Region-Based Partial Dead Code Elimination on Predicated Code

Qiong Cai[1], Lin Gao[2†] and Jingling Xue[1‡]

[1] School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia
[2] Institute of Computing Technology, Chinese Academy of Sciences

Abstract. This paper presents the design, implementation and experimental evaluation of a practical region-based partial dead code elimination (PDE) algorithm on predicated code in an existing compiler framework. Our algorithm processes PDE candidates using a worklist and reasons about their partial deadness using predicate partition graphs. It operates uniformly on hyperblocks and regions comprising basic blocks and hyperblocks. The result of applying our algorithm to an SEME 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 statistic evidence about the PDE opportunities in the 12 SPECint2000 benchmarks. In addition to exhibit small compilation overheads, our algorithm achieves moderate performance improvements in 8 out of the 12 benchmarks on an Itanium machine. Our performance results and statistics show the usefulness of our algorithm as a pass applied before instruction scheduling.

1 Introduction

EPIC architectures such as the Intel IA-64 combine explicit instruction-level parallelism (ILP) with instruction predication. In order to realise their performance potential, the compiler must expose and express increasing amounts of ILP in application programs. Region-based compilation as proposed in [12] and implemented in the IMPACT [8] and ORC [19] compilers repartitions the program into regions to replace functions as the units of compilation. By exploiting the static/dynamic profile information, the compiler can successfully create regions that more accurately reflect the dynamic behavior of the program. In particular, by forming regions containing cycles across the function boundaries, the potential for the compiler to expose more ILP is increased [12, 20]. By selecting the sizes and contents of regions appropriately, the compiler can also tradeoff compilation cost and the use of aggressive ILP techniques.

† Lin Gao is a Visiting Research Associate at the University of New South Wales since February 2003. She is supported by an ARC Grant DP0211793.
‡ Jingling Xue gracefully acknowledges Intel's donation of Itanium computers.

Predicated execution, supported by EPIC architectures, allows parallel execution of instructions from multiple control paths and aids in efficient instruction scheduling. In this architectural model, an instruction may be guarded by a boolean predicate, which 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 [1], 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 new, 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. Dead code appears frequently as a result of the optimisations applied earlier, including partial redundancy elimination (PRE), strength reduction and global copy propagation. PDE, which subsumes the dead code elimination (DCE), is an aggressive global optimisation. It is harder than PRE due to the second-order effects clearly exemplified in [18].

Existing PDE algorithms [5, 11, 18] are developed for non-predicated code assuming functions as compilation units. These algorithms are inadequate when instructions are predicated. Consider the following code snippet:

```
x = a + b (p1)
y = x + 1 (p2)
x = c - d (p3)
```

Being insensitive to the three predicates, classic PDE algorithms cannot determine if the first assignment is partially or fully dead or even dead at all.

The contributions of this work are summarised as follows:

- We introduce a practical region-based PDE algorithm on predicated code. By using the notion of Predicate Partition Graph (PPG) [10, 14], our algorithm operates uniformly on predicated blocks (known as hyperblocks [20]) and SEME (Single-Entry Multiple-Exit) regions comprising basic blocks and hyperblocks.
- The result of applying our algorithm to a region is optimal: partially dead code in the resulting program cannot be removed without changing the branching structure of the program or potentially introducing non-existent predicate defining instructions, which impair some program executions.
- We have implemented our PDE algorithm in the ORC compiler [19]. We apply PDE just before its instruction scheduling pass since doing so can potentially reduce critical path lengths along frequently executed paths [5]. We present statistic evidence about the PDE opportunities in all the 12 SPECint2000 benchmarks despite that DCE has been applied several times earlier. We obtain moderate performance improvements in 8 out of 12 benchmarks on an Itanium machine at small compilation overheads.

Our running example is the CFG depicted in Figure 1, where the three regions formed are highlighted by dashed lines. Suppose the compiler applies if-conversion [1] to eliminate the branches in block 2. This leads to Figure 1(b), where the blocks 2 – 5 have been merged into a new hyperblock HB. In addition, we have inserted the so-called *interface blocks* [4], 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 depicted in Figure 1(b). To begin with, the assignment $x = b + d$ 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. At this stage, we obtain Figure 1(c). For the other three assignments, our algorithm first sinks $a = c + e$ into blocks I_2 and I_3 to get Figure 1(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 1(a)). 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 just after the predicate defining instruction for p_4. Figure 1(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 1(f): $x = b + c$ has been removed from block 1 and then inserted into blocks I_1 and I_2.

The rest of this paper is organized as follows. Section 2 gives the background information about regions, predication, if-conversions and PPGs. Section 3 presents our PDE algorithm. Section 4 proves its correctness, termination and optimality. In Section 5, we discuss the implementation of our algorithm in ORC and present our performance results for the SPECint2000 benchmarks. Section 6 compares with the related work. Section 7 concludes the paper.

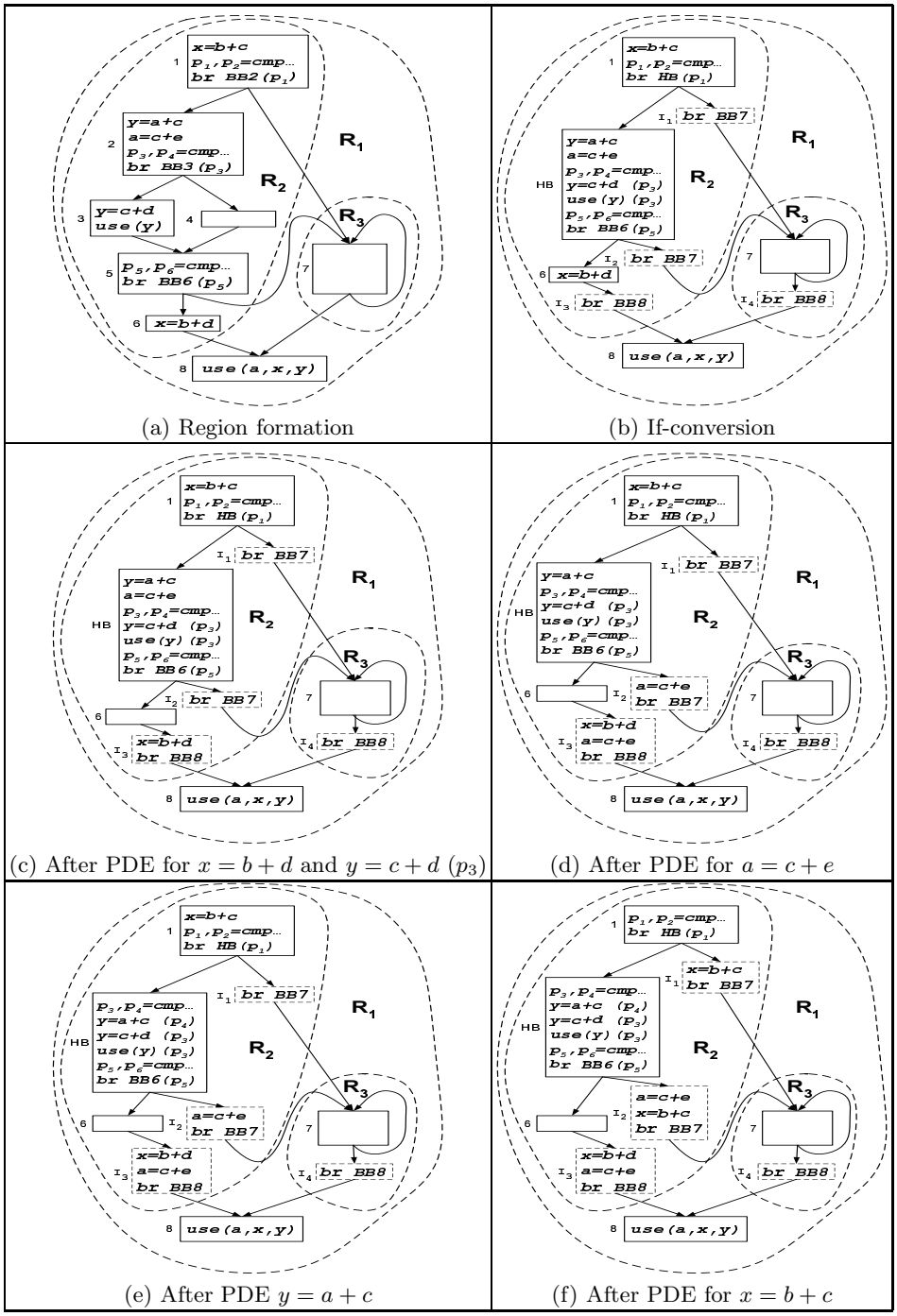
2 Preliminaries

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 the Intel's Itanium Processor family. Section 2.1 defines the regions handled by our algorithm and discusses if-conversion and regional CFGs. Section 2.2 introduces PPGs and the queries used by our algorithm.

2.1 Regions

The ORC compiler [19] uses a region formation algorithm similar to interval analysis. As a result, the division of a CFG into regions puts a hierarchical structure on the CFG, called the *region tree*. There are four kinds of regions: (1) loop regions, (2) SEME 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 every such an MEME into multiple SEMEs (with tail duplication [20] if necessary).

Consider the three regions formed in Figure 1(a). The leaf R_2 is an 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.

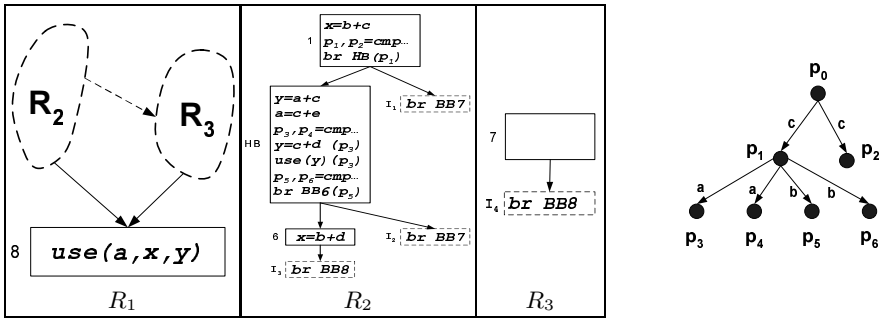


(a) Region formation (b) If-conversion

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

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

Fig. 1. A running example.



(a) Regional CFGs (b) PPG for R_2

Fig. 2. Regional CFGs and the PPG for R_2 in our running example.

We assume that if-conversion is applied to the regions created during the region formation phase wherever appropriate. Often, an SEME subregion within a region is fully converted into a single, branch-free block of predicated code. The resulting block is known as a *hyperblock* [20].

Our algorithm operates on a region by traversing the nodes in its regional CFG. Figure 2(a) shows the regional CFGs for our example. As in the ORC compiler, the back edge around block 7 is not included in the regional CFG of the loop region R_3. This is because back edges cannot be handled by if-conversion. Note that for each leaf region, we have inserted the interface blocks [4] (depicted in dashes), one for each of its region exits. As a result, the successors of every original node in a leaf region must all reside in that region.

In this study, we restrict ourselves to two kinds of leaf regions: SEMEs and innermost loop regions. So the innermost loop regions are considered as SEMEs. Therefore, our PDE algorithm is applicable to SEMEs only.

2.2 Predicate Partition Graph

Our algorithm eliminates partial deadness in an SEME region using the unique Predicate Partition Graph (PPG) [10, 14] associated with the region. From now on, the term block means either a basic block or a hyperblock. The PPG tracks uniformly control flow and explicit use of predicates in a region. A predicate assigned to a block is called a *control predicate* and a predicate which explicitly 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. By convention, the control predicate of the unique entry block of an SEME region is p_0, which denotes the always true predicate. Thus, every instruction can put into the following predicated form:

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

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

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

- BB_α: the block in which α resides
- $\mathsf{M\text{-}PRED}(\alpha)$: the materialised predicate of α
- $\mathsf{C\text{-}PRED}(BB_\alpha)$: the control predicate of BB_α
- $\mathsf{E\text{-}PRED}(\alpha)$: the *executing predicate* of α such that α is executed iff it holds:

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

Consider the regional CFG for R_2 shown in Figure 2(a), which consists of three blocks 1, HB and 6. We find that $\mathsf{C\text{-}PRED}(1) = p_0$, $\mathsf{C\text{-}PRED}(HB) = p_1$ and $\mathsf{C\text{-}PRED}(6) = p_5$. Consider the two instructions $a = c + e$ and $y = c + d$ (p_3) in HB. We find that $\mathsf{M\text{-}PRED}(a = c + e) = p_0$ and $\mathsf{M\text{-}PRED}(y = c + d\ (p_3)) = p_3$. Thus, $\mathsf{E\text{-}PRED}(a = c+e) = \mathsf{C\text{-}PRED}(HB) = p_1$ and $\mathsf{E\text{-}PRED}(y = a+c\ (p_3)) = p_3$.

In a PPG, a node represents a predicate p and a directed edge (p, q) represents that there exists a partition r in p such that q is a subset of r [10, 14]. Figure 2(b) 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: $p_1 = p_3 \cup p_4$ and $p_1 = p_5 \cup p_6$.

Our PDE algorithm relies on the following queries on a region's PPG. Our illustrating example is R_2 shown in Figure 2(a) and its PPG in Figure 2(b).

- $\mathsf{IsDisjoint}(p, q)$: asks whether the domain of predicate p overlaps with that of predicate q. Two predicates are disjoint iff they can reach a common ancestor in the PPG through different edges of the *same partition*. For example, $\mathsf{IsDisjoint}(p_3, p_4) = \mathsf{IsDisjoint}(p_4, p_2) = \mathsf{true}$ 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, $\mathsf{IsSubset}(p_3, p_1) = \mathsf{IsSubset}(p_3, p_0) = \mathsf{true}$ but $\mathsf{IsSubset}(p_3, p_4) = \mathsf{false}$. 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, 14]. *In our 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 child predicates in a partition appear in $p - q$, then these child predicates are replaced with their parent predicate. For example, $\mathsf{LUB\_Diff}(p_1, p_3) = \{p_4\}$ and $\mathsf{LUB\_Diff}(p_0, p_3) = \{p_2, p_4\}$.

We assume that all the critical edges in the regional CFG of a region have been split. This simplifies the construction of the PPG for the region and the code insertions required in code motion/sinking transformations.

3 Region-Based PDE on Predicated Code

There are two main challenges in designing a 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. Section 3.1 introduces our criterion of optimality. Section 3.2 presents our PDE algorithm. Section 3.3 illustrates it with our running example.

3.1 Scope

Our PDE algorithm operates on SEME regions, one at a time. It achieves a complete removal of partial deadness in an SEME region in the following sense.

Definition 1 (Optimality). *Let α be any instruction in an SEME region R, which is regarded as being distinct from every other instruction in R. After having completed assignment sinking and elimination for all instructions in R, let $\alpha_1, \ldots, \alpha_n$ be all the copies of α in the resulting program, which satisfy:*

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

Such a transformation, which is assumed to be semantics-preserving, is optimal if $\cup_{i=1}^{n} \mathsf{E\text{-}PRED}(\alpha_i)$ is the smallest possible.

Our PDE algorithm guarantees this optimality. Since we apply PDE before instruction scheduling, all necessary optimisations have already been performed. Therefore, we refrain from changing the branching structure of the program. We use only the existing predicates that are subsets of $\mathsf{E\text{-}PRED}(\alpha)$. Thus, we do not introduce any new predicate defining instructions. Implicit in the optimality criterion is that the dynamic count of instructions along any path is not increased.

In comparison with Knoop, Rüthing and Steffen's PDE algorithm on non-predicated code [18], our algorithm shares the same two properties as illustrated in Figures 3(a) and (b). First, we eliminate the partial deadness shown in Figure 3(a) by performing assignment sinking. In the transformed code, the executing predicate $p - q$ for the instruction in block 3 satisfies $p - q \subseteq p$. Second, we do not eliminate the partial deadness shown in Figure 3(b) since p and q are not related. However, we can remove the partial deadness of $x = a + b$ (p) along path 1-3-4 in two ways. If control flow restructuring is used as in [5], 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 may increase the pressure for predicate registers. If restructuring is not used, some new predicate defining instructions may be introduced along some path. As a

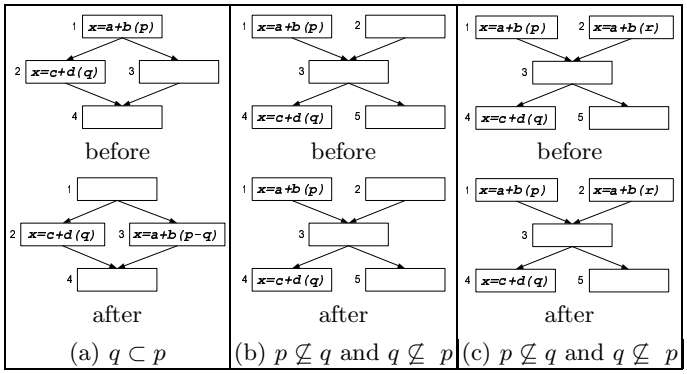


Fig. 3. Scope of our PDE algorithm.

result, the dynamic count of instructions along the path will be increased. This explains the existence of Restrictions 1 and 2 in Definition 1. Unlike [18], however, Figure 3(c) shows that we do not remove the partial deadness removable by simultaneously sinking two distinct instructions that happen to be identical. We can modify our algorithm slightly to deal with this scenario but have not done so since this case should not occur frequently in real code. This is why in Definition 1 all instructions are regarded as being distinct. Finally, Restriction 3 in Definition 1 is a basic requirement of any PDE algorithm.

3.2 Algorithm

Both code motion and deadness are governed by data dependences. There are three kinds of dependences between two distinct instructions: $\alpha : v = \pi$ and δ:

- $\mathsf{DEFINED}(v, \delta)$: v is modified by δ in BB_δ.
- $\mathsf{USED}(v, \delta)$: v is used by δ in BB_δ.
- $\mathsf{KILLED}(\pi, \delta)$: some operands of π are modified by δ in BB_δ.

which are used to define the two predicates used directly by our algorithm:

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

Figure 4 gives our algorithm, called *PPDE*, for performing PDE on an SEME region consisting of basic blocks and hyperblocks. Line 2 creates the empty interface blocks at all region exits as illustrated for R_2 in Figure 1(b). This ensures that all successors of every original node 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). Line 3 initialises W with all PDE candidates sorted in the reverse

```
 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
11         P = NULL
12         Prev = α
13     β = instruction next to α in BB_α      // β = NULL if α is the last in BB_α
14     if (¬ Sink(R, 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, 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
```

Fig. 4. The PDE algorithm on predicated code.

topological order of their dependences. *PPDE* terminates when W is empty (line 4). The PDE candidates in the worklist are processed sequentially, one at a time. As the algorithm proceeds, the candidates are removed from and new candidates added only to the beginning of the worklist in lines 43, 71 and 79. Thus, it is sufficient to understand how one PDE candidate is processed.

In lines 5 – 6, we remove the first PDE candidate α from the worklist. Like the existing PDE algorithms [5, 11, 18], we make use of live-in/out and def-use chains and assume that this information is updated wherever appropriate. Hence, in lines 7 – 8, we delete α when it is fully dead and are done with this instruction. Otherwise, line 14 calls *Sink* to perform sinking and elimination in the block containing α. If this returns false, we continue the PDE process for α in the descendant blocks BB rooted at BB_α in region R. We skip every descendant block that is not on a path starting from α since α cannot be partially dead there (line 16). The PDE process for α is completed when the call to *Sink* in line 18 returns true or when all such descendant blocks have been processed. Then the same PDE process repeats on the next candidate in the worklist.

Lines 10 – 12 are concerned with sinking the multiple copies of an instruction created at a branching node. This will be explained in Section 3.3.

At the end of *PPDE*, we do two things. In line 20, we clear up the regional CFG by deleting all empty blocks such as block 6 in Figure 1(c). 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 [5].

```
22 PROCEDURE Sink(R: SEME Region, α, β: Instruction)
23 p = E-PRED(α)
24 for(δ = β; δ ≠ NULL; δ = instruction next to 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-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 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-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, α, δ)
46             Delete α from BB_α
47             return true
48 Case 3 else if p ⊂ q
49             if DEP(α, δ)
50                 if DEFINED-NOT-USED(α, δ)
51                     Delete α from BB_α                    // α is fully dead
52                     DelInst(P, α)
53                 return true
54             else
55                 if AddInst(P, α, δ)
56                     return true
57 Case 4 else
58             if DEP(α, δ)
59                 return true
60 return false
61 PROCEDURE DelInst(P: Forest, α: Instruction)
62 Delete the node α and its parent if α is its unique child from P (if α exists)
63 PROCEDURE AddInst(P: Forest, α, δ: Instruction)
64 if the node α does not exist in P
65     Add the directed edge δ → α to P
66     Let α_1, ..., α_n be all children of δ in P
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 P
73         Delete α_1, ..., α_n from BB_{α_1}, ... BB_{α_n}
74         return true
75  return false
```

Fig. 4. The PDE algorithm on predicated code (cont'd).

```
76 PROCEDURE CompensationInsert(R: SEME Region, 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     BB = the block in which r is defined
81     if BB = BB_δ          // BB_δ is a hyperblock
82         Insert θ after the predicate defining instruction for r
83     else
84         for each successor BB_s of BB in the region CFG of R
85             if C-PRED(BB_s) = r
86                 Insert θ at the entry of BB_s
87                 break
```

Fig. 4. The PDE algorithm on predicated code (cont'd).

3.2.1 Sinking and Elimination

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. 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, the executions of α and δ are mutually exclusive. Thus, α is not partially dead with respect to δ. The next iteration of **for** loop in line 24 is executed. Otherwise, there are the four cases depending on the relations between p and q. In Case 1, lines 33 – 37 have the effect of moving α into BB_δ when α and δ are in two different blocks. Case 2 is concerned with a branching point while Case 3 a merging point. In Case 4, α cannot be sunk any further if there is a dependence from α to δ.

3.2.2 Compensation Code Insertion

The procedure *CompensationInsert* inserts copies of α at the indicated insertion points with the executing predicates in $\mathsf{LUB\_Diff}(\mathsf{E\text{-}PRED}(\alpha), \mathsf{E\text{-}PRED}(\delta))$.

3.3 Example

Let us trace briefly the execution of our algorithm on the region R_2 shown in Figure 1(b). In line 2, the three interface blocks I_1 – I_3 are created as shown in Figure 1(b). In line 3, $W = (x = b{+}d, y = c{+}d\ (p_3), a = c{+}e, y = a{+}c, x = b{+}c)$, where the instructions have been sorted in the reverse topological order of their dependences. Consider the first PDE candidate $\alpha =_{df} x = b{+}d$ in block 6, where x is live out of R_2. Since $\beta = NULL$ in line 13, the call to *Sink* in line 14 returns false immediately. I_3 is the unique successor of block 6. In line 17, β is set to point to the singleton instruction $brBB8$ in block I_3. During the call to *Sink* in line 18, Case 1 is executed so that $x = b + d$ will be moved into the beginning of block I_3. This is all that can be done for the first PDE candidate. Let us see how

PPDE deals with the second PDE candidate $\alpha =_{df} y = (c + d)$ (p_3) in W. This time, $\beta =_{df} use(y)$ (p_3) in line 13. *Sink* is called in line 14. Case 1 is executed again. Due to the flow dependence between the two instructions, *Sink* simply returns true in line 31. The resulting program so far is depicted in Figure 1(c).

At this time, $W = (a = c + e, y = a + c, x = b + c)$. *PPDE* removes $\alpha =_{df} a = c + e$ from W in lines 5 – 6 and sets $\beta =_{df} p_3, p_4 = cmp...$ in line 13. After lines 10 – 12, we have $P = NULL$ and $Prev =_{df} a = c + e$. In line 14, *Sink* is called. In the first iteration of the **for** loop in line 24, Case 1 is executed since $p = q = p_1$. So $a = c + e$ and $p_3, p_4 = cmp...$ are swapped. In the next iteration, δ points to $y = c + d$ (p_3). Hence, $q = p_3$. Since $p_1 \supset p_3$, Case 2 is executed. Note that DEFINED-NOT-USED(α, δ) = false. Thus, $a = c + e$ is removed from HB and $a = c + e$ (p_4) and $a = c + e$ (p_3) are inserted after $p_3, p_4 = cmp...$ in that order. Both instructions are added to W so that $W = (a = c + e$ $(p_4), a = c + e$ $(p_3), y = a + c, x = b + c)$. Finally, *Sink* returns true to *PPDE*.

Next, *PPDE* removes $\alpha =_{df} a = c + e$ (p_4) from W. As before, $P = NULL$ and $Prev =_{df} = a = c + e$. In line 14, *Sink* is called. Eventually, α is moved just before $\delta =_{df} p_5, p_6 = cmp...$ in Case 1. Since $p = p_4$ and $q = p_1$, we have $p_4 \subset p_1$. Then Case 3 is executed, during which *AddInst* is called. The edge set for the forest P becomes $\{p_5, p_6 = cmp... \rightarrow a = c + e$ $(p_4)\}$. *AddInst* returns false. The remaining PDE process for $a = c + e$ (p_4) does not cause any code motion. When *PPDE* removes $\alpha =_{df} a = c + e$ (p_3) from W, P remains unchanged after lines 10 – 12. In line 14, *Sink* is called. Eventually, δ will point to $p_5, p_6 = cmp...$ in Case 1. Then $q = p_1$. Since $p_3 \subset p_1$, Case 3 is executed, during which *AddInst* is called again. In line 65, the edge set for P becomes $\{p_5, p_6 = cmp... \rightarrow a = c + e$ $(p_4), p_5, p_6 = cmp... \rightarrow a = c + e$ $(p_3)\}$, where $p_3 \cup p_4 = p_1$. In the **if** statement beginning in line 67, $a = c + e$ (p_4) and $a = c + e$ (p_3) are combined and the resulting instruction $a = c + e$ is inserted just before $p_5, p_6 = cmp...$ and into W. Thus, $W = (a = c+e, y = a+c, x = b+c)$. The processing of $a = c + e$ will cause it to be split in Case 2 of *Sink* into two copies that are eventually moved into blocks I_2 and I_3 as shown in Figure 1(d).

At this time, $W = (y = a + c, x = b + c)$. By appling *PPDE* to $y = a + c$, Case 2 of *Sink* is executed. The partial deadness of this instruction is eliminated as shown in Figure 1(e). The result of performing PDE on the last candidate $x = b + c$ in block 1 is shown in Figure 1(f). Basically, $x = b + c$ is split at the end of block 1 such that one copy is moved into I_1 and the other, which is moved into HB, is dealt with in the same way as $a = c+e$ before. The only difference is that $x = b+c$, which is partially dead along path $1 - HB - 6 - I_3$, is eliminated.

Finally, all three interface blocks I_1 – I_3 are not empty. Those at the non-main region exits are moved into the parent region of R_2, i.e., R_1.

4 Correctness, Termination, Optimality

Theorem 1 (Correctness). *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

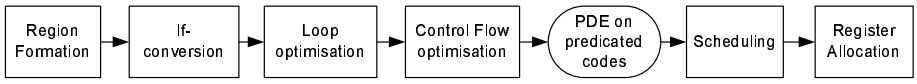


Fig. 5. The code generation (CG) module in ORC with PDE incorporated.

examine Cases 1 – 3 in *Sink*. In Case 1, we delete α because it is fully dead (line 30). The **else** statement 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 correctly by *CompensationInsert* (line 45). Otherwise, α is not partially dead. But we have also inserted a copy of α, called θ, correctly in lines 40 – 42. In Case 3, we delete α because it is fully dead (line 51), and also update the data structure P (line 52). In the **else** statement, we combine the instances of α arriving at a merging point only when the equality in line 67 holds. The semantics of the program is preserved by lines 68 – 74. □

Theorem 2 (Termination). *PPDE terminates.*

Proof. We argue that W, which starts with a finite number of PDE candidates, will be empty. Let α_i and α_{i+1} be two adjacent candidates in W. After having removed α_i from W, *PPDE* will eventually remove α_{i+1} from W. To see this, we note that during the PDE process on α_i, *Sink* can add new PDE candidates only at the beginning of W. There can only be finitely many since R is cycle-free. □

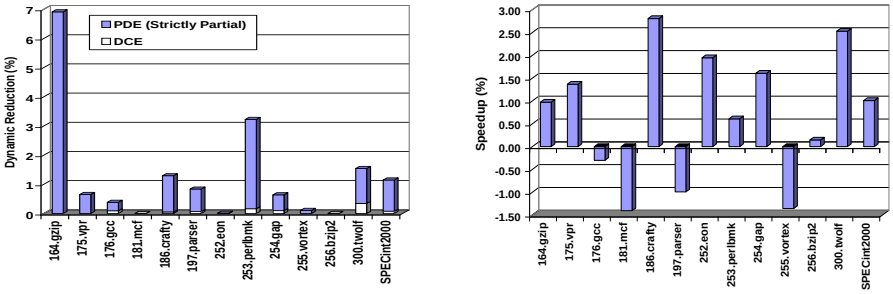
Theorem 3 (Optimality). *PPDE is optimal.*

Proof. As a loop invariant for the **while** loop in line 4, all the PDE candidates in W are always in the reverse topological order of their data dependences. Each PDE candidate is moved downwards as far as possible: some of its copies are deleted iff they are dead, some are blocked due to dependences and the remaining ones are moved into interface blocks. So *PPDE* is optimal by Definition 1.

5 Experimental Results

We have implemented our PDE algorithm in the code generation (CG) module of the Open Research Compiler (ORC) [19] (version 2.1). Figure 5 depicts the compiler framework in which our PDE algorithm is used and evaluated. 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 achieves an overall effect of reducing the cycles distributed into the eight Itanium stall categories (Figure 7).

We evaluate this work using all 12 SPECint2000 benchmarks. The benchmarks are compiled at the "O2" optimisation level with inlining switched on (except for `eon`). Inlining enables the frequently executed blocks in multiple functions to be formed into a single region. The profiling information is collected using the train inputs and is used by ORC in all profile-guided optimisations (including region formation). However, all benchmarks are executed using



(a) Opportunities (b) Speedups

Fig. 6. Opportunities and speedups for SPECint2000.

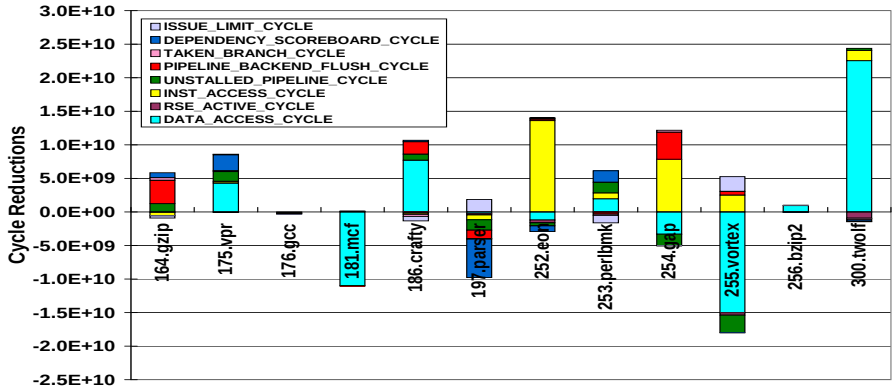


Fig. 7. Cycle reductions in stall categories for SPECint2000.

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` tool.

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 6 shows the benefits of PDE for the benchmarks. In Figure 6(a), we see convincingly the existence of PDE opportunities. This is true even though ORC has applied DCE several times earlier in the compilation process. For each

program, we measure the opportunity as the dynamic count of partially dead instructions eliminated over the dynamic count of PDE candidates executed using the profiling information from the reference inputs. In Figure 6(b), we see the execution time speedups over the ORC configuration. The positive speedups are obtained in 8 out of 12 benchmarks. The speedups for `crafty` and `twolf` are 2.81% and 2.53%, respectively. The performance degradations for `gcc`, `mcf`, `parser` and `vortex` are noted. The overall speedup for SPECint2000 is 1.01%.

The implementation of our PDE algorithm accounts for small compilation overheads for all the benchmarks, which range between 0.41% and 3.40% with an average of 1.41%. The benchmarks are cross-compiled on a 2.6GHz Pentium 4 PC with 2GB memory running Redhat Linux 8.0. There are several reasons for this efficiency. First, the leaf regions are small. Second, the average number of join nodes in the leaf regions is also small. Third, the PDE candidates are processed in the reverse topological order of their dependences. So *PPDE* on a leaf region terminates quickly.

To understand how PDE affects performance, we use `pfmon` to measure dynamic execution statistics through the eight Itanium performance monitors. Figure 7 presents the cycle reductions in the eight Itanium stall categories [13] for each program in both the ORC and ORC+PDE configurations. The dominating 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 `perlmk`, `INST_ACCESS_CYCLE` for `eon` and `gap` and `DATA_ACCESS_CYCLE` for the remaining seven 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 seven benchmarks for which `DATA_ACCESS_CYCLE` is the dominating stall category, the cycles in `DATA_ACCESS_CYCLE` are decreased in `vpr`, `gcc`, `crafty` and `twolf` but increased in `gcc`, `mcf` and `vortex`. 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.

6 Related Work

Most existing PRE algorithms [6, 7, 15, 17] and PDE algorithms [5, 11, 18] are developed for non-predicated code. They are inadequate when instructions are predicated. There are some earlier research efforts on performing PRE on predicated code [9, 16]. In particular, Knoop, Collard and Ju's PRE algorithm is based on SI-graphs, which are not as widespread as hyperblocks. By avoid introducing new predicate defining instructions, their algorithm guarantees that the dynamic count of instructions along any path is not increased. Later, Collard and Djelic [9] introduce a PRE algorithm on a single hyperblock by using first-order logical operations on predicates. They allow the synthesis of new predicate defining instructions. As a result, the instruction count along some path can be impaired.

August [3] discusses by an example how to perform PDE for a single hyperblock based on a predicate flow graph (PFG) [2]. The IMPACT compiler [8] supports DCE on predicated code. The ORC compiler performs DCE only on non-predicated IRs. We are not aware of any earlier region-based PDE algorithm on predicated code that works uniformly on basic blocks and hyperblocks.

Several approaches to predicate analysis have been described in the literature [10, 14, 21]. The predicate query system (PQS) introduced in [10, 14] is based on the PPG. This is the system implemented in the ORC compiler. PQS can accurately represent predication conforming the style of if-conversion. The Predicate Analysis System (PAS) introduced in [21] 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.

7 Conclusion

Region-based compilation increases scheduling opportunities, which is 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 sence 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 the 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 eight Itanium stall categories. We present statistic evidence about the PDE opportunities in real code. We demonstrate that our PDE algorithm can achieve moderate performance improvements for the SPECint2000 benchmarks at small compilation overheads.

References

1. J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189. ACM Press, 1983.
2. David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A framework for balancing control flow and predication. In *30th ACM/IEEE International Symposium on Microarchitecture*, pages 92–103. IEEE Computer Society, 1997.
3. David Isaac August. *Systematic Compilation For Predicated Execution*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.
4. J. Bharadwaj, K. Menezes, and C. McKinsey. Wavefront scheduling: path based data representation and scheduling of subgraphs. In *32nd ACM/IEEE International Symposium on Microarchitecture*, pages 262–271, 1999.

5. Rastislav Bodik and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *ACM SIGPLAN' 97 Conference on Programming Language Design and Implementation*, pages 159–170. ACM Press, 1997.
6. Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant computations. In *ACM SIGPLAN' 98 Conference on Programming Language Design and Implementation*, pages 1–14, 1998.
7. Qiong Cai and Jingling Xue. Optimal and efficient speculation-based partial redundancy elimination. In *1st IEEE/ACM International Symposium on Code Generation and Optimization*, pages 91–102, 2003.
8. 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, New York, NY, 1991. ACM Press.
9. Jean-Francois Collard and Ivan Djelic. A practical framework for redundancy elimination on EPIC processors. Technical report, PRiSM, 2000.
10. David M. Gillies, Dz ching Roy Ju, Richard Johnson, and Michael Schlansker. Global predicate analysis and its application to register allocation. In *29th ACM/IEEE International Symposium on Microarchitecture*, pages 114–125, 1996.
11. 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, 1997.
12. Richard E. Hank, Wen-Mei Hwu, and B. Ramakrishna Rau. Region-based compilation: an introduction and motivation. In *28th ACM/IEEE International Symposium on Microarchitecture*, pages 158–168. IEEE Computer Society Press, 1995.
13. Intel. Intel Itanium processor reference manual for software development, December 2001.
14. Richard Johnson and Michael Schlansker. Analysis techniques for predicated code. In *29th ACM/IEEE International Symposium on Microarchitecture*, pages 100–113, 1996.
15. 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.
16. Jens Knoop, Jean-Francois Collard, and Roy Dz ching Ju. Partial redundancy elimination on predicated code. In *7th International Static Analysis Symposium*. Springer-Verlag, 2000.
17. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.
18. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *ACM SIGPLAN' 94 Conference on Programming Language Design and Implementation*, pages 147–158. ACM Press, 1994.
19. Yang Liu, Zhaoqing Zhang, Ruliang Qiao, and Roy Dz ching Ju. A region-based compilation infrastructure. In *7th Workshop on Interaction between Compilers and Computer Architectures*, 2003.
20. 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 *25th ACM/IEEE International Symposium on Microarchitecture*, pages 45–54. IEEE Computer Society Press, 1992.
21. John W. Sias, Wen-Mei Hwu, and David I. August. Accurate and efficient predicate analysis with binary decision diagrams. In *33rd ACM/IEEE International Symposium on Microarchitecture*, pages 112–123. ACM Press, 2000.