Partial Dead Code Elimination using Slicing Transformations

Rastislav Bodík Rajiv Gupta

Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{bodik,gupta}@cs.pitt.edu

Abstract

We present an approach for optimizing programs that uncovers additional opportunities for optimization of a statement by *predicating* the statement. In this paper predication algorithms for achieving partial dead code elimination (PDE) are presented. The process of predication embeds a statement in a control flow structure such that the statement is executed only if the execution follows a path along which the value computed by the statement is live. The control flow restructuring performed to achieve predication is expressed through *slicing transformations*. This approach achieves PDE that is not realizable by existing algorithms. We prove that our algorithm never increases the operation count along any path, and that for acyclic code all partially dead statements are eliminated. The slicing transformation that achieves predication introduces into the program additional conditional branches. These branches are eliminated in a branch deletion step based upon code duplication. We also show how PDE can be used by acyclic schedulers for VLIW processors to reduce critical path lengths along frequently executed paths.

Keywords: partial dead code elimination, program slicing, program restructuring, path-sensitive optimization.

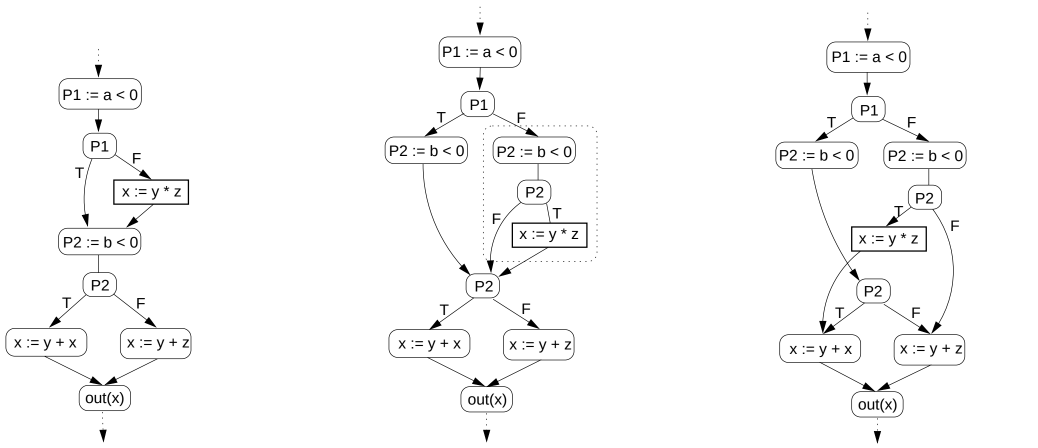
1 Introduction

We present a novel program transformation that *predicates* the execution of statements for improving the quality of generated code. The predication of a statement is a process that embeds the statement in a control flow graph to reduce the number of paths along which the statement executes, thus optimizing the program. In this paper we use predication for carrying out *partial dead code elimination (PDE)*. An assignment is partially dead if there is a path from the assignment statement to the end of the program along which the value computed by the assignment is not used. PDE through predication is achieved by embedding a partially dead statement in a *predicate flow graph* which ensures that the execution of the assignment is triggered at run time only if the program follows a path along which the value computed by the statement is live. The control flow restructuring required by the above optimization is achieved through *slicing transformations*.

The algorithm we present ensures that predication does not increase the operation count along any path as a result of PDE. Existing PDE techniques based upon code-motion reduce deadness by sinking partially dead statements to program points such that the number of paths along which the value computed by the statement is dead is reduced [2, 8, 13, 17]. Predication improves upon these existing algorithms because it is able to achieve PDE of code statements that are not sinkable. For acyclic graphs our approach generates code containing no partially dead code, which is a result that existing PDE algorithms cannot achieve. If a partially dead statement is nested inside a loop, existing techniques can only achieve PDE if sinking of the statement to the point following the loop is possible. Our approach can reduce deadness even if this sinking is not possible. PDE is an important optimization for VLIW architectures in which critical path lengths along frequently executed paths can be reduced through PDE [9, 10]. We show how our PDE optimization can improve effectiveness of hyperblock schedulers [6, 16] by moving operations that are dead on the dominant path off the scheduling region.

Insertion of the predicate flow graph into the program control flow graph introduces into some paths additional assignments and conditional branches. The *assignment statements* in the predicate graph are hoisted from other parts of the program under their original execution conditions and thus do not increase the operation count along any path. To ensure that the number of *conditional branches* that are executed along any path through the program remains the same, the predication of statements is followed by a conditional branch deletion phase. The branches introduced during the predication of a statement are copies of later branches and hence correlated to those branches. They can be eliminated through code duplication and control flow graph restructuring techniques [1, 15].

We are aware of four existing algorithms for PDE. Knoop, Rüthing, and Steffen [13] present an optimal PDE algorithm based on sinking partially dead assignments closer to their uses. Their algorithm can eliminate all partial deadness that can be removed without changing the branching structure of the program. Briggs and Cooper [2] develop an algorithm for removing partial redundancies. Their algorithm may remove some partially dead code as well but some execution paths may be impaired by sinking partially dead assignments into loops. Feigen *et al.* [8] describe an algorithm that modifies program branching structure by moving a partially dead assignment to the point of the use together with the statements that guarantee preservation of the original program semantics. The algorithm has several serious restrictions. For example, it cannot move statements out of loops or across loops. There is some similarity between their and our approach to a more aggressive PDE optimization.



(a) Unoptimized code. (b) Predicating statement for PDE. (c) Complete PDE (after branch elimination).

Figure 1: PDE through statement predication.

While they wrap the partially dead assignment into its execution condition and move it to the point of the use, we leave the assignment in place but embed it into the execution condition that characterizes the paths along which the assignment is live. Finally, Steffen [17] describes a PDE algorithm based on code restructuring that is capable of eliminating all partially dead statements, at the cost of introducing nondeterminism into the program. Nondeterministic branches are introduced even when acyclic programs are optimized. Since the program must be executed with backtracking, it may impair many execution paths.

Next, we briefly illustrate the effectiveness of our approach through an example. In Figure 1(a), the statement x:=y*z is partially dead since the value of x computed by the statement is never used if P2 is false. This deadness cannot be eliminated through code sinking techniques (the code-restructuring algorithm in [8] can handle this particular case). As shown in Figure 1(b), by predicating the execution of x:=y*z with P2, PDE can be achieved. In order to ensure that the value of predicate P2 is available before the statement, we also hoist the statement P2:=b<0. As shown in Figure 1(c), the conditional branch introduced by predication can be eliminated by restructuring the flow graph. In this graph, the operation count along each execution path has been minimized.

From the above example it should be clear that the objective of predication is to construct a flow graph structure into which a statement can be embedded to eliminate partial deadness in the statement. In the remainder of this paper we develop algorithms for PDE in acyclic code, PDE in cyclic code, and present the application of our PDE algorithm to VLIW acyclic schedulers.

2 PDE in Acyclic Code

In this section we present the algorithm for applying PDE to acyclic code segments. Our algorithms operate upon a *control flow graph* $G = (N, E, start, end)$ of the program in the *static single assignment (SSA)* form [4]. In this form, a) each program variable has a single definition and b) each use of a variable is reached by exactly one definition. The SSA property is achieved by renaming each original program variable x to a unique variable x_i at each static definition of x, and by introducing a *ϕ-assignment* $x_k := \phi(x_i, x_j)$ at control flow merge nodes that are reached by definitions of distinct variables x_i and x_j.

We use the SSA form because it eliminates all output- and anti-dependences, which allows us to freely reorder statements during the slicing transformations that achieve PDE. All ϕ-assignments are eventually eliminated in our algorithm without increasing the original operation count along any path in G. Since ϕ-assignments exist in the program only temporarily, they are not the target of PDE. Consequently, the live and dead conditions of each definition are computed transitively across a sequence of ϕ-assignments, as given by the following recursive definition. A variable x is *used along path* p if it appears along p in the right-hand side of either a non-ϕ-assignment or a ϕ-assignment whose left-hand side variable is used along path p. For the purposes of our algorithm, referencing a variable in a ϕ-assignment thus does not constitute a use. A definition node d is *live (dead) along path* p from d to *end* if the value computed by d is used (is not used) along p. A definition d is *definitely live (dead)* if it is live (dead) along each path through d.

In this section we first present the slicing transformation upon which the PDE algorithm is based. Next, we derive important properties of our algorithm. Finally, we describe an implementation of the PDE algorithm and analyze its complexity.

2.1 Slicing Transformations

The PDE algorithm eliminates partial deadness of a single assignment node d at a time. First, we identify a multi-entry, multi-exit region that characterizes the usage pattern of the value computed by d. This region is called the *use region* and has the following properties: 1) d is one of the region entries, 2) the value computed by d is not used in the region, and 3) the value computed by d is either definitely live or definitely dead at each exit. PDE is achieved through

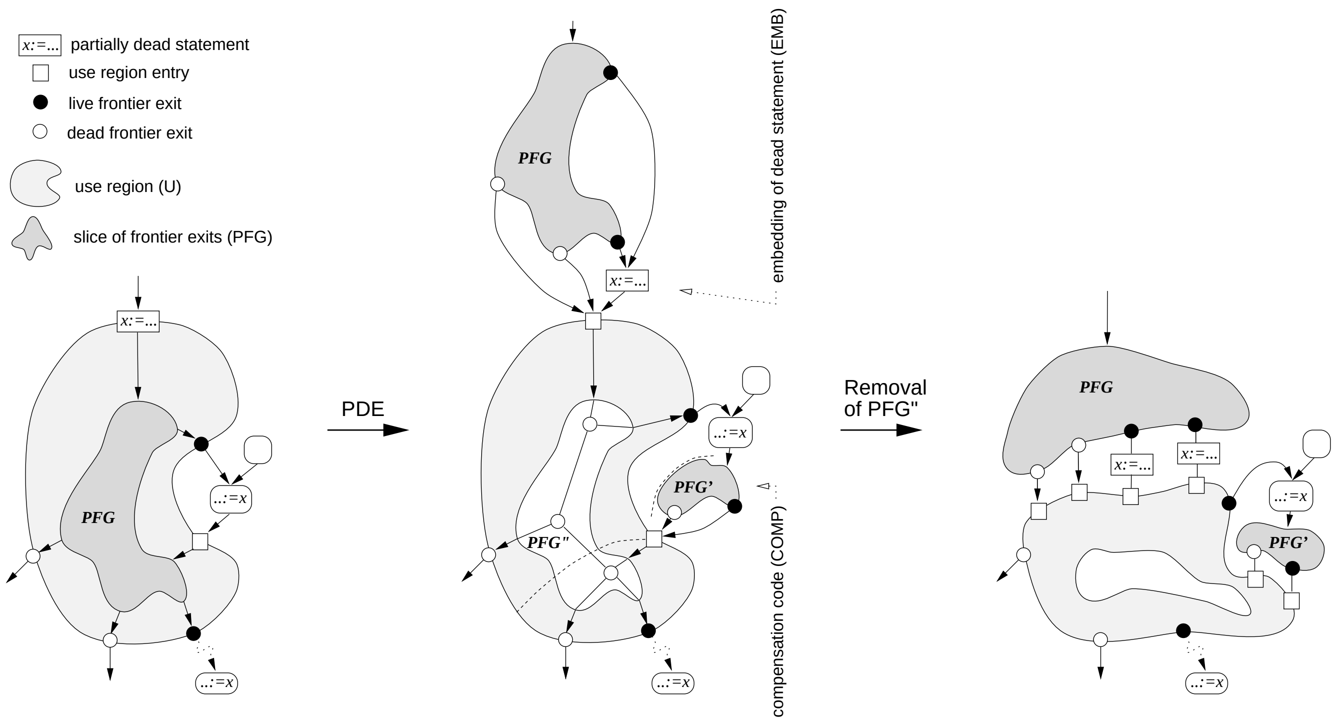


Figure 2: PDE optimization through slicing transformation.

a slicing transformation of the use region.

Definition 1. Let d be a node with an assignment to variable x. The **use region** of d, denoted $U(d)$, is a multi-entry multi-exit region of the program control flow graph with the following entry and exit edges:

- (d, v) is a *main entry edge* to $U(d)$. We assume that there is a single main entry edge and that $d \neq v$.
- (u, v) is a *live frontier exit edge* from $U(d)$ if 1) there is a path from d to u without a use of x, 2) x is definitely live at v, and 3) there is a path from u along which x is dead.
- (u, v) is a *dead frontier exit edge* from $U(d)$ if 1) there is a path from d to u without a use of x, 2) x is definitely dead at v, and 3) there is a path from u along which x is live.
- (u, v) is a *side entry edge* to $U(d)$ if 1) each path from d to u contains a frontier exit edge of d and 2) there is a path from d to v along which there is no frontier exit edge of d.

A path p from the sink node of an entry edge to the source node of an exit edge is called a *region path* if it does not contain any exit edge. The use region $U(d)$ contains exactly those nodes that lie on some region path of $U(d)$. □

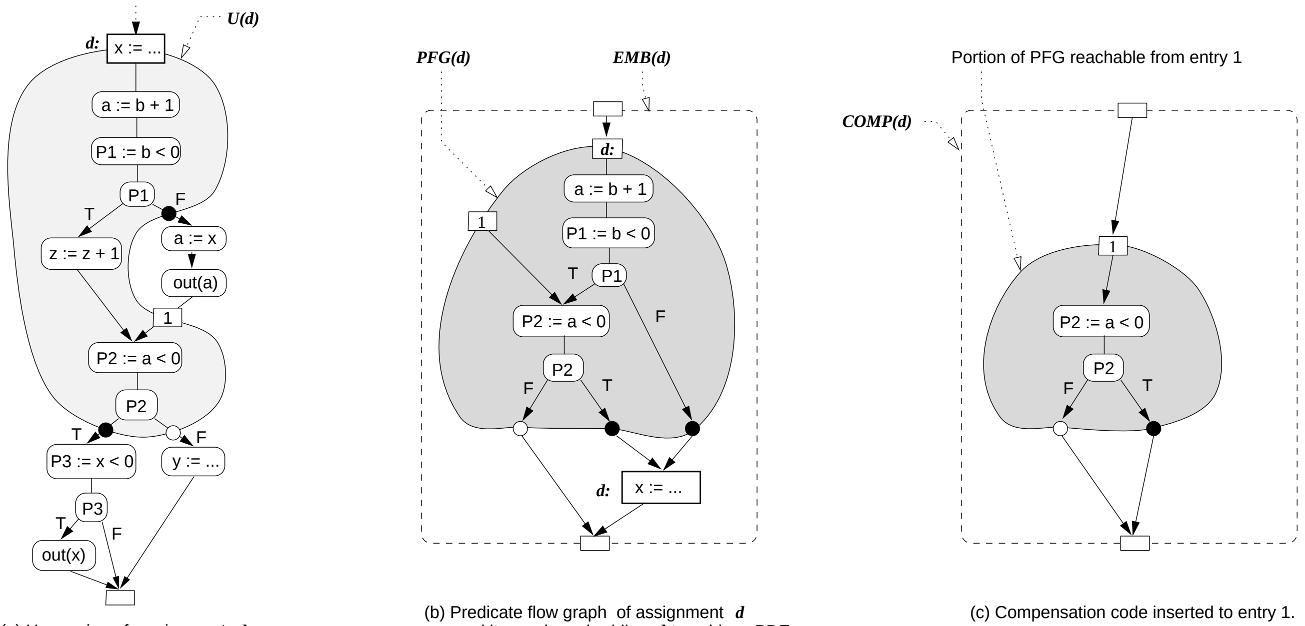
It should be noted that each frontier exit edge emanates from a node with multiple exits, that is, a conditional branch node or a case statement.

Our PDE algorithm is based upon embedding the partially dead assignment in a flow graph, called a *predicate flow graph (PFG)*, that captures execution conditions under which the assignment is definitely live. The embedding ensures that the assignment is triggered only when the control is going to follow a live path. The PFG is computed through *slicing* [19] of the use region. The *backward slice* of a program with respect to an edge is the set of statements that determine whether the edge is executed.

Definition 2. Let d be an assignment node and $U(d)$ its use region. The backward slice of $U(d)$ with respect to its live and dead frontier exit edges is called **predicate flow graph** of d, denoted as $PFG(d)$. □

Typically, the slice is computed with respect to the entire procedure or program. However, $PFG(d)$ is a subgraph of $U(d)$; it has the same set of entries and exits and contains a subset of its statements. The PFG is computed by first including into the slice the source nodes of the exit edges and then taking the closure over control and flow data dependences of the statements already in the slice, in the backward direction.

The goal of embedding d in $PFG(d)$ is to suppress deadness in d. We form a single-entry, single-exit region $EMB(d)$ that will substitute d in the control flow graph. The embedding graph $EMB(d)$ is created by adding to $PFG(d)$ the assignment node d and a new node e which will serves as the single exit node. The main entry of $PFG(d)$ serves as the entry of $EMB(d)$. The exit node e connects all dead frontier exits and and embeds d in the live frontier exit edges as shown in Figure 2 and Figure 3(b). In the program resulting



(a) Use region of assignment *d*

(b) Predicate flow graph of assignment *d* and its use in embedding *d* to achieve PDE.

(c) Compensation code inserted to entry 1.

Figure 3: An example of the use region and the predicate flow graph.

after d is replaced by $EMB(d)$, definition d is executed only if the control leaves the use region through a live frontier; otherwise it is not executed. Thus, partial deadness of d is entirely eliminated.

In the presented algorithm, the embedding and substitution of d modifies the use region. All assignment nodes that are included in $PFG(d)$ are removed (hoisted) from their original positions in $U(d)$. The conditional branches forming the skeleton of the hoisted control flow structure, denoted by PFG'' in Figure 2, remain and are removed by a subsequent branch elimination step. To preserve program semantics, all side entry edges of the region must be replaced by compensation code, denoted by PFG' in Figure 2, which is also computed with a slicing transformation as follows. First, the subgraph of the use region that is reachable from the side entry is identified. The intersection of this subgraph with $PFG(d)$ yields the desired compensation code. The compensation flow graph $COMP_s(d)$ for side entry s is constructed in a manner similar to $EMB(d)$ by using the side entry of $PFG(d)$ that correspond to s and connecting all exits into a single exit node. In Figure 2, a pictorial representation of the compensation code transformation is given. The dashed line delimits the reachable part of the use region. Finally, the branch deletion step eliminates the correlated branches in PFG''.

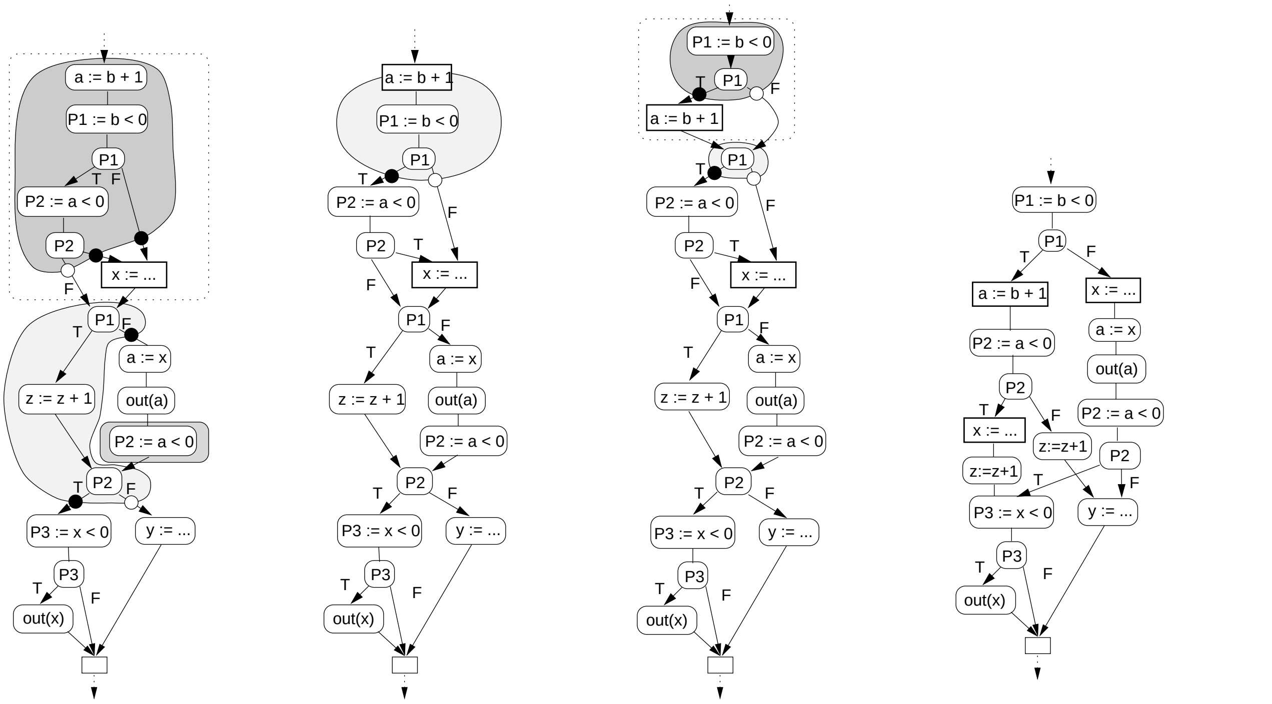
Next, we illustrate the algorithm by performing PDE on the acyclic program in Figure 3(a). First, we eliminate partial deadness present in the assignment d: x:=.... The use region of d has two entries and three exits. Figure 3(b) shows $PFG(d)$ and how the embedding graph $EMB(d)$ is constructed. To achieve PDE, the live frontier exits execute d and the dead frontier bypass d. To construct the embedding graph for d, the main entry is used as the entry to the embedding graph. The entire flow graph delimited by the dashed line is what replaces node d in the original program. Figure 3(c) shows the flow graph of the compensation code that is inserted to use region entry 1. Note that the graph in Figure 3(c) contains a spurious conditional branch P2 which our algorithm removes during the construction of the compensation graph. The resulting program where d is fully live is shown in Figure 4(a); the assignments that belong to the PFG are hoisted from the use region but the conditional branch elimination has not been performed yet. We can repeat the same process to eliminate partial deadness in the assignment a:=b+1. Its use region is shown in Figure 4(b) and the program after the embedding graph for this assignment is inserted is in Figure 4(c). Finally, the program after conditional branches are removed is shown in Figure 4(d). The cost of branch deletion is code duplication. In Section 4 we present a method for reducing the impact of code duplication.

2.2 Correctness, Safety, Termination

In this section we demonstrate the correctness and effectiveness of our PDE technique. We first show that the transformations performed by the PDE algorithm preserves program semantics (Theorem 3) and is always feasible (Theorem 4). Next we show that our approach is *safe* in that it does not increase the number of operations along any program path (Theorem 7). Finally, we show that repeated application of our algorithm to individual statements removes all partial deadness from the acyclic code segments and that the algorithm always terminates (Theorem 8).

It can be shown that each path from an assignment to the end of the program contains a live or dead frontier edge. The following theorem proves that the frontiers can navigate the predication of a partially dead assignment.

Theorem 3 (Correctness/Revival). Let p be a path from



(a) After PDE of "x := ...". (b) Use region of "a:=b+1". (c) After PDE of "a:=b+1". (d) After branch elimination.

Figure 4: PDE of the program in Figure 3(a).

start to *end* and d an assignment node from p. Executing an assignment d when it is followed on path p by a live frontier and suppressing d when it is followed on p by a dead frontier results in: a) a semantically equivalent program with b) the assignment d being definitely live.[1]

Proof. (by contradiction) [a] Assume there is a path p along which not executing d alters the program behavior because the missing value of d is needed (that is, used). By Definition 1, d cannot be followed on p by a dead frontier. Hence, d must be followed on p by a live frontier which means that d must have been executed.
[b] Assume there is a path p along which executing d produces a value that is not subsequently used. By Definition 1, d cannot be followed on p by a live frontier. Hence d must be followed by a dead frontier which means that d must not have been executed. □

Theorem 4 (Hoistability). Given an assignment d, the predicate flow graph $PFG(d)$ can be fully evaluated a) in all entries of $U(d)$ and b) prior to d.

Proof. [a] The hoisting of $PFG(d)$ to any entry is permissible because all flow data dependences on the moved assignments are satisfied by hoisting the entire slice. No output- or anti-dependences exits in the SSA form to prevent the code motion. Since $PFG(d)$ is acyclic, static single assignment guarantees that the values computed by the hoisted assignments are available at their respective uses.
[b] (by contradiction) Assume that the evaluation of $PFG(d)$ prior to d is prevented by a flow data dependence between d and any statement in $PFG(d)$. Then, d must belong to the slice of the frontier edges, which means it is used in $U(d)$, a contradiction with Definition 1. □

The insertion of the PFG into use region entries introduces additional statements along some paths. Next, we show that when the hoisted statements are removed from their original positions and the conditional branches are removed after PDE is performed, then our algorithm does not increase the operation count along any path. Lemma 5 shows that the assignments and conditional branches that are hoisted within the PFG can be removed from the use region. Lemma 6 relates to a use region with multiple entries that are not mutually exclusive, such as those in Figure 3(a). The claim of the lemma is that a statement that has been hoisted to more than one entry will execute in at most one entry, for any program input.

Lemma 5. Each statement s from $PFG(d)$ can be removed from $U(d)$ after hoisting.

Proof. Let s denote the assignment in $U(d)$ and s' the copy of s hoisted within $PFG(d)$. Consider a program execution path p from *start* to s. Since s is within the use region, p must include at least one use region entry. Let e be the last entry of $U(d)$ on p. Because the algorithm inserts $PFG(d)$ into e using the PFG entry that corresponds to e, when path p is taken by the program, s' must be executed. If s is an assignment, then due to SSA form, the value of the hoisted assignment is available at the point of the original assignment which is, therefore, redundant and can be removed. If

[1] Except that the exceptions that may be raised by d are removed from some paths.

s is a conditional branch, then due to SSA form, the copy and the original branches are correlated along all paths and the original branch can be removed using branch elimination algorithm [1, 15]. □

Lemma 6. After PDE of an assignment d has been performed, each statement s executes no more frequently than in the original program, for any program input.

Proof. If s does not belong to the PFG of d then its control condition is unchanged by the slicing transformation. If s is the assignment d, then its control condition is more strict than in the original program. If s has been hoisted within a slice, it executes under the same control condition as in the original program. Since Lemma 5 shows that each hoisted statement can be removed from its original position, it remains to be shown that if any two use region entries e_1 and e_2 are on the same path and the PFGs inserted in the two entries contain a copy of the same original assignment s, then the two copies of s cannot both execute. If there is a path p that contains both region entries e_1 and e_2, then some frontier exit edge f must lie on p between e_1 and e_2. Let the frontier f be an out-edge of a conditional branch node with predicate P. Assume that f is the true-exit. Then, the copy of s in PFG at e_1 executes only when P is false and the copy of s in PFG at e_2 executes only when P is true. Thus, for any two copies of a statement hoisted to a program path, there is always a predicate that makes their execution mutually exclusive. (e.g., the two copies of assignment P2:=a<0 in Figure 4(a) are made mutually exclusive by predicate P1.) □

Finally, we want to show that our optimization guarantees *safety*, which means that the operation count along any program path is not increased as a result of the optimization. To prove safety, we need to show that it is possible to remove all ϕ-assignments without introducing assignments to any temporary variables. First, the algorithm attempts to rename the source variable of each ϕ-assignment to the destination variable of the assignment, if such renaming is possible. Remaining ϕ-assignments are removed by an algorithm derived from the optimization framework in [17].[2] The algorithm is based on path duplication and works as follows. Consider a ϕ-assignment a and a set of nodes that use the variable assigned in a. The algorithm creates a separate path for each source variable of the ϕ-assignment. Since only one variable reaches the use along each such path, renaming can be performed and the ϕ-assignment eliminated. Elimination of ϕ-assignments is illustrated in Figure 5. Note that the multiple definitions of x3 exist because some ϕ-assignments have already been eliminated in the program.

Theorem 7 (Safety). Our PDE algorithm does not increase operation count for any program input.

Proof. Follows from Lemma 5, Lemma 6, and the fact that all SSA-assignments are removed. □

All concepts presented above relate to the optimization of a single assignment. Embedding a partially dead assignment may cause partial deadness in statements that compute values used by the assignment. Therefore, our algorithm repeats PDE for all partially dead statements in a reverse topological sort order traversal of the data dependence graph. To eliminate deadness of each assignment in the program, each assignment d (and each of its copies) needs to be optimized at most once. It is because after the optimization of d, the execution condition of d that guarantees definite liveness is unchanged by our algorithm, and so is the execution condition of all consumers of the value generated by d. The following theorem shows that such repeated applications of PDE terminates. The proof seeks to establish that, given an acyclic program, it is not possible to perform infinitely many applications of PDE such that each creates a new copy of some partially dead assignment. If this was possible, PDE would not terminate because some partially dead assignments would always exist.

Theorem 8 (Termination). Let D be a directed acyclic graph whose edges are the flow data dependences between the statements in the program. If our PDE algorithm is applied on a statement d only when no descendant of d in D is partially dead, then the PDE process terminates.

Proof. Theorem 7 implies that the operation count on any path does not increase after PDE of an assignment. Therefore, the number of conditional branch nodes on any path through the optimized acyclic program is bounded during the entire execution of the PDE algorithm by the height of the original program. It follows that the number of unique paths in such a program, after any program restructuring, is bounded by a constant. Let us assume that the PDE algorithm creates infinitely many new assignments. Given the finite number of paths, the operation count along some path would have to increase beyond its original value, contradicting the Safety Theorem 7. □

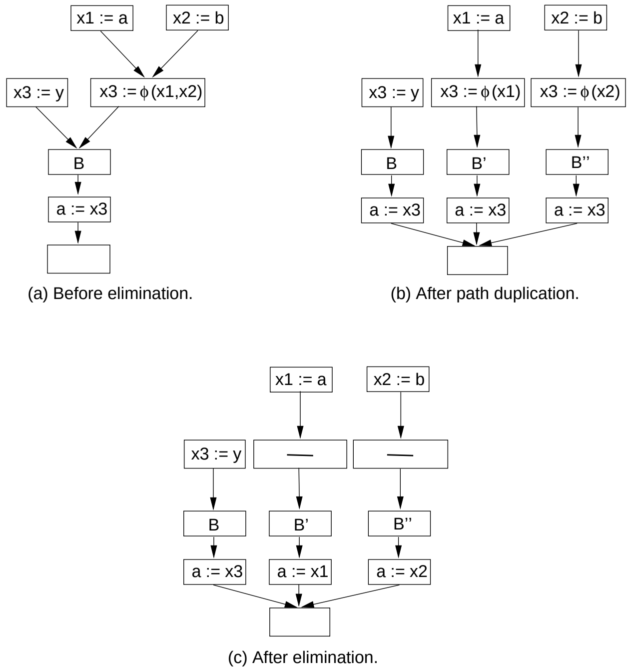


Figure 5: Elimination of ϕ-assignments.

[2] While the PDE algorithm in [17] produces nondeterministic programs, the ϕ-assignment elimination algorithm does not.

2.3 Implementation and Complexity

So far we have provided a conceptual presentation of our approach. A detailed implementation of the acyclic algorithm is presented in Figures 6 and 7. For simplicity, we assume that a node in the control flow graph cannot be both a conditional node and a merge node. Procedure *PDE* considers one partially dead assignment at a time and can eliminate partial deadness either completely or only for the desired partially dead assignments (e.g., those on the frequently executed paths). In either case, since application of PDE to a statement may uncover deadness of another assignment, assignments are selected for optimization in a reverse topological sort order of the data dependence graph so that PDE of each statement is performed at most once.

First, procedure *PDE* determines the use region by solving four data flow problems and identifying the entries and frontiers (Figure 6). Note that the four problems are not bit-vector (or *separable*) data flow problems because a ϕ-assignment $x_i := \phi(x_j, x_k)$, which is not treated in our PDE as a definition or a use, transfers data flow facts from the variable x_i to variables x_j and x_k, and vice versa. Second, *compute_PFG*(d) constructs for each entry of the use region the embedding graph in a way that it contains only those nodes from $PFG(d)$ that are reachable from the entry (see Figure 3(c) for an example of unreachable nodes). Thus, no additional elimination of unreachable nodes in the embedding graphs is necessary. Next, the assignment d is embedded in its PFG, and the PFGs representing compensation code are introduced into all other entries. Since no node is embedded into the latter PFGs, the conditional branches at the exits of the PFGs may be spurious (i.e, both exits may point to the same target) and are removed. Finally, the redundant conditional branches are eliminated and the SSA form that might have been invalidated during the restructuring is updated. After all desired deadness has been eliminated, the ϕ-assignments are removed.

The procedure *compute_PFG*(d) computes PFG for the use region using a backward traversal originating from the frontier edges and terminating at d and other entries. The PFG is gradually built as the traversal progresses, data slice being handled by $flow(n)$ and the control slice by $split(n)$. Procedure $flow(n)$ adds an assignment node n to the root of the propagated PFG if n computes a value used in the PFG. Since n is being hoisted, it can be removed (see Lemma 5). Procedure $split(n)$ checks PFGs propagated along both exits

$Entry[d]$: the set of entries in use region of d
$LF[d]$: the set of life frontiers of d
$DF[d]$: the set of dead frontiers of d

Solve data flow problems:

ANU^d: d is **Available without intercepting use**
    $GEN = \{d\}$, $KILL =$ uses, meet op $= \vee$,
    direction = forward, init value $= \perp$

FL^d: d is **Definitely live**
    $GEN =$ uses, $KILL =$ definitions, meet op $= \wedge$,
    direction = backward, init value $= \perp$

FD^d: d is **Definitely dead**
    $GEN =$ definitions, $KILL =$ uses, meet op $= \wedge$,
    direction = backward, init value $= \top$

ANF^d: d is **Available without intercept. frontier**
    $GEN = \{d\}$, $KILL =$ frontiers of d,
    meet op $= \vee$, direction = forward, init value $= \perp$

Identify the sets:

$$LF[d] = \{(u,v) | ANU^d_{OUT}[u] = \top \wedge FL^d_{IN}[v] = \top \wedge \exists (u,w) : FL^d_{IN}[w] = \perp\}$$
$$DF[d] = \{(u,v) | ANU^d_{OUT}[u] = \top \wedge FD^d_{IN}[v] = \top \wedge \exists (u,w) : FD^d_{IN}[w] = \perp\}$$
$$Entry[d] = \{(u,v) | ANF^d_{OUT}[u] = \perp \wedge ANF^d_{IN}[v] = \top\}$$

Figure 6: Computing the use region of an assignment d.

```
procedure PDE(Program P)
    while partially dead assignments exist do
        pick d such that no def-use
            descendant of d is partially dead
        identify the use region of d (Figure 6)
        compute_PFG(d)
        embed d within PFG^d[(d, succ(d))]
        for each e ∈ Entry[d] do
            remove spurious conditionals in PFG^d[e]
            insert PFG^d[e] into e
        end for
        eliminate redundant conditional branches
        update SSA form
    end while
    eliminate φ-assignments
```

```
procedure compute_PFG(assignment node d)
    initialize PFG with life and dead markers:
    for each e ∈ LF[d] do PFG^d[e] := {LF}
    for each e ∈ DF[d] do PFG^d[e] := {DF}
    for each n from the use region in reverse
        topological sort order do
        if n is a conditional node then split(n)
        else flow(n)
    end for

procedure flow(node n)
    let n be an assignment a := b + c
    let G denote PFG^d[(n, succ(n))]
    if any use of a is upward exposed in G
        add node n to the root of PFG:
        PFG^d[(pred(n), n)] := n → G
        remove n
    end if

procedure split(conditional branch node n)
    let e_t and e_f be the out-edges of n
    if PFG^d[e_t] ≠ PFG^d[e_f] then
        PFG^d[(pred(n), n)] :=
            PFG^d[e_t] ←T− n −F→ PFG^d[e_f]
    else
        PFG^d[(pred(n), n)] := PFG^d[e_t]
    end if
```

Figure 7: Implementation of acyclic PDE algorithm.

of the conditional branch n and, if they are not identical, it adds the branch to the PFG because it is part of the control slice. When the traversal reaches an entry of the use region, the computed graph represents the part of $PFG(d)$ that is reachable from the entry.

Let us consider the complexity of our algorithm. On an acyclic program with N nodes and V SSA variables, the use region is determined by data flow analysis in $O(NV)$ steps and procedure *compute_PFG* also takes $O(NV)$ steps. Elimination of branches and SSA assignments takes $O(NV)$ steps for acyclic programs. The time to eliminate deadness of a single assignment is thus $O(NV)$. Due to hoisting of assignments in the PFG and the subsequent restructuring, the size of the program may grow after PDE of a single variable. Therefore, on arbitrary input programs, complete PDE may take exponential time.

3 PDE in Cyclic Code

In the presence of loops, removal of partially dead code presents more challenges. Given a partially dead assignment d in a cyclic control flow graph G, our approach is able to eliminate d from all dead paths that exist in G prior to application of PDE. However, PDE results in a modified flow graph G' in which new paths may be created. Along these paths, d may be dead after PDE is performed. The assignment d thus becomes definitely live with respect to G, but may be partially dead with respect to G'. The difficulties with PDE in cyclic code can be traced down to two orthogonal problems: a) the cyclic nature of the predicate flow graph and b) a cycle in the graph of data dependences of the partially dead statement. These two problems are treated in the following subsections. In the last subsection we describe our contribution to partial *faint* code elimination.

3.1 Cycle in Predicate Flow Graph

Using the algorithm from Section 2, partial deadness of an assignment can be successfully eliminated along all paths in a cyclic flow graph because the embedding predicate flow graph can be hoisted prior to d as it is done for acyclic programs. Hoisting of the PFG is possible because, even in the cyclic domain, no data dependences in the use region exist that would prevent hoisting the slice of the use region to the use region entries. Therefore, Theorem 4 holds in the case when the program or the use region contain loops. However, not all assignments hoisted in the PFG can be removed from the use region, resulting in increased operation counts on some paths. When a PFG containing a loop is inserted at a use region entry, the assignments on such a loop produce multiple dynamic values that may all be required by statements that remain in the use region. For example, each hoisted loop leaves in the use region the skeleton of conditional branches (denoted PFG'' in Figure 2(b)), which is itself a loop whose statements that compute the loop exit condition have been hoisted in the PFG. To eliminate the skeleton loop, the exit conditions from each iteration of the hoisted loop must be made available, which is only possible through full (that is, infinite) unrolling of the hoisted loop. Consequently, the *safety* of Theorem 7 does not hold in cyclic code. Next, we present a modified PDE algorithm that guarantees safety in cyclic code and we show that the algorithm subsumes the sinking-based PDE algorithm [13].

Our cyclic-code PDE algorithm is based on the observation that $PFG(d)$ represents only the minimal set of use region statements that must be hoisted to predicate d. When the embedding flow graph is defined to contain all statements in the use region, PDE algorithm hoists to the region entries all statements, including the skeleton of all conditional branches. This makes elimination of statements from the use region unnecessary, resulting in a safe program transformation. Naturally, an embedding graph that is a superset of $PFG(d)$ may contain assignments that prevent its hoisting prior to d, because an embedding graph larger than the slice of the frontier exit edges may include statements that are on the def-use chain of d. According to Definition 1, these statements are restricted to ϕ-assignments. The following arguments prove that these ϕ-assignments do not prevent PDE.

The ϕ-assignments that reference the variable assigned by d and prevent hoisting of the entire $U(d)$ exist in the use region because definitions of the corresponding original variable (which was renamed to a unique SSA variable) reach $U(d)$ through the side exits. Obviously, a single-entry use region does not contain any such ϕ-assignments. If we could transform a multi-entry use region into a set of single-entry regions, we could hoist into the entries any superset of the predicate flow graph. Interestingly, the PDE algorithm described in Section 2 performs such transformation. To see that, we have to realize that in each use region entry, only one of the multiple entries of the embedding graph hoisted to that entry is connected. The other entries are unreachable, effectively transforming the embedding graph into a single-entry region in which no ϕ-assignemnts for d are necessary.

The predicate flow graph is thus the minimal subset of statements from the use region that are required for predication of d. The complete use region itself can form the embedding graph and be hoisted into all its entry edges. Since the use region is empty after such hoisting, the exits of the embedding graph can be connected directly to the exits of the empty use region. This version of the PDE algorithm is illustrated in Figure 8. A copy of the use region $U(d)'$ is created and used to embed d. The original use region $U(d)$ serves as a compensation code for the side entries. It should be noted that a single copy of the use region can now serve as compensation code for all side entries.

The PDE algorithm depicted in Figure 8 can be phrased as a sinking transformation. Given a single-entry use region, the assignment d can be sunk to the live frontier edges. Such sinking is not possible with multi-entry use regions because of the ϕ-assignments in the region. However, when a copy of the use region is created and only the main entry is connected, a single-entry region is formed through which d can be sunk to live frontiers. Actually, the PDE algorithm in [13] eliminates deadness of an assignment by sinking to live frontier edges (or further to the uses). Since this algorithm does not restructure the flow graph, PDE is permissible only on single-entry use regions. Our algorithm thus exploits all opportunities for PDE that the sinking algorithm in [13] does.

The cyclic code version of our PDE algorithm is simpler than the algorithm from Section 2 because it eliminates the need to compute the slice (PFG) and perform the subse-

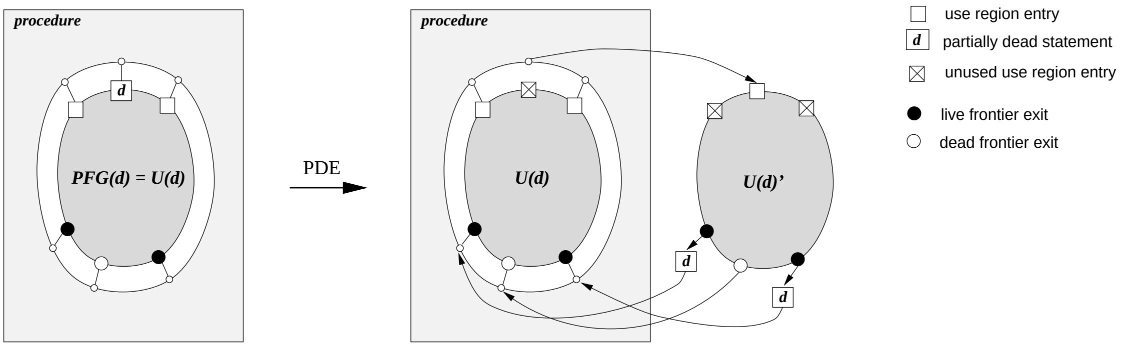


Figure 8: PDE in cyclic code.

quent conditional branch elimination. On the other hand, this algorithm is likely to result in larger code growth. Its simplicity, however, enables us to formulate PDE without the SSA form. Viewing this algorithm as a sinking transformation, the partially dead assignment d is the only statement moved during PDE. It hence suffices to ensure its freedom of motion. Given an assignment d: $x := a + b$, two new temporary variables t_a and t_b are introduced and d is rewritten to $t_a := a; t_b := b; d : x := t_a + t_b$. Since there is a single assignment to each temporary created by PDE, d can be sunk without being restricted by any anti-dependences on its operands. In other words, the temporaries provide d with the values of its operands that were current at d's original location. The temporaries can be viewed as SSA variables created on demand when d needs to be moved. They are treated by PDE the same way as the ϕ-assignments; their assignments are not the target of PDE and, after PDE is complete, they are removed.

3.2 Cycle in Data Dependence Graph

When the partially dead assignment d lies on a cycle of data dependences, PDE removes d from all dead paths that exist in the control flow graph prior to PDE. However, new paths may be created during code restructuring invoked by PDE. The assignment d (or some of its duplicates) may be dead along the new paths. Repeated application of PDE produces an identical scenario, resulting in a non-terminating PDE process. Consider the example in Figure 9. The only dead path for d is the one that leads directly out of the loop. This corresponds to d being dead only in the last iteration of the loop. PDE is performed using the cyclic code algorithm shown in Figure 8: the use region is duplicated with one entry in each copy of the region being connected to the outside program. The live frontier exit of the use region connected at the main entry (on the right in the figure) embeds the partially dead assignment. By tracing the restructured flow graph, it is clear that d was eliminated from the dead path. An alternative explanation is that the assignment d was delayed by one iteration, so that it is executed only if the following iteration is executed. However, repeated application of PDE merely peels off another iteration and leaves d dead along a newly formed path. The PDE process does not terminate because there is always some last iteration in which d is dead. In conclusion, only one dynamic instance of d is eliminated with one application of PDE. While sinking approaches terminate naturally when no new opportunities for optimization exist, our algorithm may never terminate because the restructuring may always create new opportunities for PDE. We solve the non-termination problem by marking optimized assignments and optimizing each assignment at most once.

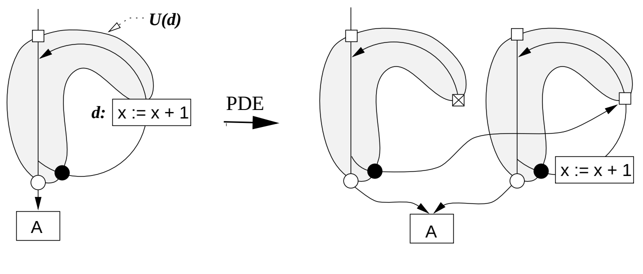


Figure 9: Non-terminating PDE in cyclic code.

3.3 Partial Faint Code Elimination

The non-termination problem can be alleviated by breaking the cycle along which PDE is repeatedly performed. While the data dependence graph cannot be made acyclic, the live condition of the partially dead assignment can be derived *transitively* from program statements that are always live (called *relevant* in [13]), rather than *directly* from assignments that may remain partially dead after each application of PDE. The transitive approach identifies for each assignment a set of *faint* paths [11, 13], which a superset of the *dead* paths. Dead paths are identified by the direct approach, and have been used in this paper so far. Under the faint notion, a definition d is *live along a path* p from d to *end* if a chain of flow data dependences leads along p to a relevant statement (e.g., an output statement or conditional). If d is not not live along p, then it is *faint along* p.

Using faint paths, the slicing approach to PDE allows meaningful definitions of live and dead frontier edges and also permits the computation of their slice for embedding an assignment d. However, the use region as specified in Definition 1 does not exist because d may appear on a (cyclic) faint path multiple times. The definition thus breaks because d cannot be both at the region entry and inside the region. The consequence for *partial faint code elimination (PFE)*

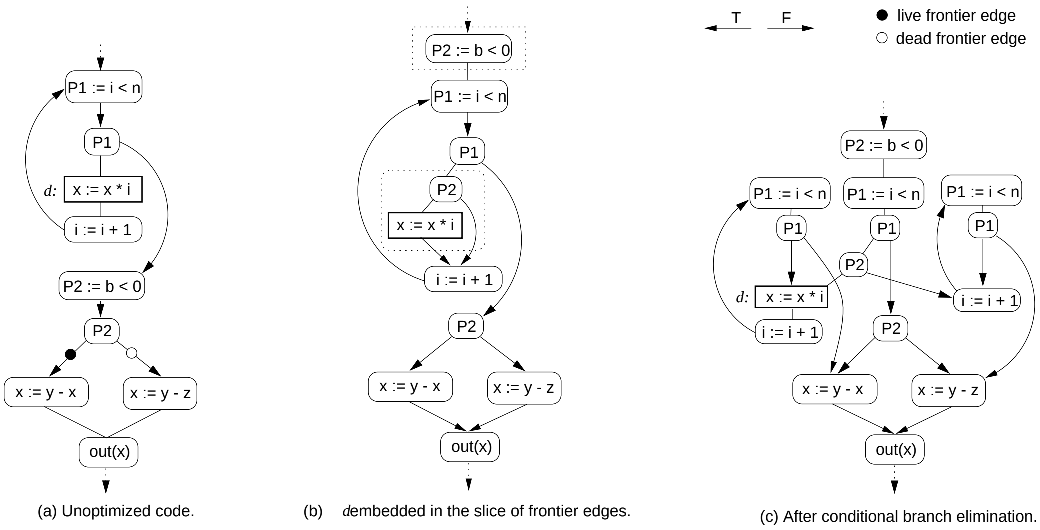


Figure 10: Partial faint code elimination.

using our approach is that d can be legally embedded in the slice of the frontier edges, removing it from all faint paths, but the safety cannot be guaranteed because not all hoisted statements can be eliminated. Consider the example in Figure 10(a). The assignment d: x:=x*i inside the loop is faint when P2 is false. Note that assignment d cannot be moved out of the loop and therefore sinking cannot be employed to achieve PFE. As shown in Figure 10(b), we can eliminate faintness by embedding d in the slice of the frontier edges. The hoisted conditional branch P2 is eliminated by restructuring the flow graph as shown in Figure 10(c). Complete and safe optimization was achieved. However, the assignment to P2 that was hoisted with the predicate flow graph into the loop had to be moved before the loop using loop-invariant code motion. We are not aware of an optimization algorithm that can remove the faintness in Figure 10.

Steffen conjectures in [17] that by code restructuring and branch reordering, all dead code can be eliminated. We have proved in this paper his hypothesis for the case of acyclic code. However, we do not believe that a complete and safe removal of dead or faint code is possible for arbitrary cyclic graphs, using a realistic model of computation. Consider the following program.

```
do {
   a = read();
   p = p + a;
   x = x * a;
} while (a != c1);
if (p != c2)
      out(x);
```

In this program, x is faint when the value of p after the loop terminates equals c2. The predication of the assignment to x within the loop must thus be based on the final value of p, which can only be obtained by executing in sequence two slices of the loop. The first loop computes p and the following loop computes the final value of x. Either the second loop must reissue the input statements (destroying program semantics) or an infinite storage area for the values of the variable a read in the first loop must be created. In either case, a safe optimization does not seem possible.

4 Using PDE in Acyclic VLIW Scheduling

Compilers for VLIW architectures use instruction schedulers that exploit instruction level parallelism for generating fast schedules of frequently executed portions of a program [3, 9, 16]. While not seen as the goal of these schedulers, a limited form of PDE is achieved as a by-product of the code motion applied during scheduling. On the other hand, the PDE optimization [13, 8, 2] is expected to be performed by an optimization phase that precedes instruction scheduling. However, separating optimization and scheduling may result in undesirable consequences. For example, the placement of statements along critical edges[3] during PDE may introduce additional branches in the program and hence degrade the quality of schedules generated. Thus instruction scheduling may in fact be a more suitable phase for performing PDE. In this section we present a PDE-based scheduling algorithm that not only takes advantage of the complete elimination of partially dead code to minimize the schedule length but also reduces the impact of code duplication incurred during PDE.

With more parallelism available in VLIW processors, recent research in code scheduling considers scheduling regions that are larger than the *trace* [9] or the *superblock* [7]. To allow inclusion of multiple frequently taken program paths in the scheduling region, the *hyperblock* scheduling region [6] contains both conditional branches and control flow merge points. The conditional branches are handled by sequentializing the region using hardware predicated execution [5, 12] or by reverse if-conversion [18]. Consider the hyperblock $R1$ in Figure 11(a). Let us assume that exit $E3$ is the most frequently taken exit while exits $E1$ and $E2$ are taken infrequently. In this situation, paths from *Entry* to $E3$ are

[3] A *critical* edge leads from a node with multiple successors to a node with multiple predecessors.

cycle 0	C if P3	c1=not(P1).not(P2)	c2a=not(P1).P2.not(P3).not(P4)	c2b=P1.not(P3).not(P4)
cycle 1	D if True	br X1 if c1	br X2a if c2a	br X2b if c2b

Figure 12: The VLIW schedule of hyperblock $R1$ after PDE optimization.

considered *critical paths* and a VLIW scheduler attempts to generate short schedules for these paths.

The scheduler in [16] improves upon hyperblock scheduling [6] by adding techniques for *critical path reduction (CPR)* that enable reordering of region exit branches. For the hyperblock in Figure 11(a), the CPR scheduler attempts to place the critical exit $E3$ (which is treated as an exit branch) in the schedule as early as possible so that the critical paths can exit the schedule early and continue to the successor region. The CPR scheduler performs an alternative version of PDE which is achieved by reordering the region exit branches and lifting them across partially dead operations. A branch can move upwards across an operation only when the operation is dead on the exit from that branch. Finding a short schedule for the critical path is thus achieved through PDE optimization.

The CPR scheduler [16] fails to move an exit branch a above another exit branch b if b does not dominate a. In Figure 11(a), $P4$ cannot be scheduled before $P2$ and, more importantly, the branch corresponding to $E3$ cannot be scheduled before $P4$ or $P2$. This may result in a longer schedule because operations dead on the critical path are scheduled before the high priority exit branch. We propose a scheduling scheme based on our PDE algorithm that removes this restriction. We apply PDE before scheduling in order to restructure the scheduling region such that the resulting schedules of the critical paths will only contain live operations. When only live operations precede the high priority exit, the scheduler can achieve minimal schedule length.

Consider again Figure 11(a) and assume that operations A and B are dead on exit $E3$. Applying PDE to the scheduling region $R1$ results in the program shown in Figure 11(b). After PDE, partially dead statements have been pushed off

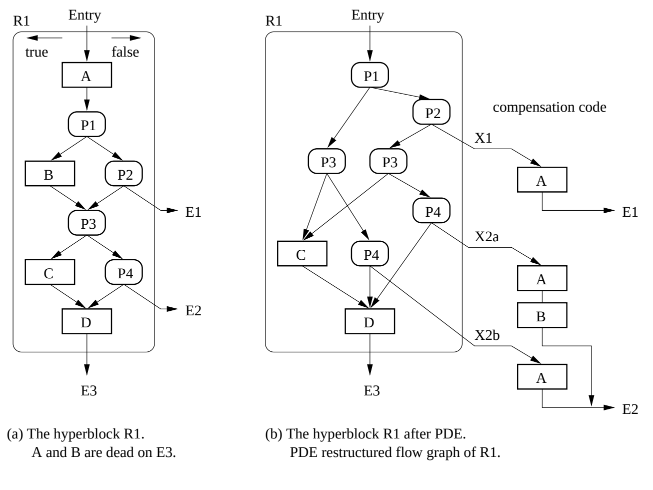


Figure 11: Reducing length of the critical path with PDE.

the hyperblock to the compensation code and the region has been restructured. Any critical path now contains only operations that are live on exit $E3$. Note that existing PDE algorithms [13] are based on code sinking and thus cannot eliminate B from the hyperblock. The CPR scheduler cannot remove A or B from the hyperblock because they are live on $P4$, and $E3$ cannot be lifted across $P4$.

The hyperblock following the removal of partially dead operations shown in Figure 11(b) can be scheduled with a slight modification of the CPR scheduler. Note that all the conditional branches will disappear because the resulting schedule uses predicated execution, as shown in Figure 12. The side exits are handled as described in [16]; they are scheduled as early as possible based on their priorities. Note that operations C and D are executed speculatively as suggested in [14] because they are issued before the side exit branches. Because of speculation, the execution conditions for C and D are very simple predicates that need not be computed by a long sequence of compare operations. Naturally, the partially dead operations pushed off-trace during PDE can be scheduled in $R1$ to fill any empty slots, giving priority to operations from higher priority exits.

The above scheduling algorithm reduces the impact of PDE-incurred code duplication in the following way. First, there is less concern for code duplication in compensation code (such as in our example) because overall performance will not be significantly impaired by a longer schedule of infrequently executed compensation code. With respect to instruction cache locality, the compiler can use a code layout in which the compensation code is not included in the window of locality and hence the window size for the critical paths is unchanged after PDE. Second, if the code duplication occurs in the hyperblock, we reduce the amount of code with predicated execution by keeping only a single copy of each duplicated basic block and computing its execution conditions as a union of the conditions for all its copies.

5 Conclusion

We have presented a powerful PDE algorithm based on program slicing transformations. We have proven that no partially dead statements exist and the operation count along any program path is not increased when our PDE algorithm is applied to acyclic code. This is a result that existing PDE algorithms cannot achieve. In cyclic code, we eliminate deadness to the extent that the operation count along any path is not increased.

Since PDE applied in early stages of compilation may, in fact, decrease code quality, we have proposed a method for combining PDE with scheduling. Based on the observation that most schedulers minimize schedule length through some form of partial dead code removal, our PDE-based scheduling improves upon existing hyperblock scheduling approaches. Our algorithm moves all partially dead statements off critical paths in cases where other PDE algorithms fail. Thus we successfully handle arbitrarily shaped regions the scheduling of which is becoming more important as parallelism available in processors increases.

We have illustrated the slicing transformation in the context of PDE optimization. We believe that the slicing approach can also be used for other optimizations that require program restructuring and movement of conditional branches. Our current research investigates criteria for determining the nature and shape of the *use regions* for other slicing-based optimizations.

6 Acknowledgment

This research has been supported in part by a National Science Foundation Presidential Young Investigator Award CCR-9157371 to the University of Pittsburgh and a grant from Hewlett-Packard. We want to thank M.L. Soffa for her help with the preparation of the final paper. We are grateful to S. Abraham, S. Anik, R. Johnson, V. Kathail, S. Mahlke, B. Ramakrishna Rau, and M. Schlansker for their helpful comments on our preliminary results. We also thank the anonymous referees for pointing us to relevant references.

References

[1] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. *SIGPLAN Notices*, 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[2] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–170, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[3] Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. Using profile information to assist classic code optimizations. *Software-Practice and Experience*, 21(12):1301–1321, December 1991.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[5] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped loop support in the Cydra 5. In Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pages 26–38, Boston, Massachusetts, 1989.

[6] Scott A. Mahlke *et al.* Effective compiler support for predicated execution using the hyperblock. In *25th Annual IEEE/ACM International Symposium on Microarchitecture*, November 1992.

[7] Wen-mei W. Hwu *et al.* The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[8] Lawrence Feigen, David Klappholz, Robert Cassazza, and Xing Xue. The revival transformation. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 421–434, Portland, Oregon, January 1994.

[9] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7), July 1981.

[10] Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.

[11] Susan Horwitz, Alan J. Demers, and Tim Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[12] Vinod Kathail, Michael S. Schlansker, and B. Ramakrishna Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, February 1994.

[13] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *SIGPLAN Notices*, 29(6):147–158, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[14] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 238–247, Boston, Massachusetts, 1992.

[15] Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. *SIGPLAN Notices*, 30(6):56–66, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[16] Michael S. Schlansker and Vinod Kathail. Critical path reduction for scalar programs. In *28th Annual IEEE/ACM International Symposium on Microarchitecture*, Ann Arbor, Michigan, November 1995.

[17] Bernhard Steffen. Property oriented expansion. In *SAS/ALP/PLILP'96*, pages 22–41, Aachen (D), September 1996. Springer Verlag. Proc. Int. Static Analysis Symposium (SAS'96).

[18] Nancy J. Warter, Scott A. Mahlke, Wen mei W. Hwu, and B. Ramakrishna Rau. Reverse if-conversion. *SIGPLAN Notices*, 28(6):290–299, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[19] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, August 1984.