MC-PRE) was proposed [3], [17]. MC-PRE considers insertion points as points dividing the original program into two and determines the point with the minimum cost using the minimum-cut. While there is a possibility that the inserted expressions will be newly computed along some execution paths, the overall program benefits from a reduction in the number of executed expressions.

Although MC-PRE has the potential to improve program execution time, it involves additional computations of minimum cuts after analyzing the availability and anticipability like LCM; therefore, it tends to have longer analysis times than the non-speculative PREs discussed earlier. An algorithm has been proposed to achieve speculative code motion while minimizing analysis costs [6]. In this algorithm, there is a trade-off between reducing analysis costs and limiting the elimination of redundancies. However, the state-of-the-art speculative PRE utilizes the bounded tree-width of the CFG, providing faster analysis with the same execution time as MC-PRE [9].

An algorithm of profile-based PRE targeting SSA form is proposed as MC-SSAPRE [19]. MC-SSAPRE is an extension of SSAPRE [4], which aims to improve analysis efficiency by analyzing partial redundancy on SSA graphs. This extension seeks to enhance the SSAPRE by determining insertion points for expressions through a minimum-cut approach.

These algorithms are effective in eliminating many redundancies that are lexically equal. However, reflecting second-order effects requires multiple applications, often in conjunction with copy propagation. Thus, the analysis time for these algorithms tends to be higher than that for non-speculative algorithms.

3.2 Combining GVN and PRE

To eliminate the second-order effects of PRE without applying copy propagation, GVN-PRE [16] and PVNRE [11] are proposed. GVN-PRE targets SSA form and assigns the same value number to expressions with identical values, even if their lexial forms differ. This algorithm eliminates many redundancies by utilizing data-flow analysis designed to leverage these value numbers. However, the data-flow equations proposed in GVN-PRE are designed outside the traditional PRE framework. To achieve results comparable to other PRE algorithms, such as LCM, it is necessary to either extend GVN-PRE or apply the algorithm iteratively. PVNRE also uses value numbering to eliminate partially redundant expressions. In PVNRE, value numbers are assigned to ϕ functions and their arguments to accommodate syntactic transformations at ϕ functions. Additionally, to prevent the movement of expressions dependent on induction variables within loops outside the loop, PVNRE ensures the transparency of the back edge. This implies that PVNRE operates under the assumption that the loop structure is reducible.

PDPRE does not have the limitations present in GVN-PRE or PVNRE, allowing it to effectively eliminate redundancies with a single application for any given program.

3.3 DDPRE

DDPRE propagates queries to examine the availability for redundancy elimination. Existing DDPREs visit the CFG in a topological order and propagate queries toward **start** whenever an expression appears. For example, PREQP [15] returns true if the query finds expressions with the same lexical representation. If the query encounters instructions that modify the values of the query's expression or visits **start**, it returns false. Another algorithm, EDDPRE [14], extends this query propagation by analyzing the occurrences of expressions with the same value number rather than lexical equality. EDDPRE first applies GVN. Then, the query returns true if it finds the same value number and false if it propagated to **start**. These DDPREs perform optimistic answer determination when encountering the same node within a loop, anticipating that the expression will be computed redundantly in the loop. While this optimistic algorithm facilitates the movement of loop-invariant expressions to outside the loop, it may lead to an inability to eliminate certain redundancies in programs with loops containing multiple exits.

To address the issues with PREQP and EDDPRE, LDPRE [18] introduces a symbol $\top$, which denotes the undecidability of the answer, to construct a lattice of answer space defined in Section 2.3. LDPRE proposes query propagation that uses the lattice. The possible results returned by these queries include true, indicating that the same expression has actually appeared; false, indicating that there is no occurrence of the same expression along the propagated path of the query; and $\top$, indicating that the query has visited the same node twice. To find the repeated visiting, LDPRE uses $\bot$ indicating that the query has already been visited.

All prior DDPREs assume that every expression within the program is executed an equal number of times.

Algorithm 1 Algorithm overview

```
 1: Function PDPRE(FrecInfo, k)
 2:   nodes ← SortingNodesByFreq(FrecInfo)
 3:   worklist ← GetTopFreqNodes(nodes, k)
 4:   GVN()
 5:   while | worklist |> 0
 6:     n ← worklist.pop() // Visiting a node
 7:     foreach e ∈ n
 8:       v_e ← val(e)
 9:       ans ← NAvail(e, v_e, n) // Query propagation
10:       if ans = true
11:         Eliminate(e)
```

4. Algorithm

PDPRE assumes that the execution count information for all nodes has been preacquired and is already available for reference before analyzing redundancies.

Algorithm 1 outlines the key steps of PDPRE, CFG traversing using execution count information, GVN application, and subsequent query propagation for redundancy analysis. PDPRE first sorts nodes based on their execution counts given as the argument *FrecInfo* (line 2). PDPRE applies the function *SortingNodesByFreq* to sort and stores the result in the list *nodes*. It then defines the stack *worklist* as collecting only top k frequent nodes by applying the function *GetTopFreqNodes* to the *nodes*. Subsequently, the algorithm applies GVN to assign value numbers to all expressions in the program (line 4). The main loop, spanning lines 5 to 11, iterates over the nodes extracted from the *worklist*. Each expression within the visiting node is retrieved individually (line 7). The algorithm then obtains the value number associated with the expression (line 8). To analyze the redundancy of this value number, the algorithm invokes the *NAvail* function for query propagation (line 9), as detailed in Section 4.3. If the query propagation result is true, indicating the presence of redundancy, the *Eliminate* function is executed to perform program transformations, replacing the occurrences of the expression with the left-hand side of the prior statements sharing the same value number (line 11).

Subsequently, we describe the processes of CFG traversal, GVN algorithm employed in PDPRE, and redundancy analysis through queries using value numbers.

4.1 CFG Traversing

PDPRE prioritizes the visits of nodes in order of high execution count to maximize the effectiveness of redundancy elimination. This is accomplished by initially recording the nodes in the *worklist*, prioritizing nodes with higher execution counts. However, visiting nodes in this order may lead to a limitation, meaning that simple lexical matching through queries is insufficient for eliminating redundancy in expressions with different lexical equality.

We discuss this issue using the example illustrated in Fig. 1(a). In this program, the expressions $\mathtt{y}_1$+1 and $\mathtt{w}_1$+1 in Node 5 have different lexical forms. However, looking at Node 4, the variables, $\mathtt{y}_1$ and $\mathtt{w}_1$, are defined by the same expression, indicating that both $\mathtt{y}_1$+1 and $\mathtt{w}_1$+1 compute the same value. If lexical matching is used for redundancy analyzing, it is necessary to visit Node 4 first to replace the right-hand side of the definition for $\mathtt{w}_1$ with $\mathtt{y}_1$. It then applies copy propagation to change $\mathtt{w}_1$+1 to $\mathtt{y}_1$+1; this result reveals the redundancy of the expression. However, because Node 5 has a higher execution count than Node 4, the visiting order prioritizes the redundancy analysis in Node 5 over Node 4. This leads to a situation in which the aforementioned expression transformation cannot be achieved.

To address this issue, as mentioned in Section 1, PDPRE uses the value numbers obtained through GVN for query analysis. GVN assigns the same value number to $\mathtt{w}_1$+1 and $\mathtt{y}_1$+1, as shown in table (b) of Fig. 2. Consequently, PDPRE eliminates this redundancy.

4.2 GVN

GVN assigns the same value number to expressions that compute the same value, regardless of their lexical equality. This feature enables the detection of redundancies without relying on the lexical representation of expressions. While traditional GVN is able to eliminate redundancies beyond nodes, we skip the process. This distinction is made because PDPRE addresses inter-node redundancies through query propagation, as explained in Section 4.3.

When another expression with the same value number exists earlier within the same node, GVN makes the later expression by referring to the previous computation to eliminate redundancy. To facilitate the elimination of such redundancies, GVN maintains a local table that records the value numbers specifically within each node. Additionally, GVN utilizes a global table denoted as *valTable*, to manage the value numbers of all expressions globally. This allows for comprehensive and coordinated handling of value numbers across the entire program.

We describe the procedure for assigning a value number to each expression in detail. For an assignment statement, the first step is to obtain the value numbers of the terms on the right-hand side. To ease presentation, we represent the assignment statement as $t_0 = t_1 \oplus t_2$. Initially, the value number for each term on the right-hand side is retrieved from *valTable*. If t_1 is not recorded in the table, a new value number *nv* is generated, and an entry with t_1 as the key and *nv* as the value is added to *valTable*. The same procedure is performed to t_2. Subsequently, it retrieves the value number for the expression; likewise, the value number for $t_1 \oplus t_2$ is acquired through *valTable*. As the left- and right-hand sides of the assignment share the same value number, an entry with t_0 as the key and the value number of $t_1 \oplus t_2$ is added to *valTable*. For trivial assignments in the form of $t_0 = t_1$, the value number on the right-hand side is just assigned to the left-hand side.

For ϕ function, if all arguments share the same value number, the variable defined by the ϕ function is assigned the same value number. Otherwise, each argument is converted into a value number using *valTable*, and then the value number for that combination is obtained using *valTable*. Finally, GVN adds an entry with the variable defined by the ϕ function as the key and this value number as the value to *valTable*.

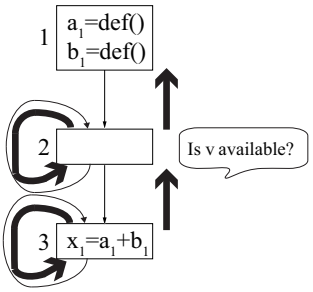


Fig. 4 Query propagation of PDPRE

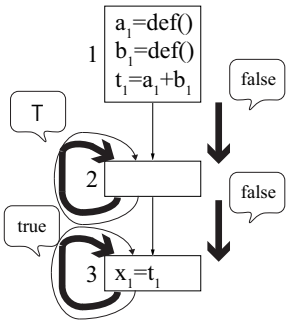


Fig. 5 Result of applying PDPRE for Fig. 4

4.3 Availability Analysis by Query Propagation

We here assume that v_e is the value number of analyzing expression e. PDPRE propagates a query *Is v_e available?* to each predecessor for analyzing its redundancy. The result of this query, similar to the LDPRE, forms an answer space over the lattice defined in Section 2.3.

Figure 4 illustrates the process of analyzing the availability in PDPRE. In this example, we focus on $\mathtt{a_1}$+$\mathtt{b_1}$ at Node 3. Let v be the value number for this expression. Because there are two predecessors (Nodes 2 and 3) in Node 3, two queries are generated to analyze whether v exists in the destination nodes. The query propagating to Node 2 continues further to predecessor Nodes 1 and 2 as Node 2 does not have any expression. Node 1 does not have an expression assigned to v and has no predecessors, leading to an inconclusive result, and the query returns false. Looking at the other query propagated from Node 2 to itself, it encounters a revisited node to analyze the same value number v. As further analysis would only confirm the same result, this query returns $\top$ to indicate an undetermined result. Consequently, the answers returned from the predecessors at Node 2 are false and $\top$. The final answer at the entry point of Node 2 is determined as false because PDPRE applies the meet operator $\sqcap$ to $\top$ = {true, false} and false. This result is also propagated from Node 2 to the entry point of Node 3. The query propagated to Node 3 from the other predecessor (Node 3) returns true because the destination Node 3 contains the expression associated with the query. Thus, the answers returned from the predecessors at Node 3 are false and true, indicating that the expression $\mathtt{a_1}$+$\mathtt{b_1}$ is partially redundant at Node 3. To insert expression, PDPRE selects a node where a query returned false. In this example, we have an assumption that the execution count of Node 1 should be fewer than Node 2 because Node 2 is a node of a loop. Thus, PDPRE inserts the statement $\mathtt{t_1}$=$\mathtt{a_1}$+$\mathtt{b_1}$ into Node 1, and then transforms the statement in Node 3 to $\mathtt{x_1}$=$\mathtt{t_1}$. The result is shown in Fig. 5.

PDPRE assumes that each node is a basic block; thus, analyzing whether v_e is available at that node is corresponding to an exit analysis. If the availability cannot be determined at the exit, an analysis of the availability at the entry point of the node is conducted by summarizing the results of propagating the query to the predecessors. The availabilities at the exit and entry points of a node are determined using the data-flow equations *XAvail* and *NAvail*, respectively.

XAvail categorizes the answer determination rules into three types: 1) If the value number v_e occurs in node n, then availability is true. 2) If queries are propagated to predecessors, the answer at the entry point is directly used as the answer at the exit point. 3) If the query repeatedly visits the same node, the algorithm uses a memorization mechanism to avoid infinite loops. To determine the last type, PDPRE checks whether queries with the same value number have already visited the same node by referencing the memorization table *visit*. This corresponds to the situation in Fig. 4, where a query propagated from Node 2 to Node 2 returns $\top$ as the answer. This memorization table is used not only to simply record $\top$, but also to record the answer when either true or false is determined as the answer. For instance, in the example in Fig. 1, the situation where the query repeatedly visited Node 1 corresponds to the use of this table. If previous query propagation for the same value number determines any answer, the current query returns the answer determined before. The sequence of application for these three types of rules is as follows: If the current query involves visiting the node for the second time, the query returns $\top$ or the answer determined during the previous visit; otherwise, it proceeds with analysis using $comp(v_e, n)$. If $comp(v_e, n)$ is false, the query is propagated to the predecessors. The data-flow equations for determining these query answers can be defined as follows:

$$XAvail(e, v_e, n) \overset{def}{\Leftrightarrow} \begin{cases} \top & \text{if } visit[(v_e, n)] = \bot \\ \text{true} & \text{if } visit[(v_e, n)] = \text{true} \\ \text{false} & \text{if } visit[(v_e, n)] = \text{false} \\ \text{true} & \text{if } comp(v_e, n) \\ NAvail(v_e, n) & \text{otherwise} \end{cases} \quad (1)$$

The determination process for *visit*, which is not explicitly represented in the above equation, is described in the pseudo-code in Algorithm 2. Lines 2 to 5 check whether a query analyzing the same value number has already been visited. This corresponds to the first three conditions of Eq. 1. If the current visit is the first visit for that value number, line 6 analyzes whether the same value number appears in the current node using the predicate $comp(v_e, n)$. If the answer for node n is still undetermined after completing the analysis of the availability within the node, line 9 records $\bot$, indicating that n has been visited, in the table $visit[(v_e, n)]$. Subsequently, when propagating the query to the predecessor with the function *NAvail* and obtaining its answer, the value of the $visit[(v_e, n)]$ is updated (line 10).

Next, we present the data-flow equation for *NAvail*. This represents the answers to the queries returned by the predecessors. To obtain the answers, the query first checks whether the visiting node has predecessors. If the visited node is **start**, there are

Algorithm 2 Pseudo codes of *XAvail*

```
1:  Function XAvail(e, v_e, n)
2:    if visit[(v_e, n)] = ⊥
3:      return ⊤
4:    if visit[(v_e, n)] = true | | visit[(v_e, n)] = false
5:      return visit[(v_e, n)]
6:    if comp(v_e, n)
7:      visit[(v_e, n)] ← true
8:      return visit[(v_e, n)]
9:    visit[(v_e, n)] ← ⊥
10:   visit[(v_e, n)] ← NAvail(e, v_e, n)
11:   return visit[(v_e, n)]
```

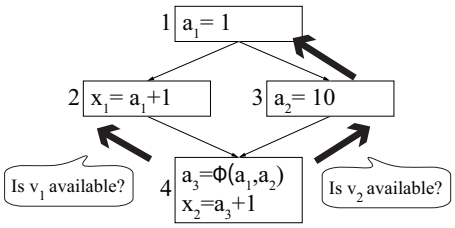


Fig. 6 Query propagation of PDPRE

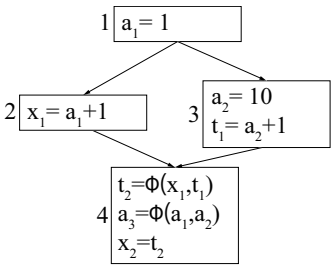


Fig. 7 Result of PDPRE application for Fig. 6

no further predecessors; thus, the answer is false. Otherwise, the query is actually propagated to the predecessors. If at least one of the predecessors returns true, indicating that it is fully or partially redundant, then the answer at the entry point of the node is also true. This is because PDPRE performs insertion to make partially redundant fully redundant. Otherwise, PDPRE applies a meet operator to the answers returned from predecessors and determines it as the answer at the entry point.

Figure 6 illustrates a detailed example of query propagating to predecessors. As PDPRE targets programs transformed into the SSA form, where what was a single variable in normal form is considered as multiple variables, ϕ functions are inserted at their merge points. In this example, the variable a has definitions in two places, Nodes 1 and 3, in the normal form program. Node 4 is a merge point where multiple definitions of a reach. If control passes through Node 2, the use of a in Node 4 should use the value defined in Node 1. If control passes through Node 3, the use of a in Node 4 should use the value defined in Node 3. In SSA form, a ϕ function is used to switch between these reaching values. Therefore, when propagating queries to predecessors, it is necessary to check whether the variable used in that expression is defined by a ϕ function. When generating queries for the expression a_3+1 in Node 4, the queries are propagated to predecessor Nodes 2 and 3. Because a_3 is defined by a ϕ function, it should be replaced with a_1 for propagating the query to Node 2. When propagating the query to Node 3, it should be replaced by a_2. As variables change, their value numbers may also change; therefore, the value numbers analyzed by the query should be updated accordingly. While performing this replacement, redundancy is analyzed, and the result of the program transformation is shown in Fig. 7. We represent these processes as the following data-flow.

$$
\begin{aligned}
e_p &\overset{def}{\Leftrightarrow} BUpdate(e, n, p) \\
v_{e_p} &\overset{def}{\Leftrightarrow} val(e_p) \\
PAns &\overset{def}{\Leftrightarrow} \prod_{p \in pred(n)} XAvail(e_p, v_{e_p}, p) \qquad (2) \\
NAvail(e, v_e, n) &\overset{def}{\Leftrightarrow} \begin{cases} \text{false} & \text{if } n = \textbf{start} \\ \text{true} & \text{if } Insert(e, v_e, n) \\ PAns & \text{otherwise} \end{cases} \qquad (3)
\end{aligned}
$$

Note that the function *BUpdate* returns the result of replacing variables defined by a ϕ function when propagating queries to predecessors. The second condition in Eq. 3 represents a scenario where queries return both true and false at a node, indicating that PDPRE should execute the insertion of the expression.

We present the pseudo-code for Eq. 3 in Algorithm 3. Lines 2∼4 address cases in which the currently visited node is **start**. The variable *alist* in line 5 records the answers returned from the predecessors. The variable *fplist* on line 6 records the nodes that are candidates of expression insertion. Lines 7∼10 perform query propagation on their predecessors. The function *BUpdate* on line 8 is used to modify the query's expression when propagating a query beyond ϕ functions. The results of this propagation are recorded on line 11. If the answer is false or ⊤, the predecessor is recorded in *fplist* as the insertion candidate node for later use. Lines 14∼29 represent the processes that follow the results of propagating queries to all predecessors. The function *isPRedundant*(*alist*) checks whether the value number v_e is partially redundant by evaluating Eqs. 6 and 7 that are defined in the next subsection. After checking that all nodes of *fplist* satisfy the down-safety, expressions are inserted to make it fully redundant (lines 16∼18), and then a ϕ function is inserted if the visiting node is a dominance frontier of the one of the insert nodes (lines 19∼22). If v_e is not partially redundant at n, line 26 uses the function *SQCAP*, which corresponds to Eq. 2, to determine the answer at the entry point of this node by applying the meet operation to the answers returned by the predecessors. If this result makes the answer of this node false, the node is added to *fplist* as an insertion candidate node. After adding nodes of *fplist* to *FalseNs* such that it can be referenced globally, the value of $visit[(v_e, n)]$ is returned (line 30).

4.4 Insertion

PDPRE inserts expressions according to the answers obtained by query propagation. Therefore, after the answers are obtained from all predecessors, the following steps are performed. Initially, the algorithm checks to see if the answers include true to confirm the analyzing expression is at least partially redundant. If the answers have false, it then checks to see if the predecessor that returned the answer is down-safe. PDPRE analyzes the down-

Algorithm 3 Pseudo codes of *NAvail*

```
1: Function NAvail(e, v_e, n)
2:   if n is start
3:     visit[(v_e, n)] ← false
4:     return visit[(v_e, n)]
5:   alist ← [] // This is used for recording answers
6:   fplist ← [] // For insertion
7:   foreach p ∈ pred(n)
8:     e_p ← BUpdate(v_e, n, p)
9:     v_{e_p} ← val(e_p)
10:    a ← XAvail(e_p, v_{e_{n_p}}, p)
11:    alist.append(a)
12:    if a = false | | a = ⊤
13:      fplist.append(p)
14:  if isPRedundant(alist) && PXDAns // Corresponding to Eq. 8
15:    isPhiNode ← false
16:    foreach p ∈ fplist
17:      p' ← InsertNodeSelection(e, p)
18:      Insert a new statement to p'
19:      if dfront(p', n)
20:        isPhiNode ← true
21:    if isPhiNode
22:      Insert a φ function to n
23:    visit[(v_e, n)] ← true
24:    FalseNs ← []
25:  else
26:    visit[(v_e, n)] ← SQCAP(alist)
27:    if visit[(v_e, n)] = false
28:      fplist.append(n)
29:    FalseNs.extend(fplist)
30:  return visit[(v_e, n)]
```

safety by propagating queries that analyze the value number from the insertion candidate node toward **end** to check down-safety. This analysis is essentially the inverse of availability analysis.

The query first checks whether the analyzing value number exists in the visiting node, indicating that it examines the down-safety at the entry point of the node. Even if the currently visited node does not have the same value number, queries are propagated to successors, meaning that the down-safety for successors is examined at the exit of the current node. These down-safety conditions at the entry and exit of the nodes are denoted by $NDSafe(v_e, n)$ and $XDSafe(v_e, n)$, respectively.

$NDSafe(v_e, n)$ is true if the value number v_e appears in the currently visited node n. As the analysis of down-safety is performed through query propagation, there can be cases in which queries examining the same value number repetitively visit the same node, such as within a loop. To prevent this repetition, a table $visit[(v_e, n)]$ is used to record the results of query propagation, similar to the availability analysis. If true or false is recorded in this table, the result is returned before analyzing the value number at n. Otherwise, $XDSafe(v_e, n)$ ensures that the results of queries propagated to the successors become answers at the entry point of this node. This definition is given in Eq. 4.

Algorithm 4 Selecting nodes for insertion

```
1: Function InsertNodeSelection(e, n)
2:   blk ← n
3:   min ← Freq(n)
4:   foreach fn ∈ FalseNs
5:     if dom(fn, n) && min > Freq(fn) && ReachTVals(e)
6:       blk ← fn
7:       min ← Freq(fn)
8:   return blk
```

$$
\begin{aligned}
IsTrue &\overset{def}{\Leftrightarrow} visit[(v_e, n)] = \text{true} \\
IsFalse &\overset{def}{\Leftrightarrow} visit[(v_e, n)] = \text{false} \\
NDSafe(e, v_e, n) &\overset{def}{\Leftrightarrow} \begin{cases} \text{true} & \text{if } IsTrue \\ \text{false} & \text{if } IsFalse \\ \text{true} & \text{if } comp(v_e, n) \\ XDSafe(e, v_e, n) & \text{otherwise} \end{cases}
\end{aligned} \tag{4}
$$

$XDSafe(e, n)$ propagates queries to successors; thus, it checks whether there are variables used in the expression being analyzed by the query are employed as arguments in ϕ functions in the propagated node s. If such a ϕ function exists, the variable in the expression of that query is replaced with the left-hand side of the ϕ function, and the analyzed value number is also updated.

The rules of answer determination of *XDSafe* are defined as follows: If it is down-safe at the entry point of all successors, it is also down-safe at the exit of n. However, if n is an exit node, and thus has no more successors, false is returned as the answer. The following data-flow equation indicates these steps.

$$
\begin{aligned}
e_s &\overset{def}{\Leftrightarrow} FUpdate(e, s, n) \\
v_{e_s} &\overset{def}{\Leftrightarrow} val(e_s) \\
SNDAns &\overset{def}{\Leftrightarrow} \prod_{s \in succ(n)} NDSafe(e_s, v_{e_s}, s) \\
XDSafe(e, v_e, n) &\overset{def}{\Leftrightarrow} \begin{cases} \text{false} & \text{if } n = \textbf{end} \\ SNDAns & \text{otherwise} \end{cases}
\end{aligned} \tag{5}
$$

The function *FUpdate* updates the analyzing expression and its value number when propagating queries to successors.

If the down-safety is satisfied, PDPRE performs the insertion. The data-flow equation for *Insert* is defined as follows:

$$
PXDAns \overset{def}{\Leftrightarrow} \prod_{p \in InsertCand(n)} XDSafe(e_p, v_e, p)
$$

$$
A_{v_{e_p}} \overset{def}{\Leftrightarrow} XAvail(e, v_e, p)
$$

$$
NoTrue \overset{def}{\Leftrightarrow} |\{A_{v_{e_p}} = \text{true} \mid p \in pred(n)\}| = 0 \tag{6}
$$

$$
HasFalse \overset{def}{\Leftrightarrow} |\{A_{v_{e_p}} = \text{false} \mid p \in pred(n)\}| > 0 \tag{7}
$$

$$
Insert(e, v_e, n) \overset{def}{\Leftrightarrow} \begin{cases} \text{false} & \text{if } NoTrue \\ PXDAns & \text{else if } HasFalse \\ \text{false} & \text{otherwise} \end{cases} \tag{8}
$$

In the equation, $|\bullet|$ is the size of set $\bullet$.

After deciding to perform the insertions, PDPRE determines the specific insertion nodes. We present the pseudo-code that searches for the node with the fewest executions to insert an expression in Algorithm 4. This function takes the predecessor n,

which has been determined to be the insertion point based on Eq. 8, as an argument. The return value of this function is node *blk*, where the expression is inserted. Lines 2∼3 initialize *blk* with *n*, assuming that *n* is the node for expression insertion, and *min* with the execution count of *n*. The loop at lines 4 to 7 iterates through nodes retrieved individually from the queue *FalseNs*, which records nodes where the query returned false or ⊤. For each retrieved node *fblk*, it checks whether *fblk* dominates *n*, whether the execution count of *fblk* is smaller than *min*, and all terms of the expressions are usable at *fblk*. The function *ReachTVals*(*e*) checks the usability of each term by traversing the dominance tree. If all these conditions are true, *fblk* is chosen as the insertion point for the expression.

5. Experimental Evaluation

5.1 Settings

To evaluate the effectiveness of PDPRE[*1], we used the COINS[*2] compiler. All DDPREs including PDPRE treat programs in the SSA form as the targets of the LIR transformer. The experiments were conducted on a machine equipped with an Intel Core i7-8700K 3.70GHz CPU running the Ubuntu 64-bit operating system. The benchmark programs used for the evaluation were selected from the SPEC CPU2000 benchmark. Although the SPEC CPU2000 includes programs written in C++ and Fortran, COINS does not support these languages. Therefore, we initially selected programs for which COINS could produce valid results. We first compiled all SPEC CPU2000 benchmark programs with no options. We confirmed that seven programs (gzip, vpr, mcf, parser, gap, bzip2, and twolf) from CINT2000 and three programs (art, equake, and ammp) from CFP2000 successfully produced results. Therefore, the 10 programs were chosen from these validated programs.

The algorithms employed in the experiments conducted are as follows:

- **PREQP**: The processes are as follows: conversion to the SSA form from the normal form, PREQP, and conversion to the normal form from the SSA form.
- **EDDPRE**: The process steps are as follows: conversion to the SSA form from the normal form, EDDPRE, and converting to the normal form from the SSA form.
- **LDPRE**: This converts from the normal form into the SSA form, LDPRE, and converts from the SSA form back into the normal form.
- **PDPRE**: This converts from the normal form into the SSA form, PDPRE, and converts from the SSA form back into the normal form.

5.2 Execution Count Information Collection

In this experiment, execution count information for each node was collected using the `cntbb` option provided by the COINS compiler. This option inserts an LIR instruction, which writes the execution count of the node to a file when the node is executed, into each node at compile time. Thus, for this experiment, LIR instructions for counting execution frequency were inserted into the SPEC CPU2000 programs using the `cntbb` option alone. The resulting programs were then executed to obtain the execution counts for each node. As `cntbb` provides the total number of executed LIR instructions for each node, the counts obtained with `cntbb` must be divided by the number of LIR instructions within each node to determine the execution counts for the nodes.

[*1] The implemented programs are available in `https://doi.org/10.5281/zenodo.13363792`

[*2] `https://sourceforge.net/projects/coins-project/`

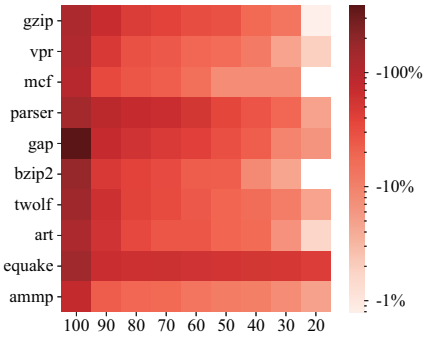


Fig. 8 Analyzing times of PDPRE changing the value of *k* by 10%.

5.3 Research Questions

We conducted the experiments based on the following research questions (**RQs**):

- **RQ1**: How does the analysis time change when varying the percentage *k* of the top *k*% of nodes with PDPRE applied?
- **RQ2**: How does the execution time of the objective code change when varying the percentage *k* of the top *k*% of nodes with PDPRE applied?
- **RQ3**: Which DDPRE algorithm requires the shortest analysis time?
- **RQ4**: How many redundant expressions can PDPRE eliminate compared with existing DDPREs?
- **RQ5**: Which DDPRE algorithm generates the objective code with the shortest execution time?

First, we conducted an evaluation following **RQs1∼2** to determine the most appropriate range of program for applying PDPRE. Based on these results, we conducted a comparative evaluation of the existing DDPREs and PDPRE following **RQs3∼5**.

5.4 Results

> **RQ1:** How does the analysis time change when varying the percentage *k* of the top *k*% of nodes with PDPRE applied?
> **A1.** As the value of *k* increases, the analysis time also increases.
> **A2.** As the value of *k* increases, the number of redundant expressions eliminated also increases.

Figure 8 shows the variation in the analysis times of PDPRE when parameter *k* is varied from 20% to 100% in increments of 10%. Logarithmic scales were employed to provide a detailed visual representation of this variation. The analysis time when $k = 10$ served as the baseline, depicted in blue for decreases in the analysis time and red for increases. The horizontal axis represents the values of *k* and the vertical axis represents the corresponding

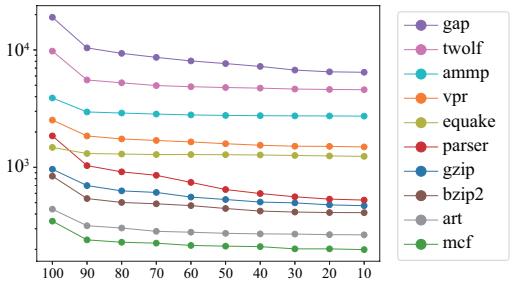


Fig. 9 Number of eliminated redundancies by PDPRE changing the value of k by 10%.

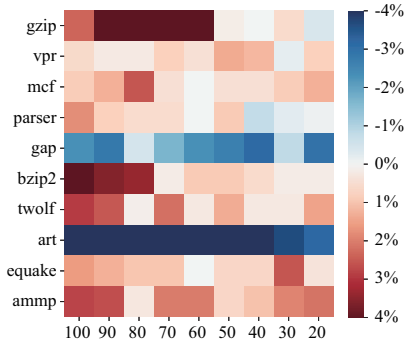


Fig. 10 Execution times of objective codes generated by PDPRE changing the value of k by 10%.

program instances.

In all programs, the observed trend indicated that, as the value of the parameter k increased, the analysis time also tended to increase. This phenomenon is natural because an increase in the number of nodes visited by PDPRE results in a corresponding increase in the generation of queries and analysis of redundant expressions, thereby leading to an increase in analysis time. The results summarizing the number of redundant expressions eliminated when varying the value of k are presented in Fig. 9. The horizontal axis represents the values of k, whereas the vertical axis denotes the logarithmically scaled number of expressions eliminated for each program. A thorough examination of this figure further confirms that, for all programs, an increase in the value of k correlates with a corresponding increase in the number of eliminated expressions.

RQ2: How does the execution time of the objective code change when varying the percentage k of the top k% of nodes with PDPRE applied?
A1. The program with the most favorable execution time was observed when the value of k was 10.
A2. Increasing the number of eliminated expressions does not necessarily guarantee a reduction in execution time.
A3. An increase in the number of eliminated expressions is associated with a corresponding increase in the number of spill occurrences and of executed numbers of load/store instructions inserted by spills.

Figure 10 shows the variation in execution time of the generated objective codes when applying PDPRE when the parameter k is varied from 20% to 100% in increments of 10%. In this figure,

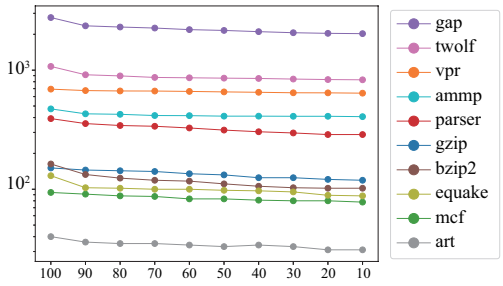


Fig. 11 Numbers of register spill applied by PDPRE.

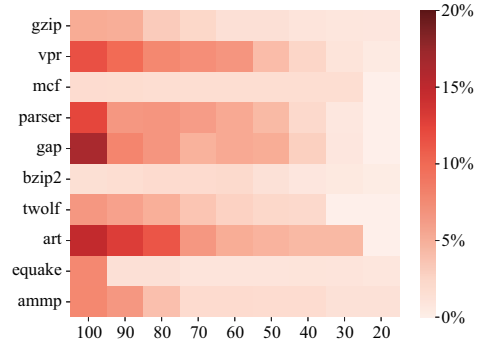


Fig. 12 Numbers of executed load and store instructions inserted by spills changing the value of k by 10%.

k=10 served as the baseline; blue and red mean that the execution time was decreased and increased, respectively. The horizontal axis represents the values of k, while the vertical axis denotes the respective programs.

Upon examining the programs to identify the ones with the fast favorable execution times, it was observed that the highest count was achieved when k equaled 10. Specifically, this included the programs mcf, bzip2, twolf, equake, and ammp. We specify the values of k that yielded the best execution times for the other programs. For gzip, art, vpr, parser, and gap, the optimal results were obtained at k=20, 20, 30, 40, and 40, respectively. In the subsequent research questions comparing with existing DDPRE, we present the results of PDPRE for k=10.

In order to understand the reasons behind the lack of execution time reduction with increasing values of k, we counted the occurrence of spills in the register allocation process. The register allocation algorithm utilized in the experiments conducted was Appel-George's iterated register coalescing, an improved version of the graph coloring algorithm.

This register allocation algorithm analyzes the program statically to induce spills, without considering the number of program executions. Thus, the store and load instructions inserted as results of spills were determined independently of the program's execution counts, implying that the occurrence of spills did not directly impact the execution time of the objective code. However, existing research [5], [7], [13] has pointed out a tendency for spills to worsen execution time; thus, this study also verified this trend.

The results of the occurrence of spills are presented in Fig. 11. The horizontal axis represents the values of k, while the vertical axis, on a logarithmic scale, denotes the number of spills for each program. As evident from the results, an increase in the number of eliminated expressions corresponds to an increase in the num-

Table 1 Analyzing times. Bold letters indicate the best results.

Prog.	PREQP	EDDPRE	LDPRE	PDPRE
gzip	773	894	893	**253**
vpr	1,509	1,961	1,798	**459**
mcf	351	348	414	**140**
parser	1,366	1,617	1,615	**453**
gap	8,194	13,350	10,057	**1,443**
bzip2	295	495	462	**108**
twolf	5,775	8,448	6,914	**1,615**
art	131	159	126	**57**
equake	367	486	432	**106**
ammp	2,454	2,574	2,923	**796**

ber of spills. As a general trend, increasing the value of k was associated with an increase in spills. Notably, when focusing on instances where execution time deteriorated significantly, we can observe that spills exhibited a pronounced increase at k=100 for bzip2, k=100 for twolf, and k=30 for equake.

Next, Fig. 12 shows the total number of executed load and store instructions inserted due to spills when the parameter k is varied from 20% to 100% in increments of 10%. In this figure, k=10 served as the baseline; red means that the executed instructions were increased. The horizontal axis represents the values of k, while the vertical axis denotes the respective programs. Similar to the occurrence of spills, the execution counts of these instructions increase with the value of k.

From the results presented in Figs. 10∼12, it is observed that while reducing the number of redundant expressions is effective for some programs (e.g., gap and art), there are programs where an increase in spills is associated with a deterioration in execution time. Thus, there is a trade-off between reducing the number of redundant expressions and the increase in load and store instructions inserted by spills. As this trade-off varies depending on the program, a key direction for future work will be to enhance the effectiveness of PDPRE by identifying which program structures exhibit this trade-off and adjusting the value of k accordingly based on the specific characteristics of the program.

RQ3: Which DDPRE algorithm requires the shortest analysis time?
A. For all programs, PDPRE exhibited the shortest time.

Table 1 presents the analysis times for all DDPREs. Across all programs, PDPRE consistently yielded the shortest results. The average overall improvement rate calculated showed that PDPRE reduced the analysis time by 71.4%. When comparing the analysis times of PDPRE with those of existing DDPRE methods, the program with the highest reduction efficiency was gap, with an average reduction of 85.7%. Specifically, PDPRE reduced analysis time on average by approximately 67.7% compared to PREQP, 74.0% compared to EDDPRE, and 72.6% compared to LDPRE.

RQ4: How many redundant expressions can PDPRE eliminate compared with existing DDPREs?
A. PDPRE eliminated redundancy to a similar extent as PRE.

Table 2 Numbers of eliminated expressions. Bold letters indicate the most numbers of eliminated expressions.

Prog.	PREQP	EDDPRE	LDPRE	PDPRE
gzip	804	**1,196**	984	471
vpr	2,974	**4,190**	3,535	1,492
mcf	232	**378**	282	199
parser	1,838	**2,242**	2,136	526
gap	26,594	**34,578**	28,901	6,452
bzip2	669	**974**	807	411
twolf	9,138	**12,249**	9,964	4,584
art	295	**449**	359	266
equake	847	**1,470**	993	1,240
ammp	3,927	**5,113**	4,247	2,730

Table 3 Execution times. Bold letters indicate the best results.

Prog.	PREQP	EDDPRE	LDPRE	PDPRE
gzip	75.3	76.9	77.2	**73.5**
vpr	**50.8**	53.3	53.2	51.1
mcf	24.4	24.1	24.1	**23.5**
parser	115	115	116	**113.1**
gap	**59.1**	**59.1**	59.4	60.1
bzip2	58.2	58.6	58.4	**56.3**
twolf	86.3	85.1	**82.5**	83.2
art	**20.8**	21.3	24.8	21.8
equake	33.8	32.2	40.5	**31.6**
ammp	80.9	**78.7**	81.4	79.8

Table 2 displays the number of redundant expressions eliminated by all DDPREs. The algorithm that eliminated the highest number of expressions was EDDPRE. This is because EDDPRE performs GVN and speculative movement of loop-invariant expressions; these features are capable of eliminating more redundancies than other DDPREs.

Focusing on the number of redundant expressions eliminated by PDPRE, it is observed that, compared to EDDPRE, PDPRE eliminated fewer expressions across all programs. On average, the number of expressions eliminated by PDPRE decreased by approximately 55.4%. Notably, in the case of gap, this reduction amounted to about 81.3%. Comparing the numbers of PREQP and LDPRE, PDPRE eliminated fewer expressions than these algorithms in programs except for equake. In the case of equake, it is notable that GVN demonstrated higher expression elimination, suggesting the presence of considerable redundancy within the nodes of this program.

RQ5: Which DDPRE algorithm generates the objective code with the shortest execution time?
A. PDPRE obtained the most number of programs achieving the shortest execution time.

Table 3 presents the execution times of the objective code generated by all DDPREs. The application of PDPRE resulted in the shortest execution times for five programs: gzip, mcf, parser, bzip2, and equake. Additionally, PDPRE achieved the highest count of programs with the shortest execution times.

In order to investigate the reasons why the abundance of eliminated expressions did not necessarily contribute to execution time reduction, Tab. 4 presents the number of spills that occurred. This result also shows that PDPRE had the best results according to the number of programs with the fewest occurrences of spills.

Table 4 Numbers of register spill. Bold letters indicate the best results.

Prog.	PREQP	EDDPRE	LDPRE	PDPRE
gzip	**115**	162	172	121
vpr	679	896	944	**644**
mcf	**74**	109	103	80
parser	305	433	442	**290**
gap	2,119	3,628	3,899	**2,037**
bzip2	108	186	175	**103**
twolf	839	1,321	1,285	**837**
art	**30**	46	39	31
equake	**50**	133	125	89
ammp	427	699	685	**410**

6. Conclusions

In this study, we propose profile-guided demand-driven partial redundancy elimination (PDPRE), which leverages runtime information for the elimination of partial redundancies. PDPRE prioritizes the elimination of redundancies in programs with higher execution counts. During this process, PDPRE generates queries for analyzing the availability of the value number for each expression to identify redundancies in expressions with different lexical forms. The insertion points, aimed at eliminating redundancies, are selected from among the candidate nodes, prioritizing those with the lowest execution counts.

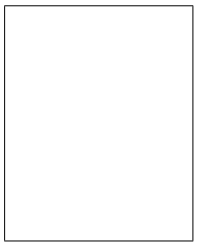
To evaluate the effectiveness of PDPRE, we applied PDPRE and existing DDPREs to the SPEC CPU2000 benchmark programs. We measured the analysis time and execution time of the generated objective codes. The results indicated that PDPRE retains the characteristic of DDPRE with shorter analysis times while achieving reductions for many programs in execution time compared algorithms.

The future work will *(a) involve extending the algorithm to address redundancies that cannot be eliminated by PRE, such as scalar replacement.* Furthermore, as indicated by the results of **RQ2**, future work will also *(b) propose a machine learning method for dynamically determining the value of k to enhance the effectiveness of PDPRE could be considered*.

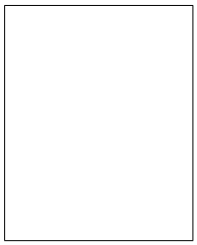
Acknowledgments. We appreciate the anonymous reviewers for their constructive and positive feedback on the content of this paper.

References

[1] Appel, A. W.: *Modern Compiler Implementation in ML: Basic Techniques*, Cambridge University Press, New York, NY, USA (1997).
[2] Bodik, R., Gupta, R. and Soffa, M. L.: Complete removal of redundant expressions, PLDI '98, New York, NY, USA, ACM, pp. 1–14 (1998).
[3] Cai, Q. and Xue, J.: Optimal and efficient speculation-based partial redundancy elimination, *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, Washington, DC, USA, IEEE Computer Society, pp. 91–102 (2003).
[4] Chow, F., Chan, S., Kennedy, R., Liu, S.-M., Lo, R. and Tu, P.: A New Algorithm for Partial Redundancy Elimination Based on SSA Form, *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, New York, NY, USA, Association for Computing Machinery, pp. 273–286 (1997).
[5] Gupta, R. and Bodík, R.: Register Pressure Sensitive Redundancy Elimination, *Compiler Construction* (Jähnichen, S., ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 107–121 (1999).
[6] Horspool, R. N., Pereira, D. J. and Scholz, B.: Fast Profile-Based Partial Redundancy Elimination, *Modular Programming Languages* (Lightfoot, D. E. and Szyperski, C., eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 362–376 (2006).
[7] Knoop, J., Ruthing, O. and Steffen, B.: Lazy code motion, *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, New York, NY, USA, ACM, pp. 224–234 (online), DOI: 10.1145/143095.143136 (1992).
[8] 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 (online), DOI: 10.1145/183432.183443 (1994).
[9] Krause, P. K.: Lospre in Linear Time, *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '21, New York, NY, USA, Association for Computing Machinery, pp. 35–41 (online), DOI: 10.1145/3493229.3493304 (2021).
[10] Morel, E. and Renvoise, C.: Global optimization by suppression of partial redundancies, *Commun. ACM*, Vol. 22, No. 2, pp. 96–103 (1979).
[11] Odaira, R. and Hiraki, K.: Partial Value Number Redundancy Elimination, *Languages and Compilers for High Performance Computing*, Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 409–423 (2005).
[12] Rosen, B. K., Wegman, M. N. and Zadeck, F. K.: Global value numbers and redundant computations, *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, New York, NY, USA, ACM, pp. 12–27 (1988).
[13] Shobaki, G., Bassett, J., Heffernan, M. and Kerbow, A.: Graph Transformations for Register-Pressure-Aware Instruction Scheduling, *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, New York, NY, USA, Association for Computing Machinery, pp. 41–53 (online), available from ⟨https://doi.org/10.1145/3497776.3517771⟩ (2022).
[14] Sumikawa, Y. and Takimoto, M.: Effective Demand-driven Partial Redundancy Elimination, *Information Processing Society of Japan Transactions on Programming*, Vol. 6, No. 2, pp. 33–44 (2013).
[15] Takimoto, M.: Speculative Partial Redundancy Elimination Based on Question Propagation, *Information Processing Society of Japan Transactions on Programming*, Vol. 2, No. 5, pp. 15–27 (2009).
[16] VanDrunen, T. and Hosking, A. L.: Value-Based Partial Redundancy Elimination, *Compiler Construction* (Duesterwald, E., ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 167–184 (2004).
[17] Xue, J. and Cai, Q.: A Lifetime Optimal Algorithm for Speculative PRE, *ACM Trans. Archit. Code Optim.*, Vol. 3, No. 2, pp. 115–155 (2006).
[18] Yanase, Y. and Sumikawa, Y.: Lazy Demand-driven Partial Redundancy Elimination, *Journal of Information Processing*, Vol. 31, pp. 459–468 (online), DOI: 10.2197/ipsjjip.31.459 (2023).
[19] Zhou, H., Chen, W. and Chow, F.: An SSA-based algorithm for optimal speculative code motion under an execution profile, *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, New York, NY, USA, ACM, pp. 98–108 (online), DOI: 10.1145/1993498.1993510 (2011).



Takuna Uemura received his B.E. degree in Computer Sciences from Takushoku University in 2024. His research interests include compiler and its implementation.



Yasunobu Sumikawa received his B.S. degree in Mathematics from Tokyo University of Science in 2010, and his M.S. and Ph.D. degrees in Information Science from Tokyo University of Science in 2012 and 2015, respectively. He is currently an Associate Professor at the department of computer science, Takushoku University, Japan. His research interests lie on compiler, information retrieval, HistoInformatics, and machine learning for history learning support.