IPSJ Transactions on Programming Vol.5 No.1 9–16 (Mar. 2012)

Regular Paper

Demand-driven Partial Dead Code Elimination

Munehiro Takimoto[1,a)]

Received: June 28, 2011, Accepted: November 8, 2011

Abstract: Partial dead code elimination (PDE) is a powerful code optimization technique that extends dead code elimination based on code motion. PDE eliminates assignments that are dead on some execution paths and alive on others. Hence, it can not only eliminate partially dead assignments but also move loop-invariant assignments out of loops. These effects are achieved by interleaving dead code elimination and code sinking. Hence, it is important to capture second-order effects between them, which can be reflected by repetitions. However, this process is costly. This paper proposes a technique that applies PDE to each assignment on demand. Our technique checks the safety of each code motion so that no execution path becomes longer. Because checking occurs on a demand-driven basis, the checking range may be restricted. In addition, because it is possible to check whether an assignment should be inserted at the blocking point of the code motion by performing a demand-driven analysis, PDE analysis can be localized to a restricted region. Furthermore, using the demand-driven property, our technique can be applied to each statement in a reverse postorder for a reverse control flow graph, allowing it to capture many second-order effects. We have implemented our technique as a code optimization phase and compared it with previous studies in terms of optimization and execution costs of the target code. As a result, our technique is as efficient as a single application of PDE and as effective as multiple applications of PDE.

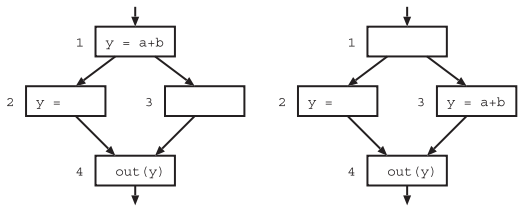
Keywords: partial dead code elimination, dead code elimination, demand-driven dataflow analysis, code optimization

1. Introduction

Dead code elimination [1] has been traditionally used as a code optimization technique used by compilers. If a variable is not used after a certain point p in a program, the variable is said to be *dead* at p. An assignment statement with a dead variable on its left-hand side is considered to be *totally dead* or simply dead. Dead code elimination not only enhances the efficiency of program execution but also reduces the program size by eliminating such totally dead assignments.

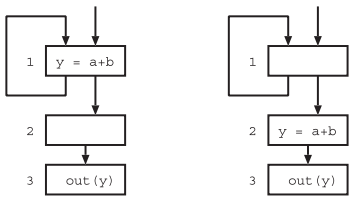
Partial dead code elimination (PDE) is a technique used to enhance the effectiveness of dead code elimination. As shown in **Fig. 1** (a), the statement at node 1 is dead on the left-hand side of the path after branching, whereas it is alive on the right-hand side of the path. Such an assignment is called *partially dead* [13]. Because a partially dead assignment is not totally dead, it cannot be removed by the traditional dead code elimination technique. PDE removes such partially dead assignments by rendering them as totally dead through the *code sinking* technique, which moves assignments forward. Thus, PDE is accomplished by combining two types of program transformations, code sinking and dead code elimination. For example, the assignment statement `y = a + b` in Fig. 1 (a) can be rendered as totally dead and is removed by sinking it from node 1 to the beginning of node 3, as shown in Fig. 1 (b).

In addition, PDE can move loop-invariant assignments out of the loop. In **Fig. 2** (a), the assignment `y = a + b` in the loop is dead on the path going back to node 1 because the assignment is killed by itself, whereas it is still alive on the path that goes out of the loop. This is because its left-hand side is used by the statement `out(y)` at node 3. This denotes that `y = a + b` is partially dead and can be removed by inserting the same statement at node 2. The transformation is equivalent to moving loop-invariant assignments out of the loop.



(a) Original code (b) Code after applying PDE

Fig. 1 Removing partially dead assignments.



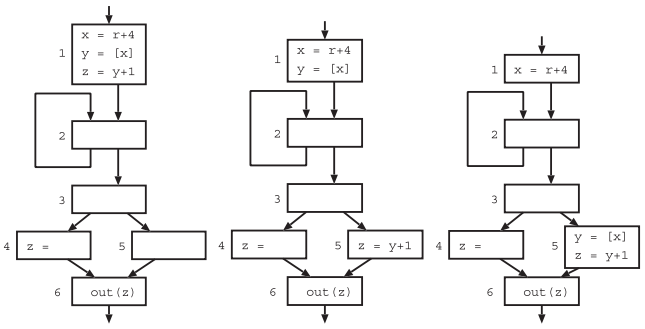
(a) Original code (b) Code after applying PDE

Fig. 2 Moving loop-invariant assignments out of a loop.

Both code sinking and dead code elimination may generate new candidates for other code sinking or dead code elimination processes. These effects can be classified into the following four types [13]:

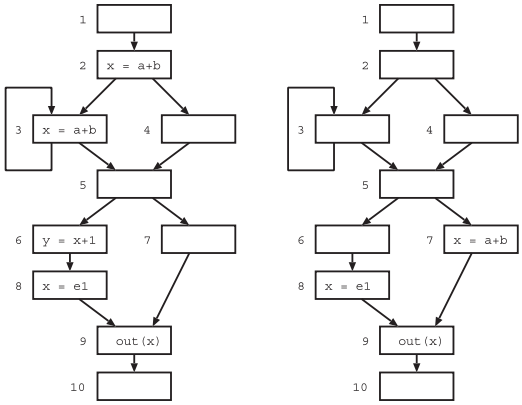
Sinking-elimination effect Sinking an assignment renders the assignment dead.

[1] Department of Information Sciences, Tokyo University of Science, Noda, Chiba 278–8510, Japan
[a)] mune@cs.is.noda.tus.ac.jp



(a) Original code (b) One-time PDE application (c) Application of PDE twice

Fig. 3 Decrease in effectiveness by incomplete application of PDE.



(a) Original code (b) Application of our technique

Fig. 4 Effectiveness of our technique.

Sinking-sinking effect For a series of assignments $s1$ and $s2$, if the variable defined by $s1$ is used or redefined in $s2$ or if the variable used in $s1$ is defined by $s2$, it is possible to further sink statement $s1$, which is blocked by $s2$, by sinking $s2$.

Elimination-sinking effect For $s1$ and $s2$ stated above, it is possible to sink $s1$ as a result of removing $s2$.

Elimination-elimination effect By eliminating the assignment s, it is possible to eliminate an assignment for a variable used in s.

Although it is possible to obtain the sinking-elimination effect by applying PDE once, it is necessary to apply it again to obtain the remaining three effects. This is because the application of PDE generates another dead assignment, providing an opportunity for further removal. These effects are called the *second-order effects* of PDE. To reflect the second-order effects, it is necessary to apply PDE repeatedly, and the analysis cost is known to be high.

In addition, the effect of PDE may be limited unless the second-order effects are reflected thoroughly. For example, in the program shown in **Fig. 3** (a), all assignments at node 1 sink to node 5 after applying PDE three times. Considering the process of each application, the first application of PDE sinks only the assignment `z = y + 1` to node 5. As a result, the *live range* of variable `y` is extended, as shown in Fig. 3 (b). An extension of the live range of variable's may hinder register allocation of the variable and increase the chance of spills. Next, the second application of PDE sinks the statement `y = [x]`, which denotes an assignment from the memory location `x`, to node 5. As a result, as shown in Fig. 3 (c), the statements `y = [x]` and `x = r + 4`, which were supposed to form one load instruction `y = [r + 4]` in the original code, are detached. These problems that arise from the incomplete application of PDE are called *incomplete PDE application problems*.

This paper introduces an algorithm that applies PDE to each assignment statement on demand and proposes a technique to efficiently reflect second-order effects.

PDE analysis is based on *dataflow analysis*. In general, the dataflow analysis is determined by the type of the solution, *minimal* or *maximal* of a *dataflow equation*. A minimal solution is represented as a union of dataflow sets propagating from different program points in a dataflow equation. Hence, it is sufficient to consider only the program points that are reachable from the relevant points of the analysis. In contrast, a maximal solution is obtained by initializing all program points with positive solutions and negating the solutions of points that do not satisfy the dataflow equation. Hence, it is necessary to calculate dataflow information for the entire program even if only information propagating from particular program points is required.

Because the dataflow analysis of PDE requires a maximal solution, it is difficult to restrict the range of analysis to obtain a result for specific assignments. Our technique tests the possibility of sinking at each program point v while tracking reachable points from point n in which the target assignment statement s resides. Once v is found to be sinkable, it is sunk further, and if it is not found to be sinkable, s is inserted at v. Because the possibility of sinking can also be tested using the demand-driven analysis, the range of analysis is restricted to a requisite minimum. Demand-driven PDE is denoted as *DDPDE*.

Using the demand-driven nature of this technique, it is possible to sequentially apply DDPDE to each statement, starting from the assignment that is closest to the end point without sacrificing efficiency. As a result, it is possible to directly reflect most of the four types of second-order effects. For example, applying DDPDE to a program sequentially from the end point, as shown in **Fig. 4** (a), results in the program given in Fig. 4 (b).

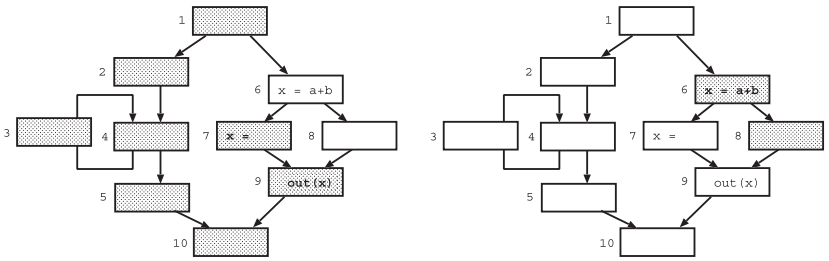
This paper is organized as follows. After presenting preliminaries in Section 2, we presents the concept of demand-driven code sinking in Section 3. In Section 4, we develop the demand-driven code sinking to DDPDE extending the notion of insertions of assignments to eliminate partially dead assignments. In Section 5, we show the actual procedures required for the transformation of a program. In Section 6, we present the evaluation results to demonstrate the effectiveness of the technique. Finally, we discuss related works in Section 7 and provide concluding remarks in Section 8.

2. Preliminaries

For ease of presentation, we assume that the input program consists of the following sets:

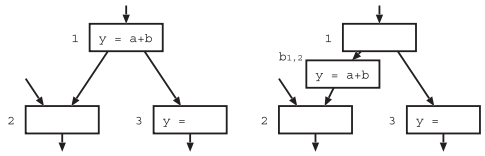
- Var: a set of variables
- C: a set of constants
- OP : a set of operators

Also, we represent a programs as a *control flow graph* (CFG), i.e., in the form $CFG = (N, E, \mathbf{s}, \mathbf{e})$ with node set N consisting of



(a) Propagation of *false* (b) Propagation of *true*

Fig. 6 Calculation of code sinking.



(a) Critical edge (b) Removing criticality

Fig. 5 Critical edges and their elimination.

a single statement, edge set $E \subset N \times N$ representing the flow of control, and **s** and **e** representing the unique start node and end node of a *CFG* with no statement, respectively. The set of predecessor nodes of node n is represented as $pred(n)$ and the set of successor nodes as $succ(n)$.

The statements in a CFG are classified into the following three groups:

Assignment statements: These statements are represented as a *three-address code*, such as $x = t$, where $x \in Var$, and t is an expression containing at most one operator.

Skip statements: These statements do not need to be processed. A CFG node that does not have a statement is considered to include a skip statement.

Relevant statements: These statements refer to statements that may change the meaning of the program if they are moved, such as those involving a memory storage operation, a function call, or branching. They are treated as being locked to the original CFG nodes, and the variables used in relevant statements are all treated as being alive (not dead). In this paper, a relevant statement is represented as an output statement, such as $out(t)$ [13].

Because this technique is based on code motion, its effect may be limited if there exist *critical edges* leading from a node with more than one successor to a node with more than one predecessor, such as an edge from node 1 to node 2 in **Fig. 5** (a) [11], [12], [14]. We assume that a critical edge is removed by inserting a synthetic node as shown by the insertion of $b_{1,2}$ in Fig. 5 (b) [*1].

3. Demand-driven Code Sinking

In this section, the demand-driven code sinking method is shown before presenting DDPDE. Subsequently, the method is extended to DDPDE by making the insertion step active selectively on the basis of deadness information. First, we summarize the traditional dataflow equation used to obtain the maximal solution of code sinking. Then, we discuss the manner in which the equation for a maximal solution is converted into one for a minimal solution, i.e., by detaching the part of testing upward safety. Finally, we present the demand-driven method based on the converted equation with a demand-driven upward safety test.

3.1 Code Sinking and Maximum Solution

The code sinking step of PDE is based on the dataflow analysis [13]. Assume that predicates $N\text{-}DELAYED_n(s)$ and $X\text{-}DELAYED_n(s)$ represent whether assignment s can be sunk to the entry and exit of the CFG node n respectively. Then $N/X\text{-}DELAYED_n(s)$ can be calculated by having the following local information $LOCDELAYED_n(s)$ and $LOCBLOCKED_n(s)$ flow to next node n:

LOCDELAYED$_n(s)$: An assignment that can be sunk exists at node n.

LOCBLOCKED$_n(s)$: The sinking of s is blocked by node n either because a variable contained in s is modified at the node n or because s modifies a variable contained in a statement at n.

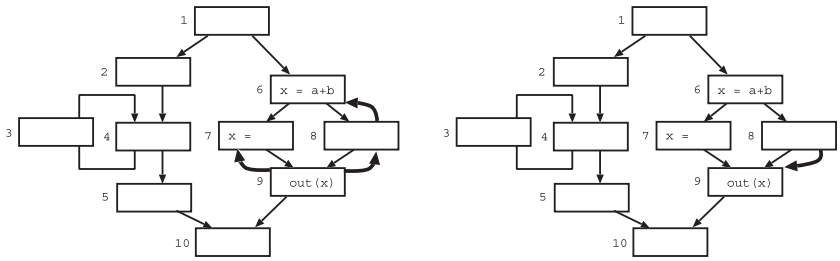
The propagation of information for $N/X\text{-}DELAYED_n(s)$ is defined by the following dataflow equations:

$$N\text{-}DELAYED_n(s) = \begin{cases} false & \text{if } n \text{ is the starting node} \\ \prod_{m \in pred(n)} X\text{-}DELAYED_m(s) & \text{otherwise} \end{cases} \quad (1)$$

$$X\text{-}DELAYED_n(s) = LOCDELAYED_n(s) \vee N\text{-}DELAYED_n(s) \wedge \neg LOCBLOCKED_n(s) \quad (2)$$

Code sinking analysis starts with the initial state $N/X\text{-}DELAYED_n(s) = true$ for all nodes other than the start node, and then the states of the nodes in the program that do not satisfy the dataflow equation are modifid to *false* The solution thus obtained from the equations is called the maximal solution. Typically, the maximal solution is sought when information propagation from adjacent nodes is represented as a product ($\prod$) in the dataflow equation. **Figure 6** shows an example of sinking the assignment `x = a + b` at node 6. In the conventional dataflow analysis, the state *false* obtained from local information is propagated through the program after initializing all nodes other than the start node with *true* for $N/X\text{-}DELAYED_n(s)$. It should be noted here that in Fig. 6, while the range over which *true* propagates is restricted to a small range reachable from node 6 (Fig. 6 (b)), the range over which *false* propagates covers almost the entire program (Fig. 6 (a)).

[*1] In the CFGs used in the figures in this paper, to simplify representation, critical edges are not removed.



(a) Query propagation for checking upward safety (b) Query propagation for checking deadness

Fig. 7 Query propagations in Fig. 6 (a).

3.2 Code Sinking Based on Minimal Solution

As shown in Fig. 6, it is possible to restrict the propagation range of dataflow if the nodes for which $N/X\text{-}DELAYED_n(s) = true$ can be directly calculated. Such a solution obtained by propagating *true* information from relevant program points in the dataflow analysis is called the minimal solution.

The dataflow equation used to obtain the minimal solution is obtained by calculating the sum ($\sum$) of information from adjacent nodes. The operation employed to obtain a product is used in Eq. (1) to guarantee upward safety, which denotes the condition for avoiding the lengthening of any execution path through insertions of s. To replace product with summation, it is necessary to test *upward safety* at the destination of sinking each time. Hence, by introducing $isUpSafe_n(s)$, which is a program that tests the upward safety of the assignment s at the node n, the dataflow equation for obtaining the minimal solution can be represented by modifying Eq. (1) as follows:

$$N\text{-}DELAYED_n(s) = \begin{cases} false & \text{if } n \text{ is the starting node} \\ isUpSafe_n(s) \wedge \sum_{m \in pred(n)} X\text{-}DELAYED_m(s) & \text{otherwise} \end{cases} \quad (3)$$

The dataflow analysis used to calculate the minimal solution for code sinking is referred to as *demand-driven code sinking*.

Here, $isUpSafe_n(s)$ can also be calculated in on-the-fly by using the demand-driven dataflow analysis [8] to obtain the solution of the dataflow equation at a particular point in the program such as node n. Therefore, the range analyzed by demand-driven code sinking is strictly restricted to a range that is reachable from the node n.

3.3 Demand-driven Dataflow Analysis for Upward Safety

The demand-driven dataflow analysis propagates a query in a direction opposite to that of conventional dataflow analysis from a particular point n to obtain the validity of a dataflow fact at n. To simplify the explanation, a query is assumed to ask whether the dataflow fact is *true* or $false$. If the answer to the query is obtained as the local solution $false$ at some point in the program, the answer to the query at the original point n is also $false$. In contrast, if the answer $false$ is not obtained for the query at any point in the propagation process, the answer at n is *true*.

To perform a demand-driven dataflow analysis to test upward safety, it is sufficient to propagate a query to check the presence of the assignment s. The query q generates a local solution by using following conditions of the node v:

(1) $q = true$ if $LOCDELAYED_v(s) = true$.
(2) $q = false$ if v is the start node.
(3) $q = false$ if $LOCBLOCKED_v(s) = true$.
(4) $q = true$ if the query has already propagated to the node v.

If the solution cannot be directly determined at v, then the query should propagate to all successors of v. **Figure 7** (a) shows an example of the upward safety test at the entry of node 9. A query to check the presence of the statement `x = a + b` propagates in a backward direction from node 9. Because $LOCBLOCKED_7$("`x = a + b`") yields the *true* at node 7, the solution $q = false$ is obtained. In contrast, the query propagated to node 8 will not yield a solution, and will therefore propagate further to node 6. Because $LOCDELAYED_6$("`x = a + b`") = true at node 6, the local solution $q = true$ will be obtained. As a result, because of a query yielding the solution $false$, the upward safety test at the entry point of node 9 yields a negative result. In practice, it is not necessary to propagate the query to nodes 8 and 6 because upward safety at node 9 is determined to be $false$ upon obtaining the local solution $q = false$ at node 7.

Program 1 below denotes the function *isUpSafe* which tests the upward safety of the assignment *target* at the node v. The function *isUpSafe* stores the candidate nodes for propagation in its worklist and it is based on the worklist algorithm that extracts a node to determine the solution of a query [8]. The predicates $LOCDELAYED_v(s)$ and $LOCBLOCKED_v(s)$ are represented as arrays $locdelayed[s, v]$ and $locblocked[s, v]$ respectively. The commented-out lines of code in lines 7–9 are explained later.

Program 1 (Upward Safety Test)

```
Function isUpSafe(target, v)
1:  worklist := {v}; query[*] := false
2:  while worklist ≠ ∅ do
3:    let n ∈ worklist; remove n from worklist
4:    query[n] := true
5:    for each p ∈ pred(n) do
6:      if locdelayed[target, p] then
7:        // if p ∉ done then
8:          // add p to done
9:          // add p to cand
10:       continue
11:     else if p = entry ∨ locblocked[target, p] then
12:       return false
13:     else if ¬query[p] then
14:       add p to worklist
15: return true
```

3.4 Implementation of Demand-driven Code Sinking

Dataflow Eqs. (2) and (3) exclude the term *isUpSafe* and can

be implemented as a worklist algorithm [2] by applying the *slot-wise* method [7] to each statement because they are ordinary dataflow equations for analyzing reachability. The function *insertForSinking* shown in Program 2 performs a slot-wise analysis of code sinking for the *target* statement at the node v.

From line 4 onward, where code sinking to the successor s is described, sinking is not performed to the entry of s if $isUpSafe(target, s) = false$ as prescribed in lines 5 and 6. In addition, because of the ensuing expansion of PDE, code sinking to the entry of s is prevented if s is the end node. Furthermore, code sinking to the entry of s is determined to be feasible ($N\text{-}DELAYED_s(target) = true$) only if these conditions are not satisfied as shown in line 8. Here the node for which code sinking is determined to be feasible is recorded in the variable *ndelayed* in line 9 and utilized in line 8 to avoid revisiting each node.

Code sinking to the exit of s is determined to be feasible ($X\text{-}DELAYED_s(target) = true$) if $locblocked[target, s] = false$ in line 12. Because this makes further sinking possible, s is added to the worklist as shown in line 12.

Program 2 (Determination of Insertion Point in Demand-driven Code Sinking)

```
Function insertForSinking(target, v)
1:  worklist := {v}; ndelayed := ∅
2:  while worklist ≠ ∅ do
3:    let n ∈ worklist; remove n from worklist
4:    for each s ∈ succ(n) do
5:      if s = exit ∨ ¬isUpSafe(target, s) then
6:        // X-INSERT_target(n) is decided to be true
7:        break
8:      if s ∉ ndelayed then
          // N-DELAYED_s(target) is decided to be true
9:        add s to ndelayed
10:       if locblocked[target, s] then
11:         // N-INSERT_s(target) is decided to be true
12:       else add s to worklist
          // X-DELAYED_s(target) is decided to be true
```

In the demand-driven analysis applied to each statement such as in Program 2, it is possible to incrementally calculate the insertion points that are conventionally determined after the analysis. Here, the predicate $N/X\text{-}INSERT$, which represents insertion at either the entry or exit point of n, is expressed as follows [13].

$$N\text{-}INSERT_n(s) = N\text{-}DELAYED_n(s) \wedge LOCBLOCKED_n(s) \quad (4)$$

$$X\text{-}INSERT_n(s) = X\text{-}DELAYED_n(s) \wedge \sum_{m \in succ(n)} \neg N\text{-}DELAYED_m(s) \quad (5)$$

In Program 2, considering the lines where $N/X\text{-}DELAYED$ becomes *true*, it is clear that $N/X\text{-}INSERT$ will be *true* in lines 6 and 11 on the basis of Eqs. (4) and (5). Hence, it is not necessary to record the results for $N/X\text{-}DELAYED$ to any variable except *ndelayed*, which is used to avoid revisiting the nodes.

4. Extension of Insertion Points

Partially dead assignments can be easily eliminated by controlling the insertion operation in demand-driven code sinking introduced in the previous section. Elimination can be achieved by inhibiting the insertion to n because of code sinking if the assignment s is dead at the node n. By using a predicate $isDead_n(s)$ to show whether the assignment s is totally dead at the exit of the node n, a predicate $USED_n(s)$ to show whether the target variable in s is used at n, and a predicate $MOD_n(s)$ to show whether the target variable in s is updated at n, Eqs (4) and (5) are extended as follows:

$$N\text{-}INSERT_n(s) = \neg(\neg USED_n(s) \wedge (MOD_n(s) \vee isDead_n(s))) \wedge N\text{-}DELAYED_n(s) \wedge LOCBLOCKED_n(s) \quad (6)$$

$$X\text{-}INSERT_n(s) = \neg isDead_n(s) \wedge X\text{-}DELAYED_n(s) \wedge \sum_{m \in succ(n)} \neg N\text{-}DELAYED_m(s) \quad (7)$$

The term $\neg USED_n(s) \wedge (MOD_n(s) \vee isDead_n(s))$ in Eq. (6) shows whether the assignment s is totally dead at the entry of the node n. For example, in the code sinking example shown in Fig. 6, the statement `x = a + b` is inserted at the exit point of node 8 on the basis of Eqs. (6) and (7).

Although the dataflow analysis is required to determine $isDead_n(s)$, it is possible to use the demand-driven dataflow analysis, similar to the case of $isUpSafe_n(s)$. That is, $isDead_n(s)$ propagates the query to check the deadness of the variable on the left-hand side of the assignment s at the exit of the node n. The query q yields a local solution on the basis of the following conditions of the next node v.

(1) $q = false$ if $USED_v(s) = true$.
(2) $q = true$ if $MOD_v(s) = true$.

If a solution is not obtained directly at the node v, the query will be propagated to all successors of v. For example, in the code sinking example in Fig. 6, the function $isDead_8$("`x = a + b`") is determined for insertion at the exit of node 8. As shown in Fig. 7 (b), the query q will propagate from node 8 to node 9 and the result $q = false$ will be obtained at node 9. Because this denotes that $isDead_8$("`x = a + b`") $= false$, the statement `x = a + b` will be inserted at the exit of node 8.

Program 3 represents the function *isDead* to test the deadness of the assignment *target* at the node v. As was the case with *isUpSafe*, *isDead* is based on the worklist algorithm. Predicates $USED_v(s)$ and $MOD_v(s)$ are represented using arrays $used[s, v]$ and $mod[s, v]$, respectively.

Program 3 (Deadness Test)

```
Function isDead(target, v)
1:  worklist := {v}; query[*] := false
2:  while worklist ≠ ∅ do
3:    let n ∈ worklist; remove n from worklist
4:    query[n] := true
5:    for each s ∈ succ(n) do
6:      if used[target, s] then
7:        return false
8:      if mod[target, s] then
9:        continue
10:     else if ¬query[s] then
11:       add s to worklist
12: return true
```

By extending Program 2 on the basis of Eqs. (6) and (7), Program 4 shows the function *insert*, which determines the insertion

point of the assignment *target* for node v. The function *insert* calculates the *target* insertion points for *ninsert* and *xinsert* on the basis of *N-INSERT* and *X-INSERT*.

Program 4 (Determination of Insertion Point in DDPDE)

```
Function insert(target, v)
1:  worklist := {v}; ndelayed := ∅
2:  while worklist ≠ ∅ do
3:    let n ∈ worklist; remove n from worklist
4:    for each s ∈ succ(n) do
5:      if s = exit ∨ ¬isUpSafe(target, s) then
6:        if ¬isDead(target, n) then
7:          add n to xinsert
8:        break
9:      if s ∉ ndelayed then
10:       add s to ndelayed
11:       if locblocked[target, s] then
12:         if ¬(¬used[target, s]
                ∧(mod[target, s] ∨ isDead(target, s))) then
13:           add s to ninsert
14:       else add s to worklist
```

The results of *isUpSafe* in line 5 and *isDead* in lines 6 and 12 can be cached only if *insert* is executed. If the solution to the query is cached, it is possible to avoid redundant query propagation by assuming the solution to be the local solution [8].

5. Code Conversion

After sinking the assignment s at a certain node, it may be necessary to sink s at other nodes. For example, because the statement `x = a + b` at node 2 is partially dead in **Fig. 8** (a), it can be eliminated by inserting it at node 5 as shown in Fig. 8 (b). However, because this insertion increases the number of calculations on the path through nodes 3 and 5 (as shown in Fig. 8 (c)), the same statement `x = a + b` at node 3 should also be eliminated by inserting it at node 6.

An assignment s', which is the target of simultaneous elimination that results from applying DDPDE to the assignment s, resides at points that are backward reachable from the insertion point. Such s' can be simultaneously found by performing the upward safety test at the insertion point. The commented-out code in lines 7–9 in Program 1 records the nodes with other assignments to be eliminated in *cand*. The variable *done* holds the processed nodes and provides the nodes to be eliminated during the program transformation, in addition to ensuring that the same assignments are not processed twice.

Program 5 shows the function *ddPDE*, which ultimately applies the function *insert* to all nodes to be considered and performs the program transformation. In the transformation, the *target*s of nodes *done* will be eliminated in lines 10 and 11, and the same assignment is inserted at the entry of nodes *ninsert* and at the exit of nodes *xinsert* in lines 12–15.

Program 5 (Transformation of a Program)

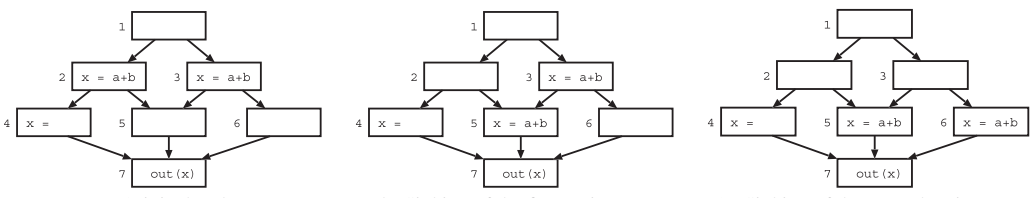
```
1: done := ∅
2: cand := ∅
3: ninsert := ∅
4: xinsert := ∅

Function ddPDE(target, v)
5:  cand:={v}
6:  done:={v}
7:  while cand ≠ ∅ do
8:    let n ∈ cand; remove n from cand
9:    insert(target,n)
10: for each org ∈ done do
11:   eliminate target at org
12: for each vin ∈ ninsert do
13:   insert target into the entry of vin
14: for each vout ∈ xinsert do
15:   insert target into the exit of vout
```

DDPDE is a code optimization technique that is applied to each occurrence of assignments. Hence, it is possible to perform exhaustive PDE that reflects the second-order effects by appropriately applying DDPDE in a sequence for all occurrences of all statements in the program.

The four second-order effects described earlier can be reflected by sequentially applying the process from the assignment closest to the end node. For example, it is possible to apply DDPDE by visiting nodes in a *reverse postorder* for a reverse CFG, in which edges are reversed. In the example shown in Fig. 4 (a), we apply DDPDE in the reverse postorder sequence of nodes 10, 9, 8, 6, 7, 5, 3, 4, 2, and 1 for a reverse CFG. First, the assignment `x = e1` at node 8 is inserted at the exit of node 8. With regard to the assignments `y = x + 1` at node 6, no insertion is performed because the assignment is dead at the entry of node 8 although it is blocked at this node. Next, regarding the assignment `x = a + b` at node 3, no insertion is performed because the assignment is dead at the entry of node 8, although it is blocked at this node. In contrast, in terms of code sinking from node 7 to node 9, the assignment `x = a + b` is inserted at the exit point of node 7, because upward safety at the entry of node 9 is not ensured. The assignment `x = a + b` at nodes 2 and 3 is processed simultaneously, yielding the same insertion point. After eliminating occurrences of the original assignments, the result shown in Fig. 4 (b) is obtained.

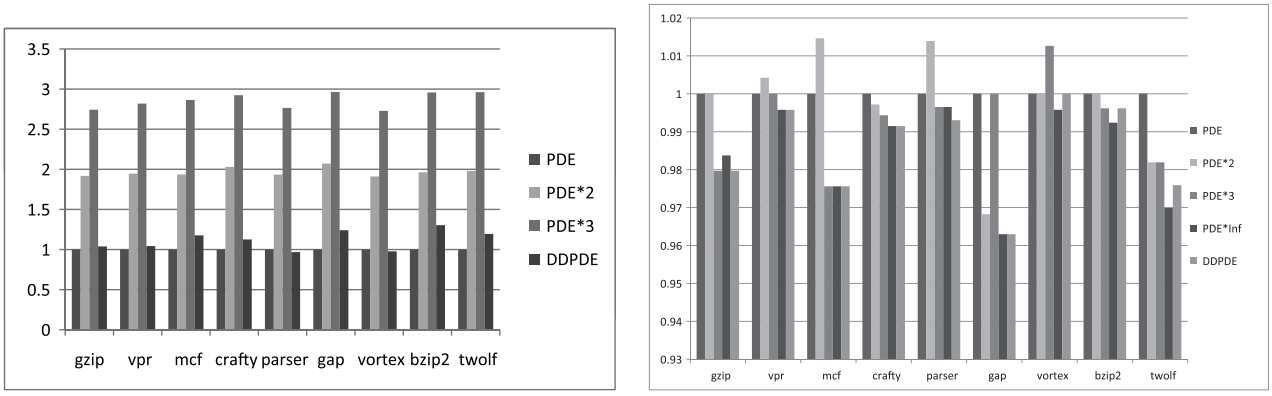
6. Evaluation

To exhibit the effectiveness of our technique, we compared its optimization cost and execution efficiency with those of the traditional method (PDE) by using a benchmark.



(a) Original code (b) Sinking of the first assignment (c) Sinking of the second assignment

Fig. 8 Repeated application of *insert*.



(a) Ratio of analysis cost for PDE

(b) Ratio of execution cost of target code for PDE

Fig. 9 Efficiency of analyses and executions.

The *COmpiler INfrastructure* (COINS) project [6] C compiler was used for implementation. The COINS compiler was implemented entirely using the Java platform and was run on Java virtual machines. The primitive input code was transformed from a *high-level intermediate representation* to a *low-level intermediate representation* (LIR) and the target code was then generated on the basis of LIR. Both our technique and the traditional methods were implemented as a transformer to generate LIR after transforming an input LIR code.

Nine SPEC CPU2000 benchmark programs (gzip, vpr, mcf, crafty, parser, gap, vortex, bzip2, and twolf) were selected for evaluation and analysis, and the target codes were run on a Solaris 10 platform with a SPARC64-V 2-GHz CPU.

The evaluation was performed by comparing our technique and the traditional method (PDE) applied multiple times.

PDE : Apply PDE once.
PDE*2 : Apply PDE twice.
PDE*3 : Apply PDE three times.
PDE*Inf : Apply PDE until the process yields the same result.
DDPDE : Apply the proposed technique.

The dataflow analysis of traditional PDE was implemented using a worklist algorithm called the word-wise method [10], which controls the dataflow using a word-size bit vector. The word-wise method is known to be efficient because it restricts the dataflow and the propagation range to be calculated while maintaining the parallel processing of a bit operation in units of words. In addition, DDPDE was applied to all assignments in the reverse postorder for the reverse CFG. The demand-driven dataflow analysis was implemented utilizing the caching technique described earlier.

Figure 9 (a) and (b) show the ratio of optimization costs and target code execution costs, respectively, for PDE. For clarity, PDE*Inf is omitted in Fig. 9 (a) because its value was too large. The number of times that it was necessary to apply PDE in PDE*Inf was 13 for gzip, 17 for vpr, 11 for mcf, 15 for crafty, 11 for parser, 20 for gap, 11 for vortex, 13 for bzip2, and 21 for twolf. A comparison of PDE, PDE*2, and PDE*3 in Fig. 9 (a) shows that the optimization cost increases proportionally. In contrast, the optimization cost of DDPDE ranged from 0.967 to 1.303 of PDE and averaged 1.118. As a result, it can be concluded that the optimization cost of our technique is approximately the same as that of a single application of PDE.

In contrast, as shown in Fig. 9 (b), the optimization effect of DDPDE was better than PDE, with the exception of vortex. In the case of vortex, because the optimization effect of PDE*3 is less that of PDE (and even more so of DDPDE), it is assumed that the problem pertaining to incomplete application was caused by many statements with a circular dependency, which is difficult to handle in DDPDE. Compared with PDE*Inf, consistent results were obtained for the four programs of vpr, mcf, crafty, and gap. Although the results for gzip and parser with DDPDE were better than those with PDE*Inf, it is assumed that these cases are examples in which the register allocation provided better results despite the inferior optimization effect of PDE*Inf.

It is worth noting that increasing the number of PDE applications from PDE and PDE*2 to PDE*3 does not guarantee a better optimization effect, and in some instances, it may even worsen the effect. This result also indicates that DDPDE, which is capable of reflecting many second-order effects at a low cost, is significant in obtaining a stable optimization effect.

7. Related Works

The significance of eliminating partially dead assignments has been reported by Feigen, L. et al. [9]. According to the technique proposed by Feigen, L. et al., the assignment s is moved to a point in the code at which the target variable in s is used more often for all paths on which s is executed, and if the motion of the statement is blocked, the entire branching structure in the CFG is moved. As a result, there is a danger that the program structure could be altered. Also, because the statement is moved only to a point in the program, it cannot eliminate certain partially dead assignments and is ineffective in moving codes out of loops. In response to these problems, Knoop, J. et al. proposed a technique to generalize the technique of Feigen, L. et al. without altering the control structure while including the effect of moving codes out of loops [13]. Because the proposed technique of Knoop, J. et al. must be applied repeatedly based on dataflow analyses, the computation cost is estimated to increase in direct proportion to the fourth power of the program size in the worst case, and if rational assumptions are made, in direct proportion to the square of the program size.

Based on the assumption that the control structure may be al-

tered, Bodik, R. et al. proposed a technique to generalize the technique of Feigen, L. et al. [3]. They implemented a method to effectively achieve PDE by using the program slicing technique. However, its computational cost is exponential to program size.

Takimoto, M. et al. proposed a technique to enable as much PDE as possible without altering the control structure [15]. Although the computational cost of this technique is directly proportional to the square of the program size, the graph representation called an extended value graph, which is used in the process of transforming the program, is complicated, and the algorithm used is also complex.

As was the case with the technique of Knoop, J. et al., the technique proposed in this paper assumes that the control structure will not be altered. The computational cost of this technique for the analysis of one assignment is directly proportional to the square of the program size if the caching technique is not used in the demand-driven dataflow analysis and is in direct proportion to the program size if the caching technique is utilized. If this technique is applied to the entire program, the computational cost will be directly proportional to the third power of the program size if the caching technique is not used and the square of the program size if the caching technique is used. In practice, the computational cost of this technique has been proven to be equivalent to that of simple dataflow analyses.

Because the demand-driven nature of this technique enables its selective application for occurrences of particular assignments, it can be applied only to the portion in which the optimization effect can be expected.

In addition, PDE for some techniques based on the *static single assignment* (SSA) form [4], [5] is obtained as a secondary effect of code sinking, which is performed to prevent excessive code hoisting when eliminating redundant expressions. However, PDE is applied restrictively until the ϕ-function is encountered in the SSA form, and for this reason, its code sinking range is limited.

8. Conclusions

This paper proposes a DDPDE technique suitable for application to different assignment. This technique makes it possible to reflect many second-order effects of PDE by applying it in sequence from an assignment closest to the end point of the program.

To evaluate effectiveness of this technique, experiments were performed by implementing it as an optimization component of the COINS infrastructure. The result shows that this technique improves the optimization effect while reducing the optimization cost.

We believe that the combination of code sinking and demand-driven tests proposed in this paper provides a guideline for modifying conventional techniques, for which an exhaustive analysis was necessary, into a demand-driven type.

Acknowledgments This work is supported in part by Grants-in-Aid for Scientific Research No.22300007 and No.22500034.

Reference

[1] Aho, A.V., Lam, M.S., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools Second Edition*, Addison Wesley (2007).
[2] Appel, A.W.: *Modern Compiler Implementation in ML*, Cambridge University Press (1998).
[3] Bodik, R. and Gupta, R.: Partial Dead Code Elimination using Slicing Transformations, *Proc. Programming Language Design and Implementation* (*PLDI'97*), pp.159–170, ACM (1997).
[4] Briggs, P. and Cooper, K.D.: Effective Partial Redundancy Elimination, *Proc. Programming Language Design and Implementation* (*PLDI'94*), pp.159–170, ACM (1994).
[5] Click, C.: Global Code Motion Global Value Numbering, *Proc. Programming Language Design and Implementation* (*PLDI'95*), pp.246–257, ACM (1995).
[6] COINS: COmpiler INfra Structure, available from ⟨http://www.coins-project.org/international/⟩.
[7] Dhamdhere, D.M., Rosen, B.K. and Zadeck, F.K.: How to Analyze Large Programs Efficiently and Informatively, *Proc. Programming Language Design and Implementation* (*PLDI'92*), pp.212–223, ACM (1992).
[8] Duesterwald, E., Gupta, R. and Soffa, M.L.: A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis, *ACM Trans. Prog. Lang. Syst.*, Vol.19, No.6, pp.992–1030 (1997).
[9] Feigen, L., Klappholz, D., Casazza, R. and Xue, X.: The Revival Transformation, *Proc. Principles of Programming Languages* (*POPL'94*), pp.421–434, ACM (1994).
[10] Khedker, U.P. and Dhamdhere, D.M.: A Generalized Theory of Bit Vector Data Flow Analysis, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.5, pp.1472–1511 (1994).
[11] Knoop, J., Rüthing, O. and Steffen, B.: Lazy Code Motion, *Proc. Programming Language Design and Implementation* (*PLDI'92*), pp.224–234, ACM (1992).
[12] Knoop, J., Rüthing, O. and Steffen, B.: Optimal Code Motion: Theory and Practice, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.4, pp.1117–1155 (1994).
[13] Knoop, J., Rüthing, O. and Steffen, B.: Partial Dead Code Elmination, *Proc. Programming Language Design and Implementation* (*PLDI'94*), pp.147–158, ACM (1994).
[14] Takimoto, M. and Harada, K.: Effective Partial Redundancy Elimination based on Extended Value Graph, *Information Processing Society of Japan*, Vol.38, No.11, pp.2237–2250 (1997).
[15] Takimoto, M. and Harada, K.: Partial Dead Code Elimination Using Extended Value Graph, *Proc. Int. Conf. Static Analysis Symposium* (*SAS'99*), *Vol.1694 of LNCS*, Venice, Springer-Verlag (1999).

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