Journal of Information Processing Vol.31 1–12 (Jan. 2023)

[DOI: 10.2197/ipsjjip.31.1]

Regular Paper

Lazy Demand-Driven Partial Redundancy Elimination

Yuya Yanase[1] Yasunobu Sumikawa[1,a)]

Received: September 4, 2022, Accepted: May 2, 2023

Abstract: Partial redundancy elimination (PRE) is a code optimization algorithm that simultaneously performs common sub-expression elimination and loop-invariant code motion. Traditional PREs analyze the entire program and eliminate redundancies. By contrast, demand-driven PRE (DDPRRE) is proposed as an algorithm to analyze only a part of the program to determine whether each expression is redundant by query propagation. The previous DDPRE reduces the analysis time because it limits the analysis range; however, it is known that redundancy may not be eliminated when the nodes in the loop are revisited. We propose a novel DDPRE, named lazy demand-driven PRE (LDPRE), which eliminates redundancy by delaying the decision of whether the analyzing expression is redundant or not when a node in a loop is revisited and redundancy cannot be analyzed. LDPRE uses a semi-lattice as the answer space during the query propagation. The semi-lattice includes not only true/false implying that the queried expression is redundant or not, but also $\top$ for undecidability. While maintaining the analytical efficiency characteristic of demand-driven analysis, our algorithm eliminates redundancy that previous DDPRE could not eliminate by determining the answer using semi-lattice.

Keywords: Code optimization, Compiler, Demand-driven data-flow analysis, Partial redundancy elimination

1. Introduction

Partial redundancy elimination (PRE) [3], [11], [13] is a powerful optimization algorithm to eliminate not only fully redundant expressions but also partially redundant ones by inserting the expressions to make partial redundancy, full redundancy. Owing to this insertion process, PRE performs loop invariant code motion (LICM) because the loop invariant expressions are partially redundant at the entry points of their loops; loop invariant expressions are redundant in their loop whereas they are not on the path before entering the loop.

Traditional PRE algorithms eliminate redundant expressions by analyzing the whole program represented by the control flow graph (CFG). By contrast, demand-driven PRE (DDPRE), named PRE-based query propagation (PREQP) [20], has been proposed to improve the analysis efficiency by analyzing only a part of the program. PREQP visits each node of the CFG in topological sort order. When an expression e occurs, the query analyzing redundancy is propagated toward the root of the CFG. The query returns true if the query visits an occurrence of the same expression e. However, if no occurrence of the same expression is found, false is returned as the answer. Because a query is simply propagated to each predecessor, a query may visit the same node in the loop twice. When it revisits a node without any occurrences of statements that might change the value of e in the loop, it assumes that *the answer will be true* and then optimistically returns true as the answer. However, the optimistic returning fails to eliminate redundancy if the loop containing the node contains multiple exits.

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

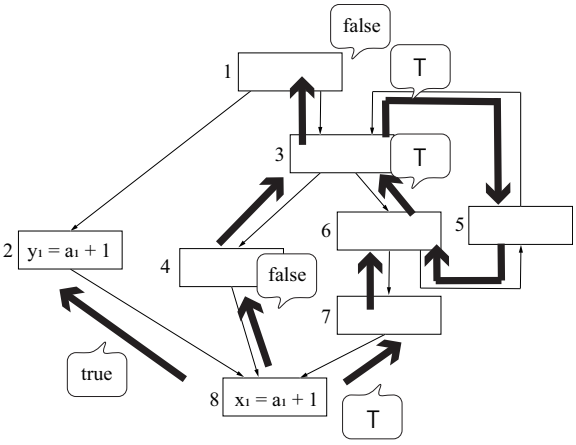


Fig. 1 Query propagation of LDPRE. Arrows indicate query propagation. The balloon represents the answer.

In this study, we propose a novel DDPRE, named the lazy demand-driven PRE (LDPRE), that performs query answer determination delaying. This algorithm returns true as the answer to the query only when the expression actually occurs indicating that the answer truly detected redundancy. If the same node is revisited, it returns $\top$ instead of true indicating that the answer is undefined. To define these answers systematically, we use semi-lattice.

Fig. 1 shows how LDPRE performs query propagation. As shown in this figure, we assume that all programs are converted into static single assignment (SSA) form [7] before applying LDPRE. LDPRE propagates three queries for expression a_1+1 on Node 8 to paths that include Nodes 2, 4, and 7, respectively. The query that visits Node 2 returns a true as a_1+1 occurs on Node 2. Looking at another query propagated to Node 3 through Node 4, it is also propagated to Nodes 5, 6, and 3. At this time, Node 3 is

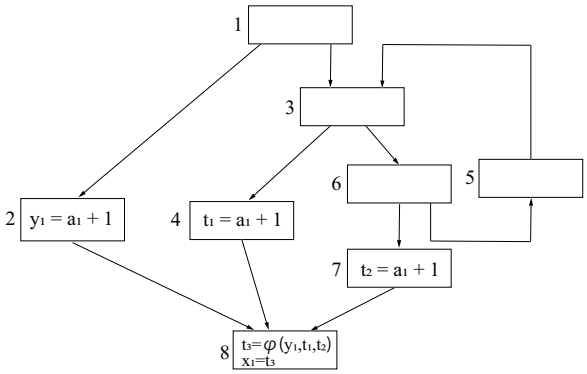


Fig. 2 Result of LDPRE

visited again, the query in LDPRE returns $\top$ to Nodes 6, 5, and 3. As false is returned from Node 1 to Node 3, LDPRE obtains $\top$ and false at Node 3. According to the definition of the operator on the semi-lattice, the final answer at Node 3 becomes false; then, this answer is also returned to Nodes 4 and 8. Next, LDPRE propagates a query from Node 8 to Nodes 7 and 6. As this is the second time of visiting Node 6 for this query, $\top$ is returned from Node 6 to Nodes 7 and 8. Finally, LDPRE obtains true, false, and $\top$ as answers at Node 8. As the obtained answers include true and false, we can say that a_1+1 is partially redundant. Once LDPRE inserts t_1=a_1+1 and t_2=a_1+1 to Nodes 4 and 7 where false and $\top$ are returned, it makes the partial redundancy, full redundancy. After inserting a ϕ function at the entry of Node 8, LDPRE eliminates the partial redundancy by replacing a_1+1 with the left-hand side of the ϕ function (see Fig. 2).

We implemented LDPRE on a compiler infrastructure called COINS and compared analysis efficiency and the number of statically eliminated redundancies using SPEC CPU2000 benchmark among LDPRE, PREQP, and traditional PRE analyzing whole programs. We confirmed that LDPRE was faster than traditional PRE in analyzing time and eliminated more redundancy than PREQP.

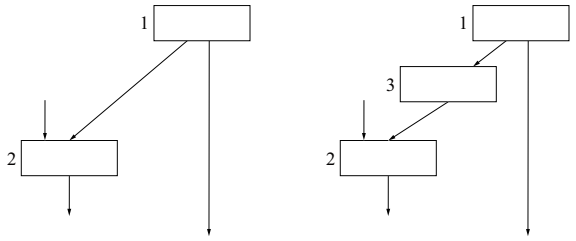
The remainder of this paper is organized as follows. Section 2 presents the definitions we used. Then, Section 3 summarizes related works. Section 4 details our algorithm LDPRE, and Section 5 discusses the experimental results. Finally, Section 6 provides the concluding remarks.

2. Background

2.1 Program Representation

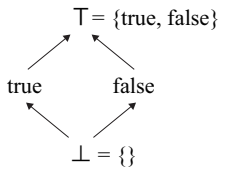
We assume that a CFG was built for each program. The CFG is represented as a quadruple (**N**, **E**, **start**, **end**), where **N** is a set of basic blocks, **E** is a set of edges $\mathbf{N} \times \mathbf{N}$, and **start** and **end** represent, respectively, a start node and an end node with an empty statement. Sets of predecessors and successors of node n are denoted by $pred(n)$ and $succ(n)$. When all paths from **start** to node n include a node m, it is said that *m dominates n* [2].

We also assume that each variable is defined exactly once by assigning a unique version to it, known as SSA form, to simplify the definitions of demand-driven data-flow analysis using semi-lattice. In SSA form, to handle cases in which several definitions can reach their uses, special functions ϕ must be inserted to merge these definitions at their dominance frontiers.



(a) Original CFG (b) CFG after critical edge removing

Fig. 3 Eliminating critical edges



$\top \sqcap \text{true} = \text{true}$ $\top \sqcup \text{true} = \top$
$\top \sqcap \text{false} = \text{false}$ $\top \sqcup \text{false} = \top$
$\text{true} \sqcap \text{false} = \bot$ $\text{true} \sqcup \text{false} = \top$
$\bot \sqcap \text{true} = \bot$ $\bot \sqcup \text{true} = \text{true}$
$\bot \sqcap \text{false} = \bot$ $\bot \sqcup \text{false} = \text{false}$
$\top \sqcap \bot = \bot$ $\top \sqcup \bot = \top$

Fig. 4 Algebra on lattice

We assume that all critical edges [18] that can block several code motions were eliminated before applying LDPRE. The critical edge is an edge leading from a node with more than one successor to a node with more than one predecessor. The critical edges were eliminated by inserting synthesized nodes. Looking at Fig. 3 (a), the edge from Node 1 to Node 2 is a critical edge. This is eliminated by inserting a node, as shown in Fig. 3 (b).

2.2 Availability and Anticipability

Expression e is *available* at node n iff e is computed on any path p from **start** to n, and no definitions are generated for e's operands as the most recent occurrence of e on p [3]. Here, we use $comp(e, n)$ and $kill(e, n)$ that represent e was computed at n, and n had definitions for e's operands, respectively. When e is available at n, n is *up-safe* with respect to e. e is *partially available* at node n iff there is at least one path from **start** to n in which e is computed without subsequent redefinition of its operands. When e is available at n, e is fully redundant and can be replaced with the variable that comprises the preceding execution result. When e is partially available at n, e is partially redundant. The partially redundant expression is eliminated after inserting expressions to make the original expression fully redundant. This paper uses $Insert(e, n)$ if n is a node in which the statement including e should be inserted. Expression e is *anticipable* at node n iff e is computed along any path r from n to **end**, and the operands of e are undefined before the first computation of e on r [3]. When e is anticipable at n, n is *down-safe* with respect to e. PRE inserts expressions at the down-safe nodes without extending the lengths of any path.

2.3 Semi-lattice

Our query propagation determines answers defined on lattice $(\mathcal{A}, \sqcap, \sqcup, \top, \bot)$, where $\mathcal{A}$ is a set of answers, $\sqcap : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is a meet operator on $\mathcal{A}$, $\sqcup : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is a join 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 $\top \sqcap a = a$, $\bot \sqcap a = \bot$, $\top \sqcup a = \top$, and $\bot \sqcup a = a$, respectively. Fig. 4 shows

relationships among $\mathcal{A}$, $\top$, and $\bot$ and all combinations between different elements and operators.

3. Related Works

3.1 Exhaustive PRE

There are numerous algorithms to eliminate redundancy [1], [16]. Morel and Renvoise proposed the original PRE using bi-directional analysis [13] to perform common sub-expression elimination [6] and LICM simultaneously without distinction. It is known that bi-directional analysis performs unnecessary code motion or remains redundancies in the program. To solve these issues, several studies have attempted to improve the original PRE algorithm. Knoop *et al.* proposed lazy code motion (LCM) [10], [11] that uses unidirectional analysis to find optimal and economical insertion points. This algorithm decomposes the bi-directional analysis into forward and backward analyses to suppress unnecessary code motion. As LCM respects that code optimization algorithms keep the meaning of the program, the insertion points should be safe. However, respecting the safety sometimes fails to eliminate redundancy as some insertions are prohibited so as not to increase the number of executed expressions on several paths. Bodik *et al.* proposed complete PRE [3] to eliminate all redundancies by duplicating parts of the program without losing the safety.

While the above algorithms are designed for normal form of programs, Chow *et al.* proposed SSAPRE [5] that allowed performing PRE on programs represented by SSA form.

3.2 Speculative Code Motion

In environments that do not support error handling, as the safety indicates that there are no paths that are lengthened by the insertion of an expression, non-safety insertion does not change the execution result of the program; it actually means that the number of statements to be executed increases. The insertion that presumably increases the number of executed statements is called speculative code motion. As LCM and complete PRE check down-safety before their insertions, these algorithms prevent any speculative code motion. Although the number of statements to be executed in some execution paths may increase, Cai and Xue proposed MC-PRE [4], [22] that performs speculative code motion. MC-PRE applies min-cut to the CFG to determine insertion points where eliminating redundancy on paths that are frequently executed. The actual insertions may lengthen some execution paths; however, it has been shown to reduce the execution time of the objective code because statements are inserted at points of infrequent execution.

These algorithms are powerful; however, it is well-known that they require much analysis time. In fact, several speculative PREs have been proposed to reduce analysis time [4], [9], [12]. Many of these algorithms reduce analysis time at the expense of redundancy that can be eliminated. The state-of-the-art algorithm of speculative PRE is proposed by Krause [12]. This algorithm uses the bounded tree-width of CFG to achieve faster analysis while obtaining the same execution time as MC-PRE.

These past studies including methods of Section 3.1 analyzed whole programs to find redundancy. Therefore, the above algorithms assume that the priority is to reduce the execution time of the objective code rather than to reduce the analysis time. On the other hand, demand-driven algorithms including our algorithm are effective in situations where it is important to achieve both redundancy elimination, which correlate with execution time, and short analysis time such as the just-in-time (JIT) compiler.

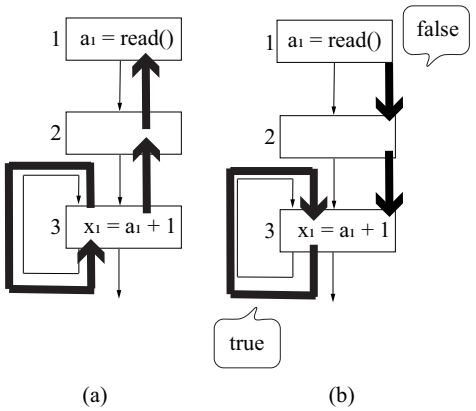


Fig. 5 (a) Loop invariant code motion by DDPRE. (b) Query answers for loop invariant code motion in previous PREQP.

3.3 DDPRE

DDPRE was proposed as an algorithm for limiting the scope of redundancy analysis. The original DDPRE (PREQP) performs query propagation [16] to check whether each expression e is available. First, PREQP traverses CFG in topological sort order. If an expression e occurs during this traverse, PREQP propagates a query *Is the expression e available?* This query returns true or false according to finding the occurrence of the same expression.

Fig. 5 (a) shows how PREQP performs LICM by query propagation. Looking at the expression a_1+1 on Node 3 where the node is included in a loop, it can be said that a_1+1 is the loop invariant because there are no definitions of the variable a_1 in the loop. PREQP propagates queries on two paths from Node 3 to Node 3 and from Node 3 to Nodes 2 and 1. Fig. 5 (b) shows the results of the queries. As the former query visits the original expression that generated the query, true is obtained from the query. By contrast, false is obtained from the query propagated on the path containing Nodes 2 and 1 as the query does not find any occurrences of the expression a_1+1 on the path. These answers indicate that the expression a_1+1 is partially redundant at Node 3; therefore, it is able to make the redundancy full redundancy by inserting t_1=a_1+1 at Node 2 where false is obtained.

Furthermore, PREQP optimistically returns true when the query for the same expression is propagated to the same node twice to perform LICM. Fig. 6 shows an example to explain why PREQP performs this optimistic answer decision. We assume that the e on Node 4 is a loop invariant code. PREQP propagates a query for this expression from Node 4 to Node 2. The query further visits Node 1, and returns false because there is no occurrence of the same expression. After visiting Node 5 from Node 2, the query visits Nodes 3 and 4. As the Node 4 includes the same expression, the query returns true. On the other hand, the query visits Node 2 from Node 3; this is the second time of visiting Node 2. As this query does not meet any definitions of e during

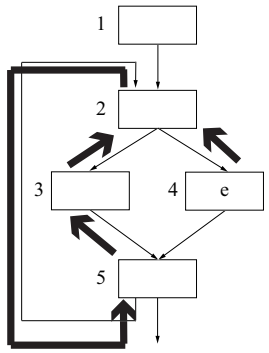


Fig. 6 Optimistic answer decision of PREQP

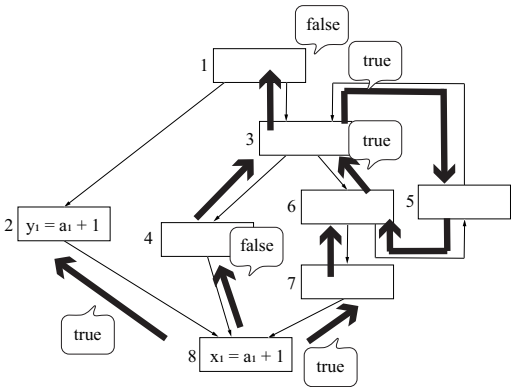


Fig. 7 Query propagation of previous DDPREs

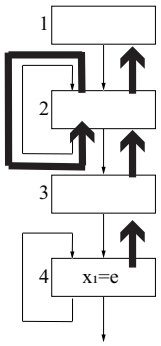


Fig. 8 Example for preventing the unnecessary code motion of PREQP

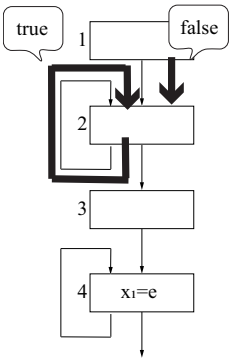


Fig. 9 Answers

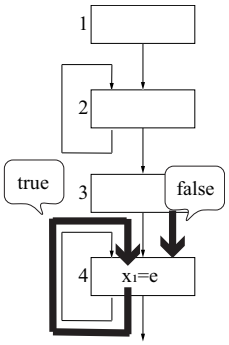


Fig. 10 Result

the propagation, PREQP decides that this query returns true. As queries return true from both Nodes 3 and 4, the query also returns true from Node 5. PREQP obtains true and false at Node 2; thus, inserting a new statement to Node 1 performs LICM for the `e`.

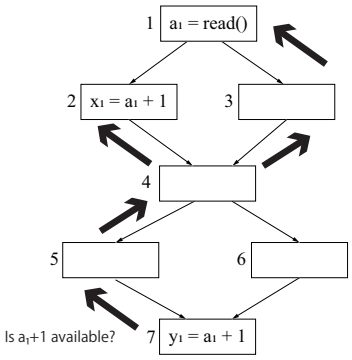
Next, Fig. 7 shows an example of redundancy that PREQP fails to eliminate. We examine at a_1+1 on Node 8. This expression is partially redundant because the same expression exists on Node 2; however, there are no expressions on all paths from Node 1 to Nodes 4 or 7. If the PREQP propagates a query on the path containing Nodes 8 and 2, the query returns true as a_1+1 occurs in Node 2. When the PREQP propagates a query on a path containing Nodes 8, 4, 3, and 1, false is obtained as Node 1 represents the **start** of the program. During the propagation, the query is also propagated from Node 3 to Node 5. Then, the query is propagated to Nodes 6 and 3. As the query revisits Node 3 without meeting the definitions of a_1, this query returns true as the answer. PREQP obtained true and false at Node 3 although its query propagation does not meet the same expressions. Thus, concluding partial redundancy at Node 3 from these answers could result in unnecessary code motion. Indeed, a_1+1 is not partially redundant at Node 3.

Figs. 8, 9, and 10 show how PREQP prevents the unnecessary code motion. PREQP propagates a query for `e` from Node 4 to Nodes 3 and 2. This query is further propagated to Nodes 2 and 1. As the visiting Node 2 is revisiting itself, the query returns true. Visiting Node 1 finds that there is not the same expression on the path; thus, it returns false. These results indicates that `e` is partially redundant at Node 2; thus, inserting a new statement to Node 1 makes `e` fully redundant. However, the ideal insertion point is at Node 3. Inserting the statement at Node 1 increases the register pressure. To suppress the unnecessary code motion, PREQP checks actual occurrence if two answers, true and false, are obtained before the insertion. In this example, there is no actual occurrence about `e` at neither Nodes 2 nor 1; thus, PREQP returns false from Node 2 as shown in Fig. 10. Finally, PREQP obtains true from the query propagated to Node 4 with the actual occurrence and false from Node 3; inserting `e` to Node 3 eliminates the partial redundancy.

Looking back to Fig. 7, the query revisited Node 3 from Node 6 returns true as the answer. However, the actual occurrence of the expression is false. As a result, PREQP returns false from Node 3 to Nodes 4 and 8 though true and false are obtained from Nodes 5 and 1. After this propagation, PREQP also propagates a query from Node 8 to Nodes 7 and 6. As the same query is already propagated to Node 6 in the previous propagation, it returns the determined answer true from Node 6 to Nodes 7 and 8. Finally, as three answers, true, false, and true, are returned to Node 8, PREQP inserts $t_1=a_1+1$ only to Node 4, which returned false, and replaces a_1+1 on Node 8 with t_1. However, this code motion is erroneous because there is no $t_1=a_1+1$ on the path from Nodes 1 to 7. To prevent this wrong code motion, PREQP revokes the insertion and replacement; PREQP does not eliminate the partial redundancy in Node 8.

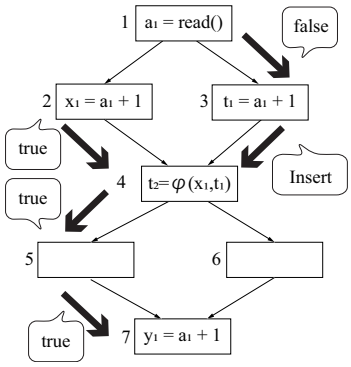
Sumikawa and Takimoto proposed effective DDPRE (ED-DPRE) [19] that combines global value numbering [1] and PREQP to analyze redundancy based on equivalence of value numbers of expressions instead of lexical equivalence.

These DDPRE algorithms eliminated redundancies in a shorter analysis time than PREs that analyzed the entire program. However, as only a part of the program was analyzed, it lacked avail-



Node	Visited Status
1	false
2	true
3	⊥
4	⊥
5	⊥
6	
7	

Fig. 11 Query propagation



Node	Visited Status
1	false
2	true
3	false
4	true
5	true
6	
7	

Fig. 12 Returning answers and determining insertion points

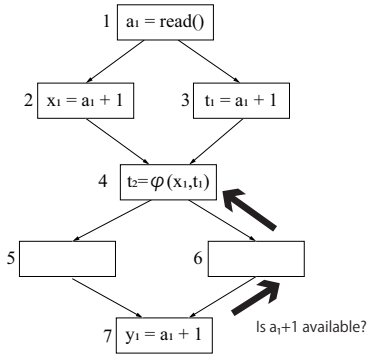
ability information and might lead to incorrect program transformations, as described above.

Here, the proposed LDPRE uses semi-lattice to analyze redundancy that previous DDPREs failed to eliminate, while retaining the feature of the previous DDPREs of eliminating redundancies in a short analysis time.

4. LDPRE

In this section, we describe how LDPRE eliminates redundant expressions by query propagation. Similar to PREQP, LDPRE propagates a query *Is e available?* toward the root of the CFG for each occurrence of the expression e to analyze its redundancy. Fig. 11 shows the propagation of a query over two paths containing Nodes 5 and 4 for the expression a_1+1 on Node 7 and states the answers obtained from all CFG nodes. Indeed, LDPRE uses a table *visited* to record the answers obtained to prevent repeated visits to the same node. Each time the query visits node n, LDPRE first checks if an answer is recorded in $visited[(e,n)]$. If an answer is recorded, LDPRE returns a result without analyzing the node; otherwise, LDPRE stores $\bot$ to the table and then analyzes the node in detail. Looking at the figure, this query propagation obtained false and true as answers in Nodes 1 and 2, respectively. After obtaining the answer at each node, LDPRE stores it to *visited* as shown in the visited status table. Looking at the other Nodes 3, 4, and 5, the current *visited* statuses are $\bot$ that indicates none of true/false, because at this point these nodes are still being analyzed by only propagating the query. Fig. 12 shows the result of the answers obtained by this query propagation. Looking at the query propagated from Nodes 4 to 2, the query returns true as an answer because this query finds the same expression at Node 2. By contrast, a query returns false if the query is propagated to the occurrence of an assignment statement to e operands or to the **start** such as the propagation from Node 4 to Nodes 3 and 1. Examining the results at Node 4, we observe that a_1+1 is partially redundant; thus, LDPRE inserts a statement $t_1=a_1+1$ into Node 3 where false is returned to make a_1+1 fully redundant, and returns true as the answer from Node 4. As a result, the answer at Node 4 is also returned to Node 5; the visited statuses of Nodes 3, 4, and 5 are false, true, and true, respectively, as shown in the table.

When a query is propagated from Node 7 to Nodes 6 and 4, as shown in Figs. 13 and 14 (a), LDPRE obtains an answer true for Node 4 by reference to *visited*. That is, each time the query is propagated to Node n, if true or false is recorded in $visited[(e,n)]$,



Node	Visited Status
1	false
2	true
3	false
4	true
5	true
6	true
7	

Fig. 13 Query propagation to the other path

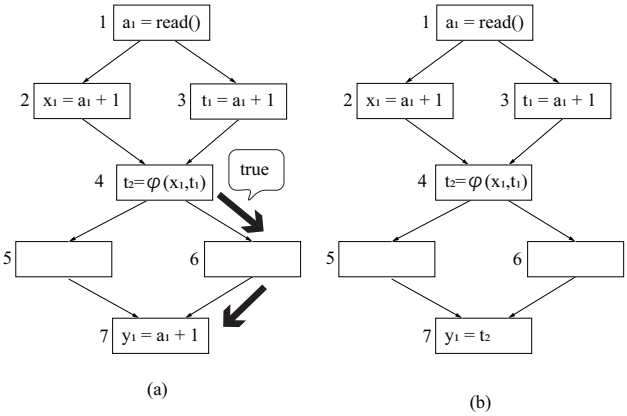


Fig. 14 (a) Revisiting to Node 4 and obtaining the answer. (b) Result.

the value is returned. If $\bot$ is recorded in $visited[(e,n)]$, $\top$ is returned. Otherwise, LDPRE stores $\bot$ in $visited[(e,n)]$ and continues query propagation. As we assume that all programs are represented in SSA form, LDPRE propagates queries to predecessors after replacing operands of e with the arguments of the ϕ function. This replacement changes the lexical representation of the expression though the original query is the same. The above procedures about checking *visited* indicate whether *visited* records a query about the same lexical representation of the expression that has already been visited in the same node. Fig. 14 (b) shows the result of the insertions and replacement by LDPRE.

As described in Fig. 1, when LDPRE revisits a node where no answer is determined, it returns $\top$ to indicate that it is undecided at this time. The intuitive idea is that when LDPRE obtains answers to two or more queries later, using $\top$ that can represent either true or false allows us deciding the final answer at the node according to the other answers. Fig. 15 (a) presents statuses of

Node	Visited Status
1	false
2	true
3	⊥
4	⊥
5	⊥
6	⊥
7	
8	

(a)

Node	Visited Status
1	false
2	true
3	⊤
4	⊥
5	⊤
6	⊤
7	
8	

(b)

Fig. 15 (a) *visited* statuses of Fig. 1 when the query is revisited to Node 3. (b) *visited* statuses of Fig. 1 after the query revisiting to Node 3.

visited for the query propagation of Fig. 1 to describe how ⊤ is used as answer. The status table makes the following three assumptions: 1) queries had visited Nodes 2 and 1, 2) the query visited Node 1 through Nodes 4 and 3, and 3) the query is deciding the answer for Node 3 after visiting Nodes 5 and 6 because Node 5 is the other predecessor of Node 3. At this time, the query refers to the *visited*$[(e, n)]$ and finds that ⊥ is recorded in the table. This result indicates that the query revisited the same node; it updates ⊥ to ⊤ of *visited*$[(\mathtt{a}_1{+}1, 3)]$ and returns ⊤ as the answer to Nodes 6, 5, and 3 as shown in Fig. 15 (b). Although already described in Section 1, the subsequent determination of the answer is as follows. As false and ⊤ are returned from Nodes 1 and 5, respectively, LDPRE applies the meet operator ⊓ for the two answers (see Fig. 4 for details of the calculation results) and obtains false as the answer for Node 3.

4.1 Data-Flow Equations

We now present formal definitions of query propagation used in LDPRE. In the definitions, we assume that the query analyzes the availability of the expression e of node n.

As LDPRE assumes that CFG nodes are basic blocks, we analyze two types of availability: availability at the exit of node n (*XAvail*(e, n)) and availability at the entry of node n (*NAvail*(e, n)).

XAvail(e, n) analyzes the occurrence of e in n and whether there exists an assignment statement (kill) to the operand of e between its occurrence and exit. At the beginning, *XAvail*(e, n) checks whether a query on the same expression e has already been visited in node n. As we described the definitions of *visited* before Section 4.1, it returns the answer according to the results of the previous visit if the same query is already visited. If there is an occurrence of e without encountering kill assignment statements, the query returns true. If there is a kill assignment statement or the propagated node n is **start**; then, the query returns false. Looking at Fig. 12, this corresponds to the query propagation to Node 1 as Node 1 is **start**. If there is neither an occurrence of e nor kill assignment statements in n, LDPRE further propagates the query to the predecessors. If the query is propagated to predecessors, LDPRE determines its answer by *NAvail*(e, n).

The data-flow equation for *XAvail* is defined as follows:

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

NAvail(e, n) collects the query answers obtained from the predecessors of n. Similar to PREQP, if both true and false are obtained, and nodes where false are obtained are down-safe, then LDPRE performs expression insertion to convert the partial availability to full availability. After the insertion, it returns true as e becomes fully available at the entry of node n. To do this, LDPRE uses *XAvail*(e, p) to determine whether e is fully available at the entry of the node n. Looking at Fig. 12, this situation corresponds to the result of propagating two queries from Node 4 to Node 2 and from Node 4 to Nodes 3 and 1. Since LDPRE already has both true and false as answers at the exits of Nodes 2 and 3, the predecessors of Node 4, LDPRE performs insertions when determining the answers of Node 4. As it inserts an expression at Node 3 where false is returned as an answer to the query, LDPRE returns true as an answer at Node 4.

Following equation presents the data flow equation for *NAvail*.

$$e_{n_p} \overset{def}{\Leftrightarrow} BUpdate(e,n,p)$$

$$NAvail(e,n) \overset{def}{\Leftrightarrow} \begin{cases} \text{false} & \text{if } n = \textbf{start} \\ AllPDSafe(e,n) & \text{else if } Insert(e,n) \\ \prod_{p \in pred(n)} XAvail(e_{n_p}, p) & \text{otherwise} \end{cases}$$

where *BUpdate*(e, n, p) is a function that replaces the operand of e with the argument of the ϕ function corresponding to node p to propagate the query to p if node n has a ϕ function defining the operand of e. *AllPDSafe*(e, n) is a function that returns true if all predecessors of n, into which the expression is inserted, satisfy down-safety; otherwise, it returns false. The $\prod_{p \in pred(n)} XAvail(e_{n_p}, p)$ of this equation applies the meet operator ⊓ described in Section 2.3 to all answers. This operation is performed when the combinations of answers returned from predecessors do not meet the *Insert* condition.

To keep the semantics of the program, LDPRE analyzes down-safety at the insertion point. The down-safety is checked by propagating the query from the exit of the node that is the candidate for insertion of the expression to **end**. Similar to the availability analysis, this query examines if there are occurrences of the same expression. In addition, this query propagation analyzes two types of down-safety for e: down-safety at the entry of node n (*NDSafe*(e, n)) and down-safety at the exit of node n (*XDSafe*(e, n)).

NDSafe(e, n) becomes true if there are no assignment statements for the operand of e between the occurrence of e in n and the entry of the node. Even if there is no occurrence of e in n, the predicate also becomes true when *XDSafe*(e, n) is true and there are no assignment statements for the operand of e.

LDPRE defines *NDSafe* as follows:

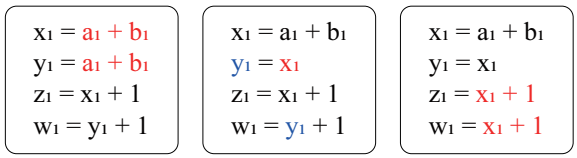


Fig. 16 Capturing second-order effects. (a) An Original code (b) Eliminating redundancy (c) Applying copy propagation to (b)

$$NDSafe(e,n) \overset{def}{\Leftrightarrow} \begin{cases} \text{true} & \text{if } visit[(e,n)] = \text{true} \\ \text{false} & \text{else if } visit[(e,n)] = \text{false} \\ \text{true} & \text{else if } comp(e,n) \\ \text{false} & \text{else if } kill(e,n) \\ XDSafe(e,n) & \text{otherwise} \end{cases}$$

$XDSafe(e,n)$ becomes true if all entries of the successors of n are down-safe. However, if n is an **end** node, it is not possible to propagate the query to successors anymore; thus, it returns false.

We define the data-flow equation for *XDSafe* as follows:

$$XDSafe(e,n) \overset{def}{\Leftrightarrow} \begin{cases} \text{false} & \text{if } n = \textbf{end} \\ \prod_{s \in succ(n)} NDSafe(e_{s_n}, s) & \text{otherwise} \end{cases}$$

$$e_{s_n} \overset{def}{\Leftrightarrow} FUpdate(e,s,n)$$

The function *FUpdate* is a function that updates operands to propagate the query in the opposite direction of *BUpdate*; $FUpdate(e,s,n)$ replaces the operand of e with the variable defined by the ϕ function of the node s if the operand is used as the argument of the ϕ function corresponding to node n.

If both true and false are obtained in node n, and the predecessors that returns false or $\top$ are down-safe, then LDPRE inserts the expression into the predecessors.

The data-flow equation for *Insert* is defined as follows:

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

$$Insert(e,n) \overset{def}{\Leftrightarrow} \begin{cases} \text{false} & \text{if } |\{A_{e_p} = \text{true} \mid p \in pred(n)\}| = 0 \\ \text{true} & \text{else if } |\{A_{e_p} = \text{false} \mid p \in pred(n)\}| > 0 \\ \text{false} & \text{otherwise} \end{cases}$$

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

4.2 Application to the Entire Program

It is well-known that applying PRE and copy propagation has the effect of exposing new redundancies, known as *second-order effects* [16]. Fig. 16 shows an example of capturing second-order effects. If LDPRE traverses the program in (a), it first finds that the right-hand side of y_1=a_1+b_1 is redundant. After eliminating this redundancy by replacing it with x_1, LDPRE applies copy propagation to replace y_1 with x_1 in the following expressions. This application updates the expression y_1+1 to x_1+1; LDPRE eliminates this redundancy later. As shown in this example, there is redundancy that becomes parsable by changing the lexical representation of the equation, called second-order effects. Once PREQP revokes eliminating redundancies as described in Section 3.3, the number of redundancies that could have been eliminated by reflecting the second-order effects is also reduced.

To capture the second-order effects, LDPRE visits each CFG node in topological sort order, and performs copy propagation after eliminating redundancy as well as previous DDPREs.

Algorithm 1 Pseudo codes of *XAvail* and *NAvail*

```
1:  Function XAvail(e, n)
2:    if visit[(e, n)] = ⊥
3:      return ⊤
4:    if visit[(e, n)] = true | | visit[(e, n)] = false
5:      return visit[(e, n)]
6:    if comp(e, n)
7:      visit[(e, n)] ← true
8:      return visit[(e, n)]
9:    if kill(e, n)
10:     visit[(e, n)] ← false
11:     return visit[(e, n)]
12:   visit[(e, n)] ← ⊥
13:   visit[(e, n)] ← NAvail(e, n)
14:   return visit[(e, n)]
15: Function NAvail(e, n)
16:   if n is start
17:     visit[(e, n)] ← false
18:     return visit[(e, n)]
19:   alist ← [] // This is used for recording answers
20:   for p in pred(n)
21:     e_{n_p} ← BUpdate(e, n, p)
22:     alist.append(XAvail(e_{n_p}, p))
23:   if InsertCond(alist) // Checking down-safety in this function
24:     new statements and a phi function are inserted
25:     visit[(e, n)] ← true
26:     return visit[(e, n)]
27:   return SQCAP(alist)
```

4.3 Algorithm Overview

As the data-flow equations describe how to determine the answers, we present implementations of our *XAvail* and *NAvail* in Algorithm 1.

Looking at the function *XAvail*, it first checks whether the same query has already been propagated to node n (lines 2~5) to obtain an answer without checking the statements of n in detail. If this is the first time visit for e, LDPRE checks whether there is the same expression or kill statement occurrence in n. If it finds the occurrence, LDPRE stores the answer as true or false according to *comp* and *kill* functions, respectively, and then it returns the answer (lines 6~11). Otherwise, LDPRE stores $\bot$ in $visit[(e,n)]$ to indicate that this node has already been visited by e (line 12) and calls *NAvail* to propagate queries to the predecessors of n (lines 13~14).

As the function $NAvail(e,n)$ obtains answers from predecessors, it first checks whether n is the **start** (lines 16~18). If n is not the **start**, it calls *XAvail* for each predecessor (lines 19~22). After collecting the answers, this function determines the answer at this node. If any expression or a ϕ function is inserted at the predecessors or this node, the answer is true (lines 23~26). Otherwise, the answer is determined by function *SQCAP* corresponding to $\prod_{p \in pred(n)} XAvail(e_{n_p}, p)$ defined in the data-flow equation of *NAvail* (line 27).

5. Experimental Evaluation

5.1 Settings

We implemented our algorithm[*1] as a low-level intermediate representation (LIR) converter using a COINS compiler[*2]. We conducted all experiments on a machine equipped with a Intel Corei7-8700K 3.70GHz CPU and an Ubuntu 64 bit operating system. We evaluated the effectiveness of our algorithm using seven programs (gzip, vpr, mcf, parser, gap, bzip2, and twolf) from CINT2000 and three programs (art, equake, and ammp) from CFP2000 in the SPEC CPU2000 benchmark. SPEC CPU2000 benchmark includes programs written in C++ and Fortran, which are not supported by the COINS compiler. The objective codes of the above 10 programs all behaved correctly in the environment mentioned above when COINS was used with no options and all PRE options that were used in this evaluation described below.

As the purpose of the experiment was to compare the performance of PRE's algorithms, we implemented the following five algorithms.

- **PRE** applies LCM that uses bit vectors to analyze the whole program, converting to the SSA form from the normal form, and converting to the normal form from the SSA form. This LCM uses a data-flow analysis that improves the efficiency of initialization, which is a costly process in data flow analysis [14].
- **PRE*2** applies LCM, copy propagation, LCM, converting to the SSA form from the normal form, and converting to the normal form from the SSA form.
- **PRE*3** applies LCM, copy propagation, LCM, copy propagation, LCM, converting SSA form from the normal form, and converting to the normal form from the SSA form.
- **PREQP** performs conversion to the SSA form from the normal form, PREQP, and converting to the normal form from the SSA form. Note that the original algorithm performs speculative code motion only for loop invariant codes to move them out of their loops; however, this study prohibits the speculative code motion by adding down-safety check for all insertions for fair comparison.
- **LDPRE** converts from the normal form to SSA form, LDPRE, and converts from the SSA form back to the normal form.

For fair comparisons, we applied the SSA/normal form conversion to the three exhaustive PREs to capture effects of coalescing in translating ϕ functions [17]. Note that these five algorithms commonly apply making three-address code for each statement and eliminating critical edges before they perform. As this paper compares the analysis time and the number of redundant expressions eliminated by each of the above PREs, we did not perform any optimizations for LIR not described above. To perform the above optimizations, COINS provides the following options.

- `divex` makes each statement three-address code
- `esplt` eliminates critical edges
- `pre` performs LCM
- `cpyp` performs copy propagation
- `prun` converts to the SSA form from the normal form
- `srd3` converts to the normal form from the SSA form
- `preqp` performs PREQP

Thus, to perform the above five algorithms, we set the following option combinations.

- **PRE**: `divex`, `esplt`, `pre`, `prun`, `srd3`
- **PRE*2**: `divex`, `esplt`, `pre`, `cpyp`, `pre`, `prun`, `srd3`
- **PRE*3**: `divex`, `esplt`, `pre`, `cpyp`, `pre`, `cpyp`, `pre`, `prun`, `srd3`
- **PREQP**: `divex`, `esplt`, `prun`, `preqp`, `srd3`
- **LDPRE**: `divex`, `esplt`, `prun`, `ldpre`, `srd3`

Note that we require only the two options of making each statement three-address code and eliminating critical edges before the application of LDPRE. However, it is possible to run other optimizations before or after LDPRE. In fact, COINS applies branch optimization, instruction selection, and register allocation after PRE in this evaluation [15].

[*1] All codes and configuration files used in this evaluation are available at `https://github.com/sumilab/programs/tree/master/ldpre`

[*2] The document is available at: `http://coins-compiler.osdn.jp/050303/index.html`. The source codes are opened at `https://sourceforge.net/projects/coins-project/`

Table 1 Number of eliminated expressions. Bold letters indicate the best results.

Program	PRE	PRE*2	PRE*3	A. PREQP	B. LDPRE	(B-A)/A
gzip	455	544	584	804	**984**	22.4%
gap	12,200	17,456	20,004	26,594	**28,901**	8.7%
bzip2	277	376	419	669	**807**	20.6%
vpr	1,452	1,873	2,056	2,974	**3,535**	18.9%
mcf	118	160	161	232	**282**	21.6%
parser	718	1,079	1,196	1,838	**2,136**	16.2%
twolf	4,210	5,618	6,306	9,138	**9,964**	9.0%
art	121	141	147	295	**359**	21.7%
ammp	870	1,335	1,544	3,927	**4,247**	8.1%
equake	186	239	251	847	**993**	17.2%

5.2 Results

In the remainder of this section, we first show the number of redundant expressions eliminated by the five algorithms. We then present the analysis time for each PRE algorithm. Next, focusing on DDPRE, we analyze the number of nodes queries of PREQP and LDPRE actually visited.

> **Q.** How many redundant expressions could LDPRE eliminate?
> **A.** It eliminated the most redundancy in all programs among the five algorithms.

Tab. 1 shows the numbers of eliminated expressions. It can be seen that LDPRE eliminated the most redundant expressions in all programs. Compared to PREQP, which may cancel redundancy elimination, LDPRE eliminated redundancy in approximately 10~20% more expressions. The cancellation of PREQP may reduce the number of reflecting second-order effects; thus, LDPRE eliminated more redundancy than PREQP. Note that PRE, PRE*2, and PRE*3 denote the number of expressions to be replaced by the predicate *Replace* that represents the replacement of expressions in LCM. As the number of replaced expressions increased when applying LCM and copy propagation for all programs, this result indicated that the capturing second-order effects

Table 2 Results of analysis time. The unit is seconds.

Program	PRE	PRE*2	PRE*3	A. PREQP	B. LDPRE	(A-B)/A
gzip	804.3	1,436.7	1904.9	406.2	**406.0**	0.0%
gap	17,545.4	35,340.4	50,073.0	**4,779.1**	4,788.3	-0.2%
bzip2	541.2	1,065.9	1,541.5	**161.9**	162.1	-0.1%
vpr	1,473.0	2,838.3	3,749.6	**798.7**	809.4	-1.3%
mcf	332.7	486.0	600.7	185.5	**179.4**	3.3%
parser	1,157.5	2,113.5	2,791.2	**644.6**	656.2	-1.8%
twolf	17,982.5	36,462.2	50,790.9	3,325.7	**3,307**	0.6%
art	145.3	290.2	397.6	55.6	**55.1**	0.9%
ammp	3,912.8	7,778.4	10,508.0	**1,308.8**	1,315.8	-0.5%
equake	741.9	1,536.6	2,176.2	167.2	**163.2**	2.4%

increases the eliminable redundancy.

> **Q.** Which algorithm achieved the shortest analysis time?
> **A.** Each LDPRE and PREQP obtained the shortest analysis time for half of the programs.
> **A.** Comparing LDPRE and PREQP, seven programs only had 1% change in analysis time. The remaining three programs also showed a maximum change of only approximately 3%.

Tab. 2 shows that analysis times of all algorithms. Each number in this table is the average of 10 runs of each program.

Exhaustive PREs vs. LDPRE: First, we compare the exhaustive PREs and LDPRE to see if LDPRE keeps a short analysis time that is a feature of demand-driven analysis achieved by limiting the analyzing range. We can see that LDPRE obtained shorter analysis times for all programs; thus, we can say that LDPRE retains the benefit of demand-driven analysis. Note here that both PREQP and LDPRE had shorter analysis times than the three PREs that use bit vectors.

PREQP vs. LDPRE: We next compare the analysis times of PREQP and LDPRE. The 7th column in this table shows the percentage of analysis time for LDPRE compared to PREQP. These results show that seven programs had only 1% change in analysis time while the remaining three programs showed a maximum change of only approximately 3%. As shown in Tab. 1, LDPRE increased the number of eliminated expressions compared to PREQP; it is natural that the number of queries propagated by LDPRE would also be increased. In fact, for the five programs, the analysis times of LDPRE were longer than that for PREQP though the increasing ratios were approximately 1∼2%. By contrast, LDPRE took less time to analyze in the remaining five programs, specifically taking approximately 3.3% less time for mcf. By delaying answer determination when a query for the same equation revisits the same node, LDPRE eliminates the need to check whether the answer is consistent, which was necessary in PREQP, and thus the analysis time was reduced by that amount.

> **Q.** Did the number of nodes visited by queries propagated by LDPRE change compared to those of PREQP?
> **A.** The total number of visited nodes increased for all programs.
> **A.** The average number of nodes visited by one query also increased in LDPRE for many programs.

Table 3 Number of nodes query visited and average.

Program	A.PREQP	B.LDPRE	(A-B)/A	C.PREQP	D.LDPRE	(C-D)/C
gzip	39,900	41,424	-3.8%	21.8	21.7	0.5%
gap	1,291,858	1,529,511	-18.4%	29.1	33.5	-15.1%
bzip2	35,439	41,116	-16.0%	25.8	28.8	-11.6%
vpr	91,350	103,658	-13.5%	17.8	19.5	-9.6%
mcf	9,271	10,140	-9.4%	16.3	16.8	-3.1%
parser	82,084	93,253	-13.6%	19.2	21.1	-9.9%
twolf	533,079	58,0955	-9.0%	42.1	45.1	-7.1%
art	8,952	9,098	-1.6%	17.9	18.0	-0.6%
ammp	138,426	143,628	-3.8%	21.5	22.0	-2.3%
equake	23,109	26,974	-16.7%	37.0	41.0	-10.8%

Next, we see how the increase of redundancy that LDPRE was able to eliminate affected the number of nodes visited by each query. Tab. 3 shows the numbers of nodes queries of PREQP and LDPRE visited and their averages per a query. First, the total numbers of nodes visited by the query are shown in 2∼3th columns. The overall trend was that query propagation by LDPRE visited more nodes than the ones by PREQP. In fact, as shown in the 4th column, the number of query visiting was approximately 10∼20% higher for the seven programs. This increase may be due to the greater number of redundant expressions that LDPRE was able to eliminate.

Next, columns 5∼6 shows the average number of nodes visited by a single query. The 7th column shows the percentage decrease in the mean. In nine programs, the average numbers of nodes visited by queries of LDPRE were higher than that of PREQP. Focusing on the gzip result, the average number of nodes visited per query decreased, even though the total number of visited nodes increased. This result simply indicates that the fact that the generated queries were increased by capturing the second-order effects although these queries obtained answers near the generated program points.

6. Conclusions

In this paper, we have proposed a novel demand-driven partial redundancy elimination (DDPRE) called lazy demand-driven PRE (LDPRE). LDPRE does not analyze the entire program to eliminate redundancy; it analyzes only parts of it by propagating query that checks for occurrences of the same expression. The query returns true if the same expression appears; otherwise, it returns false. Although PREQP has been proposed as a DDPRE, PREQP optimistically returns true when a query about e revisits the same node twice. However, this optimistic return of the answer may cause the expression not to be inserted even though it should have been. LDPRE returns $\top$, an element of the semilattice representing the inclusion of all answers, so that when a query on e revisits the same node, it can later determine the answer instead of true. The use of $\top$ makes it possible to delay the determination of the answer, allowing it to be true only when there is an actual occurrence of the expression, rather than optimistically determining it to be true. This extension allows the elimination of redundancies that could not be done by PREQP.

To evaluate the effectiveness of LDPRE, we used PRE analyzing the entire programs and PREQP as baselines. Using the SPEC CPU2000 benchmark to measure the number of eliminated expressions and analysis time, we found that LDPRE eliminated

more redundancy than the previous PRE and DDPRE in a short analysis time, which is a characteristic of demand-driven analysis. PREQP requires analysis to determine whether the insertion of equations can be done without problems by determining optimistic answers; however, LDPRE does not require this. This benefits further reduces the analysis time for many programs.

Future Works. Redundancy elimination involves two phases: analyzing redundancies and eliminating them. This paper focused on the former phase with the assumption that all analyzed redundancies are eliminated. Thus, we will improve the latter phase in the next study. Indeed, it is well known that eliminating redundancies does not always lead to shorten the execution times because the elimination tends to increase the number of register spill that would worsen execution efficiency of the objective code [10], [11]. Therefore, we can consider an extension of LDPRE to analyze execution frequency used in the optimization algorithms of JIT compilers. In recent years, JIT compilers have been widely used in situations where both shorter analysis time and execution time are required such as for Web browsers. In such usage, the shorter the analysis time involved in optimization, the better; thus, demand-driven analysis is desirable for using a powerful optimization such as PRE. In general, a JIT compiler provides frequency information for optimizations to perform them focusing on hot code paths. If we perform query propagation only for frequently executed nodes, we can suppress the number of redundancies to be eliminated by giving up analyzing less effective areas while maintaining analyzing areas where optimization is more effective. In addition, a method can be considered to reduce the execution time of the objective code by eliminating redundancy while suppressing register spills. For example, it is interesting to use a register allocator generating more spills in cold paths to improve the execution time.

References

[1] Alpern, B., Wegman, M. N. and Zadeck, F. K.: Detecting equality of variables in programs, POPL '88, New York, NY, USA, ACM, pp. 1–11 (1988).

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

[3] Bodik, R., Gupta, R. and Soffa, M. L.: Complete removal of redundant expressions, PLDI '98, New York, NY, USA, ACM, pp. 1–14 (1998).

[4] 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).

[5] 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).

[6] Cocke, J.: Global Common Subexpression Elimination, *Proceedings of a Symposium on Compiler Optimization*, New York, NY, USA, ACM, pp. 20–24 (online), DOI: 10.1145/800028.808480 (1970).

[7] 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).

[8] Gupta, R. and Bodík, R.: Register Pressure Sensitive Redundancy Elimination, *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, CC '99, Berlin, Heidelberg, Springer-Verlag, pp. 107–121 (1999).

[9] 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).

[10] 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).

[11] 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).

[12] 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).

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

[14] Morgan, B.: *Building an Optimizing Compiler*, Digital Press (1998).

[15] Mori, K., Abe, S. and Nakata, I.: A Guide for Compiler Developers Using the Latest Tool - The COINS Compiler Infrastructure - : LIR (Low-level Intermediate Representation) and the Outline of the Backend, *IPSJ magazine*, Vol. 47, No. 6, pp. 662–669 (2006). (in Japanese).

[16] 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).

[17] Sreedhar, V. C., Ju, R. D.-C., Gillies, D. M. and Santhanam, V.: Translating Out of Static Single Assignment Form, *Static Analysis* (Cortesi, A. and Filé, G., eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 194–210 (1999).

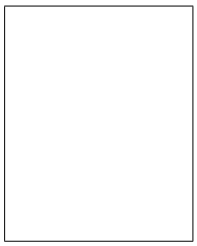
[18] Steffen, B., Knoop, J. and Rüthing, O.: Efficient Code Motion and an Adaption to Strength Reduction, *Proceedings of the International Joint Conference on Theory and Practice of Software Development on Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Development (CCPSD): Vol. 2*, TAPSOFT '91, Berlin, Heidelberg, Springer-Verlag, pp. 394–415 (1991).

[19] 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).

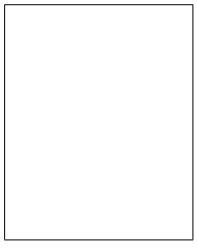
[20] Takimoto, M.: Speculative Partial Redundancy Elimination Based on Question Propagation, *Information Processing Society of Japan Transactions on Programming*, Vol. 2, No. 5, pp. 15–27 (2009).

[21] Wimmer, C. and Mössenböck, H.: Optimized Interval Splitting in a Linear Scan Register Allocator, *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, New York, NY, USA, Association for Computing Machinery, pp. 132–141 (online), DOI: 10.1145/1064979.1064998 (2005).

[22] 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).



Yuya Yanase received his B.E. degree in Computer Sciences from Takushoku University in 2022. His research interests includes 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 Assistant Professor at the department of computer science, Takushoku University, Japan. His research interests lie on compiler, information retrieval, computational history, and history learning.

Appendix

We analyze the execution time of the objective code generated by applying each PRE.

Q. How different the execution time of the objective code generated by the application of LDPRE compared to other algorithms?
A. The overall trend was similar execution times of LDPRE to the ones of comparators; however, LDPRE obtained the best results for the four programs.
A. LDPRE got better execution times for six programs compared to three PREs analyzing the entire program.
A. Compared to PREQP, LDPRE's execution time differed by approximately 2∼3% for the nine programs; however, LDPRE's execution time was 25.5% longer for equake.

Tab. A·1 shows the execution times of the objective code generated by applying each algorithm. The numbers in this table are the average of 10 runs of each program. Overall, the LDPRE results were similar in execution time to the comparators; however, LDPRE achieved the best results for the four programs: gap, twolf, art, and ammp. Comparing the results of PRE, which apply LCM only once, and LDPRE, LDPRE obtained slower times for the vpr, parser, and equake; however, the difference was only approximately 0.8∼1.5%. An overall comparison of the three PREs analyzing the entire program and LDPRE shows that the LDPRE had comparable or better execution times for six programs. The three PREs show that the capturing second-order effects affects execution time, because repeated LCM and copy propagation tends to reduce execution time for many programs. As DDPRE applies copy propagation for each elimination, all programs with the shortest execution time for the objective code were obtained by DDPRE. Furthermore, DDPRE achieved better results on not only the execution time of the objective code, but also the analysis time, as we have already seen the results in Tab. 2.

Next, we compare the results of PREQP and LDPRE. LDPRE achieved better execution times for four programs whereas it worse the times for six ones. However, the difference in the nine programs, excluding earthquake, was approximately 2∼3%. In equake, LDPRE got 25.5% longer execution time than the one of PREQP. To better understand why LDPRE worsened the execution time for equake, we counted the number of expressions eliminated by the algorithms. Tab. 1 shows this result. We can see that LDPRE eliminated more redundant expressions than PREQP. Focusing on equake, LDPRE eliminated 1.2 times more redundancies than PREQP. In general, register pressure tends to be higher when copy propagation is applied after redundant statements have been eliminated [8]. We then counted the number of register spills the algorithms generated and listed them in Tab. A·2. Tab. A·2 shows that LDPRE generates approximately 54% more register spills than PREQP for equake. We can conclude that LDPRE eliminated more redundancy than PREQP but also generated more register spills as an effect of the elimination, resulting in longer execution times. Register spills are less likely to occur on machines with a large number of registers; thus, LDPRE may have a shorter execution time than PREQP on other CPUs, even equake.

Table A·1 Results of execution time of objective code. Bold letters indicate the number with the shortest execution time. The unit is seconds.

Program	PRE	PRE*2	PRE*3	A. PREQP	B. LDPRE	(A-B)/A
gzip	80.8	76.5	79.6	**74.6**	74.8	-0.2%
gap	61.2	60.6	59.6	59.8	**58.8**	1.6%
bzip2	59.5	57.6	57.7	**56.3**	57.0	-1.2%
vpr	52.6	55.4	53.1	**51.8**	53.3	-2.8%
mcf	24.3	24.0	**23.8**	**23.8**	24.0	-0.8%
parser	116	116	117	**114**	117	-2.6%
twolf	85.2	87.1	85.5	84.9	**83.5**	1.6%
art	20.3	**20.1**	**20.1**	20.3	**20.1**	0.9%
ammp	83.4	82.7	84.6	83.7	**82.3**	1.4%
equake	40.6	35.3	33.8	**32.9**	41.3	-25.5%

Table A·2 Number of register spills.

Program	PRE	PRE*2	PRE*3	A. PREQP	B. LDPRE	(A-B)/A
gzip	146	150	146	108	138	-27.7%
gap	2,898	3,118	3,026	1,973	2,671	-35.3%
bzip2	132	129	140	97	132	-36.0%
vpr	751	763	744	663	744	-12.2%
mcf	91	96	93	73	88	-20.5%
parser	280	288	285	284	376	-32.3%
twolf	1,132	1,170	1,179	784	986	-25.7%
art	35	35	34	28	33	-17.8%
ammp	570	455	472	400	535	-33.7%
equake	83	87	100	50	77	-54.0%

Q. How much register spill LDPRE generated?
A. The number of times LDPRE occur tends to be less than those of PREs that analyze the entire program
A. LDPRE generated the spills 10∼50% more than PREQP in all programs

In order to provide a deeper analysis on the execution time of the objective code, we present here the results of measuring the number of spills actually generated by the register allocation equipped with COINS. The register allocation determines spills by static analysis because the algorithm is based on Appel-George's Iterated Register Coalescing, an improved version of the Graph Coloring algorithm. Therefore, the numbers shown here have less correlation with execution time of the target code. However, as LDPRE is designed as a optimization technique applied before register allocation, a discussion on the number of static spills caused by LDPRE is useful when selecting register allocation algorithms. First, we compare the three PREs that analyze the entire program and DDPREs. The result shows that DDPREs generate fewer spills in many programs. While PRE, which analyzes the entire program, targets normal-format programs, both PREQP and LDPRE target SSA-format programs. In the elimination of redundancy on the SSA format, the algorithms insert ϕ functions. All programs represented by SSA format need to be reverted to normal format. During that conversion, each ϕ function is converted to a copy statement by its destination and argument variables. Thus, a possible reason for the lower number of spills for DDPRE is that this copy statement divides the live-range of

the variable and reduces register pressure. In fact, a technique has been proposed to improve the performance of register allocation by partitioning the live-range of a variable [21]. Comparing PREQP and LDPRE, the spills that occurred in all programs were higher in LDPRE. In particular, LDPRE increased them approximately 54% for equake. This reason is due to the large number of redundancies that could be eliminated.