SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; **36**:1655–1685 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Partial dead code elimination on predicated code regions

Jingling Xue *,†, Qiong Cai and Lin Gao

Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia

SUMMARY

This paper presents the design, implementation and experimental evaluation of a practical region-based partial dead code elimination (PDE) algorithm on predicated code in the ORC compiler framework. Existing PDE algorithms are not applicable on predicated code due to the existence of if-converted branches in the program. The proposed algorithm processes all PDE candidates in a worklist and reasons about their partial deadness using predicate partition graphs. Our algorithm operates uniformly on individual hyperblocks as well as regions comprising basic blocks and hyperblocks. The result of applying our algorithm to a SEME (single-entry multiple-exit) region is optimal: partially dead code cannot be removed without changing the branching structure of the program or potentially introducing new predicate defining instructions. We present statistical evidence about the PDE opportunities in the 17 SPEC95 and SPEC00 integer benchmarks. Our algorithm achieves performance improvements in 12 out of the 17 benchmarks on an Itanium machine at small compilation overheads. Our results indicate that our algorithm can be used as a practical pass before instruction scheduling.

KEY WORDS: code optimisation; partial dead code elimination; predicated code; predication; predicate partition graphs; hyperblocks; regions; performance evaluation

1. Introduction

Based on Explicitly Parallel Instruction Computing (EPIC) technology, the Itanium architecture was designed to combine explicit instruction-level parallelism (ILP) with instruction predication. In order to realise the performance potential of the Itanium processors, the compiler must expose and express increasing amounts of ILP in application programs. Region-based compilation as proposed in [1] and implemented in the IMPACT [2] and ORC [3]

*Correspondence to: Jingling Xue, Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia
†E-mail: jxue@cse.unsw.edu.au

Received
Revised

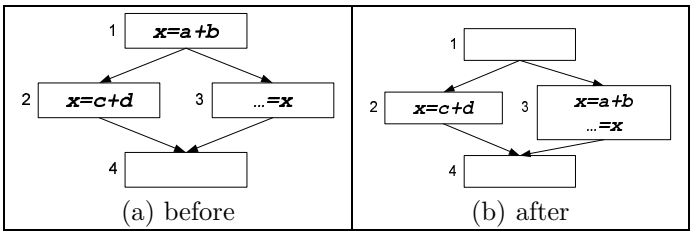


Figure 1: Partial dead code elimination for non-predicated code.

compilers repartitions the program into regions to replace functions as the units of compilation. By exploiting static/dynamic profile information, the compiler can create regions that reflect more accurately the dynamic behavior of the program. In particular, by forming regions containing cycles across function boundaries, the potential for the compiler to expose more ILP is increased [1, 4]. By selecting the sizes and contents of regions appropriately, the compiler can also tradeoff the compilation cost and the use of aggressive ILP techniques.

Predicated execution, supported by the Itanium architecture, allows parallel execution of instructions from multiple control paths and aids in efficient instruction scheduling. In this architectural model, each instruction may be guarded by a boolean operand, its qualifying predicate; the value of this predicate determines whether the instruction is executed or nullified. The values of predicates are manipulated by a set of predicate defining instructions. To explore predication, the compiler generally uses a technique called if-conversion [5], which eliminates branch instructions and replaces affected instructions with predicate defining instructions and predicated forms. This technique enlarges the scope of instruction scheduling and avoids branch misprediction penalties that may be caused by the eliminated branches.

This paper discusses the design, implementation and experimental evaluation of a practical region-based partial dead code elimination (PDE) algorithm on predicated code. An assignment is *partially dead* if there is a path along which the value computed by the assignment is never used, and is *fully dead* if it is partially dead along all such paths. Consider the CFG for non-predicated code as depicted in Figure 1. In Figure 1(a), the assignment $x = a + b$ in block 1 is partially dead along the left branch. This partial deadness can be eliminated by sinking this assignment into block 3, where it is blocked by the use of x due to the flow dependence on x. Dead code appears frequently as a result of the optimisations applied earlier by the compiler, including partial redundancy elimination (PRE) [6], strength reduction and global copy propagation. PDE, which subsumes the standard (full) dead code elimination (DCE), is an aggressive global optimisation. PDE is harder than PRE due to the second-order effects clearly exemplified in [7]. Basically, the code motion for one assignment may be blocked by another due to the data dependences between the two assignments. Therefore, if a complete removal of all partial deadness is to be achieved, the effects of code motions for some assignments on those for the others must be taken into account.

Existing PDE algorithms [8, 9, 7] are developed for non-predicated code with functions as compilation units. These algorithms are not applicable when instructions are predicated. Consider, for example, the following sequence of predicated instructions:

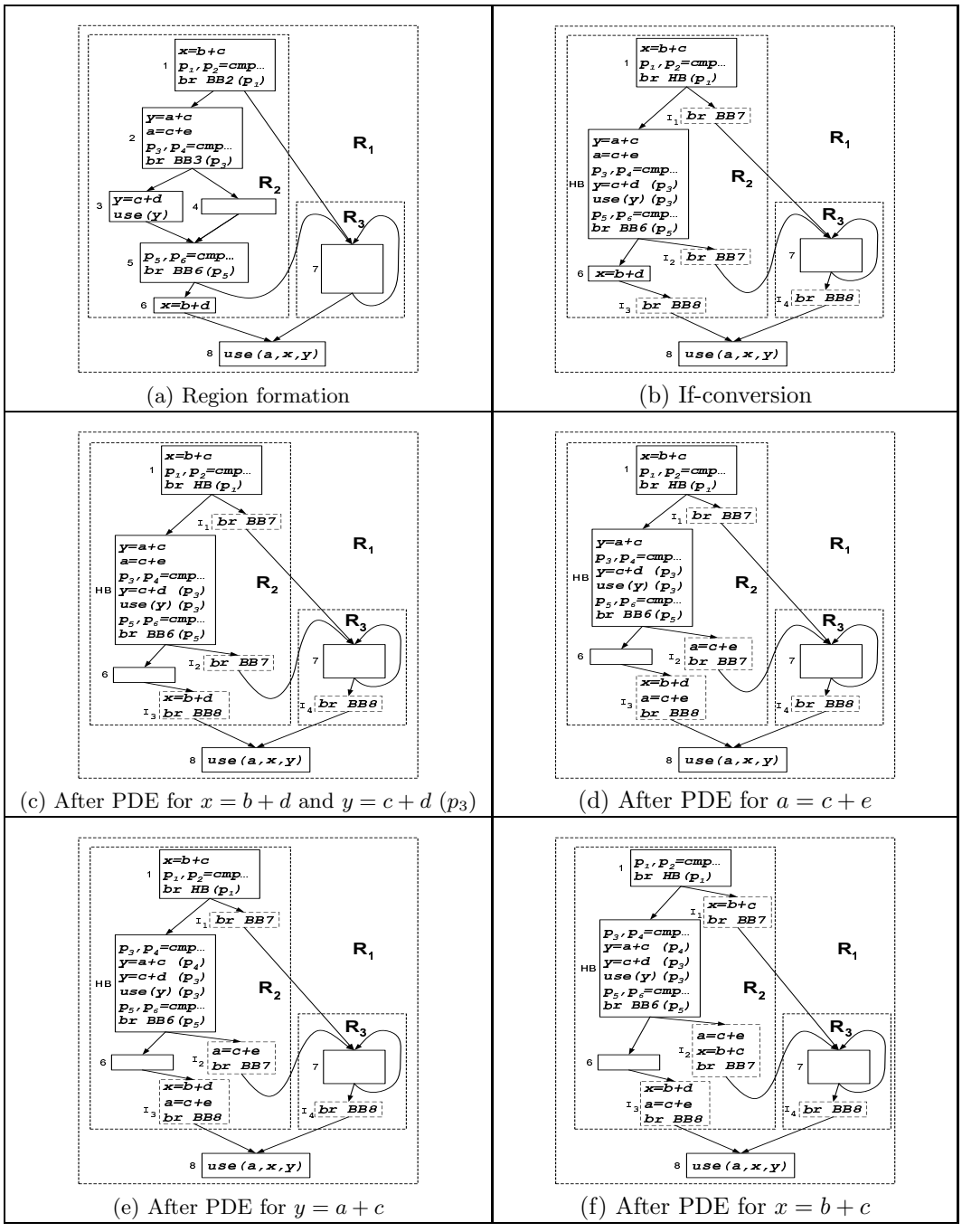
```
x = a + b   (p)
y = x - 1   (q)
x = c + d   (r)
```

Being insensitive to the qualifying predicates, p, q and r, classic PDE algorithms cannot determine whether or not the first assignment $x = a + b$ is partially or fully dead or even dead at all. For example, if $p = q \cup r$, where q and r are disjoint, then $x = a + b$ is partially dead. if $p \cup q = r$, where p and q are disjoint, then $x = a + b$ is fully dead. If $p = q = r$, then $x = a + b$ is not dead at all.

The contributions of this work are summarised as follows:

- We introduce a region-based PDE algorithm on predicated code. Based on the notion of Predicate Partition Graph (PPG) [10, 11], our algorithm is designed to operate uniformly on both hyperblocks (i.e., fully predicated blocks) [4] as well as SEME (single-entry multiple-exit) regions comprising basic blocks and hyperblocks. In the ORC compiler for the Itanium architecture, the regions processed by our algorithm are the leaf, i.e., innermost regions in the region trees of a program [12]. These regions are frequently executed parts of the program and thus demand the use of aggressive optimisations.
  When compiling a program using a region-based compiler such as IMPACT or ORC, (full) dead code elimination (DCE) is often invoked several times in order to reap the benefits of some optimisations applied before each DCE pass. However, PDE is not supported in these compiler frameworks. The PDE algorithm developed in this research is the first that works on regions comprising both basic blocks and hyperblocks.
- The result of applying our algorithm to a region is optimal: partially dead code in the resulting region cannot be removed without changing the branching structure in the program or potentially introducing non-existent predicate defining instructions, which may impair some program executions.
- We have implemented our PDE algorithm in the ORC compiler [3]. We apply PDE just before its instruction scheduling pass since doing so can potentially reduce critical path lengths along frequently executed paths [8]. We present statistical evidence about the PDE opportunities in the 17 SPEC95 and SPEC00 integer benchmarks despite the fact that DCE has been applied several times earlier in the ORC framework. We obtain performance improvements in 12 out of the 17 benchmarks on an Itanium machine with the top three speedups 5.75% and 2.81%, 2.53% being attained by `compress`, `crafty` and `twolf`, respectively. The compilation overhead for all the benchmarks is very small.

Our running example is the CFG (control flow graph) depicted in Figure 2, where the three regions formed by the compiler are highlighted by dashed boxes. For illustration purposes, `use(`y`)` in block 3 indicates that y is used at that point. Similarly, `use(`a, x, y`)` in block 8 indicates that the three variables are used in that block. Suppose that the compiler applies if-conversion [5] to eliminate the branches in block 2. This leads to Figure 2(b), where the blocks 2 – 5 have been merged into a new hyperblock, *HB*. As result, the branching instruction



(a) Region formation

(b) If-conversion

(c) After PDE for $x = b + d$ and $y = c + d$ (p_3)

(d) After PDE for $a = c + e$

(e) After PDE for $y = a + c$

(f) After PDE for $x = b + c$

Figure 2: A running example

"*brBB3*(p_3)" in block 2 has been eliminated. The two instructions in block 3 are now guided by the predicate p_3. Similarly, all instructions in block 4 will be guided by p_4; these instructions are not shown since they are assumed to be irrelevant to our discussions. In Figure 2(b), we have also introduced the so-called *interface blocks* [13], I_1 – I_4, at all region exits to simplify the design and implementation of our algorithm.

Let us explain briefly the effectiveness of our algorithm using the SEME region R_2 in Figure 2(b). We will examine this example in more detail in Section 3.3. There are five assignments in region R_2, which are processed in the reverse topological order of their data dependences, one at a time. To begin with, $x = b + d$ in block 6 is moved into block I_3. As for the $y = c + d$ (p_3) in block *HB*, nothing needs to be done since there is a use of y in the ensuing instruction with predicate p_3. At this stage, we obtain Figure 2(c). For the other three assignments, our algorithm first sinks $a = c + e$ into blocks I_2 and I_3 to get Figure 2(d). By sinking $a = c + e$, we are able to eliminate the partial deadness of $y = a + c$ (along the edge $(2, 3)$ in Figure 2(a)). Otherwise, $a = c+e$ would block $y = a+c$ from being moved downwards (due to the anti dependence on a), preventing the partial deadness of $y = a + c$ from being eliminated. By processing the PDE candidates in the reverse topological order of their data dependences, we can achieve optimal results by avoiding the second-order effects in the PDE problem. In actuality, $y = a + c$ is first removed from *HB* and the so-called *compensation code* $y = a + c$ (p_4) is inserted into *HB* after the predicate defining instruction for p_4. Figure 2(e) depicts the resulting program so far. Finally, $x = b + c$ in block 1 is partially dead along path $1 - HB - 6 - I_3$. The elimination of this partial deadness is illustrated in Figure 2(f): $x = b+c$ has been removed from block 1 and inserted into blocks I_1 and I_2.

The rest of this paper is organised as follows. Section 2 gives the background information about the regions, predication, if-conversions and PPGs. Section 3 presents our PDE algorithm. Section 4 proves its correctness and optimality and argues that the algorithm always terminates. In Section 5, we discuss the implementation of our algorithm in ORC and present our performance results and statistics on the SPEC95 and SPEC00 integer benchmarks. Section 6 compares our work with the related work. Section 7 concludes the paper.

2. Background

Our algorithm applies to any SEME region comprising basic blocks and hyperblocks. However, this work is carried out in the context of ORC, which is a region-based compiler for Intel's Itanium processors. Therefore, this section presents the background information for this work and defines the program representations used in our algorithm. In Section 2.1, we define the regions handled by our algorithm and discuss if-conversion and regional CFGs. In Section 2.2, we introduce PPGs and the queries on the PPGs used by our algorithm.

2.1. Regions

A region-based compiler incorporates a *region formation* phase for partitioning a CFG into regions. Several region formation algorithms have been proposed [14, 1, 15, 4]. In particular, the ORC compiler [3] uses an algorithm similar to interval analysis [16]). As a result, the division of a CFG into regions serves to put a hierarchical structure on the CFG, called the

region tree (also known as the control tree [16]). There are four kinds of regions in ORC [12]: (1) loop regions, (2) SEME (single-entry multiple-exit) regions, (3) MEME (multiple-entry multiple-exit) regions, and (4) improper regions (or irreducible regions). By default, ORC produces MEMEs only temporarily in an intermediate step and eventually converts each of MEMEs into multiple SEMEs (with tail duplication [4] if necessary). While it is possible to force ORC to create MEMEs explicitly, no optimisations will be applied to them.

Consider the three regions formed in Figure 2(a), where R_2 and R_3 are contained in R_1. The leaf R_2 is a SEME consisting of blocks 1 – 6. The leaf R_3 is a loop region formed by block 7 alone. The root region R_1 contains R_2, R_3 and block 8.

We restrict ourselves to two kinds of leaf regions: SEMEs and innermost loop regions. As in the ORC compiler, back edges are not included in loop regions since they cannot be handled by if-conversion. So loop regions are SEMEs. Thus, our PDE algorithm is applicable to SEMEs.

There are several reasons for applying PDE to leaf regions only. First, the most benefit of performing PDE should come from leaf regions. These regions, which include the innermost loop regions as a special case, represent the hottest parts in a program. Second, a region-based PDE algorithm that focusses on leaf regions is consistent with the philosophy of the region-based compilation strategy in the sense that the regions are the units of optimisations. Finally, the leaf regions contain enough PDE opportunities (and particularly so in light of the first point). Some statistical evidence for justifying these claims will be presented in Section 5.

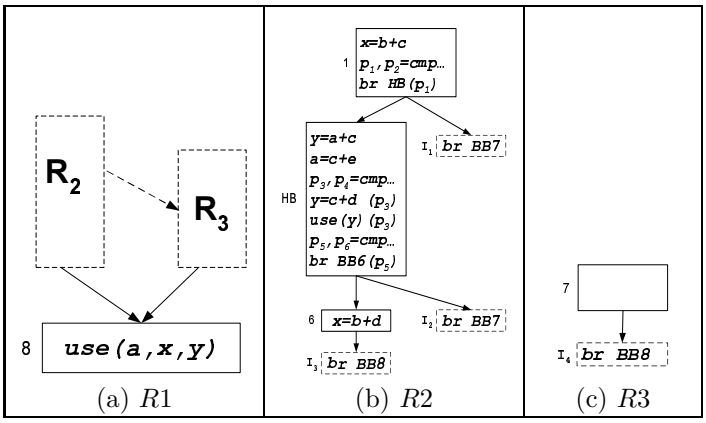
We assume that if-conversion is applied to the regions created during the region formation phase. In particular, a SEME subregion within a region is often fully converted into a single, branch-free block of predicated code. The resulting block is known as a *hyperblock* [4]. In Figure 2(a), blocks 2 – 5 can be regarded as a SEME subregion contained within region R_2. After the two branches in block 2 have been if-converted, the four blocks are combined to form one single hyperblock, *HB*, as shown in Figure 2(b).

Our algorithm operates on a region by traversing the nodes in its regional CFG [12]. Every region has a regional CFG, which is essentially the CFG for the region except that some nodes are regions themselves. The concept can be easily understood by an example. For the running example given in Figure 2, there are three regions. Figure 3 shows the corresponding regional CFGs for the three regions. Note that the back edge around block 7 is not included in the regional CFG for the loop region R_3. Note also that for each leaf region, we have inserted the so-called *interface blocks* [13] (depicted in dashed boxes), one for each region exit. As a result, the successors of every original block in a leaf region must all reside in that region. This simplifies the design and implementation of our PDE algorithm as will be clear later.

2.2. Predicate Partition Graphs

Unlike the existing PDE algorithms on non-predicated code [8, 9, 7], our algorithm uses the notion of Predicate Partition Graph (PPG) [10] to eliminate partial deadness in predicated code. When operating on leaf regions, our algorithm assumes that each leaf region is associated with a unique PPG, which tracks relations among predicates. In ORC, PPGs can be built for all SEME regions. From now on, by a block we mean either a basic block or a hyperblock.

The PPG tracks uniformly control flow and explicit use of predicates in a SEME region. A predicate assigned to a block is called a *control predicate* and a predicate which explicitly



(a) $R1$ (b) $R2$ (c) $R3$

Figure 3: Regional CFGs in our running example.

appears in the instruction is called a *materialised predicate*. The control predicate of a block is viewed as a predicate combining all of the conditions which control whether the block will be executed or not. The control predicate of the unique entry block of a SEME region is denoted by p_0. By convention, p_0 always denotes the true predicate.

Due to the presence of p_0, every instruction can be expressed to have the predicated form:

$$v \quad = \quad \pi \quad (p)$$

where v is a variable, π an expression and p its qualifying predicate. If the materialised predicate of the instruction is $p = p_0 = \mathsf{true}$, we simply write $v = \pi$ (without the qualifying predicate).

Let α be an instruction in a block. The following notations are used:

- BB_α: the block in which α resides
- M-PRED(α): the materialised predicate of α
- C-PRED(BB_α): the control predicate of BB_α
- E-PRED(α): the *executing predicate* of α such that α is executed iff E-PRED(α) = true:

$$\text{E-PRED}(\alpha) = \begin{cases} \text{C-PRED}(BB_\alpha) & \text{if M-PRED}(\alpha) = p_0 \\ \text{M-PRED}(\alpha) & \text{otherwise} \end{cases} \tag{1}$$

Consider the regional CFG for R_2 depicted in Figure 3(b). There are three blocks: 1, HB and 6. Their control predicates are C-PRED$(1) = p_0$, C-PRED$(HB) = p_1$ and C-PRED$(6) = p_5$. Let us take a look at the materialised and executing predicates for the two instructions $a = c + e$ and $y = c + d$ (p_3) in HB. We find that M-PRED$(a = c + e) = p_0$ and M-PRED$(y = c + d\ (p_3)) = p_3$. In addition, E-PRED$(a = c + e) =$ C-PRED$(HB) = p_1$ and E-PRED$(y = a + c\ (p_3)) = p_3$.

The notion of PPG for a region is defined based on execution traces. An execution trace includes all of the instructions being executed. A trace belongs to the *domain* of a predicate

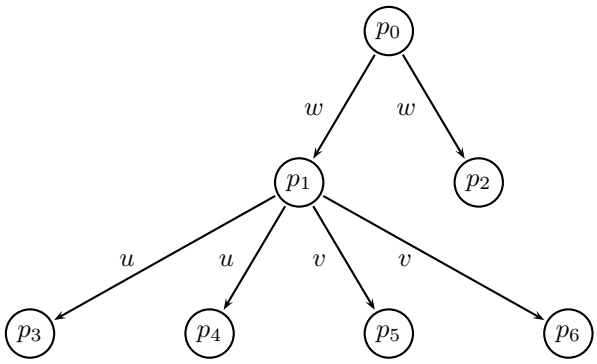


Figure 4: The PPG for the SEME region R_2 in our running example.

p if all of the instructions on this trace are executed when p is true. Following [10, 11], we will use p to mean the domain of p. A *partition* of a predicate p divides the domain of p into mutually disjoint subsets. In a PPG, each node represents a predicate p and each directed edge (p, q) represents the fact that there exists a partition r in p such that q is a subset of r. Figure 4 depicts the PPG for the region R_2 in our example, where the edges from the same partition are conventionally decorated to have the same label. For example, p_1 has two distinct partitions, which are denoted by $p_1 = p_3 \cup p_4$ and $p_1 = p_5 \cup p_6$, respectively.

Our PDE algorithm relies on the following queries on PPGs. We illustrate these operations on predicates using the regional CFG for R_2 shown in Figure 3(b) and its PPG in Figure 4.

- $\mathsf{IsDisjoint}(p, q)$: asks whether the domain of predicate p overlaps with that of predicate q. Two predicates are disjoint if they can reach a common ancestor in the PPG through different edges of the *same partition* and *not disjoint* otherwise. For example, $\mathsf{IsDisjoint}(p_3, p_4) = \mathsf{true}$ and $\mathsf{IsDisjoint}(p_4, p_2) = \mathsf{true}$ but $\mathsf{IsDisjoint}(p_4, p_5) = \mathsf{false}$.
- $\mathsf{IsSubset}(p, q)$: asks whether the domain of p is a subset of the domain of q. For example, $\mathsf{IsSubset}(p_3, p_1) = \mathsf{true}$ and $\mathsf{IsSubset}(p_3, p_0) = \mathsf{true}$ but $\mathsf{IsSubset}(p_3, p_4) = \mathsf{false}$.
  Following [10, 11], we shall write $p \subseteq q$ if $\mathsf{IsSubset}(p, q)$ holds and $p \subset q$ if $p \subseteq q$ but $p \neq q$.
- $\mathsf{LUB\_Diff}(p, q)$: returns the set of predicates such that the union of their domains is the smallest superset of the domain of p subtracted by the domain of q, i.e., $\mathsf{LUB\_Diff}(p, q) \supseteq p - q$, where the equality holds when $q \subseteq p$ [10, 11]. In our PDE algorithm, $\mathsf{LUB\_Diff}(p, q)$ is called only when $q \subset p$. In addition, the result of this operation is simplified such that if all of the child predicates in a partition appear in $p - q$, then these child predicates are replaced with their parent predicate. For example, $\mathsf{LUB\_Diff}(p_1, p_3) = \{p_4\}$, where $p_1 - p_3 = p_4$. As another example, $\mathsf{LUB\_Diff}(p_0, p_3) = \{p_2, p_4\}$, where $p_0 - p_3 = p_2 \cup p_4$. Our algorithm uses this query to find the points for the insertion of compensation code.

We assume that the critical edges in a regional CFG have been split. These are the edges leading from nodes with more than one immediate successor to nodes with more than one

immediate predecessor [6]. This simplifies the construction of the PPG for a region and makes it easy to perform the code insertions required in code motion/sinking transformations.

3. Region-based PDE on Predicated Code

There are two main challenges in designing a worklist-based PDE algorithm that works for regions comprising both basic blocks and hyperblocks. First, we must handle uniformly explicit branches and if-converted branches. We solve this first problem by using a region's PPG to guide the PDE process. Second, sinking an instruction across a branching node and later a join node is not straightforward in a worklist solution. Once again the branches at these branching and join nodes can be explicit or if-converted branches. We solve this second problem by sinking copies of an instruction with appropriate predicates at a branching node. We use a forest as a data structure to record the arriving copies at a join node. We combine the arriving copies at the join node into a single instruction once we have detected that a copy has arrived from each of its incoming edges by comparing predicate relations.

3.1. Scope

Our PDE algorithm operates on SEME regions, one at a time. When working on a region, our worklist-based algorithm eliminates partial deadness for the instructions in the region, one at a time. As in [7], the process of performing PDE for an instruction, which is typically an assignment, involves two basic operations: sinking and elimination. The sinking operation moves the instruction along the control flow on all possible control flow paths. Multiple copies of the instruction guided by mutually disjoint executing predicates are created at branching nodes and moved along the appropriate branches (explicit or implicit). In assignment elimination, copies of the instruction that become fully dead on some paths are eliminated.

We eliminate partial deadness in a SEME region by using its PPG to guide assignment sinking and elimination. We assume that every compare instruction of the form "$p_1, p_2 = cmp \ldots$" generates both true and false predicate values, as is typically the case after if-conversion has been fully performed. The proposed algorithm achieves a complete removal of partial deadness in a SEME region, and is thus optimal in terms of exactly the same criterion used in [7] under the assumption that all PDE candidates are considered to be distinct. This notion of optimality for predicated code regions is introduced below and then illustrated by a few examples.

Definition 1 (Optimality) *Let α be an instruction in a SEME region R – all instructions are distinct. During assignment sinking and elimination, α may be split into multiple copies with mutually disjoint executing predicates. Upon the completion of the PDE algorithm on R, let $\alpha_1, \ldots, \alpha_n$ be all such copies of α existing in the resulting program, which always satisfy:*

1. $\mathsf{E\text{-}PRED}(\alpha_1), \ldots, \mathsf{E\text{-}PRED}(\alpha_n)$ *are the existing predicates in the PPG of R,*
2. $\mathsf{E\text{-}PRED}(\alpha_1), \cdots, \mathsf{E\text{-}PRED}(\alpha_n)$ *are mutually disjoint, and*
3. $\mathsf{E\text{-}PRED}(\alpha_1) \subseteq \mathsf{E\text{-}PRED}(\alpha), \ldots, \mathsf{E\text{-}PRED}(\alpha_n) \subseteq \mathsf{E\text{-}PRED}(\alpha)$.

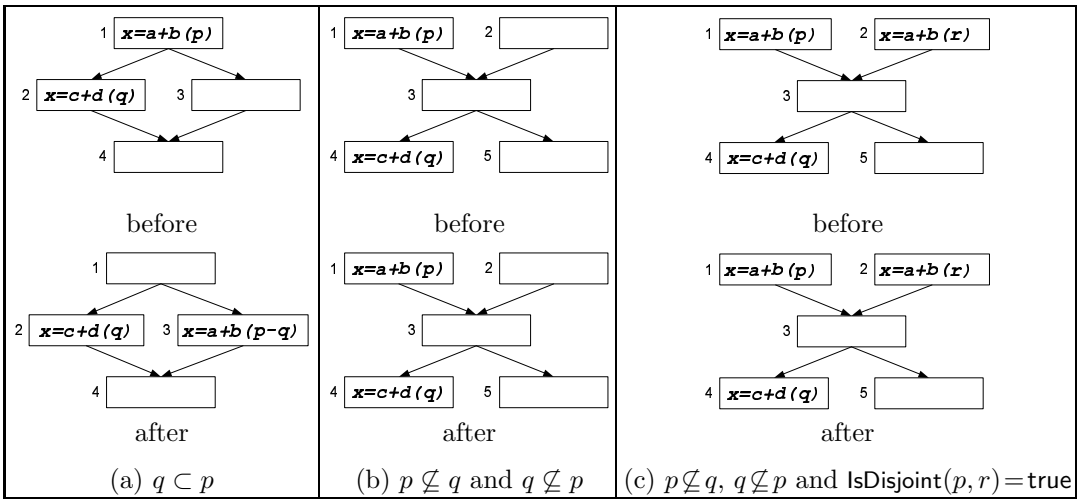


Figure 5: Scope of our PDE algorithm.

Such a transformation (assumed to be semantics-preserving) is optimal if every α_i is not partially redundant when optimised using only assignment sinking and elimination.

Our PDE algorithm guarantees this optimality. The three restrictions as stated in this definition are explained as follows. Since we apply PDE before instruction scheduling, all optimisations have already been performed. Therefore, when eliminating the partial deadness of an instruction α, we refrain from changing the branching structure of the program. We use only the existing predicates that are subsets of E-PRED(α). Thus, we do not introduce any new predicate defining instructions. Hence, Restriction 1 in Definition 1. Implicit in the optimality criterion is that the dynamic count of instructions along any path is not increased. When sinking an instruction across a branching node, multiple copies of that instruction will be created at the branching node and pushed downwards across the respective branches. These copies will be guided by mutually disjoint executing predicates. Obviously, the domains of these executing predicates are subsets of that of the executing predicate of the instruction under consideration. Hence, Restrictions 2 and 3 in Definition 1.

In comparison with the prior work on non-predicated code, we eliminate partial deadness by performing assignment sinking and elimination as illustrated in Figure 5(a) as in Knoop, Rüthing and Steffen's PDE algorithm [7]. In the transformed code, the executing predicate $p - q$ for the instruction in block 3 satisfies $p - q \subseteq p$. However, as in [7], we do not attempt to eliminate the partial deadness illustrated in Figure 5(b) since p and q are not related. There are two approaches to eliminating the partial deadness of $x = a + b$ (p) along path 1-3-4. If control flow restructuring is used as in [8], it is possible to ensure that the dynamic count of instructions is not increased along any path. But the new predicates introduced due to restructuring will

```
1 int x, y, z;
2 z = x + y;     TN717 :- add TN257(p0) TN713 TN715 ;
3 if (z > y)     TN718 TN719 :- cmp4.le TN257(p0) TN717 TN713 ;
4   x = z + x;   TN715 :- add TN719 TN717 TN715<defopnd> ; cond_def
5 else
6   x = z + y;   TN715 :- add TN718 TN717 TN713 ; cond_def
7 z = x + y;     TN720 :- add TN257(p0) TN713 TN715 ;
```

Figure 6: CGIR for a code snippet in ORC. Initially, the mappings of three variables to registers are: `(TN713) = y, (TN715) = x` and `(TN717) = z`. However, `z` in line 7 is renamed from `TN717` to `TN720`. Thus, the two high-level assignments that are syntactically identical in lines 2 and 7 become syntactically distinct in the low-level CGIR representation.

increase the pressure for predicate registers. If restructuring is not used, some new predicate defining instructions may be introduced along some paths. As a result, the dynamic count of instructions along the path will be increased. This further explains the motivation behind Restriction 1 stated in Definition 1. Finally, we do not aim at removing the partial deadness removable by simultaneously sinking multiple occurrences of the same instruction. This is why all instructions are regarded as being distinct in Definition 1. However, Knoop et al. [7] can sink the two instructions in blocks 1 and 2 together into block 5 (as illustrated in Figure 5(c)). As shown in Figure 14, our PDE pass is used in the code generation (CG) module of the ORC compiler. It operates on its intermediate representation (IR), called *code generation IR* (CGIR). Figure 6 gives a code snippet and its corresponding CGIR instructions, in which $TNxyz$ represents a virtual register, where TN stands for *Temporary Name* and xyz is an integer. When translating a program into such a register-based IR, a compiler typically keeps local variables and temporaries in registers by assuming the existence of infinitely many virtual registers. In Figure 6, two occurrences of assignment `z=x+y` are translated into instructions that are no longer syntactically identical (with their predicates ignored). In all 17 SPEC benchmark programs used in our experiments, we have found no single SEME region that contains two identical PDE candidates (with their predicates ignored). Such a property facilitates the practical implementation of the PDE optimisation in optimising compilers. When all PDE candidates are distinct, our algorithm achieves exactly the same optimality results in SEME regions as traditional PDE algorithms on non-predicated codes [8, 9, 7] (Theorem 3).

In our illustrations, basic blocks with explicit branches are used. However, our algorithm works uniformly even when all these branches are if-converted (into implicit branches).

3.2. Algorithm

Our algorithm works on SEME regions one at a time. All PDE candidates in a region are kept in a worklist and processed in the reverse topological order of their data dependences. A

PDE candidate is an instruction that is both movable and free of side effects. In Section 5, we shall examine all kinds of PDE candidates considered by our algorithm. When working on a PDE instruction, our algorithm essentially sinks the instruction along the control flow on all possible paths. When sinking an instruction across a branching node (explicit or if-converted), multiple copies of the instruction with mutually disjoint executing predicates are created and moved along their designated branches. Each copy of the PDE candidate instruction is moved downwards as far as possible until it is either eliminated because it becomes fully dead, or blocked due to data dependences or has been moved into an interface block (as shown in Figure 2). If one copy has arrived at at every incoming edge of a join node, then these copies will be merged and the merged instruction will be processed in the same manner.

Let $\alpha : v = \pi$ be the PDE candidate that is presently processed by our algorithm. Let β be a different instruction that may be executed later on a control flow path. Both are not necessarily in the same block. The following predicates about the two instructions are used:

- $\mathsf{DEFINED}(v, \beta)$: v is modified by β, i.e., there is an output dependence from α to β.
- $\mathsf{USED}(v, \beta)$: v is used by β, i.e., there is a flow dependence from α to β.
- $\mathsf{KILLED}(\pi, \beta)$: some operands of π are modified by β, i.e., there is an anti dependence from α to β.

To facilitate the presentation of our algorithm, the following two predicates are also used:

- $\mathsf{DEP}(\alpha, \beta) =_{df} \mathsf{DEFINED}(v, \beta) \vee \mathsf{KILLED}(\pi, \beta) \vee \mathsf{USED}(v, \beta)$. There is a data dependence from α to β if and only if $\mathsf{DEP}(\alpha, \beta)$ holds.
- $\mathsf{DEFINED\text{-}BUT\text{-}NOT\text{-}USED}(\alpha, \beta) =_{df} \mathsf{DEFINED}(v, \beta) \wedge \neg\mathsf{USED}(v, \beta)$. Essentially, α is partially dead with respect to β only if $\mathsf{DEFINED\text{-}BUT\text{-}NOT\text{-}USED}(\alpha, \beta)$ holds.

Figures 7, 8 and 13 give our algorithm, called *PPDE* (Predicate-Based PDE), for performing PDE on a SEME region, R, consisting of basic blocks and hyperblocks. There are two data structures shared by all procedures: W and $\mathcal{F}$. W is a worklist consisting of all PDE candidates to be processed and $\mathcal{F}$ contains copies of a PDE candidate waiting to be merged during sinking and elimination. The procedures of our algorithm are described below and illustrated with examples.

3.2.1. Main Program (Figure 7)

In line 2, the empty interface blocks are created at all region exits as illustrated in Figure 2(b). This ensures that all successors of every basic block or hyperblock in R are contained in R itself. This simplifies the design of our algorithm so that we can move code out of R easily (line 21). In line 3, the worklist W is initialised with all PDE candidates sorted in the reverse topological order of their dependences. Such an order guarantees the optimality of our algorithm. This is because by eliminating a partially dead instruction, another instruction on which the eliminated instruction is data-dependent may become partially dead. As a result, such so-called second-order effects [7] are dealt with appropriately, resulting in the optimality of *PPDE* (Theorem 3). Finally, *PPDE* terminates when W is empty in line 4.

The PDE candidates in the worklist are processed sequentially, one at a time. As the algorithm proceeds, the candidates are always removed from the worklist in line 6 and new

```
1   PROCEDURE PPDE (R: SEME Region)
2   Create an empty interface block (with a branching instruction) for each region exit
3   Initialise the worklist W with all PDE candidates in R
    sorted in the reverse topological order of their data dependences
4   while W is not empty
5       α = first PDE candidate from W, which, say, has the form v = π (p)
6       Remove α from W
7       if v is not live out of R and the def-use chain of v in R is empty
8           Delete α from BB_α                    // α fully dead
9           continue
10      if α does not have the same form v = π as Prev // initialised to NULL
11          F = NULL
12          Prev = α
13      β = instruction following α in BB_α       // β = NULL if α is the last in BB_α
14      if (¬ Sink(R, W, F, α, β))
15          for each descendant block BB of BB_α in R sorted in topological order
16              if ¬IsDisjoint(E-PRED(α), C-PRED(BB))
17                  β = first instruction of BB
18                  if Sink(R, W, F, α, β)
19                      break
20  Delete the empty blocks (including empty interface blocks)
21  Move the interface blocks at the non-main exits of R into the parent region of R
```

Figure 7: The PDE algorithm on predicated code.

candidates always added at the beginning of the worklist in lines 43, 71 and 79. Thus, to understand the *PPDE* algorithm, it suffices to understand how one PDE candidate is handled.

In lines 5 – 6, we remove the first PDE candidate α from the worklist. Like the existing PDE algorithms [8, 9, 7], our algorithm makes use of live-in, live-out and def-use chains and assumes that this information is updated wherever appropriate. Hence, in lines 7 – 9, α is deleted when it is fully dead. Then the next iteration of **while** in line 4 is executed. As a loop invariant, α *itself* cannot be fully dead if the procedure *Sink* is called.

Lines 10 – 12 are concerned with sinking the multiple occurrences of an instruction created at a branching node and combining them at a join node. This will be discussed in Section 3.2.2.

In line 13, β is initialised to be the instruction that immediately follows the PDE candidate α in block BB_α. If α is the last instruction in the block, then β is set to be *NULL*.

In line 14, the procedure *Sink* is called to perform assignment sinking and elimination in the block containing α. If this returns false, then α can be further moved downwards. So we continue the PDE process for α in all the descendant blocks *BB* rooted at BB_α in the region *R*. The test in line 16 is performed since α can be partially dead only along the execution paths starting from α. The PDE process for α is declared to be complete when either the call to *Sink* in line 18 returns true or all the blocks rooted at BB_α in *R* have been processed. Then the same PDE process will be repeated on the next PDE candidate in the worklist.

At the end of PDE, we do two things. In line 20, we clear up the regional CFG by deleting all empty blocks such as block 6 in Figure 2(f). In line 21, we move the interface blocks at the non-main region exits into the parent region of R. This strategy tends to reduce the critical path lengths along the frequently executed paths leaving the main exit(s) of the region R .

3.2.2. Sinking and Elimination (Figure 8)

The procedure *Sink* aims at eliminating the partial deadness of α in BB_α with respect to the instruction δ starting from β in BB_β (line 24). In lines 23 and 25, p and q are the executing predicates of α and δ, respectively. The procedure is driven entirely by comparing the predicate relations between p and q. If $\mathsf{IsDisjoint}(p, q) = \mathsf{true}$ in line 26, then the executions of α and δ are mutually exclusive. Therefore, α is not partially dead with respect to δ. Nothing needs to be done. So the next iteration **for** loop in line 24 will be executed. Otherwise, there are four cases depending on the relations between the predicates p and q as explained below. All illustrating CFGs for these four cases given in Figures 9 – 12 consist of explicit branches only. However, the versions of these examples in which the explicit branches are if-converted will be equally applicable.

Case 1 (lines 27 – 37): $p = q$. If there is a dependence from α to δ (line 28), then α cannot be moved beyond δ. Hence, we return true to *PPDE* to start processing the next PDE candidate. However, if $\mathsf{DEFINED\text{-}BUT\text{-}NOT\text{-}USED}(\alpha, \delta)$ also holds, α is fully dead and then deleted. If there is no dependence from α to δ (line 32), there are two cases. If δ is a branching instruction, this must be the only exit for α (since $p = q$). Hence, we move α just before δ (which has the effect of moving α into BB_δ when both are in distinct blocks). We then return false to enable *PPDE* to process the descendant blocks of BB_δ in lines 15 – 19. This situation, which may occur in a program, is illustrated in Figure 9. As a special case, this allows instructions to be moved into interface blocks as shown in Figure 2. Otherwise, in line 37, we move α after δ and and set δ to point to α. Thus, in the next iteration of the **for** in line 24, δ will point to the instruction following α. By swapping α and δ, we are essentially sinking α downwards along the flow of control. A functionally equivalent but less efficient replacement for line 37 would be:

```
Move α after δ
Insert α at the beginning of W
return true
```

However, this alternative will only cause *Sink* to be called immediately.

Case 2 (lines 38 – 47): $q \subset p$. There are two possible scenarios, which are illustrated in Figure 10, depending on whether α is partially dead or not:

(a) If $\mathsf{DEFINED\text{-}BUT\text{-}NOT\text{-}USED}(\alpha, \delta) = \mathsf{true}$ (line 44), then α is partially dead with respect to δ. The elimination of the partial deadness is accomplished as illustrated in Figure 10(a). First, the so-called compensation instruction(s) are inserted in

```
22  PROCEDURE Sink(R: SEME Region, W: worklist, F: Forest, α, β: Instruction)
23  p = E-PRED(α)
24  for(δ = β; δ ≠ NULL; δ = instruction following δ in instruction list of BB_β)
25      q = E-PRED(δ)
26      if ¬IsDisjoint(p, q)
27  Case 1 if p = q
28            if DEP(α, δ)
29               if DEFINED-BUT-NOT-USED(α, δ)
30                  Delete α from BB_α                  // α is fully dead
31               return true
32            else
33              if δ is an exit (i.e., a branching instruction) of BB_β
34                  Move α before δ
35                  return false
36              else
37                  Move α after δ and then set δ to point to α
38  Case 2 else if p ⊃ q
39            if ¬DEFINED-BUT-NOT-USED(α, δ)
40               θ = Create a copy of α
41               C-PRED(θ) = C-PRED(δ), M-PRED(θ) = M-PRED(δ)
42               Insert θ before δ
43               Insert θ at the beginning of W
44            else do nothing                           // α is partially dead
45            CompensationInsert(R, W, α, δ)
46            Delete α from BB_α
47            return true
48  Case 3 else if p ⊂ q
49            if DEP(α, δ) ∧ DEFINED-BUT-NOT-USED(α, δ)
50               Delete α from BB_α                     // α is fully dead
51               DelInst(R, F, α)
52               return true
53            else
54               AlreadyMerged = AddInst(R, F, α, δ)
55               if AlreadyMerged ∨ DEP(α, δ)
56                  return true
57  Case 4 else
58            if DEP(α, δ)
59               return true
60  return false
61  PROCEDURE DelInst(R: SEME region, F: Forest, α: Instruction)
62  Delete the node α from F (if α exists in F) of R
63  PROCEDURE AddInst(R: SEME region, F: Forest, α, δ: Instruction)
64  if the node α does not exist in F (of R)
65     Add the directed edge δ → α to F
66     Let α_1, ..., α_n be all children of δ in F
67     if ⋃_{i=1}^{n} E-PRED(α_i) = E-PRED(δ)
68        θ = Create a copy of α
69        C-PRED(θ) = C-PRED(δ), M-PRED(θ) = M-PRED(δ)
70        Insert θ before δ
71        Insert θ at the beginning of W
72        Delete the nodes α_1, ..., α_n and δ from F
73        Delete α_1, ..., α_n from BB_{α_1}, ... BB_{α_n}
74        return true
75   return false
```

Figure 8: The PDE algorithm on predicated code (cont'd).

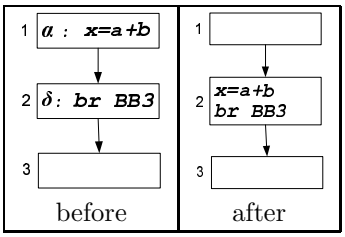
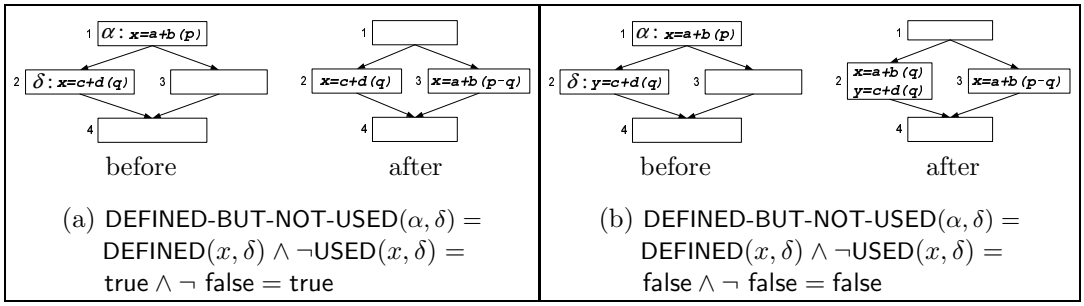


Figure 9: Case 1 of *Sink* when $\mathsf{DEP}(\alpha, \delta) = \mathsf{false}$ and δ is a branching instruction.



(a) $\mathsf{DEFINED\text{-}BUT\text{-}NOT\text{-}USED}(\alpha, \delta) = \mathsf{DEFINED}(x, \delta) \wedge \neg\mathsf{USED}(x, \delta) = \mathsf{true} \wedge \neg\ \mathsf{false} = \mathsf{true}$

(b) $\mathsf{DEFINED\text{-}BUT\text{-}NOT\text{-}USED}(\alpha, \delta) = \mathsf{DEFINED}(x, \delta) \wedge \neg\mathsf{USED}(x, \delta) = \mathsf{false} \wedge \neg\ \mathsf{false} = \mathsf{false}$

Figure 10: Case 2 of *Sink* ($q \subset p$).

the code and into the worklist in the call to the procedure *CompensationInsert* made in line 45. In this example, the compensation instruction $x = a + b\ (p - q)$ is inserted into block 3. This instruction is also inserted at the beginning of the worklist (line 79). Then, in line 46, α is removed from block BB_α. In line 47, we return true to *PPDE* so that we will continue to perform the assignment sinking and elimination for all the compensation instructions (queued now all at the beginning of the worklist).

(b) If $\mathsf{DEFINED\text{-}BUT\text{-}NOT\text{-}USED}(\alpha, \delta) = \mathsf{false}$ (line 39), α is not partially dead with respect to δ. Figure 10(b) illustrates this scenario for the case when $\mathsf{DEFINED}(v, \delta) = \mathsf{USED}(v, \delta) = \mathsf{false}$. In comparison with Figure 10(a), the only difference is that a copy of α with the executing predicate q must also be inserted before δ. As a result, we can create more than one copy of α with mutually disjoint predicates. For example, two copies of α are created in Figure 10(b). Of all occurrences of α created at these branching points, some may be deleted later because they are dead, some may be blocked during their code motions due to data dependences and the others will eventually arrive at some control flow merging points. If there

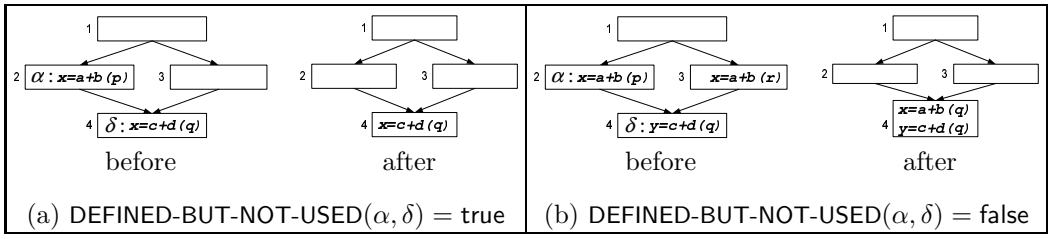


Figure 11: Case 3 of *Sink* ($p \subset q$).

is an instruction occurrence arriving at each distinct incoming execution path at a merging point, these occurrences will be combined as described in Case 3 below.

Case 3 (lines 48 – 56) : $p \subset q$. There are also two possible scenarios as explained below:

(a) If DEFINED-BUT-NOT-USED(α, δ) = true (line 49), then α is fully dead with respect δ as illustrated in Figure 11(a). This is dealt with in lines 50 – 52. We return true in line 52 so that the next PDE candidate in the worklist can be processed.

(b) If DEFINED-BUT-NOT-USED(α, δ) = false (line 53), then α cannot be fully dead with respect to δ. This is dealt with in lines 54 – 56 and illustrated in Figure 11(b) when DEP(α, δ) = false. The instruction δ represents a merging point (at a join node) for copies of the PDE candidate α created earlier and arriving at the incoming edges of the join node. To guarantee the optimality of our algorithm, we must merge these copies once they have arrived along *all* the incoming edges of the join node. However, these copies arrive (i.e., are processed) sequentially. When working on one copy, we do not know in advance if the others will eventually arrive or not since some of them may have been deleted because they are dead or blocked during their code motions due to data dependences. Thus, this part of our algorithm has been designed to deal with all these possibilities optimally. In line 54, we call *AddInst* to record this arriving instruction and also check to see if we can combine the instruction occurrences of α that were created earlier in Case 2 as illustrated in Figure 10. In *AddInst*, we use a forest, denoted $\mathcal{F}$, as a data structure to keep track of the instructions arriving at the merging point δ from all its incoming edges. If this is so (line 67), we combine them into θ (lines 68 – 69) and insert it before δ and also into the worklist (lines 70 – 71). To clean up, we delete these arriving instructions and associated nodes in lines 72 – 73. We return true in line 74 so that *PPDE* can process immediately the next PDE candidate, i.e., the instruction θ that we have just inserted into the worklist. If the equality in line 67 does not hold, then at least one copy of α is not found just before δ along some incoming edge at the merging point δ. This copy may or may not arrive eventually. It is possible

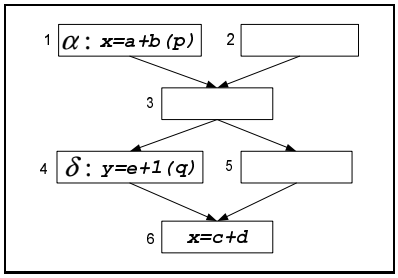


Figure 12: Case 4 of *Sink* when $\mathsf{DEP}(\alpha, \delta) = \mathsf{false}$.

that no copies of α were ever created earlier on the paths leading to that edge or copies were created but some were later deleted due to partial deadness or blocked in their code motions due to data dependences. Hence, we return false in line 75 to continue the PDE process on α. In this case, α can only be fully dead (with respect to some instruction that will be executed after δ) or not dead at all. In the former case, $\mathit{AlreadyMerged} \vee \mathsf{DEP}(\alpha, \delta)$ in line 55 will evaluate to false. Then continuing the PDE process on α will cause it to be eliminated eventually. Similarly, all the other copies of α that arrive at this merging point later will also be eliminated. In the latter case, $\mathit{AlreadyMerged} \vee \mathsf{DEP}(\alpha, \delta)$ will evaluate to true. Then these instruction copies will be merged at the merging point if there is one copy arriving at every incoming edge of the join node. The same PDE process will be repeated on the merged instruction. Note that in the case when $\mathsf{DEP}(\alpha, \delta) = \mathsf{true}$, we will still call *AddInst* in line 54 to see if it is possible to combine the instruction copies from all the incoming edges of the merging point so that we can insert the merged instruction just before the data-dependent instruction δ.

Recall that in lines 10 – 12, we destroy (or re-initialise) $\mathcal{F}$ if two adjacent PDE candidates in the worklist are not identical (with their predicates being ignored). This explains the purpose of *Prev* used in *PPDE*, which is initialised to *NULL*.

Case 4 (lines 57 – 59) : p and q are not disjoint but neither is a subset of the other. If there is a dependence from α to δ, α cannot be moved beyond δ. We return true so that *PPDE* can process the next candidate in the worklist. Otherwise, i.e., when $\mathsf{DEP}(\alpha, \delta) = \mathsf{false}$, we return false since we must continue to perform the assignment sinking and elimination for α due to the optimisation opportunities that may happen later. The rationale behind this is illustrated in Figure 12, where we need to eliminate the full deadness of α with respect to $x = c + d$. If all these basic blocks are part of a hyperblock (due to if-conversion), then $y = e + 1$ (q) can appear earlier before $x = c + d$ in the hyperblock. Hence, the PDE process for α must continue after $y = e + 1$ (q) has been examined.

```
76  PROCEDURE CompensationInsert(R: SEME Region, W: worklist, α, δ: Instruction)
77  for each predicate r in LUB_Diff(E-PRED(α), E-PRED(δ))
78      θ = Create a copy of α such that E-PRED(δ) = r
79      Insert θ at the beginning of W
80      if BB_α = BB_δ          // BB_δ is a hyperblock
81          Insert θ before (or after) δ
82      else
83          for each successor BB_s of BB_α in the regional CFG of R
84              if C-PRED(BB_s) = r
85                  Insert θ at the entry of BB_s
86                  break
```

Figure 13: The PDE algorithm on predicated code (cont'd).

3.2.3. Compensation Code Insertion (Figure 13)

The procedure *CompensationInsert* is called with four arguments: the region R being processed, the worklist W for R and the PDE candidate instruction α (with the executing predicate $p = \mathsf{E\text{-}PRED}(\alpha)$) and the instruction δ (with the executing predicate $q = \mathsf{E\text{-}PRED}(\delta)$). This procedure is responsible for inserting copies of α with appropriate executing predicates such that the executions of these so-called compensation instructions have exactly the same effect as the execution of α in BB_α when its executing predicate is changed to $p - q$.

When this procedure is called in line 45, α has just been sunk to a branching node. There are two cases. If both α and δ are in a common hyperblock, i.e., $BB_\alpha = BB_\delta$, then δ is the instruction that immediately follows α (due to assignment sinking in general and instruction swapping in line 37 in particular). If α and δ are in two distinct blocks, i.e., $BB_\alpha \neq BB_\delta$, then δ must be the first instruction of BB_δ, which is a successor block of BB_α (line 15). Therefore, *CompensationInsert* has been simply designed to handle these two cases.

We use $\mathsf{LUB\_Diff}(\mathsf{E\text{-}PRED}(\alpha), \mathsf{E\text{-}PRED}(\delta))$ to find all the insertion points with the required executing predicates. In line 77, we go through the predicates in $\mathsf{LUB\_Diff}(\mathsf{E\text{-}PRED}(\alpha), \mathsf{E\text{-}PRED}(\delta))$. For each predicate r, we create a compensation instruction θ with r being its executing predicate. We insert this instruction into the worklist in line 79. If $BB_\alpha = BB_\delta$, where BB_δ must be a hyperblock, then the compensation insertion takes place inside BB_δ (line 81). It does not matter whether the compensation instruction θ is inserted before or after δ since both have disjoint predicates. Otherwise, we traverse the successors of BB_α in the regional CFG of R in line 83 to locate the *unique* successor with the control predicate r. If BB_s is the desired successor (line 84), the compensation code is inserted at the entry of BB_s. In both cases, the compare instruction for defining predicate r must have been defined earlier. Since all critical edges have been split, BB_s has BB_α as its unique immediate predecessor. We are done with the predicate r. Hence, the break in line 86.

3.3. Example

Let us trace the execution of our algorithm on the region R_2 shown in Figure 2(b). In line 2, the three interface blocks I_1 – I_3 are created as shown in Figure 2(b). There are five PDE candidates in the region. So in line 3, $W = (x = b+d, y = c+d\ (p_3), a = c+e, y = a+c, x = b+c)$. These five instructions appear in the list in the reverse topological order of their data dependences. Consider the first PDE candidate $\alpha =_{df} x = b + d$ in block 6, where x is live out of R_2 (since there is a use of x in block 8). This PDE candidate is the only instruction in block 6, so $\beta = NULL$ in line 13. As a result, the call to *Sink* in line 14 returns false immediately. Block 6 has one successor, I_3. In line 17, β is pointing to the instruction "*br* `BB8`" in the interface block I_3. In line 18, *Sink* is called. Case 1 will be executed so that $x = b + d$ will end up at the beginning of block I_3. This is all that can be done for this first PDE candidate. Consider now how *PPDE* deals with the second PDE candidate $\alpha =_{df} y = (c + d)\ (p_3)$ in the worklist W. This time, $\beta =$ `use`$(y)\ (p_3)$ in line 13. *Sink* is called in line 14. Case 1 is executed. Due to the flow (i.e., true) dependence between the two instructions, *Sink* simply returns true in line 31. The result of performing PDE on these first two PDE candidates is shown in Figure 2(c). At this time, there are three PDE candidates in the worklist: $W = (a = c+e, y = a+c, x = b+c)$.

PPDE then takes the candidate $\alpha =_{df} a = c + e$ from W and sets β to be the instruction "$p_3, p_4 = cmp...$" – both instructions are in block *HB*. In the first iteration of the **for** loop in line 24, Case 1 is executed since $p = q = p_1$. This causes $a = c + e$ and "$p_3, p_4 = cmp...$" to be swapped. In the next iteration of the **for** loop, δ will be pointing to $y = c + d\ (p_3)$. Hence, $q = p_3$. The PPG for region R_2, i.e., the region under consideration, is shown in Figure 4. We can see that $p_3 \subset p_1$. So Case 2 is executed. Then the situation as illustrated in Figure 10(b) will take place except that the branching is if-converted rather than explicit. This means that $a = c + e$ will be removed from *HB* and $a = c + e\ (p_3)$ and $a = c + e\ (p_4)$ inserted just after "$p_3, p_4 = cmp...$". In addition, both instructions will also be inserted into the worklist W. What happens at this branching point is that $a = c + e$ has been effectively split into two copies, $a = c + e\ (p_3)$ and $a = c + e\ (p_4)$, with disjoint executing predicates. *Sink* then returns true to *PPDE*. At this time, $W = (a = c + e\ (p_3), a = c + e\ (p_4), y = a + c, x = b + c)$.

We will not describe all the details about how the first two PDE candidates in W, i.e., $a = c + e\ (p_3)$ and $a = c + e\ (p_4)$, are handled. Looking at Figure 2(a), $a = c + e\ (p_3)$ will be moved conceptually to the bottom of basic block 3 and $a = c + e\ (p_4)$ to the bottom of basic block 4. In line 54 of Case 3, both will be combined into one single instruction $a = c + e$, which will be inserted just before "$p_5, p_6 = cmp...$". Then the combined instruction $a = c + e$ is the next PDE candidate to be processed. It will be split again into two copies, which will be moved eventually into the interface blocks I_2 and I_3, respectively, as shown in Figure 2(d). At this time, $W = (y = a + c, x = b + c)$. By applying *PPDE* to $y = a + c$, Case 2 of *Sink* will be executed. The partial deadness of this instruction is eliminated as shown also in Figure 2(e). In essence, the executing predicate of the instruction has been changed from p_1 to p_4. Figure 2(f) shows the result of performing PDE on the last candidate $x = b + c$ in block 1.

Finally, all three interface blocks I_1 – I_3 are non-empty. Those at the non-main region exits are moved into the parent region of R_2, which is the root region R_1 in this particular case.

3.4. Time Complexity

Suppose that a region R contains n instructions and m PDE candidates. Note that R is a SEME region free of cycles. During assignment sinking and elimination for a particular candidate, multiple copies of the candidate may be created (with disjoint predicates). These copies then move along distinct branches of a branching node according to their predicates. However, some of these copies are merged, whenever possible, at a join node (line 54). This implies that no two copies of the same candidate, which must have disjoint predicates, may move along a common branch. Hence, the worst-case time complexity for processing one PDE candidate is $O(n)$. This gives rise to an overall time complexity of $O(nm)$ for processing all m candidates in the region R. A linear algorithm does not appear to be possible due to the second-order effects as explained briefly in Section 1 [7]. Despite its non-linear time complexity, we shall see in Section 5 that the *PPDE* algorithm is efficient for benchmark programs.

4. Termination; Correctness; Optimality

Theorem 1 (Termination) *The algorithm PPDE terminates.*

Proof. It is sufficient to prove that the worklist W will eventually be empty. In line 3, W is initialised with a finite number of PDE candidates in the region R. In lines 5 – 6, the first PDE candidate α is removed from W. Let η be the next PDE candidate (i.e., the candidate after α) in W. During the calls to *Sink* in lines 14 and 18, *PPDE* may add new PDE candidates to W in lines 43, 71 and 79. But it will add them only at the beginning of W. We will prove the theorem by arguing that *PPDE* will eventually remove η from W in lines 5 – 6 and start performing the PDE on it. This is trivially the case if lines 7 – 8 are executed. Otherwise, when α is processed, all the instructions created (in Cases 2 and 3, if any) are its copies with mutually disjoint predicates. Due to the nature of assignment sinking and elimination, some of these are eliminated since they are dead, some are blocked due to data dependences and the others are eventually moved out of the region R. Since R is a SEME region free of cycles, any new PDE candidates (i.e., copies of α) added to W will be removed eventually. Then, the candidate η will be processed, i.e., removed from W in lines 5 – 6 eventually. □

Theorem 2 (Correctness) *The algorithm PPDE preserves the semantics of the program.*

Proof. We argue that every assignment sinking or elimination preserves the semantics of the program. In line 8, we delete α because it is fully dead. Let us examine Cases 1 – 3 in *Sink*. In Case 1, we delete α because it is fully dead (line 30). The **else** statement beginning in line 32 is justified due to the lack of dependences between α and δ. In Case 2, we delete α because it is partially dead (line 46). In addition, the compensation instructions are inserted by *CompensationInsert* (line 45) correctly according to the predicate relations. Otherwise, α is not partially dead. But we have also inserted a copy of α, called θ, in lines 40 – 42 correctly according to the predicate relations. In Case 3, we delete α because it is fully dead (line 50), and also update the data structure $\mathcal{F}$ (line 51). In the **else** statement beginning in line 53, we combine the instances of α arriving at a merging point only when the equality in line 67

holds. The simple data structure, $\mathcal{F}$, always keeps tracks of the instruction instances that have arrived at that point via the call to *DelInst* in line 51 and the call to *AddInst* in line 54. The semantics of the program is preserved by lines 68 – 74. □

The following lemma will be used to prove the optimality of our algorithm.

Lemma 1. *During sinking and assignment elimination, every PDE candidate, once blocked by an instruction due to data dependences, must always be positioned just before that instruction.*

Proof. The code motion for the PDE candidate α happens only in the four cases in the procedure *Sink*. In Case 1, α is moved downwards and blocked only just before a data-dependent instruction δ (lines 34 and 37). In Case 2, the required compensation instructions are inserted and they become new PDE candidates to be dealt with immediately afterwards. In Case 4 (as illustrated in Figure 11), no code sinking is explicitly done. To complete the proof, we need to deal with Case 3, the most important case for this lemma. In the **else** statement (lines 53 – 56), we merge the copies of the PDE candidate instruction arriving at all the incoming edges of a merging point. We then apply the same PDE process to the merged instruction by returning true in line 56. In the special case when $\mathsf{DEP}(\alpha, \delta) = \mathsf{true}$, we also return true (lines 55 – 56) since α cannot be moved further downwards. By returning true, the other copies of the PDE candidate can be processed next. If there will be one copy arriving at each incoming edge of the merging point, all these copies will be merged and positioned just before the data-dependent instruction δ, which will prevent α from being moved downwards. By combining all four cases, the claim of this lemma has been established. □

Theorem 3 (Optimality) *The algorithm PPDE is optimal.*

Proof. As a loop invariant for the **while** loop in line 4, the PDE candidates in the worklist for a region are always sorted in the reverse topological order of their data dependences. This is possible since the region is a SEME region free of cycles. Each PDE candidate is moved downwards as far as possible: (1) some copies are deleted if and only if they are dead, (2) some are blocked due to data dependences, and (3) the remaining ones are moved into some interface blocks inserted at the exits of the region. To complete the proof, it suffices to show that in the case of (2), the blocked candidate is always positioned just before another instruction such that both are related by data dependences. This result is established in Lemma 1. □

5. Experimental Results

We have implemented our PDE algorithm in the code generation (CG) module of the Open Research Compiler (ORC) [3] (version 2.1), a compiler for the Itanium Processor Family (IA-64). Figure 14 depicts the compiler framework in which our PDE algorithm is recommended to be used and evaluated. In the CG module, the major passes are region formation, if-conversion, loop optimisation, control flow optimisation, scheduling and register allocation. Our PDE pass is invoked just before the instruction scheduling pass. This phase-ordering not only eliminates all partial deadness before scheduling (Theorem 3) but also tends to reduce the cycles that the instructions wait for the source operands from the memory subsystem (Figures 18 and 19).

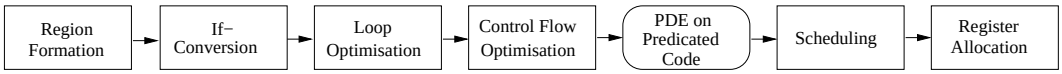


Figure 14: The code generation (CG) module in ORC with PDE incorporated

SPEC	Benchmark	Language	Description
00	164.gzip	C	Compression
	175.vpr	C	FPGA Circuit Placement and Routing
	176.gcc	C	C Programming Language Compiler
	181.mcf	C	Combinatorial Optimisation
	186.crafty	C	Game Playing: Chess
	197.parser	C	Word Processing
	252.eon	C++	Computer Visualisation
	253.perlbmk	C	PERL Programming Language
	254.gap	C	Group Theory, Interpreter
	255.vortex	C	Object-Oriented Database
	256.bzip2	C	Compression
	300.twolf	C	Place and Route Simulator
95	099.go	C	An Internationally Ranked Go-Playing Program
	124.m88ksim	C	A Chip Simulator for the Motorola 88100 Microprocessor
	129.compress	C	An In-Memory Version of the Common UNIX Utility
	130.li	C	Xlisp Interpreter
	132.ijpeg	C	Image Compression/Decompression on In-Memory Images

Table I: All 17 SPEC95 and SPEC00 integer benchmark programs.

We evaluate this work using all 17 SPEC95 and SPEC00 integer benchmarks as shown in Table I. The benchmarks are compiled at the "-O2" optimisation level with inlining switched on (except for `eon`). Inlining enables the frequently executed blocks across the function boundaries to be formed as regions. The profiling information is collected using the train inputs. However, all benchmarks are executed using the reference inputs. The measurements were performed on an Itanium machine equipped with a 667MHz Itanium processor and 1GB of memory. We report the PDE opportunities, performance speedups and compilation overheads for the benchmarks. We also collect dynamic execution statistics to understand how PDE impacts various cycle metrics for a program using the `pfmon` performance monitoring tool [17].

Our algorithm is designed to work on cycle-free SEME regions consisting of basic blocks and hyperblocks. In the ORC compiler, these are exactly the innermost (i.e., leaf) regions in the region trees of a program. The only exception is that PDE is not applied to the innermost regions that are loops when they are already software-pipelined. Figure 15 demonstrates the

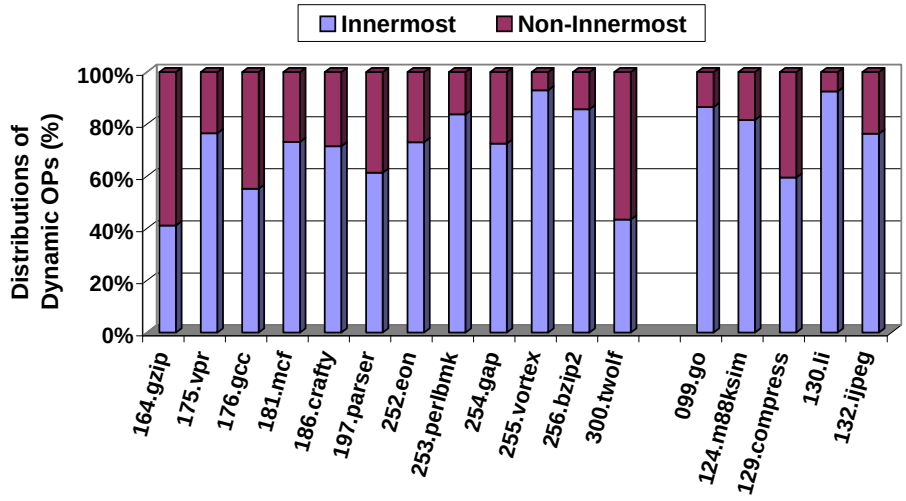


Figure 15: Innermost vs non-innermost regions for which the PDE algorithm can be applied.

sufficient benefits of performing PDE on the innermost regions, where "innermost" represents all those innermost regions processed by our algorithm and "non-innermost" the remaining non-leaf regions. The percentage of the dynamic number of instructions processed by our algorithm over the total in both categories in a benchmark ranges from 41.15% for `gzip` to 93.00% for `vortex` with an average of 79.00% for all the benchmarks. These statistics show that the innermost regions are the hottest regions in a program worth being optimised.

In our implementation, the PDE candidates are rather comprehensive, including instructions on logical operations, arithmetic operations, shift operations, move operations (between registers), float conversion operations (e.g., `fcvt`), zero-extension operations (e.g., `zxt`) and multimedia operations. The non-PDE candidates are typically those with side effects, including instructions on memory operations (e.g., load and store) and cache operations. Another reason for excluding load instructions is that the instruction scheduler in ORC tends to move them up against the control flow. The other non-PDE candidates are compare and branch instructions and any instructions marked as being non-movable by the ORC compiler.

Figure 16 shows the benefits of PDE for the benchmarks. We see convincingly the existence of PDE opportunities in the benchmarks. This is true even though ORC has applied DCE several times earlier before our PDE algorithm is applied. For each program, we measure the opportunity as the dynamic count of instructions which were found to be partially dead and eliminated. The bottom bar represents the amount of full deadness. All benchmarks except `eon` are dominated by the strictly partial deadness, which is removable by our PDE algorithm.

Figure 17 shows the execution time speedups of the ORC+PDE configuration (with PDE being included) over the default ORC configuration. The positive speedups are obtained in 12

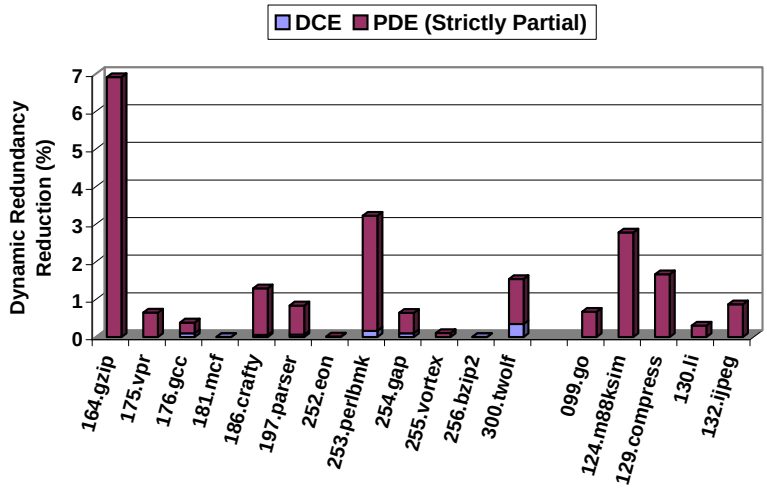


Figure 16: PDE opportunities in benchmarks.

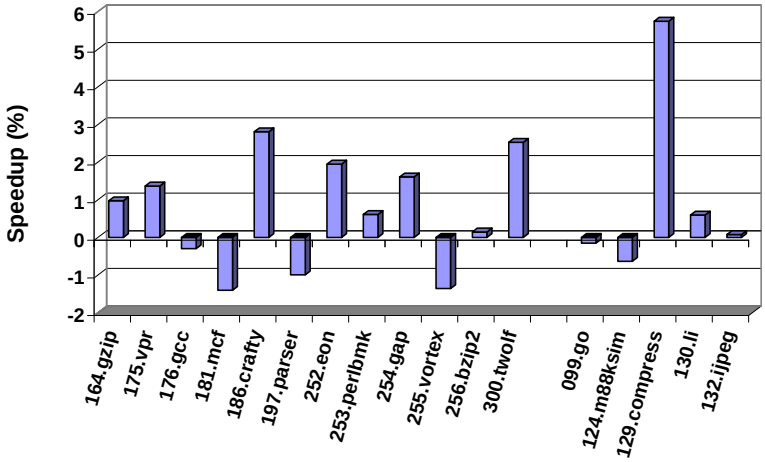


Figure 17: Execution time improvements in benchmarks.

Benchmark	ORC (secs)	ORC+PDE (secs)	Overhead (%)
164.gzip	5.44	5.49	0.92
175.vpr	16.02	16.09	0.44
176.gcc	159.24	162.59	2.10
181.mcf	2.89	2.93	1.38
186.crafty	27.17	27.26	0.33
197.parser	13.35	13.61	1.95
252.eon	136.23	137.09	0.63
253.perlbmk	84.84	86.37	1.80
254.gap	58.74	59.15	0.70
255.vortex	61.27	62.36	1.78
256.bzip2	3.33	3.35	0.60
300.twolf	33.01	33.06	0.15
099.go	24.27	24.71	1.81
124.m88ksim	20.35	20.39	0.20
129.compress	0.77	0.77	0.00
130.li	7.79	7.79	0.00
132.ijpeg	22.99	23.29	1.30

Table II: Compilation overheads "ORC+PDE" over "ORC" (`eon` is not inlined).

out of the 17 benchmarks. The best three speedups are 5.75% and 2.81%, 2.53% which are achieved by `compress`, `crafty` and `twolf`, respectively. The performance degradations for `gcc`, `mcf`, `parser`, `vortex` and `m88ksim` are observed (due to some complex interactions between PDE and other later passes used in the ORC compiler).

As shown in Table II, the implementation of our PDE algorithm accounts for small compilation overheads for all the benchmarks. The benchmarks are cross-compiled on a 2.6GHz Pentium 4 PC with 2GB memory running Redhat Linux 8.0. According to [3], the cross compiler is more stable since "the native compiler has gone through less testing due to lack of resources." There are two main reasons for this efficiency. First, the leaf regions are small as shown in Table III. Second, Table IV lists the number of PDE candidates in a benchmark and shows how far they are moved downwards during assignment sinking and elimination. These statistics together show that our PDE algorithm completes quickly on SEME regions in benchmark programs.

To understand how PDE affects performance, we use `pfmon` to measure dynamic execution statistics through the eight Itanium performance monitors on an Itanium 1 system. Figure 18 presents the dynamic cycles distributed into the Itanium stall categories [18] for each program in both the ORC and ORC+PDE configurations (cf. Table II). Figure 19 shows more clearly the reductions in the eight categories for each program. The dominant category (i.e., the one with the largest cycle reduction in absolute value) is `PIPELINE_BACKEND_FLUSH_CYCLE` for `gzip`, `DEPENDENCY_SCOREBOARD_CYCLE` for `parser` and `perlbmk`, `INST_ACCESS_CYCLE` for `eon` and `gap` and `ISSUE_LIMIT_CYCLE` for `compress`. In the case of `m88ksim`, both `PIPELINE_BACKEND_FLUSH_CYCLE` and `DATA_ACCESS_CYCLE` are reduced

Benchmark	R	P	P/R	M
164.gzip	142	4,335	30.53	147
175.vpr	455	16,142	35.48	162
176.gcc	7,801	211,885	27.16	319
181.mcf	28	1,106	39.50	149
186.crafty	732	26,101	35.66	399
197.parser	501	11,366	22.69	123
252.eon	3,442	111,943	32.52	627
253.perlbmk	2,764	82,192	29.74	787
254.gap	2,308	80,053	34.69	264
255.vortex	2,295	109,237	47.60	229
256.bzip2	107	2,949	27.56	102
300.twolf	705	25,506	36.18	239
099.go	902	29,034	32.19	146
124.m88ksim	536	19,739	36.83	167
129.compress	20	885	44.25	139
130.li	381	10,216	26.81	105
132.ijpeg	521	21,197	40.69	301

Table III: Sizes of innermost regions processed by the PDE algorithm (i.e., PDE regions). For each benchmark, R and P represent the number of PDE regions and the total number of (static) OPs in these regions, respectively. Thus, P/R is the average number of OPs in a PDE region. For each benchmark, M is the maximum number of OPs in a PDE region.

equally more significantly than the other six categories. However, `DATA_ACCESS_CYCLE` enjoys the largest reduction among all the eight stall categories in the remaining 11 benchmarks. Clearly, PDE affects the cycles in the stall category `DATA_ACCESS_CYCLE` more profoundly than the other seven categories. This category counts the number of cycles that the pipeline is stalled when instructions are waiting for the source operands from the memory subsystem. Of the 11 benchmarks for which `DATA_ACCESS_CYCLE` is the dominant (or equally dominant) stall category, the cycles in `DATA_ACCESS_CYCLE` are decreased in `vpr`, `gcc`, `crafty`, `twolf`, `m88ksim`, `compress`, `li` and `ijpeg` but increased in `gcc`, `mcf`, `vortex` and `go`. This phenomenon may be attributed to the aggressive nature of code sinking inherent in our PDE algorithm. By sinking instructions as low as possible along the control flow, the lifetimes are decreased for some variables but increased for the others. However, the overall impact of such a code motion is positive: the performance improvements are achieved in 12 out of the 17 benchmarks.

6. Related Work

Region-based compilation may potentially reduce expensive compilation costs by focusing aggressive optimisations on regions rather than functions. The scope of regions ranges from

Benchmark	C	V	V/C	M
164.gzip	1,251	3,468	2.77	74
175.vpr	4,385	10,979	2.50	101
176.gcc	49,813	100,948	2.03	157
181.mcf	363	599	1.65	11
186.crafty	9,104	35,583	3.91	265
197.parser	2,850	6,562	2.30	46
252.eon	26,850	48,692	1.81	208
253.perlbmk	17,943	47,440	2.64	97
254.gap	25,830	52,418	2.03	116
255.vortex	20,089	42,700	2.13	95
256.bzip2	802	2,058	2.57	28
300.twolf	8,865	26,978	3.04	188
099.go	8,993	19,887	2.21	95
124.m88ksim	4,871	13,841	2.84	92
129.compress	326	1,002	3.07	39
130.li	1,141	1,881	1.65	68
132.ijpeg	7,746	58,492	7.55	165

Table IV: The number of (static) OPs encountered by PDE candidates during code motions. For each benchmark, C represents the number of PDE candidates and V the number of OPs they encounter during their code motions. Thus, V/C denotes the average number of OPs encountered by a PDE candidate. For a benchmark, M denotes the maximum number of OPs encountered by some PDE candidate.

simple trace [14], superblock [15] and hyperblock [4] to more general multiple-entry multiple-exit (MEME) and single-entry multiple-exit (SEME) regions [1, 12]. Region-based compilation has been used in the IMPACT and ORC compilers [2, 19, 3]. In practice, function inlining is often performed in order to create regions spanning multiple functions [1] and new blocks may be introduced through tail duplication (to turn MEME into SEMEs regions, for example) [4]. Therefore, duplication ratios must be controlled in order to avoid excessive code expansion.

Most existing PRE algorithms [20, 21, 22, 6] and PDE algorithms [8, 9, 7] are developed for non-predicated code. These algorithms are designed to operate directly on the explicit branches in a CFG and are thus not applicable when instructions are predicated.

Some earlier research efforts on performing PRE on predicated code can be found in [23, 24]. In particular, Knoop, Collard and Ju's PRE algorithm [24] is based on SI-graphs. By avoid introducing new predicate defining instructions, their algorithm guarantees that the dynamic count of instructions along any path is not increased. Collard and Djelic [23] introduce a PRE algorithm on a single hyperblock by using first-order logical operations on predicates.

August [25] discusses by an example how to perform PDE for a single hyperblock based on a predicate flow graph (PFG) [26]. The IMPACT compiler [2] supports DCE on predicated code. The ORC compiler performs DCE only on non-predicated IRs. We are not aware of any other

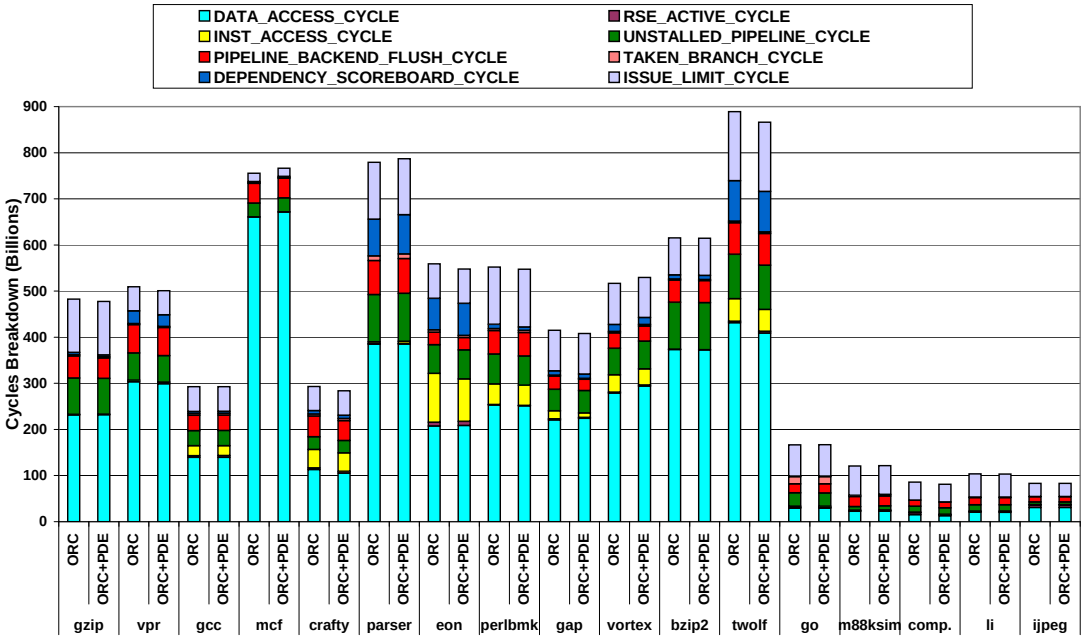


Figure 18: Cycle breakdown in stall categories.

region-based PDE algorithm on predicated code that works uniformly on both hyperblocks and/or regions containing basic blocks and hyperblocks.

Several approaches to predicate analysis have been described in the literature [10, 11, 27]. The predicate query system (PQS) introduced in [10, 11] is based on the PPG. This is the system implemented in the ORC compiler. PQS can accurately represent predication conforming to the style of if-conversion. The Predicate Analysis System (PAS) introduced in [27] is more powerful since it can accurately accommodate arbitrary predicate formulations. Our algorithm can be easily adapted when a PAS-based system is used provided it also supports the queries on control and materialised predicates at the same time.

The existence of dynamically dead codes in application programs has also been dealt with in the microprocessor community [28, 29]. In particular, Butts and Sohi [28] show that the majority of the dynamically dead instructions arise from a small set of static instructions. They use a dead instruction predictor to avoid the execution of predicted-dead instructions. The proposed hardware implementation achieves an average of 0.6% performance speedup for SPEC00 integer benchmarks if their base architecture is used and an average of 3.6% performance speedup if the limited-resource architecture is used instead.

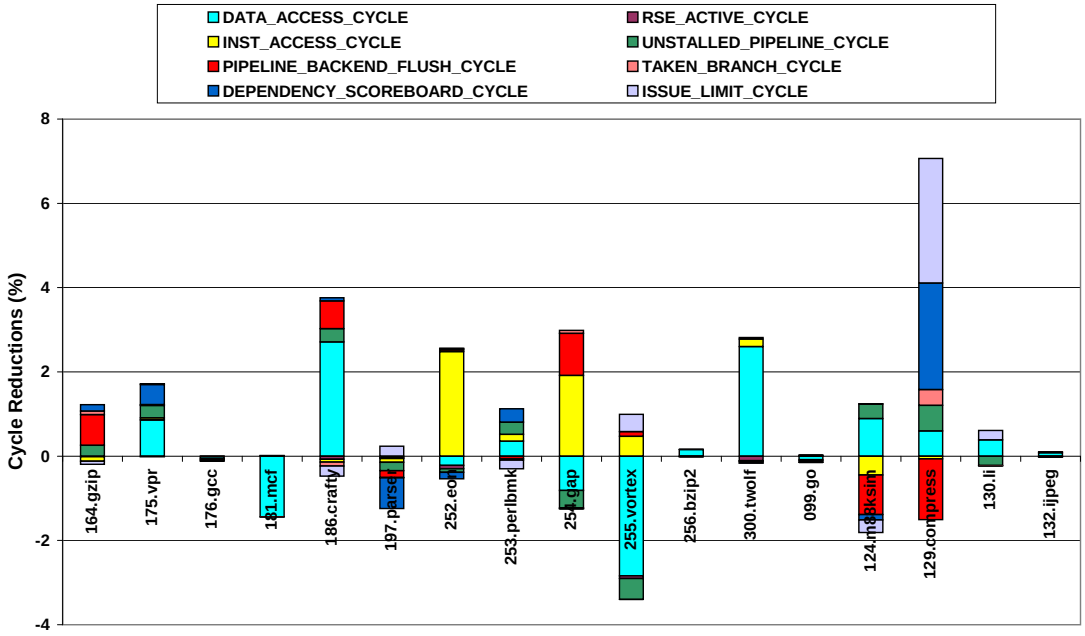


Figure 19: Individual performance improvements of ORC+PDE over ORC measured separately in terms of the eight event categories. An event bar for a benchmark represents the speedup achieved by ORC+PDE over ORC as a result of reducing the cycles in that event category. The eight bars for a benchmark add up to the corresponding speedup given in Figure 17.

7. Conclusion

Region-based compilation increases scheduling opportunities, which can be critical for improving the performance of programs running on ILP architectures. Predicated execution on these architectures is an effective technique for dealing with conditional branches. The contribution of this research is the development of a practical algorithm for performing region-based PDE on predicated code. This algorithm is optimal in the sense that it can eliminate all partial deadness that can be removed without changing the branching structure of the program or potentially introducing new predicate defining instructions. We have implemented this algorithm in the ORC compiler for Intel's Itanium Processor family. In our implementation, PDE is applied just before instruction scheduling. This strategy not only eliminates all partial deadness but also achieves an overall effect of reducing the cycles distributed into the Itanium stall categories on Itanium 1. We present statistical evidence about the PDE opportunities in SPEC benchmark programs. We demonstrate that our PDE algorithm can achieve performance improvements in application programs. The implementation of our PDE algorithm accounts

for only a small fraction of the total compile time for each benchmark. Therefore, our algorithm can be used as a practical pass in a region-based compiler for EPIC architectures.

8. Acknowledgements

We wish to thank the reviewers for their comments and suggestions on the initial version of this paper. This work is supported in part by an ARC Grant DP0452623.

REFERENCES

1. Richard E. Hank, Wen-Mei Hwu, and B. Ramakrishna Rau. Region-based compilation: an introduction and motivation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 158–168, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.
2. P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and Wen-Mei Hwu. Impact: An architectural framework for multiple-instruction-issue processors. In *18th International Symposium on Computer Architecture (ISCA)*, pages 266–275. IEEE Computer Society, 1991.
3. ORC. Open Research Compiler for Itanium Processor Family, 2005. http://ipf-orc.sourceforge.net.
4. Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, New York, NY, USA, 1992. IEEE Computer Society Press.
5. J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, New York, NY, USA, 1983. ACM Press.
6. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: theory and practice. *ACM Transactions on Programming Languages and Systems*, pages 147–158, 1994.
7. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 147–158, New York, NY, USA, 1994. ACM Press.
8. Rastislav Bodik and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 159–170, New York, NY, USA, 1997. ACM Press.
9. Rajiv Gupta, David A. Berson, and Jesse Fang. Path profile guided partial dead code elimination using predication. In *5th International Conference on Parallel Architectures and Compilation Techniques*, pages 102–113, Washington, DC, USA, 1997. IEEE Computer Society Press.
10. David M. Gillies, Roy Ju, Richard Johnson, and Michael Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 114–125, Washington, DC, USA, 1996. IEEE Computer Society Press.
11. Richard Johnson and Michael Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 100–113. IEEE Computer Society Press, 1996.
12. Yang Liu, Zhaoqing Zhang, Ruliang Qiao, and Roy Ju. A region-based compilation infrastructure. In *Proc. of the 7th Workshop on Interaction between Compilers and Computer Architectures*, pages 75–84, Washington, DC, USA, 2003. IEEE Computer Society Press.
13. Jay Bharadwaj, Kishore Menezes, and Chris McKinsey. Wavefront scheduling: path based data representation and scheduling of subgraphs. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 262–271, Washington, DC, USA, 1999. IEEE Computer Society Press.
14. Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE transactions on Computers*, 30(7), July 1981.
15. Wen-Mei Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M.

Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.
16. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
17. pfmon. http://www.hpl.hp.com/research/linux/perfmon/pfmon.php4, 2005.
18. Intel. Intel Itanium processor reference manual for software development, December 2001.
19. Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for a Java just-in-time compiler. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 312–323, New York, NY, USA, 2003. ACM Press.
20. Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 1–14, New York, NY, USA, 1998. ACM Press.
21. Qiong Cai and Jingling Xue. Optimal and efficient speculation-based partial redundancy elimination. In *Proceedings of the international symposium on Code generation and optimization*, pages 91–102, Washington, DC, USA, 2003. IEEE Computer Society Press.
22. Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, and Peng Tu. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999.
23. Jean-Francois Collard and Ivan Djelic. A practical framework for redundancy elimination on EPIC processors. Technical Report 2000/02, PRiSM, 2000.
24. Jens Knoop, Jean-Francois Collard, and Roy Ju. Partial redundancy elimination on predicated code. In *Proceedings of the 7th International Static Analysis Symposium*, London, UK, 2000. Springer-Verlag.
25. David Isaac August. *Systematic Compilation For Predicated Execution*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.
26. David I. August, Wen-Mei Hwu, and Scott A. Mahlke. A framework for balancing control flow and prediction. In *30th ACM/IEEE International Symposium on Microarchitecture*, pages 92–103, 1997.
27. John W. Sias, Wen-Mei Hwu, and David I. August. Accurate and efficient predicate analysis with binary decision diagrams. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 112–123, New York, NY, USA, 2000. ACM Press.
28. J. Adam Butts and Guri Sohi. Dynamic dead-instruction detection and elimination. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 199–210, New York, NY, USA, 2002. ACM Press.
29. A. Yoaz, R. Ronen, R. Chappell, and Y. Almog. Silence is golden? In *Proceedings of Work-in-Progress Workshop in conjunction with the 7th Symposium on High Performance Computer Architecture*, Washington, DC, USA, 2001. IEEE Computer Society Press.