PARTIAL CODE ELIMINATION IN THE IMPACT COMPILER FRAMEWORK

BY

SHANE RYOO

B.S., University of Illinois at Urbana-Champaign, 2000

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

ACKNOWLEDGMENTS

I first would like to thank my advisor, Professor Wen-mei Hwu, for teaching me the fundamentals of computer architecture and optimizing compilers, for giving me the opportunity for graduate study and research in the IMPACT research group, and for his continued guidance and support.

Many thanks go to the members of IMPACT, who have supported and encouraged me through my studies. Special thanks go to John Sias for his mentoring, direction, and advice. Thanks also to Sain-Zee Ueng for his support and our many technical discussions, and to Hillery Hunter, Ronald Barnes, and Erik Nystrom for their encouragement and support. Thanks to past members Marie Conte and Christopher Shannon for their patience and help when I first started my studies. I would also like to thank the numerous past and current members of IMPACT that have contributed to the compiler infrastructure, which enables this research and that of future generations of IMPACT members.

I would like to thank my family for their continued support during my graduate studies.

Finally, I would like to recognize and thank the U.S. Department of Defense, American Society for Engineering Education, the University of Illinois and its Department of Electrical and Computer Engineering for their financial support of my graduate studies.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Programs often contain instructions that are partially redundant, meaning that their results have been previously generated along some, but not all, incoming paths. It is often possible to remove this code and increase performance through *partial redundancy elimination (PRE)* techniques. A symmetric opportunity exists when programs contain instructions whose results are partially dead, or sometimes not used, via *partial dead code elimination (PDE)*. Together, these are termed *partial code elimination (PCE)*. Numerous publications have addressed the PCE problems, either singly or together, and differing by their extent of analysis and transformation. This thesis discusses the implementation and effects of motion-based partial code elimination in the IMPACT compiler framework.

Initial publications on PRE and PDE provided "optimal" motion-based algorithms for code elimination; they are optimal in that no program path is impeded as a result of the code motion [1, 2]. Later works perform more aggressive optimization by utilizing profile information to improve frequent paths at the cost of impairing infrequent ones, or by transforming control flow [3–9]. Both PRE and PDE have been implemented in the IMPACT compiler. The first versions implement the optimal code motion techniques for PCE, based on dataflow published by Knoop et al. [2, 10]. The second versions are more aggressive, using profile information to move code to "unsafe" locations, where they may introduce additional executions along some

paths. These speculative techniques anticipate that the motion will result in a net execution savings. They use architectural features of the Intel Itanium Architecture to enable this motion and maintain correct execution [11]. This thesis will describe in detail the implementation of these optimizations and their effects on an industry-standard benchmark suite, SPECint 2000 [12].

PCE can improve program performance in two ways: reduce computing resource contention in an instruction schedule, and reduce the data dependence height of a series of instructions [13]. These may directly improve performance by lowering instruction count, and may also have synergistic effects with other IMPACT optimizations. Thus, I hypothesize that PCE will be able to achieve performance improvements for the target architecture and applications. However, the experimental results will show that these situations are generally limited and explain why this is the case.

This work proceeds as follows: Chapter 2 briefly describes previously published methods of addressing the PRE and PDE problems, with examples. Chapter 3 presents IMPACT's speculative PRE implementation. Chapter 4 presents a new method for performing PDE, termed profile-directed, predicated partial dead code elimination (P3DE). Chapter 5 explains the experiments that evaluate the PCE algorithms and empirical results. Chapter 6 suggests future work in the area of PCE and related analyses. Chapter 7 makes some important concluding observations.

CHAPTER 2

BACKGROUND

IMPACT performs loop-invariant code motion and global common subexpression elimination, as well as a form of loop-biased register promotion called Loop Variable Migration (LVM), which is described in Chapter 5. Thus, the primary purpose of integrating PCE into IMPACT would be to capture additional opportunities which can be optimized only by PCE methods. Desirable properties of PRE for any environment are relatively low overhead, aggressive code motion, and good control over any negative effects of the optimization, such as increases in register pressure. In addition, PCE should have little code growth; IMPACT performs several optimizations which rely on a certain amount of code expansion to obtain higher performance, and PCE optimizations which signficantly increase code will likely exacerbate the situation. In this section, published PCE methods are surveyed to determine candidates for implementation in IMPACT.

2.1 PRE Methods

Partial Redundancy Elimination, or PRE, is a classical optimization first established by Morel and Renvoise [14], subsuming loop invariant code motion and global common subexpression elimination. A version of the optimzation utilizing unidirectional bit-vector analyses on a graph of individual instructions was published by Knoop et al. [1], which is the basis for

much subsequent PRE work. Later work by Knoop et al. [10] extended the algorithm for basic blocks, and is the basis for the much of the nomenclature and dataflow analyses in this thesis.

2.1.1 Lazy code motion

Knoop et al.'s lazy code motion [1] is considered the seminal work of modern PRE, replicating code to minimize temporary variable lifetimes and using only unidirection dataflow analyses. A significant amount of research has been done to make these dataflow analyses faster, using fewer dataflow analyses, or removing the need to split critical edges (described later).

For this work, an "expression" is defined as a logical, arithmetic, or memory operation on source operands with an arbitrary register destination. Two instructions are computations of the same expression if they have the same operator and identical source operands. In Figure 2.1(a), the expression `x+y` is computed in both blocks `1` and `3`, even though they write to different variables. Motion-based PRE operates by inserting computations of expressions at locations that make partially redundant computations completely redundant, thus enabling their removal. All irredundant computations that reach a redundant computation write a virtual "temporary" register, and redundant computations are "removed" by converting them into moves from the expression's temporary register to the original destination of the computation.

Conceptually, Knoop's method hoists partially redundant computations and hoists them until either the partial redundancy is removed or further hoisting would introduce the expression

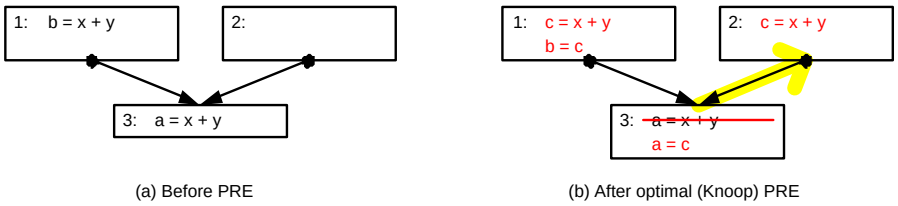


Figure 2.1 Example of PRE

onto a path where it would not have been originally computed. These motions are termed "unsafe." Knoop's algorithm is considered *optimal* since it never introduces additional expression computations onto any path. It is also *nonrestructuring*: it does not transform the flow structure of the program to perform the optimization.

Figure 2.1(a) shows an example of a partial redundancy. The computation of the expression x + y in block 3 is redundant through block 1, but is not redundant through block 2. To perform PRE, the expression is first inserted, or hoisted, into block 2. Next, all expression computations that reach block 3, which are the existing computation in 1 and the newly inserted one in 2, write the result of the computation to a new virtual register c. Finally, the computation in block 3 is replaced with a move operation from c to its original destination a, resulting in the code in Figure 2.1(b).

The primary limitation of Knoop's basic algorithm is that it cannot move expressions to speculative locations, even if a significant savings can come from it, because the optimal algorithm does not take into account program bias. It is generally accepted that programs are often heavily biased and profiles can be quite accurate [15], so more aggressive optimizations may be possible that make the common progam paths faster and improve overall program performance.

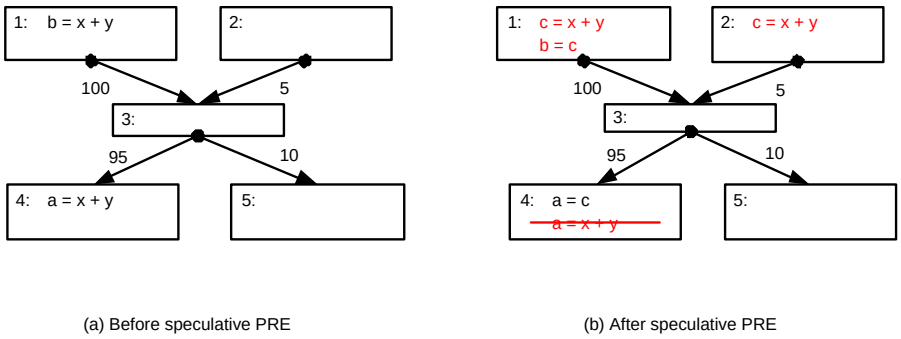


Figure 2.2 Example of Speculative PRE

Consider the code example in Figure 2.2(a). By moving the computation of x + y from 4 to 2, the total number of executions from 195 to 105. However, PRE cannot remove the partial redundancy, because moving the expression to 2 would introduce new computations on the path 2-3-5. A speculative PRE method that takes into account profile information and performs the appropriate cost-benefit analysis could potentially hoist the expression to the block. The next section will discuss methods of performing speculative, motion-based PRE, or *SM-PRE*.

2.1.2 Speculative motion PRE

More aggressive partial redundancy elimination can be performed if expression computations are hoisted to locations where the computation may be introduced into paths where it was not previously computed. This is termed speculative, motion-based PRE, or SM-PRE. SM-PRE can greatly reduce execution of the expressions beyond the capabilities of optimal PRE, but the algorithm may be restricted to moving expressions that will not cause exceptions in the target architecture, or be required to utilize architectural support to avoid or otherwise handle

improper exceptions. In addition, reasonably accurate profile information and a good cost-benefit analysis is necessary to prevent motion that impairs program performance. By hoisting expressions beyond "safe" points, SM-PRE introduce partially dead code into the program.

Consider the example in Figure 2.2(a). As stated in the previous section, Knoop's optimal method cannot hoist the computation of `x + y` from `4` to `2` because it introduces new computations onto the path `2-3-5`. SM-PRE utilizes profile information to identify this motion as profitable and make the transformation, resulting in the code in Figure 2.2(b). If exceptions can occur for additions in the target architecture, the new instruction must be marked to ignore or defer any exceptions.

Horspool and Ho [16] present the first known SM-PRE work, utilizing edge weights for a cost-benefit analysis. They analyze an expression computation and potential insertion-points pairwise, performing the motion if there is a net savings in expression computation execution. For the example in Figure 2.2, their algorithm would recognize and optimize the partial redundancy. However, their method cannot take into account situations involving multiple computations or multiple potential insertion points, which are required for a code motion that minimizes expression computation. Horspool and Ho acknowledge that a minimum cut of a reduced control flow graph may result in a profitable placement for a given profile, but do not demonstrate a technique to do so. Later publications show different techniques, as well as the optimality of this minimum cut algorithm.

Gupta et al. [8] later presented a method to perform a speculative PRE based on path profile information. A cost-benefit analysis is performed by determining the location that minimizes

the sum of the path weights that compute the expression. Although this algorithm has the benefit of rapidly isolating opportunities for optimization, the path dataflow analyses are less intuitive than an edge-based cost-benefit analysis. It also cannot minimize expression computation, as it cannot hoist computations through the backedges of loops. Thus, it is not described in detail here.

Bodik, in his Ph.D. thesis [13], is the first to evaluate the minimum-cut algorithm for speculative motion PRE. He proves its optimality, in terms of minimization of expression computations for a given profile, for nonrestructuring PRE. Cai and Xue [17] later presented a similar algorithm to Bodik's, evaluated as part of a runtime optimizer. Because these techniques perform essentially the same operations and result in the same code motion, the description of the algorithm is saved for Chapter 3. This type of minimum-cut PRE will likely have higher overhead than optimal PRE, but it is also capable of much more aggressive optimization.

2.1.3 Extending PRE

Previous work has addressed methods for matching expressions which differ lexically [18, 19, 20]; these are not covered in this work. I will make observations on the potential benefit for these techniques in the Section 5. Work by Bodik has presented a restructuring PRE [7] that replicates code and transforms control flow to creating regions of full and no redundancy, which are optimized as necessary. His Ph.D. thesis [13] conjectures that a speculative, nonrestructuring, code motion-based PRE is nearly as effective at reducing the amount of redundancy, so the

restructuring algorithm was not implemented. Finally, Lin presents a method for speculating loads past may-alias stores [21]: its potential is discussed in Chapter 5.

2.2 PDE Methods

PDE methods originated in parallel with PRE methods, which have received more attention from the program optimization community. Several papers have presented optimization methods that either directly address the PDE problem or perform a subset of PDE optimizations as a secondary effect of another optimization. These methods are reviewed in this section.

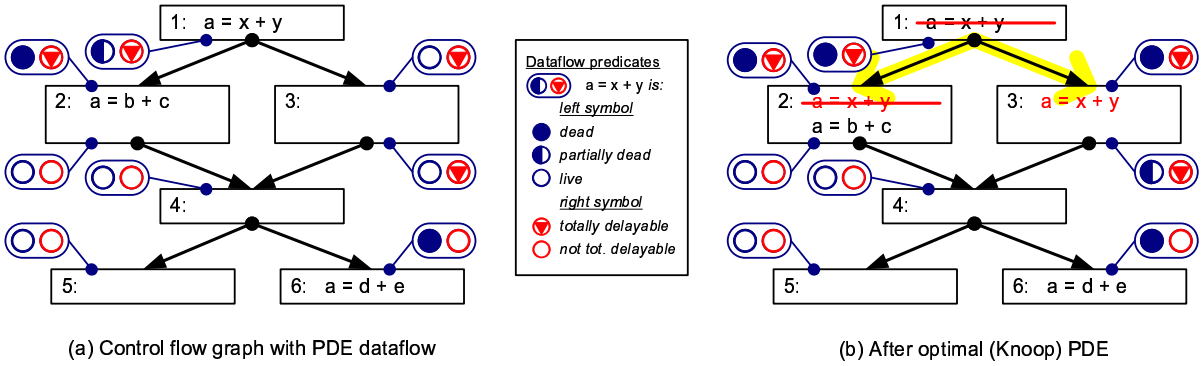
Prior to describing previous PDE schemes, several terms must be defined. The term "assignment" in this work is defined as a register-writing operation or a store to memory. Two instructions are considered to be computations of the same assignment if they write to the same register or memory location, perform the same operation, and have identical operands. An assignment is more strictly defined than an expression, as expressions do not have a restriction on destination operands. In this thesis, the PDE optimization is assumed to operate on assignments by removing those which are partially dead. PDE methods based on code motion operate by "sinking" or "delaying" assignment computations, moving them to later control flow locations; these terms are used interchangably throughout this work. An assignment is sinkable to another location if no intermediate assignment prevents the motion; motion is prevented if an assignment's result is used, a source or destination operand is redefined, or, in the case of memory assignments, another assignment may access the same memory location. Code motion is safe as long as it does not introduce new computations along any path of the

program and only eliminates computations which are completely dead. Although PRE and PDE are problems with many parallel properties, one can have specific requirements for aggressive optimization that the other does not have, due to the differences between expressions and assignments.

2.2.1 Knoop PDE

Knoop et al. [2] provide the most widely recognized, systematic work in partial dead code elimination. Prior works performed limited PDE, often as a byproduct of another optimization. Similar to Knoop's PRE algorithm, Knoop's PDE is both optimal and nonrestructuring.

The method presented by Knoop is composed of two stages. The first stage performs a dataflow analysis to find the latest position(s) at which all assignments can safely be delayed along all paths. It then replicates the assignment computations at those locations. The second stage performs standard dead code elimination, removing the original occurrence of the assignment and any new dead occurrences. In cyclic code regions, if an assignment is computed in every path through a loop body and is not used in the same or subsequent iterations, it can be sunk out of the loop. The sinking and dead code stages are separated because fully-live and sinkable assignment computations may block the motion of partially dead assignments due to antidependence. One disadvantage of the separation of stages is that self-antidependent assignments (those that write to their own operands) cannot be migrated without splitting the assignment into two separate components, since the original computation is not made dead by the copies introduced by the code replication.



(a) Control flow graph with PDE dataflow

(b) After optimal (Knoop) PDE

Figure 2.3 Example of Knoop PDE

Consider the example in Figure 2.3(a). The assignment computation a = x + y in basic block 1 is dead along the program path that passes through 2, but may be live through 3. Using Knoop's method, this assignment can be sunk to blocks 2 and 3. The subsequent dead code elimination removes the computations in 1 and 2, resulting in the code in Figure 2.3(b). Nothing more can be done in this code region, since no assignment computation can be safely sunk without introducing a computation on a path where it did not exist previously.

One disadvantage to Knoop et al.'s method is that it may sink computations without any code removal. This may create nonbeneficial code growth. Motion-based PDE in general also pushes the computations of values closer to their uses, which may expose stalls.

Knoop et al.'s method also has limitations which prevent it from removing all partially dead code. First, it is limited by the structure of the control flow graph: in Figure 2.3(b), the assignments in basic blocks 2 and 3 are partially dead, but since the computations cannot be safely moved into 4, no optimization can be performed by code motion alone. By transforming control flow or utilizing predication, this partially dead code can be removed. However, such transformation has negative tradeoffs, such as potentially exponential code expansion or the

insertion of many predicate operations. In addition, not all cyclic code can be completely unrolled, so some partially dead code can never be entirely removed. In these situations, the PDE optimization is directed by specializing for program bias. Later work attempts to address these issues.

2.2.2 Transformation-based PDE

Just prior to the publication of Knoop's PDE algorithm, Feigen et al. [22] present various examples of partial dead code removal, which they term the *revival transformation*. This involves "detaching" assignment computations and "reattaching" them closer to their uses, under more restrictive control conditions. In some cases, it moves "superstructure" around the partially dead code to enable elimination that is not possible with Knoop et al.'s algorithm.

An example of the restructuring aspect of the revival transformation is presented in Figure 2.4, which uses the same notation as Bodik and Gupta's [5] paper for comparison. Each branch is divided into two components: a calculation of the predicate used to decide branch direction, and the branch itself, which is denoted by the name of its predicate. The assignment `x = a + b` is dead along the `F` path leading from the branch on `P2`, but is live along the T path. Because the assignment is not safely sinkable along the T path of `P1`, Knoop's method cannot be used. The revival transformation first moves the branch and following hammock region of `P1` downward into both the T and F paths of `P2`, resulting in the code in Figure 2.4(b). The assignment computation under `P2`'s false condition is exposed and eliminated, resulting in the code in Figure 2.4(c).

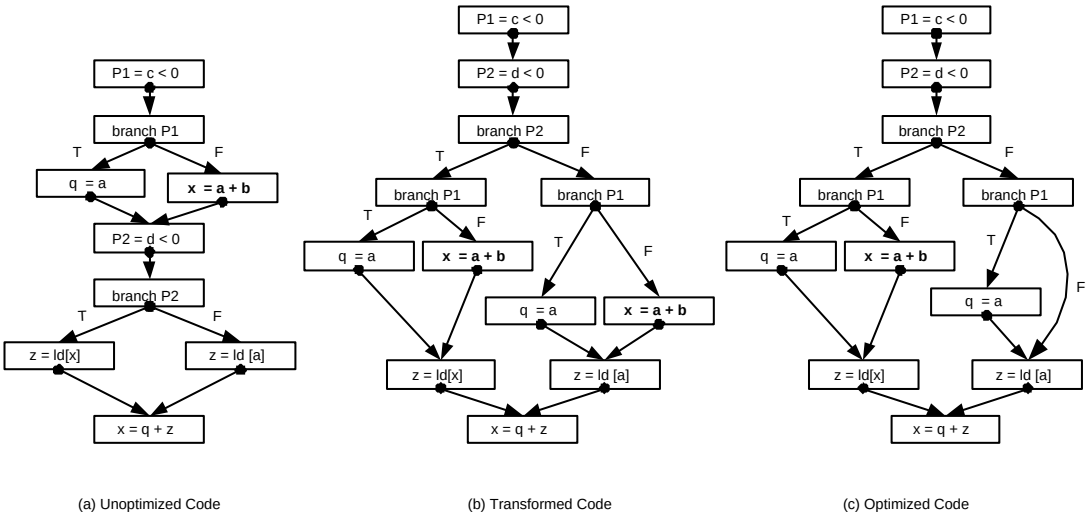


Figure 2.4 The Revival Transformation

Although the authors present some interesting cases and properties of PDE that were not addressed by Knoop et al., such as the restructuring of control flow to expose more PDE opportunities, they do not present a unified algorithm to perform the PDE optimization. The paper also does not move assignments out of or across loops or other cyclic regions, missing significant optimization opportunities. This paper does show some of the possibilities of transformation-based PDE but does not help in performing a global optimization of partial dead code.

Bodik and Gupta [5] later presented a PDE algorithm to address the shortcomings of the revival transformation, called the *slicing transformation*. Figure 2.5 shows an example of their method, using the same code region as the revival transformation to illustrate the differences. Again the computation `x = a + b` is dead along the false path leading from `P2`, but is live along the T path. The slicing transformation replicates and hoists the predicate computation and branch for `P2` around the assignment, resulting in Figure 2.5(b). As a result, the statement

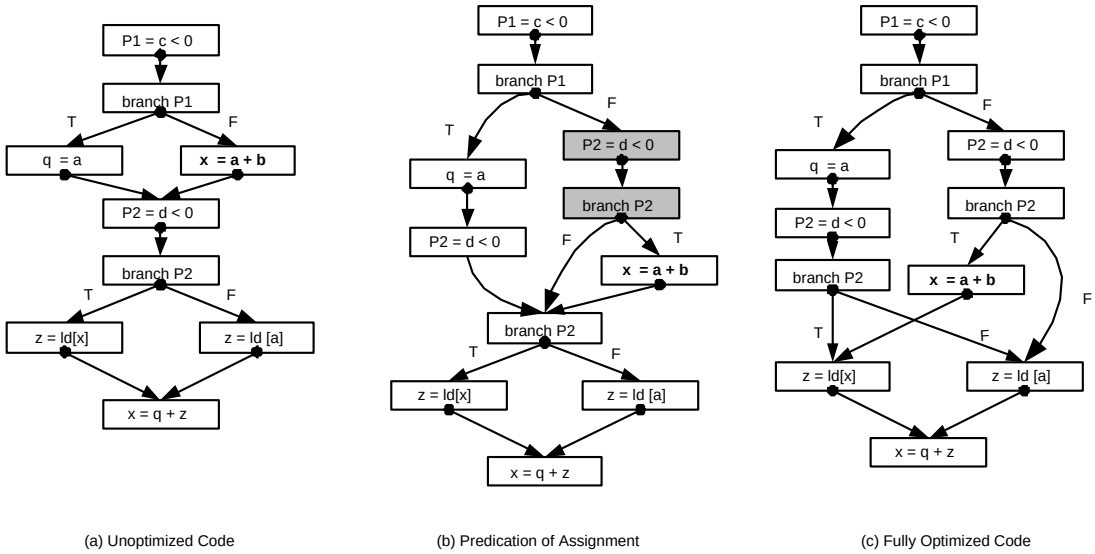


Figure 2.5 The Slicing Transformation

is no longer partially dead, since it is only executed if P2 is true. There is an extra, redundant branch when P1 and P2 are false, so branch elimination [23] is performed to obtain the code in Figure 2.5. Although an extra copy of the P2 predicate computation and branch have been introduced, no additional instructions will be executed along any path and the size of the resulting code is slightly smaller than the result of the revival transformation.

The slicing transformation can address all acyclic partial dead code, at the cost of potentially exponential complexity and code expansion. However, it does not needlessly replicate code which does not contribute to the PDE optimization. Bodik and Gupta also discuss the tradeoffs for removal of cyclic partially dead code. Thus, Bodik and Gupta present a PDE scheme that can remove all acyclic PDE cases without the introduction of additional computationsditional computations along any path and duplicates a targeted, minimal amount of code.

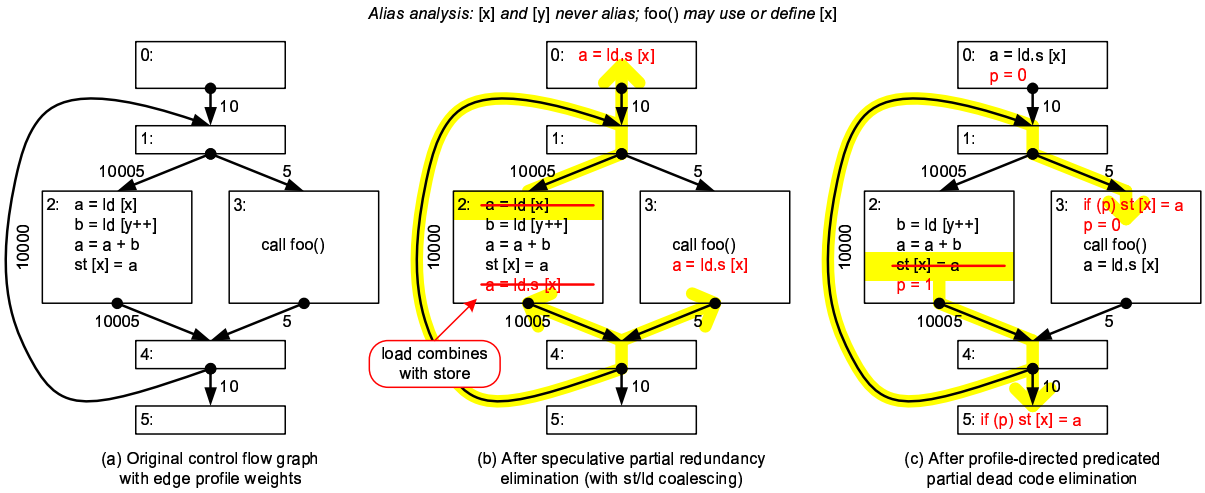


Figure 2.6 Example of Cyclic PDE

Since the slicing paradigm projects future control flow back onto original computation sites rather than sinking computations into “future” control flow, it cannot address cases in which future computations cannot safely be performed earlier. Thus, it does not allow the restructuring of code in loop bodies that is partially dead because of computations in subsequent loop iterations. For example, in Figure 2.6(b), the assignment st[x] = a is partially dead along the left-hand path, but the condition of the branch at the end of block 1 may not be moved to a prior iteration. Only by sinking code can the PDE situation be eliminated. In loops, therefore, Bodik’s approach falls back on a limited amount of replication followed by Knoop’s basic sinking technique. For the example in Figure 2.6(b), the store cannot be removed from the loop by the slicing transformation. It is possible to remove the store from some loop iterations via loop unrolling, but it cannot be entirely removed via transformation-based PDE methods.

2.2.3 Property-oriented expansion PDE

Steffen [24] presents a PDE algorithm based on code expansion in his work on property oriented expansion (POE). His scheme performs code duplication and transformation to turn partially dead assignment computations into completely dead and completely live computations, as opposed to Knoop's code motion. The method is enabled by the transformation of the code into a more abstract transition model. This POE PDE method allows removal of partial dead code that cannot be addressed by generic code motion techniques, but with a worst-case cost of exponential complexity and code expansion. The conversion to and from a transition model also creates complexity, as does the introduction of nondeterminism in the model. Because Bodik and Gupta's transformation method [5] is capable of the same transformations at lower code replication cost and in a more intuitive fashion, POE-based PDE is not as desirable for implementation in IMPACT and is not described here.

2.2.4 Predicate-enabled PDE

Gupta et al. [4] present a path profile-guided partial dead code elimination using predicated instructions. For frequently executed paths in the program, they find the estimated benefits and costs of each computation and delay it to the most profitable location(s). This can introduce calculations of the assignment on paths where it was previously absent, so predication is used to prevent execution on those paths.

Consider the example in Figure 2.7(a). Using profile information, moving the assignment a = x * y from block 3 to 5 will remove the 93 executions of the assignment for the path 1-3-4-6,

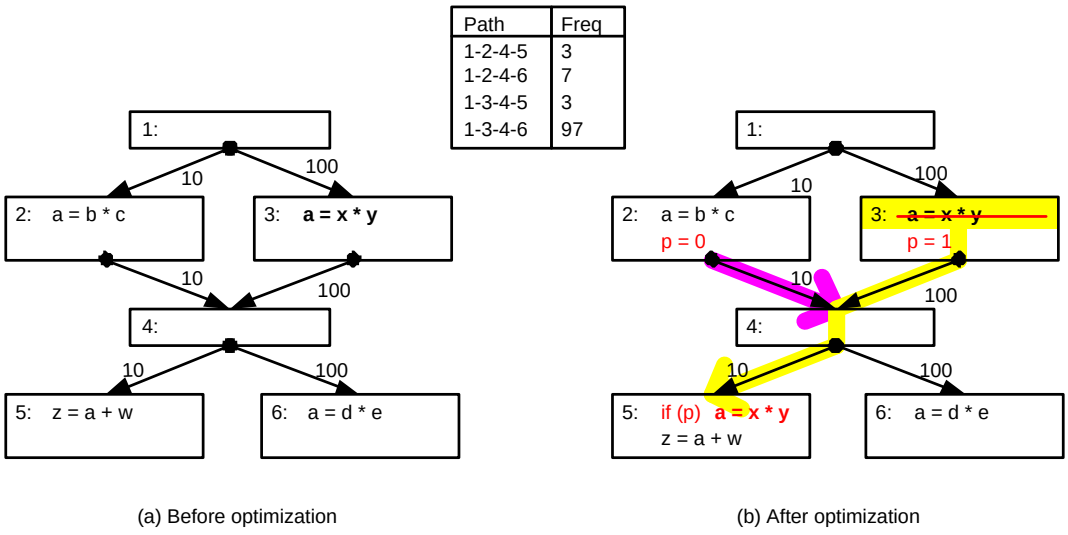


Figure 2.7 Predicated PDE

but introduces 3 executions of the assignment on path 1-2-4-5. Since this is a net savings of 90 executions, it is profitable to perform the optimization; the assignment is moved to block 5, a predicate set instruction is inserted in block 3, and a predicate guard instruction is inserted in block 2. Thus, the number of executions for the assignment is reduced and program correctness is maintained. If an attempt is made to sink the assignment a = b * c from block 2 to block 5, the analysis will find that 7 executions will be saved on path 1-2-4-6, but 7 executions will be incurred on path 1-3-4-5. Since this results in no net execution savings, the transformation would not be performed.

One important note with regard to using path-profile information to perform the cost-benefit analysis is that edge profiling is no less capable of finding the optimal number of computations. This has been shown for the dual PRE problem by both Bodik [13] and later by Cai and Xue [17]; the proofs for the PRE situation apply similarly to the PDE case.

Gupta et al.'s method in its presented form does not recognize the relative cost of predicate operations in its cost-benefit analysis. There are situations where sinking an assignment to reduce execution by a small amount can require a large execution count of predicate clear operations to be introduced, reducing performance of the program. For the example in Figure 2.7, if the weights of edges 2→4 and 4→6 were changed to 1010 and 1100, respectively, Gupta's PDE would result in the same code motion as shown. However, 1010 predicate clear operations would have been introduced for a reduction of 90 in the assignment's execution, which may impair the common path through the code region. Recognizing and controlling this cost is necessary to permit general usage of a predicate-based PDE algorithm.

Another limitation of Gupta's method is that it is conservative within cyclic code regions. It requires that an assignment be dead through a loop back edge in order to sink it past a loop exit. Although this may simplify predicate analysis, it does not allow motion of statements that may be dead on some paths through a loop body but are used in other paths. Consider the example in Figure 2.6(b): the store assignment `st[x] = a` is dead on the frequent, left-hand path of the loop, but is potentially live on the infrequent, right-hand path. It is possible to use predication to sink this store assignment to blocks 3 and 5, obtaining the code in Figure 2.6(c). However, Gupta's technique does not allow the motion of the assignment across the loopback edge 4→1, because `st[x] = a` is live on exit 4→5.

2.2.5 Opportunistic PDE

August et al. [25] mention a PDE scheme that works locally and opportunistically on already-predicated code, rather than actively predicating for the purpose of dead code removal. If a partially dead assignment computation has been included as part of an if-converted region, it can be scheduled after any succeeding branches in its hyperblock if its destination is dead along the taken path of the branch. This may enable a scheduler to generate a more compact static code schedule. In the code example of Figure 2.6(b), the store assignment cannot be sunk because it may be live across the loopback edge.

2.3 Combined PCE Methods

Lo et al. [9] present PRE and PDE for loads and stores, respectively, in the context of register promotion. This work terms the partial dead code problem for stores as a PRE problem, although the concepts involved are still similar to previous PDE work. The hoisting of loads is done within the SSA framework, as specified by Chow et al. in their SSAPRE work [6]. For the PDE of stores, their method uses a static single use (SSU) representation, the dual of SSA form, to identify partially dead stores. As the name implies, the SSU form creates a unique identifier for each used instance of a variable in a program: those assignments that have the same operator and whose results have the same uses can be treated as the same assignment, even though they may have lexically different source variables. This assists in building a graph of stores and their uses. They use this SSU form to perform the *SSUPRE* optimization, which speculatively sinks stores referencing safe locations to the exits of partial-dead regions.

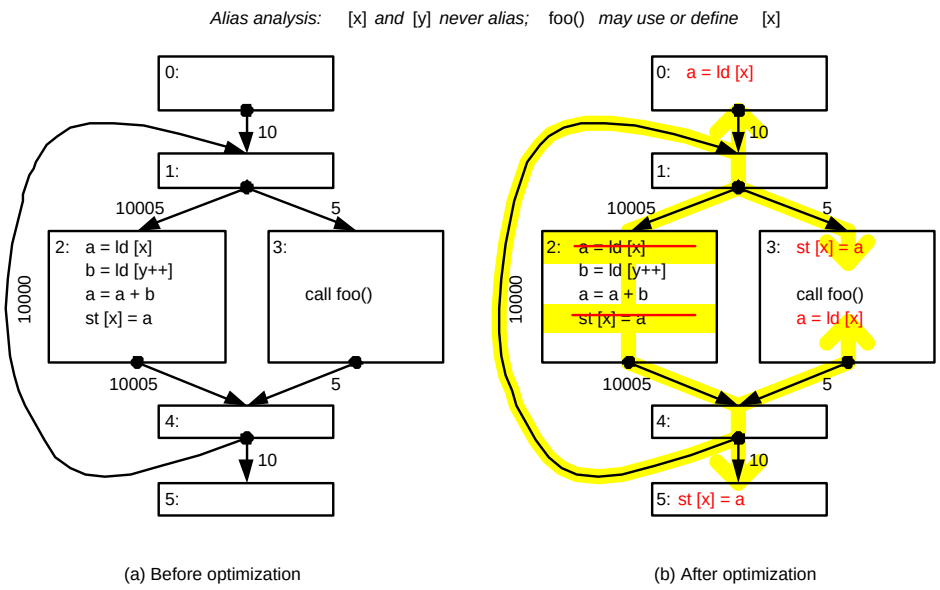


Figure 2.8 Speculative Register Promotion

Consider the example in Figure 2.8(a). There are repeated accesses to the memory location x in block 2, but the function foo() potentially aliases with that memory location, preventing a simple register promotion. However, the hot path through the loop, incorporating blocks 1, 2, and 4, does not have any may-alias memory operations. If x is an address which is always valid (such as a local or global variable), a register promotion can take place in this region, denoted by the yellow-tinted path. The compiler can speculatively insert loads from x to a promotion variable at the entrances of this region (blocks 0 and 3), place stores from the variable to x at the exits (blocks 3 and 5), and change all loads and stores of x within the region to uses of the variable. This results in the code in Figure 2.8(b). The decision to speculate is based on the execution counts at the entrances and exits of the region: if the sum of execution at those locations is less than the current sum, then the memory operations are speculated to those locations, and the memory location is promoted to a register within the region. The execution

of loads and stores to [x] have been reduced from 20 000 to 30, so the transformation is found to be beneficial. Since the register is always loaded at the entrances of the region and always stored back to memory at the exits with no modifications of the corresponding memory location in between, this is a safe transformation.

For PRE on loads, this method operates similarly to other PRE schemes. It has two drawbacks as a general partial dead code elimination scheme. First, when performing speculative motion it performs only an "all-or-nothing" motion in which assignments are moved to the furthest positions possible in the partial redundance/dead code region. It is possible to have an optimization opportunity such that the optimal speculative locations are not at the entrances and exits of the region, but rather within the region. Similar to PRE, a minimum-cut computation is capable of finding the optimal insertion locations for given profile data. Second, Lo's speculative store sinking cannot move memory operations with potentially unsafe addresses and other types of assignments, due to lack of guarding predication. If these problems could be corrected, it would result in a general motion-based PDE scheme which may be more extensive than other existing motion-based PDE methods.

Bodik et al., in work prior to [4, 5], perform a load-store coalescing for array references [3], similar to Lo's method. This method is powerful in that array references have better alias information between each other, due to value-based dataflow analysis that determines precisely which memory operations in a loop iteration and across loop iterations will alias. However, this method only applies to store assignments that reference array elements. It also requires prologue and epilogue code for loops in order to prevent invalid accesses.

Briggs and Cooper [26] present a limited form of PDE in the context of their partial redundancy elimination optimization. Their PRE scheme propagates expressions forward to their use points, in an effort to build larger expressions and associate more computations together. In the PDE context, this ensures that all computations of an assignment are used. However, this may inadvertently impair some program paths by sinking assignments into loops, from which they might not be removed without a speculative PRE algorithm (which introduces partially dead code). Because this method is inherently linked to the PRE algorithm, it will not address partial dead code that is not also partially redundant. Since the store in the cyclic code example of Figure 2.6(b) cannot be hoisted via PRE, it will not be addressed by expression propagation.

2.4 Implementation Candidates

PRE and PDE techniques exist in many forms, varying from Knoop's optimal methods to more aggressive techniques. The optimal and speculative motion-based PRE methods are good candidates for implementation in IMPACT, as they satisfy the criteria established at the beginning of the section. Optimal PDE, modified to limit motion only to partially dead assignments, is also a candidate. However, no PDE method exists in literature that can treat all assignments and control flow structures generally and has minimal code growth. A new form of PDE is necessary to satisfy all these criteria; it is discussed in Chapter 4.

CHAPTER 3

SPECULATIVE PARTIAL REDUNDANCY ELIMINATION

Presented in Figure 3.1 is the algorithm for the IMPACT compiler's SM-PRE optimization. It is an adaptation of Cai and Xue's SM-PRE algorithm [17], modified to operate at basic block granularity, have reduced overhead, enable coalescing of different memory types, and mark all speculated computations. The latter is necessary to speculate potentially excepting expressions.

The SM-PRE algorithm consists of three phases: dataflow analysis, motion graph construction and minimum cut calculation, and code motion. The dataflow analysis provides the information to construct the motion graph, which represents the possible range of motion for all expression computations. Each edge in the motion graph has an associated weight, obtained from control flow graph edge profiling. A minimum cut across this motion graph finds the optimal placement of computations for the expression with the given representation. Finally, SM-PRE performs code motion, utilizing two additional dataflow analyses to find the final positions for new and replaced computations.

Prior to the algorithm, all critical edges (edges connecting a multiple-successor node to a multiple-predecessor node) have been split by inserting a new node in the "middle" of each such edge. This ensures that for each edge there exists a control-equivalent node, either the predecessor or the successor, into which operations may be inserted as if they were inserted onto the edge itself. Cai and Xue assume edge placement.

```
 1: split all critical edges
 2: repeat
 3:    find Avail, PartialAvail, Antic, and PartialAntic
 4:    for each partially redundant expression do
 5:       construct motion graph
 6:       if motion graph not empty then
 7:          find motion graph min-cut
 8:          if min-cut is not at original computation points then
 9:             mark new computations
10:          end if
11:       end if
12:    end for
13:    find Isolated
14:    insert new computations and moves
15:    reaching expression analysis and attribute copying
16:    remove old computations
17:    perform coalescing optimizations
18: until no code motion in current iteration
19: absorb empty blocks resulting from critical edge splitting
```

Figure 3.1 SM-PRE Algorithm

The motivating example for this presentation of SM-PRE is a store-load coalescing performed as part of loop-biased register promotion. The example is shown in Figure 3.2. The expression ld [x] is moved off the hot path of the loop, 1-2-4, into colder sections, effectively promoting the memory location pointed to by x to a register. A PDE algorithm will be used to move the store to x out of the loop.

3.1 Expressions in SM-PRE

SM-PRE operates on all hoistable occurrences of an expression simultaneously and raises them to early program points to minimize execution weight for a given profile. Concurrent examination is necessary to accurately measure the benefit of motion, as hoisting motions of several occurrences may only collectively have a benefit justifying the movement cost.

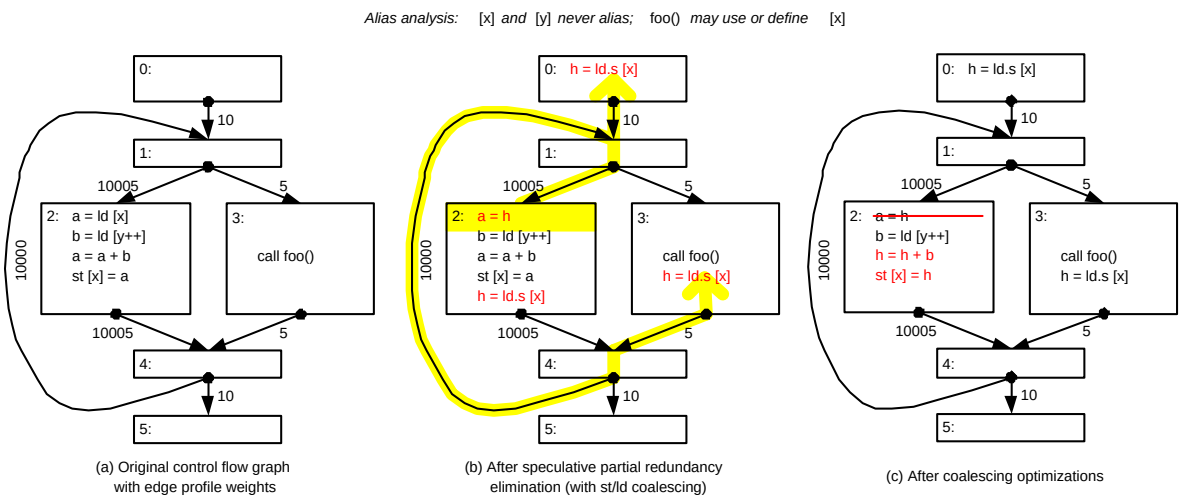


Figure 3.2 Motivating Example for SM-PRE

SM-PRE operates on a low-level, virtual register-based internal representation. An expression can either be a computation which writes to an arbitrary register or a store operation. The cost-benefit analysis operations on one expression at a time, where each expression may consist of multiple computations (each a static instruction). Computations are grouped into the same expression if they perform the same operation and have identical source operands (or destination address, in the case of stores). Commutative operations are expressed in canonical form to enhance matching. A pair of accesses to memory are recognized as being to the same location (must-def/must-use) if they reference the same global label or stack location, or if their operands are the same and there are no intervening writes to those operands. Aliasing (may-use/may-def) information, used to identify code motion constraints relating to other stores, loads, and subroutine calls, is obtained from a previous interprocedural analysis [27].

Because a signed store can supply the value of a signed or unsigned load to the same location, IMPACT's SM-PRE algorithm modifies Cai and Xue's algorithm to allow certain

expressions to "imply" other expressions. A load or store expression implies all other load expressions of the same length or smaller, either signed or unsigned, to the same location. An implied expression is treated as being present for providing the value of an expression, but is not moved. An example of this is shown in Figure 3.2: the expression `st[x] = a` implies the expression `ld[x]`.

3.2 SM-PRE Dataflow Analysis

The SM-PRE dataflow analyses identify the locations where expressions are partially or completely redundant, preparing for the cost-benefit analysis in the second stage of the algorithm. They are bitvector dataflow analyses, run for all all expressions in parallel. Table 3.1 shows the local (single basic block) properties of an expression used in the dataflow analyses. The **Place** properties are found during the minimum-cut computation, while the other properties are calculated before the initial dataflow analyses. In the code example, the expression ld [x] is **N_Comp**, **X_Comp**, and **Implied** in block 2. The expression is not **Trans** in blocks 2 or 3, because the store to [x] aliases and the call to foo() potentially aliases wth the load expression.

There are three types of dataflow analyses performed for SM-PRE, shown in Table 3.2. The first set, the availability analyses, determines the locations where an expression has been previously computed. Availability analysis indicates where an expression has been computed along all incoming paths. Partial availability analysis indicates where an expression has been computed along some incoming paths, indicating where code motion may be profitable. These

Table 3.1 SM-PRE Local Dataflow Sets For Block n

Expression $e \in$	iff
$\mathbf{N\_Comp}_n$	an expression's result is computed in the block and is not blocked from being hoisted to the beginning of the block.
$\mathbf{X\_Comp}_n$	an expression's result is computed or otherwise available in the block but cannot be hoisted to the beginning of the block, due to a blocking memory expression or a definition of a source operand.
$\mathbf{Implied}_n$	an expression's result is implicitly available in the block due to the presence of a different expression. For example, a load expression is implied when a store expression referencing the same memory location is present. This allows load-store coalescing. Expressions that are $\mathbf{Implied}_n$ are also $\mathbf{X\_Comp}_n$.
$\mathbf{Trans}_n$	an expression cannot be moved through the block, due to a definition of a source operand or an aliased memory access.
$\mathbf{N\_Place}_n$	an expression will be computed at the top of the block after code motion is complete: it is either $\mathbf{N\_Comp}_n$ and not profitable for motion or a new computation inserted by the algorithm.
$\mathbf{X\_Place}_n$	an expression will be computed at the end of the block after code motion is complete: it is either $\mathbf{X\_Comp}_n$ or a new computation inserted by the algorithm.

Table 3.2 SM-PRE Global Dataflow Analyses

$$\mathbf{Avail}_m^{\mathbf{out}} = \mathbf{Trans}_m \cap \left(\mathbf{N\_Comp}_m \cup \mathbf{Avail}_m^{\mathbf{in}}\right) \tag{3.1}$$

$$\mathbf{Avail}_m^{\mathbf{in}} = \bigcap_{n \in \mathrm{pred}(m)} \left(\mathbf{X\_Comp}_n \cup \mathbf{Avail}_n^{\mathbf{out}}\right) \tag{3.2}$$

$$\mathbf{PartialAvail}_m^{\mathbf{out}} = \mathbf{Trans}_m \cap \left(\mathbf{N\_Comp}_m \cup \mathbf{PartialAvail}_m^{\mathbf{in}}\right) \tag{3.3}$$

$$\mathbf{PartialAvail}_m^{\mathbf{in}} = \bigcup_{n \in \mathrm{pred}(m)} \left(\mathbf{X\_Comp}_n \cup \mathbf{PartialAvail}_n^{\mathbf{out}}\right) \tag{3.4}$$

$$\mathbf{Antic}_m^{\mathbf{in}} = \mathbf{N\_Comp}_m \cup \left(\mathbf{Trans}_m \cap \mathbf{Antic}_m^{\mathbf{out}}\right) \tag{3.5}$$

$$\mathbf{Antic}_m^{\mathbf{out}} = \mathbf{X\_Comp}_m \cup \bigcap_{n \in \mathrm{succ}(m)} \mathbf{Antic}_n^{\mathbf{in}} \tag{3.6}$$

$$\mathbf{PartialAntic}_m^{\mathbf{in}} = \mathbf{N\_Comp}_m \cup \left(\mathbf{Trans}_m \cap \mathbf{PartialAntic}_m^{\mathbf{out}}\right) \tag{3.7}$$

$$\mathbf{PartialAntic}_m^{\mathbf{out}} = \mathbf{X\_Comp}_m \cup \bigcup_{n \in \mathrm{succ}(m)} \mathbf{PartialAntic}_n^{\mathbf{in}} \tag{3.8}$$

$$\mathbf{Isolated}_m^{\mathbf{in}} = \mathbf{X\_Place}_m \cup \mathbf{Isolated}_m^{\mathbf{out}} \tag{3.9}$$

$$\mathbf{Isolated}_m^{\mathbf{out}} = \bigcap_{n \in \mathrm{succ}(m)} \left(\mathbf{N\_Place}_n \cup \left(\mathbf{Isolated}_n^{\mathbf{in}} \cap \overline{\mathbf{N\_Comp}_n}\right)\right) \tag{3.10}$$

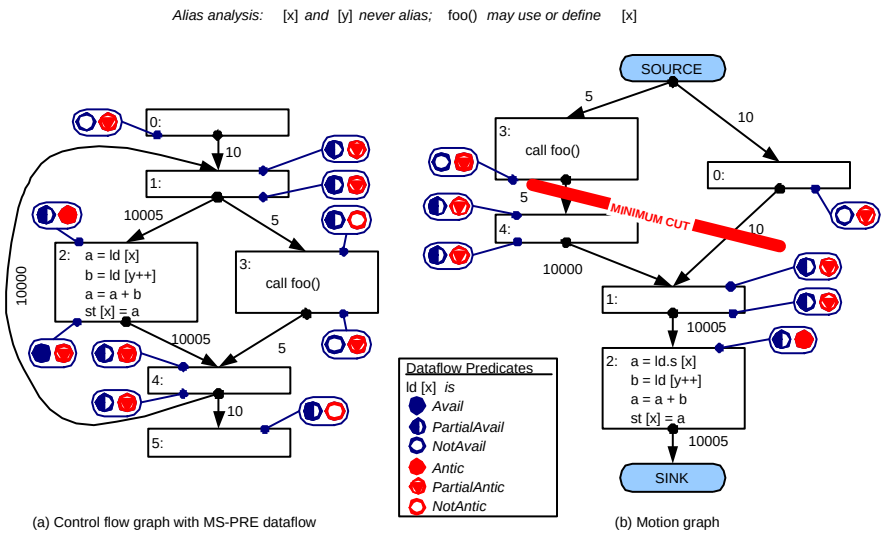


Figure 3.3 SM-PRE Dataflow and Motion Graph

are both forward bitvector propagations, and are described by Equations (3.1) – (3.4). An important note here: an expression computation that is $\mathbf{N\_Comp}$ and $\mathbf{Avail}^{\mathbf{in}}$ is completely redundant, and is marked for removal immediately after the dataflow analysis is completed. These computations are not addressed by the motion graph representation and must be handled in this manner. Figure 3.3(a) shows the results of dataflow analyses on the code example: note that only the exit of block 2 is fully available, all other block entrances and exits are either not available or only partially available. The exit of block 3 is not partially available because the call to foo() may overwrite [x], so the block is not transparent to the expression.

The second set of analyses are the anticipability analyses, indicating where an expression will be computed in the future. Anticipability indicates if an expression will be computed at or after a location. Partial anticipability indicates when an expression may be computed along some future path. If an expression is not partially anticipatable, it cannot be hoisted into the region, excluding it from the motion graph. Both analyses are backward dataflow propagations,

as indicated in Equations (3.5) – (3.8). In the code example in Figure 3.3(a), `ld[x]` is fully anticipable only at the entrance of block 2. It is not partially anticipable at the entrance to block 3 because the call to foo() may overwrite [x].

The final analysis, called isolated analysis, is performed after the cost-benefit analysis and is used for determining proper insertion of writes to expression temporaries. The same analysis is performed by Knoop et al.'s basic-block PDE algorithm [10]. Its use is explained in Section 3.4.

It is important to note that both the **PartialAvail** and **Antic** analyses are not required from a code motion standpoint: the algorithm is capable of placing code in the optimal locations, which Cai and Xue [17] demonstrate. However, the **Antic** dataflow analysis is necessary to find whether an inserted expression is completely anticipatable (will always be calculated) during code motion; if a new computation is inserted at a location where it is not anticipable, then there is at least one execution path through that location that did not compute the expression prior to code motion. If an expression can cause an exception, a new computation of the expression placed at a nonanticipable location must be marked to defer or ignore exceptions to preserve program correctness. The **PartialAvail** dataflow analysis is used to restrict the motion graph to those locations that are partially available and thus potentially profitable. By restricting the motion graph to blocks that are partially available or that border partially available regions, the motion graph can be significantly reduced in size. The same minimum cut closest to the sink node will be found without utilizing this property, but since the minimum cut computation has higher computational complexity than a bitvector dataflow analysis and is performed for every partially redundant expression, this reduction is desirable. In experiments

on the SPECint 2000 benchmark suite this has significantly reduced the algorithm overhead. Cai and Xue do not utilize this property: it is likely that the smaller size of their benchmark suite and the limitations of speculation in a Java framework do not provide as large or as many motion graphs as manifested in the SPEC 2000 benchmark suite.

3.3 The SM-PRE Motion Graph and Minimum Cut

For each expression with some partially redundant computation, a weighted, directed *motion graph* is constructed. This motion graph respresents the potentially profitable hoisting paths from original computations to points where they are no longer redundant. An empty motion graph signifies that an expression has no partial redundancy in the program. A minimum cut of the edges of the graph that renders the sink node unreachable from the source indicates the most profitable location for computation of the assignment. This may occur at the original computation points (indicating that no code motion is profitable for that expression) or at another location, which represents the location for profitable expression hoisting. The motion graph serves as a surrogate for the control flow graph; one of its most important properties is that it treats forward and back edges uniformly, allowing motion through either type of edge.

The motion graph for an expression r is derived using the $\mathbf{Avail}$, $\mathbf{PartialAvail}$, and $\mathbf{PartialAntic}$ sets from the control flow graph. For each edge $e_{u,v}$, $n_u \rightarrow n_v$, in the control flow graph, add $e_{u,v}$ and nodes n_u and n_v to the motion graph iff

$$r \in \left(\mathbf{PartialAntic}_v^{\mathbf{in}} \cap \mathbf{PartialAvail}_v^{\mathbf{in}}\right) - \mathbf{Avail}_u^{\mathbf{out}}$$

Each edge is assigned a weight equal to the corresponding branch or fallthrough weight from profiling. Partial anticipatability indicates that an expression is computed at some later point and can be hoisted into the given edge. Available edges are not profitable for hoisting because expression computations already exist on those paths; conversely, edges that are not partially available are not profitable for hoisting because no redundant computations exist on those paths. Thus, the motion graph is limited to those edges where motion is possible and may be profitable.

Because dataflow analysis is performed at the granularity of basic blocks, a single node can be both an original computation point and a hoisting point for the same expression. For the motion graph to accurately represent this situation, such blocks must be split into an entrance/exit pair: node n is split iff

$$r \in \mathbf{N\_Comp} \cap \overline{\mathbf{Trans}}$$

This edge selection process may result in a multiple-source, multiple-sink graph. To provide the correct format for the minimum cut calculation, the graph is converted to have single source and sink nodes. All nodes that have no incoming edges are connected to the source and and all nodes with no outgoing edges (which must be $\mathbf{N\_Comp}$) are connected to the sink node. The additional edges are assigned weights equal to the execution weights of the basic blocks to which they are attached.

A minimum cut that cuts all possible paths from the source to the sink node represents the optimal location for expression computations. The mechanism used in IMPACT is based

on Goldberg and Tarjan's push-relabel algorithm for finding maximum-flow [28]. It utilizes several heuristics published by Cherkassky and Goldberg [29] to speed computation.

Figure 3.3(b) shows the motion graph for the code example, with the corresponding minimum cut. The minimum cut crosses edges with a combined weight of 15, which is much lower than the original execution of 10 005, resulting in a significant execution savings.

3.4 SM-PRE Code Motion

The minimum cut of the motion graph supplies the desired locations for insertion of hoisted expression computations. A virtual "temporary" register is defined to hold the result of the expression.[1] Expression computations are inserted "on" edges indicated by the minimum cut; they are placed in the control-equivalent blocks exposed by critical edge splitting. These new computations write to the temporary variable. If the location is not **Antic**, then the computation is marked as speculative. Computations which are replaced are those which were not at the minimum cut in the motion graph, as well as those which were found as available by the dataflow.

Figure 3.2(b) shows the result of SM-PRE code motion for the code example. Computations are placed on the edges 0→1 and 3→4, which correlate to the exits of blocks 0 and 3. Since those locations are not anticipatable, the loads must be marked as speculative. A computation must also be placed at 2→4, since `st[x] = a` implies `ld[x]` and must provide the result of the expression.

[1] IMPACT's copy propagation optimizations often replaces existing virtual registers with the temporary register, which exposes more opportunties for expression matching and motion.

In addition to new computations, writes to the expression's temporary variable by preexisting computations must be inserted at all original, not-removed expression computations that can reach a deleted expression. This requires the isolated analysis as shown in Equations (3.9) – (3.10). Moves are inserted for $\mathbf{N\_Comp}$ locations when those are $\mathbf{N\_Place} \cap \overline{\mathbf{Isolated}^{\mathbf{in}}}$ and for $\mathbf{X\_Comp}$ locations when those are $\overline{\mathbf{Isolated}^{\mathbf{out}}}$. The $\mathbf{Implicit}$ predicate indicates when an expression is implied and cannot be lexically matched; these are inserted and removed by subsequent local optimization. No computations in the code example are isolated, so that dataflow analysis is not shown.

Next, a reaching expression dataflow analysis, similar to a conventional reaching definition analysis, finds for each newly inserted computation which to-be-removed computations can be reached. Because the new computation reaches the to-be-removed computation, it was effectively generated by hoisting the to-be-removed computation and must have similar behavior, especially for deferral of or ignoring exceptions. Cai and Xue do not perform this because they do not move expressions which may cause exceptions. After performing the reaching expression analysis and finding all pairs of computations, attributes are copied from the to-be-removed computation to the new computation:

- Synchronization arcs connect memory operations which may conflict. These must be copied to the new computation to prevent illegal reordering of memory operations.
- If a to-be-removed computation is marked as speculative and thus has special exception handling, all new computations that reach it must also be marked as speculative. Note

that the new computations may be completely anticipable, so they would not have been marked speculative previously.

- Other attributes which assist later optimization stages are also copied.

Expression computations are then removed at original computation points that are included in the motion graph but are not at the minimum cut, and at locations where the computation is fully available. The removal consists of converting the expression computation to a move of the expression's temporary variable to the destination of the computation. The temporary variable has been written by previous computations on all paths leading to the removed computation's location, maintaining correctness, and new computations have been marked speculative as necessary to maintain correct execution. This results in the code in Figure 3.2(b). Finally, coalescing optimizations such as copy propagation are performed to propagate the new "temporary" value locally and globally, and a global dead code removal is also performed, exposing new opportunities for PRE. This completes the optimization; for the code example, assuming the variable a is dead outside of the loop, this results in Figure 3.2(c). Note that all occurrences of a have been transformed to h; if a is live after the loop exit and h is not, then all occurrences of h can be changed to a. This is the transformation shown in later sections.

It is well known that PRE can increase register pressure via the introduction of temporary registers. Because SPRE determines profitable expression placement one at a time, it allows for more precise control of register pressure than the entirely parallel optimal PRE algorithm. When register pressure reaches a critical point, further PRE motions are prevented by terminating the minimum cut for expressions and leaving all computations at their original locations,

which requires only a single pass through the control flow graph. Expressions can be sorted so that those most likely to be profitable have their minimum cuts calculated first. Additional virtual registers are introduced by later IMPACT optimization stages, complicating fine-grain tuning of register pressure; this method is only meant to control significant register growth.

CHAPTER 4

PROFILE-DIRECTED PREDICATED PARTIAL DEAD CODE ELIMINATION[1]

As noted earlier, PDE methods are often developed as duals of PRE methods. No dual PDE method has been published for the minimum cut-drive SM-PRE, which inspired the creation of a new profile-driven PDE algorithm. This algorithm is called *Profile-Directed Predicated Partial Dead Code Elimination*, or *P3DE*. It utilizes predication to enable a broad and general scope of transformation. As noted previously, the "optimal" PDE algorithm is only so when one ignores bias in program execution; more aggressive transformation is possible when utiliing an accurate execution profile. Previous methods are constrained by practical limits on code growth, or have safety requirements that may unduly restrict optimization. In particular, most prior methods of PDE have difficulty dealing with cyclic code regions. P3DE attempts to address these issues.

P3DE:

- uses a control flow edge profile to specialize program paths according to detected execution bias, performing sinking transformations that go beyond those allowed by the "optimal" PDE approach.

[1] This chapter has been adapted from a conference submission. It was originally written in collaboration with John W. Sias. Significant sections have been preserved from the original.

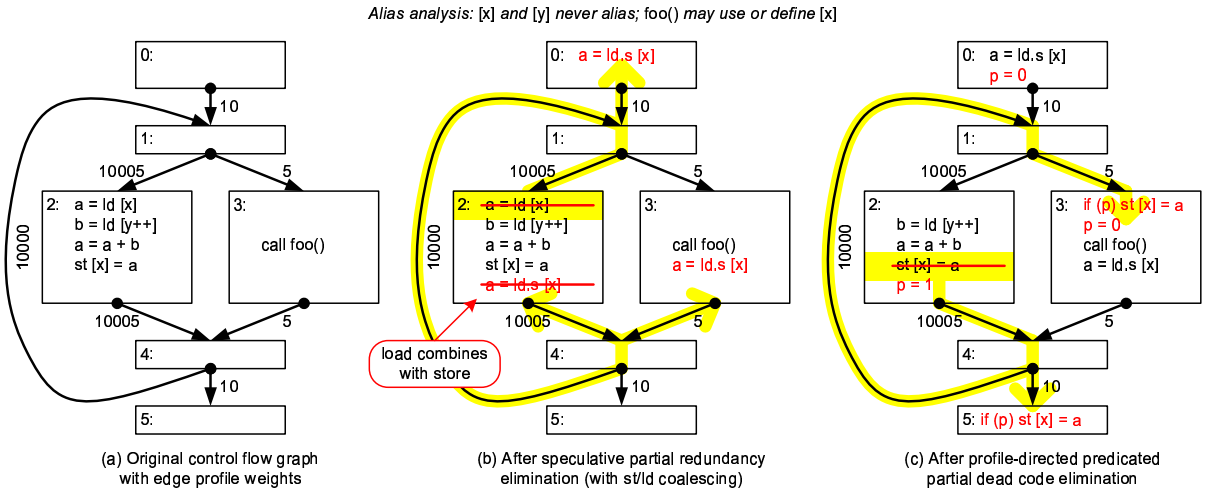


Figure 4.1 P3DE Motivating Example

- uses predication support in the compiler and target architecture, if available, to sink any assignments (under Boolean guards) into previously inaccessible locations, including those across loop back edges.

- minimizes predication overhead inserted and provides some degree of enhanced sinking in nonpredicated environments.

- incorporates a uniform cost-benefit model taking into account both profile and predication overhead, in both acyclic and cyclic control flow.

Figure 4.1 shows the same register promotion example as presented in Chapter 3. SM-PRE is able to speculate the loads out of the common path of the loop. However, an unnecessary store still exists in the path. Execution of the store can be reduced be sinking it to less frequent locations, in this case blocks 3 and 5. However, because this introduces computations on the store on paths where it did not exist previously, such as 0-1-3, predication must be used to guard its execution. Thus, the sunken stores are guarded with a predicate p, which is written

at the store's original location, and cleared at locations which begin paths on which the store is not executed.

Although this code motion may not seem to be much of a savings in execution, as the common case has simply replaced the store with a predicate write operation, there are some gains achieved. The predicate write does not occupy memory bandwidth, can often be performed by more functional units, and does not need to wait for the variable a to be computed, thus potentially enabling a shorter schedule. In addition, a chain of sunken operations can be guarded with a single predicate.

The P3DE algorithm is summarized in Figure 4.2. It is an iterative algorithm mirroring SM-PRE, consisting of three phases: dataflow analysis, motion graph construction and minimum cut calculation, and finally code motion. The dataflow analysis reveals opportunities for profitable sinking motion. The motion graph is a representation of the potential sinking paths for all computations of an assignment, and the minimum cut finds the most profitable location for sinking. Code motion performs the actual optimization. As with SM-PRE, P3DE has access to the results of control flow graph edge profiling. To simplify the presentation it is assumed that all locally dead computations have already been removed and that all critical edges have been split by inserting a new node in the middle of the edge.

4.1 Assignments in P3DE

P3DE operates on all delayable occurrences of an assignment simultaneously and delays them to minimize execution weight under a given profile. Concurrent examination is necessary

```
 1: split all critical edges
 2: repeat
 3:    find Dead, PartialDead, PartialDelay, and Avail
 4:    for each delayable assignment do
 5:       construct motion graph
 6:       if motion graph not empty then
 7:          find motion graph min-cut
 8:          if min-cut is not at original computation points then
 9:             if unsafe paths to min-cut exist then
10:                insert predicate initializations
11:             end if
12:             insert new computations
13:          end if
14:       end if
15:    end for
16:    reaching assignment analysis and attribute copying
17:    delete original computations
18: until no code motion in current iteration
19: absorb empty blocks resulting from critical edge splitting
```

Figure 4.2 P3DE Algorithm

to accurately measure the benefit of motion, as sinking motions of several occurrences may only collectively have a benefit justifying the movement cost. The task of moving multiple occurrences of an assignment is made more difficult by the fact that assignments may not be the same lexically. Some of the benefit in doing a value-flow analysis to find similar assignments is shown by generalizing store operations that access the same address as a single assignment and sinking then simultaneously. Live assignments may exist which block the sinking of partially dead assignments; P3DE cannot optimize these code patterns.

P3DE operates on a low-level, virtual register-based internal representation. An assignment can therefore be either a register-writing operation or a store to memory. The cost-benefit

analysis operates on one assignment at a time, where each assignment may consist of multiple computations (each a static instruction). Computations are grouped into the same assignment if they write the same location, perform the same operation, and have identical source operands. Commutative operations are expressed in canonical form to enhance matching. A pair of accesses to memory are recognized as being to the same location (must-def/must-use) if they reference the same global label or stack location, or if their addresses are defined by the same computation. Aliasing (may-use/may-def) information, used to identify code motion constraints relating to other stores, loads, and subroutine calls, is obtained from a previous interprocedural analysis [27].

To increase the efficiency of store sinking, the notion of assignments is generalized slightly. Store operations that write the same location, but store potentially different values, are associated together as *generalized assignments*. Each generalized assignment is treated as a single assignment during motion graph construction and cost-benefit analysis, enabling a number of lexically different stores to the same location to sink as a unit. Generalized assignments are revisited later to show how they impact the implementation of code motion.

4.2 P3DE Dataflow Analysis

Dataflow analysis, the first phase of P3DE, identifies the locations in which assignments are partially and totally dead, providing an outline for potentially profitable sinking transformations. Figure 4.3 illustrates the outcome of the analyses and is a useful example for this section. This information also helps bound the motion graph constructed in the next phase.

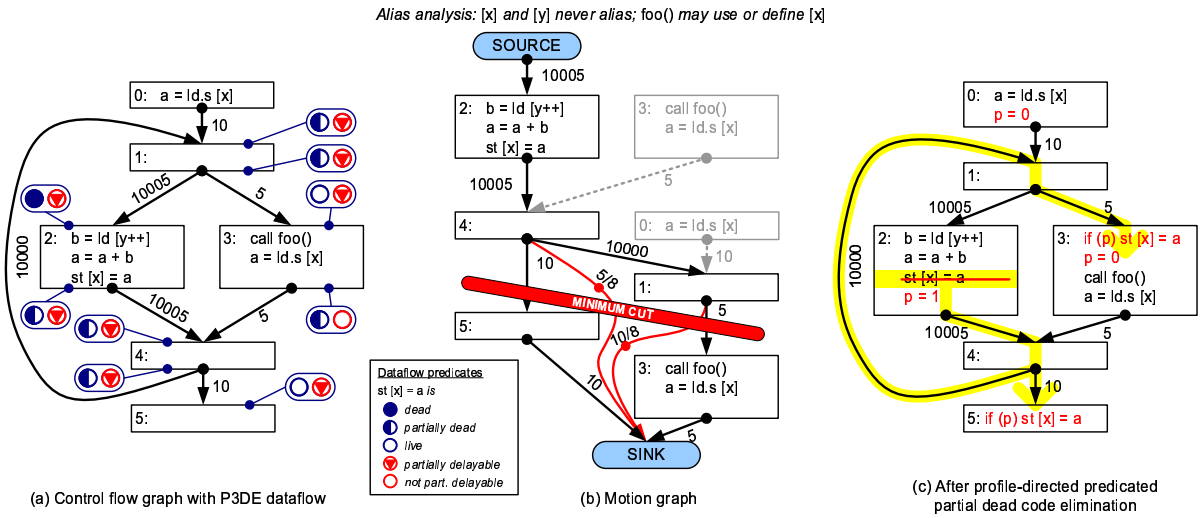


Figure 4.3 P3DE Dataflow and Motion Graph for Loop Register Promotion Example

Table 4.1 shows the local (single basic block) properties of an assignment used in the various dataflow analyses.

Dead code analysis is performed first to identify opportunities for profitable code sinking. Conventional dead and partial dead code analyses are performed, both backward bitvector propagations, as indicated by Equations (4.1) – (4.4) in Table 4.2. Partial dead code analysis indicates the paths along which sinking a computation may be profitable. A path on which an assignment is totally live offers no opportunity for removing a computation, since it is required everywhere down such a path. A path on which an assignment is totally dead requires no computation; sinking a computation to such a point renders it removable, so sinking further offers no additional gain.

Partial delay analysis is a forward derivation based on partial deadness. It reveals the extent of the blocks into which a computation might profitably be sunk. Equations (4.5) and (4.6) in Table 4.2 show the analysis rules. Note the difference from Knoop's PDE, which in its total

Table 4.1 P3DE Local Dataflow Sets For Block n

Assignment $a \in$	iff
$\mathbf{LocalDead}_n$	an assignment is dead at the beginning of the block, either because its destination is overwritten before any use or because the block ends with a return and the assignment is not used.
$\mathbf{Trans}_n$	an assignment is not made live (its destination is not used) in the block. For store assignments, the destination memory location must not have any *may-use* in the block. Also, for store assignments, address operands may not be modified. This prevents the sinking of a store computation to a point where it would write a different location.
$\mathbf{LocalDelayed}_n$	an assignment is computed in, and can be delayed out of, the block without altering program outcome.
$\mathbf{LocalBlocked}_n$	an assignment cannot be moved through the block, due to use of its destination operand, overwriting of a source or destination operand, or an aliased memory access.
$\mathbf{LocalAvail}_n$	an assignment is always computed in a block and its operands or memory location are not modified afterwards.
$\mathbf{AvailTrans}_n$	an assignment's availability is not affected in the block. Its source and destination operands are not overwritten; for memory assignments its memory location may not be overwritten.

Table 4.2 P3DE Global Dataflow Analyses

$$\mathbf{Dead}_m^{\mathbf{in}} = \mathbf{LocalDead}_m \cup \left(\mathbf{Dead}_m^{\mathbf{out}} \cap \mathbf{Trans}_m\right) \tag{4.1}$$

$$\mathbf{Dead}_m^{\mathbf{out}} = \bigcap_{n \in \mathrm{succ}(m)} \mathbf{Dead}_n^{\mathbf{in}} \tag{4.2}$$

$$\mathbf{PartialDead}_m^{\mathbf{in}} = \mathbf{LocalDead}_m \cup \left(\mathbf{PartialDead}_m^{\mathbf{out}} \cap \mathbf{Trans}_m\right) \tag{4.3}$$

$$\mathbf{PartialDead}_m^{\mathbf{out}} = \bigcup_{n \in \mathrm{succ}(m)} \mathbf{PartialDead}_n^{\mathbf{in}} \tag{4.4}$$

$$\mathbf{PartialDelay}_m^{\mathbf{out}} = \left(\mathbf{PartialDead}_m^{\mathbf{out}} - \mathbf{Dead}_m^{\mathbf{out}}\right) \cap \left(\mathbf{LocalDelayed}_m \cup \left(\mathbf{PartialDelay}_m^{\mathbf{in}} - \mathbf{LocalBlocked}_m\right)\right) \tag{4.5}$$

$$\mathbf{PartialDelay}_m^{\mathbf{in}} = \bigcup_{n \in \mathrm{pred}(m)} \mathbf{PartialDelay}_n^{\mathbf{out}} \tag{4.6}$$

$$\mathbf{Avail}_m^{\mathbf{out}} = \mathbf{LocalAvail}_m \cup \left(\mathbf{Avail}_m^{\mathbf{in}} \cap \mathbf{AvailTrans}_m\right) \tag{4.7}$$

$$\mathbf{Avail}_m^{\mathbf{in}} = \bigcap_{n \in \mathrm{pred}(m)} \mathbf{Avail}_n^{\mathbf{out}} \tag{4.8}$$

delay analysis required delayability on *all* (not *any*) predecessor paths for a block to be marked delayable. This difference results in more sinking possibilities, including ones potentially *increasing* the execution of dead computations. The minimum cut algorithm described in the next section selects from these candidates a safe and beneficial set of insertion locations: one satisfying program requirements and minimizing the number of computations.

PartialDead dataflow analysis is not strictly necessary for proper functionality of the algorithm: skipping it and assuming it is always true in the **PartialDelay** computation would increase the size of the motion graph, but the same minimum cut closest to the source would still be identified. The partial dead dataflow analysis is included because it is an intuitive representation of the region offering potential benefit and can reduce the size of, or even eliminate, the motion graph. One common case of this is a computation that is sinkable but completely live: since sinking it cannot reduce its execution, the motion graph is empty. Because the minimum cut on the motion graph has higher computational complexity than bitvector dataflow and the bitvector dataflow is performed for all assignments in parallel, this graph reduction is performed; in practice this provides some time savings.

Finally, an availability analysis is performed on the assignments. Extending the conventional expression-based definition [30] to assignments, an assignment a is available at a point p if every path to p computes the assignment and no path connecting the most recent computation of a to p contains an assignment to the source or destination operand(s) of a, or may write to the memory location referenced by the assignment. An available assignment may be recomputed

with no effect on program outcome. Section 4.4 will demonstrate how this property is applied to avoid unnecessary predication of sunken computations.

4.3 P3DE Motion Graph and Minimum Cut

The $\mathbf{PartialDelay}$ sets derived for each assignment as a result of the previous bitvector dataflow analysis define the scope of potential assignment-sinking transformations. For each delayable assignment, a weighted, directed *motion graph* is constructed. The minimum cut of the edges of this graph that renders the sink node unreachable from the source indicates the most profitable location for computation of the assignment: a cut at the original computation points indicates that no motion is profitable. A minimum cut elsewhere indicates an opportunity for profitable code motion and the appropriate locations for sunken computations and predication insertion. As with SM-PRE, the motion graph treats forward and back edges uniformly, enabling the sinking of assignments through back edges. Predication support enables the resulting, previously impossible, transformations. The motion graph for the code example is illustrated in Figure 4.3 and may aid in understanding this section.

The motion graph for an assignment a is derived using the $\mathbf{PartialDelay}$ and $\mathbf{Dead}$ sets from the control flow graph. First, relevant nodes and edges are extracted: In general, for each edge $e_{u,v}$, $n_u \rightarrow n_v$, in the control flow graph, add $e_{u,v}$ and nodes n_u and n_v to the motion graph iff ($a \in \mathbf{PartialDelay}_u^{\mathbf{out}} - \mathbf{Dead}_v^{\mathbf{in}}$). Some special restrictions apply: Predicate assignments are not allowed to undergo sinking that requires guarding predicates, since allowing such motion will cause the algorithm not to terminate. Certain other inexpensive assignments,

the sinking of which was not judged to be worth the cost of inserting a predication apparatus, are likewise not allowed to sink into unsafe locations that require predication.[2] In these two cases, an edge $e_{u,v}$ is required to satisfy $\mathbf{Avail}_v^{\mathbf{in}}$ to be included, preventing the occurrence of any sinking requiring predicates (because any computation at a location where the assignment is available is safe to recompute unconditionally). In an architecture offering no support for predication, this restriction would be applied to all assignments. Further implications of availability for code motion are discussed in Section 4.4.

Second, because dataflow analysis is performed at the granularity of basic blocks, a single node can be both a maximal sinking location and an original computation point for the same assignment (e.g., when an operand of an assignment is redefined before the assignment is computed). In order for the delaying paths to be modeled accurately in the motion graph, such a node must be split into an entrance/exit pair: node n is split iff ($a \in \mathbf{LocalDelayed}_n \cap \mathbf{LocalBlocked}_n$). Splitting the block between the blockage and the computation allows the graph to be constructed correctly.

Finally, the multiple-source, multiple-sink graph is connected to single source and sink nodes to provide the correct format for the minimum cut procedure. All original computation nodes ($\{n | a \in \mathbf{LocalDelayed}_n\}$) are connected to the source node. All nodes without outgoing edges (latest delayable nodes) are connected to a single sink node. These new edges are assigned weights equal to the execution weights of the blocks to which they are attached.

[2] In this work, moves, integer additions, integer subtractions, and shifts by constants were restricted. There is a possibility that this restriction may prevent profitable sinking of other operations; this will be evaluated and addressed in future work.

At this point, because critical edges have been split, each edge is associated with a control-equivalent node which will receive its apportioned computations.

The motion graph represents the extent of the control flow subgraph over which the assignment computations may be profitably delayed. Every node connected to the source of the graph represents the initial location of a delayable computation of the assignment. The nodes connected to the graph sink comprise the limit of profitable delayability. The graph has several useful properties. Completely dead edges are excluded because the assignment is irrelevant to them. An empty graph signifies that an assignment cannot be profitably moved. Most importantly, since the graph connects all original computation points to all locations below which the assignment may not be delayed (including all nearest consumers), any bisection of the graph rendering the sink node unreachable from the source node provides a legitimate set of delayed computation points. The bisection that crosses the minimum amount of forward flow weight yields the best placement of computations.

The motion graph treats forward and backward control flow edges uniformly and makes no special consideration of sinking operations to locations that may not be dominated by the original placement of computations. Motions exploiting this feature may need to use predication to guard the execution of delayed computations (inserting predicate defining operations into blocks adjacent to the motion graph), incurring some additional transformation cost. However, there is a potential for the cost of this predication apparatus to far outweigh the benefit of sinking an assignment, such as sinking into loop bodies. No equivalent issue exists for SM-PRE. Thus, the decision-making mechanism is improved by factoring predication costs into the

graph. This is done by including *cost edges* that represent the cost incurred in predicate clear evaluations if the cut includes those edges. The weight of a cost edge is the potential number of predicate clear operations inserted multiplied by an empirically determined *reweighting factor* between 0 and 1. This factor reflects the consideration that predicate initializations are much cheaper than the typical assignment being sunk, since they have no incoming dependences and can issue in any slot of the target architecture. For this work, a reweighting factor of $\frac{1}{8}$ was chosen, which allows aggressive sinking but prevents motion when the weight of introduced predicate initializations would be nearly an order of magnitude higher than the number of computations removed by the sinking. Cost edges are connected such that a cut requiring the insertion of predicate clearing operations will necessarily cut the representative edge. Since cost edges are only of positive weight, and no edges connect the source to any other node, a minimum cut of the motion graph is guaranteed to have a weight equal to or less than the combined execution of the original computation locations. There are two situations where cost edges are needed:

- Loop back edges *in the motion graph*[3] receive predicate clearing operations if the cut is placed within the loop. Since the min-cut need only make the sink of the (directed) motion graph unreachable from its source, backward edges do not contribute to the profitability metric. A distinct cost edge is therefore created with the same endpoints as the loop back edge but in the opposite direction, with a weight equal to the loop back edge weight multiplied by the reweighting factor. A cut through the loop incurs the penalty.

[3]Loop back edges in the motion graph may be forward edges in the control flow graph and vice-versa.

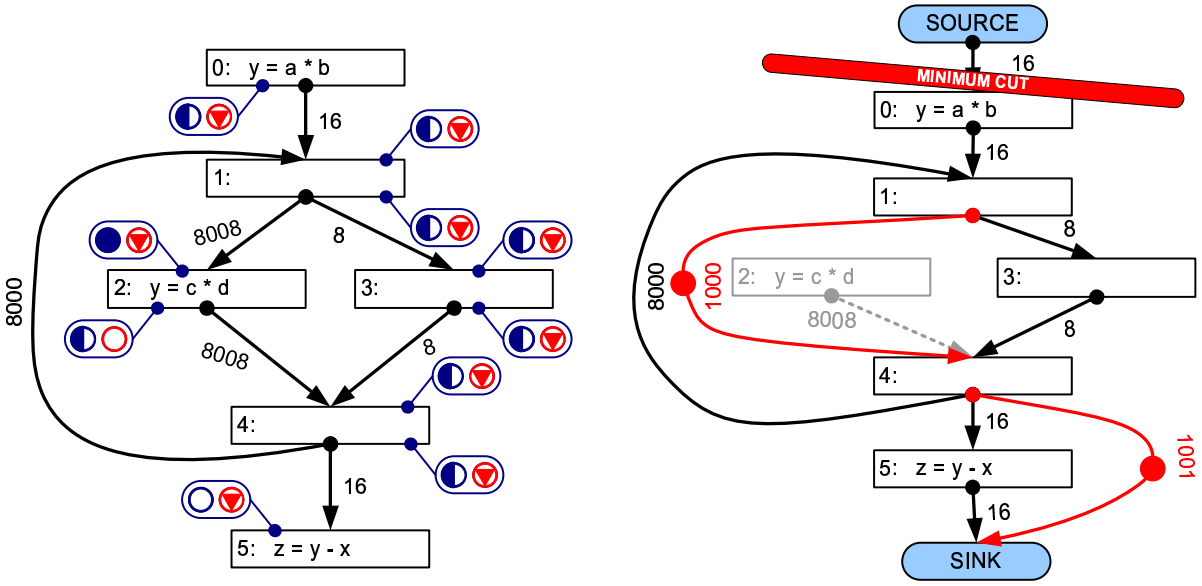


Figure 4.4 Example of Motion Prevented by Cost Edges

- Nodes above the cut that are the entry points of unsafe path(s) (discussed in Section 4.4) will also receive predicate clearing operations. A cost edge is therefore created from the node to the sink of the motion graph with a weight equal to the sum of the incoming unsafe edges multiplied by the reweighting factor. A cut below this node incurs the penalty.

Figure 4.4 shows a code example and the corresponding motion graph for the assignment "y = a * b". Cost edges, marked with circles, have been inserted into the motion graph with a reweighting factor of $\frac{1}{8}$. In the motion graph, the loop back edge has a corresponding cost edge in the opposite direction with a weight of 1000. This represents the cost of the predicate clearing operations needed on the loop back edge if a computation is placed on an edge within the loop. For the edge that enters the motion graph at block 4, a cost edge is included from block 4 to the sink of the graph with a weight of $\frac{1}{8}$ the edge's weight. This represents the cost

of the predicate clears needed in block 2 if a computation is placed after block 4. In this case, these added costs prevent the assignment from being sunk, since the predicate initialization cost outweighs the potential sinking benefit. Conversely, the example in Figure 4.3 is a case where the cost of migrating an assignment is vanishingly small compared to the potential benefit.

An issue that impairs more precise modeling of costs is that the need to insert predicate setting operations at original computation sites is unknown until after the minimum cut is found. Since the count of predicate setting operations is bounded by the initial computation count for the assignment being delayed, this is not as significant a problem as the problem managed by the cost edges, the danger of introducing an unbounded number of predicate initializations.

To find optimal placement the same minimum cut mechanism used for SM-PRE, based on Goldberg and Tarjan's push-relabel algorithm for finding maximum-flow [28], is utilized. It is actually run in the opposite direction from SM-PRE, since the desire is to find the minimum cut closest to the source node. Because most motion graphs constructed are primarily acyclic, the runtime is generally fast. Once the minimum cut is computed, code motion can be performed.

4.4 P3DE Code Motion

The minimum cut of the motion graph supplies the desired locations for insertion of sunken computations. Before this code motion is performed, it is necessary to determine whether a guarding predicate will be required and, if so, where its definitions should reside. This predicate guard is required if there exists a path in the control flow graph from the function start node to a sunken placement site along which, in the original program, the assignment is not computed.

This is called an *unsafe* path. If such a path exists, a predicate is created to guard the sunken computation(s). At each edge incident to the boundary of the motion graph that is part of an unsafe path, a predicate clear operation (p = 0) is inserted. A predicate set operation (p = 1) is inserted in place of each original computation ($\{n|a \in \mathbf{LocalDelayed}_n\}$), each guarded by the guard predicate of its location's original computation. This predication apparatus ensures that computations will be executed only under the proper conditions, regardless of their sunken location, both squashing potentially dead executions and preventing unsafe evaluations and definitions. Predicate clear and set operations are of a type that has no effect if guarded by a predicate that evaluates to *false*, a requirement for subsequent if-conversion based hyperblock formation [31].

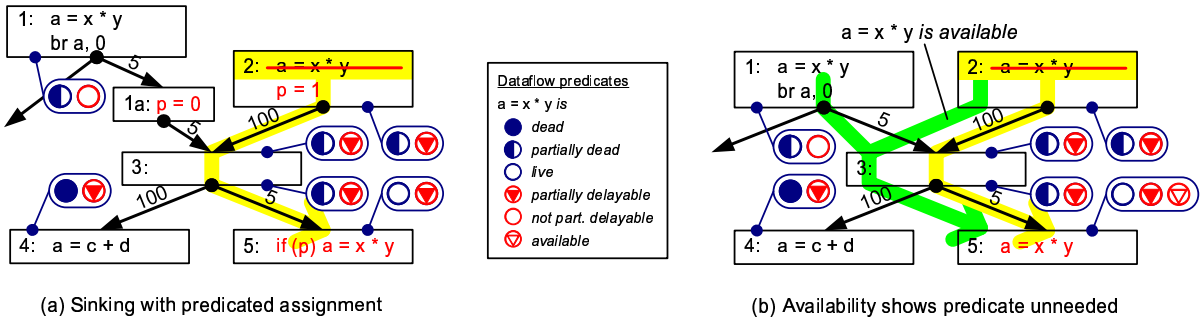
An unsafe path can exist in two ways, both of which are immediately apparent in the motion graph. First, considering initial computation nodes in the motion graph, those connected immediately to the source node, an unsafe path exists if the original computation therein is predicated and the predicate may be false. Second, for other nodes *above the minimum cut*, any incoming control flow edge not incorporated in the motion graph or not originating above the cut[4] indicates an unsafe path. (The acyclic case is illustrated in Figure 4.3(b) and a potential cyclic case in Figure 4.4.) In either case, a predicate clear operation is inserted on the unsafe edge.

The seamless, uniform handling of unsafe edges using predication is the key to the generality of the motion graph cutting scheme. An example of the second unsafe edge case occurs

[4]Edges crossing the cut from below the minimum cut can occur in the case of a cut through a cycle in the motion graph.

in Figure 4.3(b), in which the unsafe edges from the "ghosts" of nodes 0 and 3 are indicated as dotted lines. Comparing this motion graph to the transformed code, as shown in Figure 4.3(c), predicate clears are inserted in node 0 and node 3.

Since allocating a predicate register for the motion predicate and executing predicate definitions both have nonnegligible costs, it is desirable to avoid using predication unnecessarily. The availability dataflow analysis allows identification of delayed computation locations where predication is not required, even though an assignment is not sinkable along all paths to those locations (recalling that an assignment may be recomputed safely at any point where it is available). If the assignment is available at all sunken placements, there is no need to allocate or set a predicate at all. Figure 4.5 shows an example. The computation of assignment a = x * y in block 2 can be sunk, but not the computation in block 1. This being the case, it would seem necessary to guard the delayed computation on a predicate, since the assignment was blocked on some sinking paths (Figure 4.5(a)). Since the assignment is *available* (as indicated by the superimposed, dark arrows), however, the computation can be sunk to block 5, achieving a profitable code motion, without unnecessarily incurring predication overhead (Figure 4.5(b)). Only Gupta et al. [4] have a cost-benefit analysis capable of performing such a transformation. The tests have shown availability to allow a substantial decrease in predication overhead on some benchmarks; this also means P3DE can perform useful code motion even without architectural support for predication. A final note is that in order to remove predicates due to availability under generalized assignment motion, the compiler must ensure that all original computation sites that contribute to availability at a given delayed computation site maintain



(a) Sinking with predicated assignment

(b) Availability shows predicate unneeded

Figure 4.5 Availability Motion

correct execution by moving the value to be stored to the generalized assignment's "temporary" source register.

To further reduce the number of predicate definitions and uses inserted, a forward and a backward reachability propagation determine which insertion points are reachable from unsafe paths and which original computations reach these insertion points, respectively. Only insertion points reached in this analysis require predication, and only original computation sites reached require predicate set operations. It is important to note that an assignment may be available at an unsafe path, but an insertion point reachable from that unsafe path may not be available. Although program correctness is maintained if the predicate is neither set nor cleared at the unsafe path, nondeterministic execution of instructions is generally undesirable; for this work these unsafe paths clear the guard predicate.

P3DE is not aware of predicate register pressure; because P3DE is performed prior to hyperblock formation, which generally uses predication to a much greater degree, P3DE is not in a suitable position for controlling predicate register pressure beyond the use of rough heuristics. However, Chapter 5 will show that predicate register pressure does exist and has an effect on the compiler's final output code, so predicate register pressure needs to be addressed. If

P3DE has advance knowledge of predicate usage in hyperblock formation, or is moved after that stage, it may be desirable to tailor P3DE to recognize and target the optimization to minimize predicate usage or perform only the most profitable motions given a limited number of predicate registers. Work by Gupta et al. [32, 33] has addressed this problem for general registers, but the predicate register pressure problem has an additional property: multiple predicates can potentially be combined into a single predicate under certain conditions. There are several options for combining predicates that guard different assignments into a single predicate: these are discussed in Future Work, Chapter 6.

The code motion process can be summarized as follows: Delayed computations are inserted "on" edges in the cut-set, as facilitated by prior critical edge splitting. If the resulting computation is reachable from an unavailable, unsafe entry edge, it is guarded with the motion predicate. For generalized store assignments, the value stored is the value in a temporary register created to merge the stored values of the associated, specific assignments. Next, a reaching assignment analysis, similar to a conventional reaching assignment analysis, is performed to find for each sunken computation, which original, removed computations contribute to that sunken computation. Attributes from the original computations are copied to the sunken computations:

- Synchronization arcs, drawn between memory operations that may conflict, must be copied to prevent illegal reordering of memory operations.
- If an original computation is marked as speculative and thus has special exception handling, all sunken computations that it contributes to must also be marked as speculative.

- Other attributes which assist later optimizations stages are also copied.

Finally, original computations not on the minimum cut are removed. If any new computation point requires a guarding predicate and is reachable from the original computation point, the deleted computation is replaced with a predicate set instruction. If the original assignment is part of a general assignment motion, a compensating move is also inserted to initialize the general assignment temporary register with the value originally stored at that location.

4.5 Correctness, Termination, and Complexity

The P3DE algorithm is correct in that it will only remove assignment executions under the conditions under which they are dead. It will only introduce new or additional computations on paths when re-executing those assignments will not change program behavior or when their guard predicates prevent erroneous execution. Consider that the motion graph for an assignment consists of all delayable occurrences of the assignment and all paths of possible motions that are not completely dead. Thus, any path that leads to a use of the assignment leads into the sink node of the motion graph. A cut bisecting the source and sink of the motion graph provides a correct set of code sinking insertion points, creating a single computation of the assignment along any path from a delayable assignment to any use. To guard execution, we must ensure that the guard predicate is set when the assignment would originally be executed and cleared in any other case (unsafe paths). Without predication, a cut across the graph will guarantee that the assignment will be executed before its output would used. With predication, we replace all original computations with a guard predicate set and clear the guard predicate

for all other possible control flows (either from the computation points or incoming paths to the motion graph), so the assignment will only be executed if it would have been executed originally. Thus, correctness is maintained for the program.

Code motion is only performed when a reduction in the profile execution count of assignments can be obtained by delaying. The algorithm terminates when no profitable motions can be found. Edge weights can only be integral values, there are a finite number of cuts of the control flow graph, and an assignment's execution can be reduced at most to zero, so there is a finite number of motions possible for each assignment. An upper bound for this is s, the sum of the edge weights in the control flow graph. The algorithm starts with a finite number of assignments, a, in the program. For these assignments, the maximum number of iterations is as, which is finite.

However, this does not consider the effects of new assignments that P3DE may introduce. These are predicate set and clear operations and compensating moves for generalized assignment motion. For the worst case upper bound, there are at most a stores in the program, each of which may generate a compensating move each time it is moved. This results in a^2s compensating moves, a^2s^2 motions for those moves, and $as + a^2s^2$ nonpredicate motions. If motion of predicate assignments is allowed, there are potentially $2\left(as + a^2s^2\right)$ total predicate assignments introduced during the algorithm. These cannot generate predicate assignments themselves by the algorithm. The predicate assignments can undergo at most $2s\left(as + a^2s^2\right)$ motions. Thus, the upper bound of total motions is $2s\left(as + a^2s^2\right) + as + a^2s^2$, which can also be represented as $as\left(1 + 2s\right)\left(1 + as\right)$.

The previous paragraph shows that there is a polynomial limit to the number of motions that the P3DE algorithm can perform. Worst-case execution of the P3DE algorithm would be to perform a single motion on a single assignment for each iteration, so the upper iteration bound of the algorithm is the same as the upper bound of total motions. Since one iteration of the algorithm runs in polynomial time, the P3DE algorithm runs in polynomial time. Presented here is an explanation of the polynomial runtime of the algorithm.

A single iteration of the algorithm consists of four dataflow analyses on the control flow graph and, for each delayable assignment, one dataflow analysis (detecting dominators to find cycles for cost edges), a linear pass to detect unsafe paths, one minimum cut solution, and two breadth-first graph traversals. Reaching assignment analysis is another dataflow analysis on all assignments. Each primary dataflow analysis has a runtime that is $O\left(n\left(d+2\right)\right)$, where n is the number of blocks in the control flow graph and d is the maximum number of back edges on any acyclic path in the control flow graph [34]. The dominator analysis and minimum cut algorithm operate on a reduced graph $G_{rd} \equiv (N_{rd}, E_{rd})$. The dominator analysis runs in $O\left(|N_{rd}|\left(d+2\right)\right)$. There exist various polynomial-time minimum cut algorithms; this work uses one based on Goldberg and Tarjan's push-relabel algorithm, using a list for node traversal, which executes in time $O\left(|N_{rd}|^3\right)$ [28]. Graph traversals locating predicate set and clear locations are $O\left(|N_{rd}| + |E_{rd}|\right)$. This minimum cut is at worst performed as many times as the maximum number of assignment motions, which has been shown to have polynomial bound. The assignment reaching propagation checks each pair of deleted and sunken assignments, which is $O\left(A * n^2\right)$, A being the maximal number of assignments. Since every component of

a single iteration has polynomial time, the overall algorithm runs in polynomial time, bounded by $O\left(n^4\right)$.

CHAPTER 5

EXPERIMENTS

This chapter explains the experiments which evaluate our PRE and PDE schemes. A subset of the SPECint2000 C benchmarks suite was compiled within the IMPACT compiler framework for the Itanium Processor Family. SPEC's training inputs were used to drive profile-guided optimization.[1] IMPACT performs hyperblock and superblock transformations [35] and other aggressive ILP transformations to produce very high quality code for the Itanium Processor Family. IMPACT's baseline performance is generally comparable to that of Intel's production compiler for the Itanium family.

Estimated SPEC performance numbers were gathered on a Hewlett-Packard zx6000 workstation with 8-GB RAM and two 900-MHz Itanium 2 processors, each with a 1.5-MB L3 cache. These numbers are estimated because they do not exactly follow SPEC's requirements for reported results, but are run within a SPEC framework. The system runs RedHat Linux Advanced Server v.2.1, with a custom 2.4.21 Linux kernel supporting general speculation of loads (a model of control speculation that does not require speculation checks or recovery code). Results involving real-time instruction counts are obtained from the *perfmon* performance monitoring tool, provided by Hewlett-Packard Laboratories [36]. This tool enables the gathering of detailed real-machine performance evaluations.

[1] Two benchmarks were excluded: *176.gcc* had errors with missing synchronization arcs that allowed illegal ordering of memory operations, and *252.eon* is a C++ benchmark, which IMPACT cannot currently compile.

Compile times are presented only for rough comparison to the overhead of classical compiler optimizations. Since IMPACT is a prototype research compiler, its compilation speed does not necessarily reflect the ultimate speed at which the algorithm can be executed. In addition, Lopti occupies a small portion of the overall compile time of the compilation process: in that scope the overhead of the PRE and PDE methods is very small. Runs were performed on a dual-processor 1.7-GHz Pentium 4 system with 2 GB RAM. IMPACT was compiled using gcc version 2.96 with -03 optimization.

All PCE optimizations were performed at the intraprocedural level only. Unless otherwise noted, all evaluations presented here reflect execution of the SPEC reference inputs.

5.1 IMPACT Optimizations

IMPACT performs optimizations in several stages in an intermediate form similar to machine assembly language. The basic structure operated on is the control block, which is a region of code with a single entry point and potentially multiple exits.

The first optimization stage is Lopti, which performs classical optimizations such as those found in standard compiler texts [30, 34]. All PCE optimizations were performed during this stage: PRE optimizations were performed iteratively in conjunction with IMPACT"s loop and classical local and global optimizations, while the PDE optimizations were performed after all other optimizations.

Two optimizations of note in Lopti are loop-invariant code removal (LICM) and loop variable migration (LVM). Both are subsumed by PCE optimizations, and were disabled for particular situations. LICM speculatively hoists expressions out of loops if the profile weight of the loop entrance is smaller than the sum of the profile weights of its calculations inside the loop. It is subsumed by SM-PRE. It is also heuristically disabled when the number of control blocks exceeds 100. LGVM performs a loop-biased register promotion by moving memory accesses which are constant throughout a loop to the entrances and exits of the loop, and around any potentially aliasing memory operations and subroutine calls. It determines a profitable transformation by comparing the weights of the existing calculations to the final transformation locations. LVM does not sink store operations when no corresponding loads to the location exists in the loop: when SM-PRE and LVM are both performed without P3DE, LVM is performed first before SM-PRE hoists loads out of loops. A combination of SM-PRE and P3DE subsumes LVM.

After Lopti, IMPACT performs Hyperblock formation [31], which performs if-conversion of code regions and utilizes predication to allow overlap of independent control constructs. This removes branch instructions and reduces branch mispredictions.

The final stage of optimization, just prior to code generation, is Superblock formation [37]. It performs various code motions, duplications, and transformations to expose ILP for scheduling. Superblock formation can potentially subsume some PRE opportunities by moving or duplicating partially redundant instructions into locations where they are completely redundant, but no significant cases of this were exhibited in the experiments.

5.2 PRE Results

For these PRE results, we compare the performance of optimal PRE and SM-PRE to the baseline IMPACT compilation. The algorithm used for optimal PRE is Knoop et al.'s basic-block algorithm [10]. LVM was performed for all compilation formats, removing some of the cases that PRE would subsume. LICM was enabled for PRE, but disabled for SM-PRE. The intent in doing this was to isolate and evaluate those cases which only PRE can address.

Register pressure was observed and counteracted in our PRE algorithms by not allowing PRE motion (not transparent in the dataflow analyses) that might create a new live register through a basic block that has at least 64 live virtual registers. The Itanium's register stack engine (RSE) enables the processor to only spill and fill a subset of the total register file of the processor at function boundaries. Gupta and Bodik have published previous work to control register pressure in PRE [33], which assumed a constant number of architectural registers that are spilled and filled at function boundaries. In this case, however, increasing the total number of registers needed in a function can negatively impact performance even when the total is less than the register file. Before this register pressure metric was implemented, the performance loss due to RSE stalls outweighed the redundancy elimination gains on several benchmarks. 64 registers is a heuristic which has been relatively effective in our experiments. For SM-PRE, the number of motions per interation was limited so that the 64 live register limit would not be exceeded in any block.

Table 5.1 Lopti PRE Compile Time (in seconds)

Benchmark	Baseline	PRE	SM-PRE	PRE Pct. Inc.	SM-PRE Pct. Inc.
164.gzip	34.56	44.85	47.72	29.77%	38.08%
175.vpr	168.08	204.42	239.74	21.62%	42.63%
181.mcf	12.77	16.71	17.85	30.85%	39.78%
186.crafty	658.43	735.11	771.27	11.65%	17.14%
197.parser	109.74	134.88	142.35	22.91%	29.72%
253.perlbmk	1373.69	1580.01	1750.25	15.02%	27.41%
254.gap	658.91	854.71	880.02	29.72%	33.56%
255.vortex	470.33	644.79	663.62	22.54%	23.74%
256.bzip2	52.45	64.27	64.90	22.54%	23.74%
300.twolf	627.02	810.66	827.68	29.29%	32.00%

5.2.1 PRE compilation time

Table 5.1 shows the time spent during the Lopti compilation stage with PRE. In general, the overhead of SM-PRE is not significantly higher than PRE. The exception to this is *175.vpr*, where SM-PRE has nearly twice the overhead of optimal PRE. The reason for this is because the register pressure control mechanism allows only one motion at a time in the function try_place(). Because those motions introduce temporaries that can replace existing registers, the live register count stays under 64, which results in a full iterations of the SM-PRE algorithm for the motion of a single expression. In this case, Lopti terminated because it hit an iteration limit, rather than reaching a stable optimized code state. The effects of this on code performance are discussed in a later section.

5.2.2 Load reuse analysis

The purpose of PRE is to eliminate redundant instructions, which recompute the same values multiple times in a program. Load reuse (consistently accessing the same memory location) is one phenomenon that can be reduced by PRE. To find the amount of load reuse eliminated in a program, we profile memory access addresses. When a load's address has been previously accessed, it denotes a potential opportunity for the PRE optimization to recognize the reuse and remove a partial or full redundancy.

Loads are tracked in two ways:

- Track addresses for each static load, noting a reuse when the address has been accessed within the visible history of the load. High reuse in this domain represents an opportunity for register promotion of the memory location. For these experiments, each instruction's history is four addresses.

- Track an address history of dynamic memory accesses, noting a reuse when the address has been accessed by a previous instruction (itself or another static instruction). High reuse in this domain represents an opportunity for PRE to borrow the loaded value from another instruction (or itself). The accesss history in our experiments is 32 addresses.

Bodik et al., in previous work [13, 20], also tracked the history of each static load to determine the amount of load reuse and show an upper-bound potential for register promotion. However, this can be deceptive, since a load that consistently accesses the same location may be prevented from being promoted due to may-alias stores or subroutine calls. Also, this method

will not detect reuse between two different static instructions, which PRE can detect and optimize. For this reason, two different metrics are used to determine load reuse.

Figure 5.1 shows the measured reuse by the same static load as a percentage of the total dynamic loads in the baseline compilation. Results for both optimal PRE and SM-PRE are shown. The SPEC training inputs were used for this simulation. Reuse is divided into the three categories:

- Interprocedural reuse, where an instruction will load the same address as a prior execution from a previous call of the function. This is the top portion of the bars in the figure.

- Intraprocedural reuse, where an instruction is within a loop and loads the same address as a prior execution from the same call of the function. This is the bottom portion of the bars in the figure.

- Intraprocedural reuse which is unoptimized because of the need to control register pressure. This is measured by finding the differences in reuse when the register pressure control mechanism is either enabled or disabled. It is the middle section of the bar in the figure, represented as “reg press intraprocedural.”

Although there is an enormous amount of reuse in several benchmarks, the majority of that reuse is interprocedural and cannot be addressed by our intraprocedural PRE techniques. In general, PRE does not make many significant reductions in the reuse at static instructions. One reason for this is that intraprocedural reuse occurs within loops, and the two previously

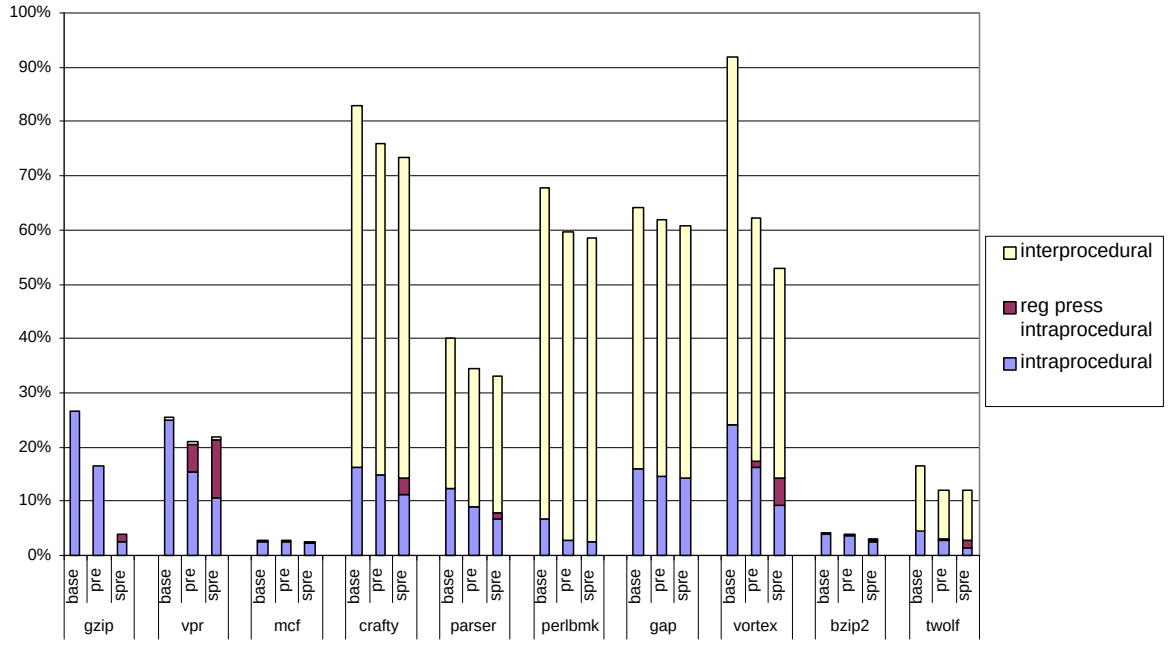


Figure 5.1: Detected Dynamic Reuse By the Same Static Loads as a Percentage of Total Dynamic Loads in Baseline

mentioned loop optimizations that promote memory locations to registers have already eliminated many of these cases in our baseline compilation. Thus, few opportunities remain for the PRE algorithms to reduce reuse at static instructions. Because of may-alias memory operations, particularly subroutine calls, some amount of intraprocedural reuse will exist after PRE optimization.

For *175.vpr*, the SM-PRE reuse is slightly higher than optimal PRE's due to the register pressure control mechanism. Because SM-PRE operates on one expression at a time, it can stop code motion exactly when the live registers in a block reaches 64, the heuristic chosen for these experiments. PRE, however, operates on all expressions in parallel, so it sometimes

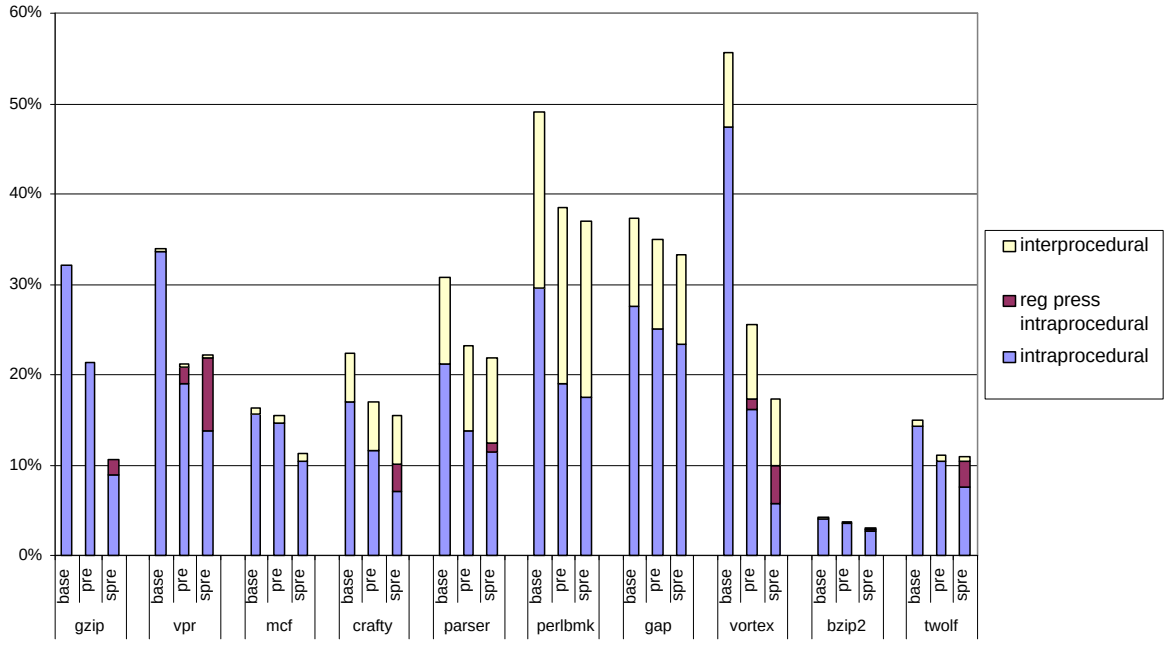


Figure 5.2 Detected Dynamic Reuse from the Previous 32 Dynamic Accesses

exceeds the limit of 64 live registers before the control mechanism can prevent further motion. In these cases, SM-PRE has more unoptimized PRE opportunities.

Figure 5.2 shows the percentage of dynamic load reuse from the previous 32 dynamic accesses, again based on the SPEC training inputs of each benchmark. A dynamic history of memory accesses was kept during execution, indicating the address and function call of the access. Stores invalidate all previous accesses to the same address in the history, and detected intraprocedural reuse takes precedence over interprocedural reuse in the data collection mechanism. For this experiment, more intraprocedural reuse is visible in many of the benchmarks than for the static instruction detection. It is important to note that because the PRE algorithms reduce the total number of memory operations, they can expose reuse that was not visible with

the baseline compilation due to a finite history. Some evidence of this was observed during initial experiments, but a history of 32 accesses was large enough to minimize this effect.

An inspection of significant reuse has revealed several sources of dynamic load reuse remaining after PRE. These are:

- In PRE, reuse is blocked due to safety requirements. SM-PRE is able to take advantage of these opportunities by utilizing speculation.
- Unoptimized reuse because of the need to control register pressure.
- Reuse blocked by may-alias stores or subroutine calls.
- Reuse hidden by lexical differences.
- Data-dependent reuse: two instructions that exhibit reuse under one profile may not exhibit it under different program inputs.

As shown in Figure 5.2, PRE is relatively effective at removing load reuse from several benchmarks. SM-PRE is able to remove a portion of the remaining reuse, but is often blocked by the register pressure control mechanism. In some cases, such as *175.vpr* and *255.vortex*, nearly half of the remaining reuse is unoptimized for this reason.

The reuse blocked by may-alias memory operations is probably the most prominent portion of remaining reuse. The benchmarks *175.vpr* and *254.gap* in particular have a significant amount of this type of reuse, as does *253.perlbmk* (for which we do not have fine-grained alias analysis). Reuse blocked by may-alias stores can be optimized via data speculation when the stores seldom alias with the load. Lin et al. [21] show a methodology to perform this

optimization within a SSA representation. In a few cases, a may-alias store will usually alias with a load; this reuse cannot be removed without introducing additional code which will likely impair overall execution.

Lexically different expressions can be separated into three different categories. The first can be addressed by value numbering: two virtual registers have the same value in all cases, but copy propagation optimizations are unable to merge these variables together. *186.crafty* has some reuse which could be identified in this manner. Simpson addresses means of identifying and optimizing these cases in his Ph.D. thesis [18]. A few significant cases of this reuse are visible in SPEC, although it is important to note that many more of these cases would exist if copy propagation were not performed between iterations of PRE.

The second category of lexically different expressions is array element reuse that is carried across separate iterations of loops. This happens when the an array element may be read or written across multiple iterations of a loop, and was observed to some degree in *186.crafty*, *197.parser*, and *256.bzip2*. Bodik et al. [3] have published an analysis and method for optimizing these cases: this uses an array dataflow analysis to find all writes and uses of an array element, then promotes it to a register. To preserve values across iterations, these promoted elements are shifted to other registers; a register will carry a particular element for at most a single iteration. This scheme requires constructing prologue and epilogue code for the loop to correctly populate the promotion registers.

The final category are those which cannot be identified due to the flow of values. The previous case of loop iteration-carried reuses falls into this category, but value-flow opportunities also include cases which do not necessarily come from loops. These cases cannot be identified via value numbering because they do not contain the same value along all paths. A significant portion of *164.gzip*'s remaining reuse is comprised of this case, as is the largest single point of reuse in *197.parser*. Some cases also exist in *175.vpr*. Bodik et al., in previous work [13, 19, 20], discusses means to identify and optimize these cases. His methods subsume both lexical and value numbering-based PRE. In some cases, value-flow opportunities are also blocked by may-alias stores, so a method combining data speculation and value-flow analysis would be required to remove the redundancy.

Data-dependent reuse does occur in several benchmarks, and is a prominent percentage of *181.mcf*'s reuse. Although that reuse might be optimizable, it cannot be addressed by a PRE mechanism. The reason for this is because PRE depends on instructions always computing the same expression, originating from the same program variables, for a result to be borrowed. For these cases the measured reuse originates from different expressions which have the same numerical values, and may not match often if a different set of program inputs are used.

5.2.3 PRE SPEC performance

The ultimate measure of the effectiveness of PRE is the increase it contributes in wall-clock performance. Presented in Table 5.2 shows the estimated SPEC performance results for PRE and SM-PRE compared to the baseline compilation.

Table 5.2 PRE Estimated SPEC Results

Benchmark	Baseline	Optimal PRE	SM-PRE	Opt. PRE Pct. Increase	SM-PRE Pct. Increase
164.gzip	665	675	692	1.50%	4.06%
175.vpr	517	536	521	3.68%	0.77%
181.mcf	239	240	242	0.42%	1.26%
186.crafty	613	602	611	-1.79%	-0.33%
197.parser	448	451	450	0.67%	0.45%
253.perlbmk	546	548	556	0.37%	1.83%
254.gap	391	397	396	1.53%	1.28%
255.vortex	918	905	941	-1.42%	2.51%
256.bzip2	556	538	550	-3.24%	-1.08%
300.twolf	625	642	650	2.72%	4.00%

As stated by Bodik in his Ph.D. thesis [13], PRE can increase performance by either freeing up computing resources or shortening the data dependence height of a sequence of instructions. Because of the high width of the Itanium architecture, one can anticipate that computing resources will generally be widely available and that reduction of data dependence height will be PRE's contribution to performance. Our results support this: the most significant source of performance gain in *175.vpr* is due to the removal of a chain of expressions in a small loop in the function route_net(). This removal enables a shorter schedule and better performance. In general, however, PRE and even SM-PRE are incapable of increasing performance by more than a few percent.

As noted by many researchers, PRE has the effect of introducing new temporary variables into programs and increasing register pressure. If all expressions are operated on before register pressure controls limit motion, register pressure can increase beyond desirable limits. This was

observed for SM-PRE on the benchmark *186.crafty* before these experiments were run, which prompted a control mechanism that checks register pressure after each expression motion.

PRE can have some synergistic effects with other optimizations: in the case of *164.gzip*, 40% of SM-PRE's performance improvement comes from fewer branch misprediction flushes, due to a more effective Hyperblock formation.

PRE considers the execution of an expression computation to be the same at every location in the program. However, SM-PRE may speculatively hoist a load into paths where the result will often not be used, but the load address is invalid or is in a page that is not in memory. This requires invoking a software page table handler, and the total penalty can be easily in the thousands of cycles. This is the cause of SM-PRE's low performance in *175.vpr*: the benefits of code motion are offset by calls to the kernel in try_place() to handle loads of this nature. Approximately five times as many cycles are spent processing these loads for the SM-PRE version.

The experiments show that PRE can have a positive effect on performance. However, the control of register pressure remains an issue for IMPACT's PRE techniques, as does the potentially high cost of speculating load instructions. There is opportunity for more advanced PRE mechanisms to further optimize the program and reduce reuse. Such methods already exist in literature [3, 20, 21], but individually address a limited set of reuse in a small number of benchmarks. The largest potential for gain is from data speculation, particularly in *254.gap*, where potentially aliasing stores prevent optimization by IMPACT's PRE methods.

5.3 PDE Results

Performance results obtained using optimal PDE and the P3DE algorithm are presented here. To the author's knowledge, no prior related work has shown empirical runtime results for partial dead code elimination. LVM was disabled for the first set of experiments, so that the efficacy of P3DE for register promotion could be evaluated. IMPACT's SM-PRE was enabled for all builds. This was chosen because P3DE was not expected to have much effect beyond store sinking out of loops; results at the end of this section appear to support this belief.

The baseline for the comparison includes, in place of P3DE, an implementation of Knoop et al.'s "optimal" partial dead code elimination [2], modified to sink only partially dead computations. This is a design decision based on empirical study. Delaying all computations, as opposed to only partially dead ones, may cause needless code growth; however, choosing to sink only partially dead assignments may prevent useful motion of computations blocked by antidependence.

To judge the necessity of using predication in the algorithm, presented alongside the baseline and experimental compile paths is a modified P3DE algorithm that operates similarly to Lo et al.'s speculative store sinking [9]. This will be termed the *nonpredicated optimization*; it only allows motion which is possible without the use of predication. This consists of availability-based sinking and speculative sinking for nonnpredicated stores whose addresses are known to be safe, such as local and global variables. The latter are treated as generalized store assignments. The reweighting factor (see Section 4.3) for cost edges is set at 1, to recognize that loads are generally at least as expensive as stores. During code motion, loads to the memory

location with the generalized assignment's temporary variable as the destination are inserted instead of predicate clear operations. No predicate set operations are inserted and no guard predicates are placed on sunken stores. Thus, a speculatively sunken store will either store the value loaded earlier from the same location or a new value that would have been stored at an original computation point.

5.3.1 PDE compilation time

Presented first are compile-time overhead costs for the algorithm. Table 5.3 shows the time spent in the Lopti compilation stage. The SM-PRE optimization is active, but LVM is not. A final speculative PRE iteration and local optimization are performed after P3DE to remove any remaining redundancy; this is often useful for loads inserted during nonpredicated optimization. Thus, the additional time of the P3DE compilation includes the P3DE algorithm, additional speculative PRE and local optimization overhead, and any second-order effects during the merging of empty blocks.

5.3.2 Store removal

Figure 5.3 shows the change in the execution frequency of store operations as a result of applying the P3DE algorithm. The number of store operations issued is separated into two parts: $p = 1$ shows the percentage of store operations that execute and retire, while $p = 0$ is the percentage of dynamic store operations squashed because of a false predicate evaluation. The numbers are normalized against the number of store operations issued by the baseline. In nearly all cases the number of store operations issued after applying P3DE is less than the

Table 5.3 Lopti PDE Compile Time (in seconds)

Benchmark	Baseline	P3DE	Increase	Pct. Increase
164.gzip	42.11	48.52	6.41	15.22%
175.vpr	291.09	311.96	20.87	7.17%
181.mcf	18.44	19.34	0.90	4.88%
186.crafty	1137.91	1183.31	45.40	3.99%
197.parser	197.80	208.09	10.29	5.20%
253.perlbmk	5501.80	5759.81	258.01	4.69%
254.gap	1231.56	1303.10	71.54	5.81%
255.vortex	1001.90	1170.08	168.18	16.79%
256.bzip2	55.22	61.58	6.36	11.52%
300.twolf	1040.82	1115.05	74.23	7.13%

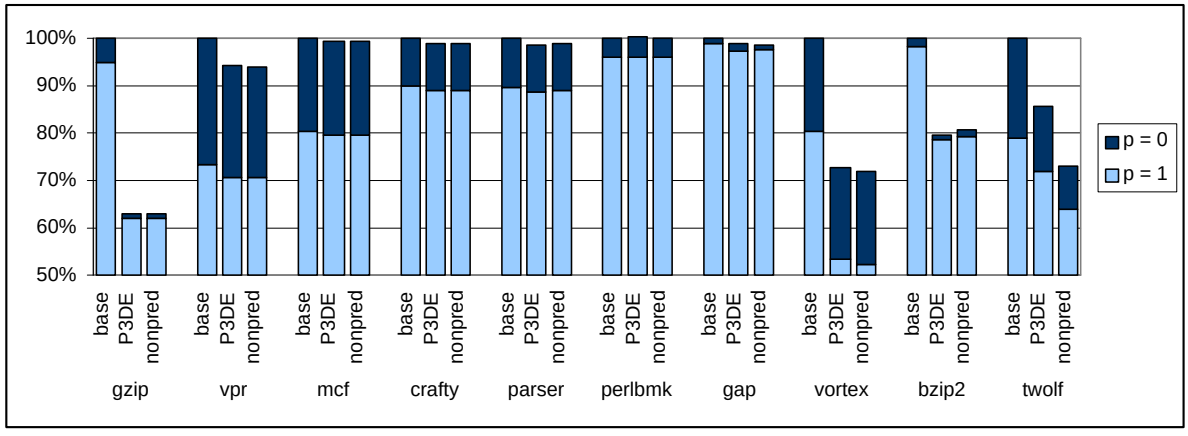


Figure 5.3 Retired (“p = 1”) and Predicate-Squashed (“p = 0”) Store Operations

baseline. This indicates healthy profile information and proper code motion. The total number of stores issued is reduced by up to 36%, with a reduction in both the number of squashed and the number of executed store operations. This reflects some interaction with subsequent hyperblock formation [25]. As expected, an increase can sometimes be observed in the number of squashed stores (as in *254.gap*) due to the predication of sunken stores lying on unsafe paths.

In benchmarks where store reduction is significant, the nonpredicated optimization is capable of performing the same degree of store removal as P3DE. This is evidence that the majority of stores removed in these cases are those to safe addresses. The apparent further reduction of stores by the nonpredicated optimization in *300.twolf* is an artifact of IMPACT's predicate register live range detection, which generates unnecessary spill code for individual predicates in P3DE. These stores only slightly lengthen the schedule and are seldom loaded back, so this does not have a major impact on performance. Better control of predicate register pressure is a subject for future work.

Figure 5.4 shows the ratio of the increase in predicate operations to the number of store executions removed when P3DE is applied. This is a convenient measure of the efficiency of the predication apparatus involved in aggressive sinking of partially dead stores. An important note here is that the predicate operations represented in the graph are those of *all* predicate operations, including those introduced from the sinking of nonstore operations, as well as those from hyperblock formation and other optimizatins. In general, an excessive number of predicate operations is not executed in place of migrated stores. The use of the availability property to avoid using predicate assignments helps reduce the amount of predication introduced.

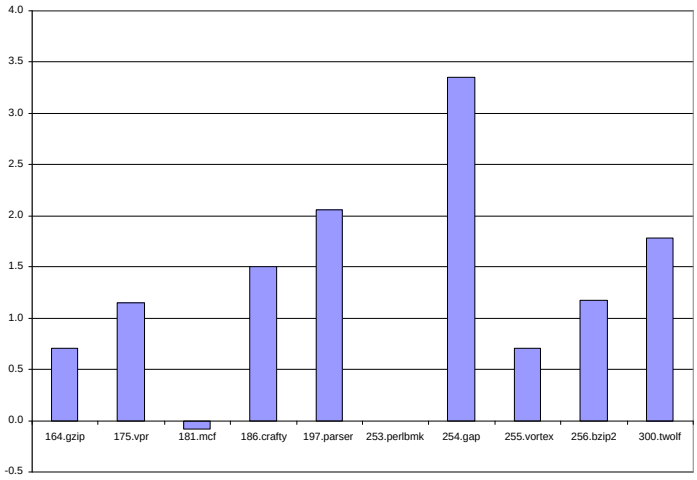


Figure 5.4 Approximating Predicate Defines Inserted per Store Removed

Subsequent optimizations in the IMPACT compiler may also combine, reduce, or eliminate predicate operations introduced by P3DE: these optimizations help to further control predicate usage and assist in lowering the ratio of predicate operations executed to stores. The apparently dramatic increase in predicate operations per store removed in *254.gap* is an effect not of P3DE but of a subsequent optimization (triggered by the effects of code motion) that generates several predicate assignments. The actual percentage increase in the number of predicate defining operations was 3.54%, so the magnitude of this change is greatly exaggerated by the low number of stores removed in those benchmarks, as seen in Figure 5.3. The same optimization is not triggered after P3DE *181.mcf*, resulting in an overall reduction of predicate operations. Because *253.perlbmk* has both a decrease in predicate operations and an increase in store operations (99.63% and 100.22% of baseline, respectively), its ratio does not represent a relevant relationship and is excluded from the figure.

Table 5.4 P3DE Estimated SPEC Results with SM-PRE and without LVM

Benchmark	Baseline	P3DE	Nonpred	P3DE Pct. Increase	Nonpred Pct. Increase
164.gzip	660	666	668	0.91%	1.21%
175.vpr	540	542	540	0.37%	0.00%
181.mcf	242	243	244	0.41%	0.83%
186.crafty	582	598	601	2.75%	3.26%
197.parser	423	424	424	0.24%	0.24%
253.perlbmk	528	528	529	0.00%	0.19%
254.gap	400	399	400	-0.25%	0.00%
255.vortex	1002	1016	1009	1.40%	0.70%
256.bzip2	523	527	519	0.76%	-0.76%
300.twolf	626	649	643	3.67%	2.72%

5.3.3 PDE SPEC performance

Finally, Table 5.4 shows the net performance results for the P3DE scheme. Larger estimated SPEC numbers indicate better performance. The results presented are the median score of five consecutive runs of the benchmark. The Itanium 2 microarchitecture is capable of issuing no more than two store instructions per clock cycle, but is capable of issuing up to six predicate instructions per clock cycle. Thus, migration of stores out of heavily store-laden regions can allow certain code segments to be executed in fewer cycles. This is the cause of performance increases in *164.gzip* and *300.twolf*; P3DE was able to sink stores out of the primary loop of the benchmark. *300.twolf* in particular has basic blocks comprised mainly of long sequences of store assignments: a significant source of store reduction is the inlining of the function ucxx2() into uloop(). P3DE was able to sink many global pointers out of the section of code originally belonging to ucxx2() and out of the primary loop of uloop(); disallowing this inlining would prevent motion of the stores and results in no performance gain in this code region.

There are several reasons why the P3DE optimization generally does not have a more significant effect on program performance. PDE schemes reduce the execution of code which does not have an effect on the result of the program. Because of the large width of the Itanium 2 implementation, partially dead instructions can often be statically scheduled without lengthening the schedule. This is the reason why *164.gzip*, although having a reduction of 36% in dynamic stores, barely shows a 1% performance gain. PDE will have an effect when partially dead instructions occupy limited resources that prevent a tighter schedule, such as the store-laden regions in *300.twolf*.

In several cases, nonpredicated optimization is capable of achieving similar performance gains as P3DE. In these cases, the majority of P3DE's performance gains are from sunken stores to safe locations. The reason for the difference in program performance between the P3DE and nonpredicated versions of the benchmarks are due to the different tradeoffs made in the optimization: P3DE utilizes predication to guard execution of stores, while the nonpredicated optimization promotes the variable to a register, loading the value at the entrances of the region. The hot path of a code region often stores a new value to the promotion register, rendering the load inconsequential: in this case, the predicate operations used to guard execution in P3DE is pure overhead. The situation is exacerbated when predicate register spill and fill code is present. *164.gzip* has better performance with the nonpredicated optimization for this reason.

Tables 5.5 and 5.6 show the SPEC performance of optimal PDE and P3DE with loop variable migration active, to see their benefit as incremental optimizations. As expected, PDE and

Table 5.5 PDE Estimated SPEC Results with Loop Variable Migration

Benchmark	Baseline	Optimal PDE	P3DE	Opt. PDE Pct. Increase	P3DE Pct. Increase
164.gzip	665	656	652	-1.35%	-1.95%
175.vpr	517	517	518	0.00%	0.19%
181.mcf	239	240	240	0.42%	0.42%
186.crafty	613	624	619	1.79%	0.98%
197.parser	448	452	452	0.89%	0.89%
253.perlbmk	546	564	548	3.30%	0.37%
254.gap	391	399	399	2.05%	2.05%
255.vortex	918	928	938	1.09%	2.18%
256.bzip2	556	523	531	-5.94%	-4.50%
300.twolf	625	654	624	4.64%	-0.16%

Table 5.6 PDE Estimated SPEC Results with SM-PRE and Loop Variable Migration

Benchmark	SM-PRE	Optimal PDE	P3DE	Opt. PDE Pct. Increase	P3DE Pct. Increase
164.gzip	692	688	688	-0.58%	-0.58%
175.vpr	521	490	501	-5.95%	-3.84%
181.mcf	242	242	243	0.00%	0.41%
186.crafty	611	609	605	-0.33%	-0.98%
197.parser	450	453	450	0.67%	0.00%
253.perlbmk	556	559	544	0.54%	-2.16%
254.gap	396	402	403	1.52%	1.77%
255.vortex	941	992	994	5.42%	5.63%
256.bzip2	550	553	544	0.55%	-1.09%
300.twolf	650	635	611	-2.31%	-6.00%

P3DE by themselves have little effect on performance. However, when SM-PRE and PDE are combined, there is a synergistic effect on *255.vortex* with a result that is greater than the sum of its parts. This is evidently the result of more efficient Hyperblock formation.

CHAPTER 6

FUTURE WORK

In the results, the PCE methods were not shown to have a large positive effect on performance. In some cases, the negative effects were quite pronounced. However, this thesis only analyzes PRE and PDE for a specific benchmark on a specific architecture, so future work may be able to achieve worthwhile improvements in other situations. There are several remaining avenues of exploration available for both PRE and PDE.

A major problem encountered in PRE performance was the speculation of loads whose addresses forced invocation of a software page table handler. These loads have a cost which is greatly disproportionate to their unspeculated execution cost. The cost edges of P3DE could be adapted to factor this cost into the minimum cut calculation of SM-PRE, but this would limit profitable motion if performed indiscriminately. An analysis that could determine when a load's address is likely to generate this kind of problem would be needed to direct the placement of cost edges.

As stated in Chapter 5, no published method exists to specifically balance register pressure and redundancy removal via PRE for an architecture with a register stack engine. One method to do this is to determine whether motion would increase the number of allocated registers in a function, then factor in the spill and fill costs at function boundaries. An offshoot of this might be to aggressively perform all possible PRE, then introduce redundancy to reduce register

pressure in low trip-count areas, via rematerialization [38], in order to reduce the number of registers.

One idea not previously demonstrated in either PRE or PDE is to identify chains or traces of expressions/assignments which are then migrated as a single unit. One hidden cost of performing PCE is the need to split critical edges, which may result in decreased performance: knowing how many computations can be moved into the split edge may result in better trade-offs. In addition, being able to move traces in PDE would save the effort of doing predicate recombination and allows the compiler to perform optimizations based on the total cost/benefit of moving assignments. For example, individual assignments in the chain may not be worth the cost of predication, but replacing several assignments with a single predicate write operation would likely be cost-effective.

For PDE, replacement of the largely lexical assignment identification mechanism with a system capable of recognizing lexically different assignments that compute the same results would extend the algorithm's ability to match and sink assignments. This is similar to the use of Bodik's value-flow analysis for PRE, as presented in his Ph.D. work [13]. One approach may be to adapt the static single use (SSU) representation presented by Lo et al. [9], as described in Chapter 2. The results also suggest interactions between P3DE and if-conversion; this might be leveraged with predicate optimizations to enable even more efficient sinking.

Currently, certain kinds of predicate operation optimizations are not performed that would help the P3DE algorithm further reduce the usage of predicates. There are two situations that have not yet been addressed:

1. Predicates which occur in the same locations but are associated with different assignments (such as a chain of delayed assignments): these could be combined. This is not limited to those predicate operations which occur in the same locations, but can also be applied to supersets and intersecting predicates which could be combined.

2. Sets and clears performed in alternate sides of a hyperblock hammock structure: these could borrow the value of the hammock predicate.

Although there was not a significant increase in the number of introduced predicate operations, further predicate combination optimization would help reduce predicate spill and fill code and improve performance.

A more accurate way to find an optimal placement for the P3DE scheme might be to perform the initial stage of hyperblock formation to mark where predicates will be formed as part of that optimization and could be reused for P3DE.

CHAPTER 7

CONCLUSION

This thesis presents implementations of partial code elimination techniques in the IMPACT compiler, consisting of two different methods of both partial redundancy elimination and partial dead code elimination. It presents a new PDE algorithm, P3DE, which removes partial dead code by taking advantage of edge profile data and predication support, covering both acyclic and cyclic code in a systematic manner. The motion graph abstraction, inherited and adapted from SM-PRE, incorporates decision factors and predication requirements into an easily manipulable form, allowing convenient and effective PDE in and around loops. When combined together, SM-PRE and P3DE are capable of subsuming other optimizations such as loop-biased register promotion.

Empirical, real-machine measurements are presented for PCE. The experiements demonstrate removal of up to 2/3 measured reused loads for the PRE methods. For PDE, up to 36% of dynamic stores were removed without a disproportionate increase in predicate operations. For real-time clock performance, PCE is shown to sometimes have a positive effect on performance, particular when synergies are found with other optimizations. However, this occurs in limited cases when compiling SPECint 2000 for the Itanium Architecture. In addition, PCE can also have negative effects on performance, due to increased register pressure or other factors.

PRE may be valuable for its ability to subsume multiple classical optimizations, but the additional benefits it provides are limited. PDE generally does not provide much additional performance. Thus, as incremental optimizations in IMPACT for SPECint 2000, they have a limited use as benchmark-specific optimization but cannot provide consistent positive improvement to the suite.

REFERENCES

[1] J. Knoop, O. Rüthing, and B. Steffen, "Lazy code motion," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementaton*, June 1992, pp. 224–234.

[2] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementaton*, June 1994, pp. 147–158.

[3] R. Bodik and R. Gupta, "Array data-flow analysis for load-store optimizations in superscalar architcetures," in *Proceedings of the Eighth Annual Workshop on Languages and Compilers for Parallel Computing, LNCS 1033*, August 1995, pp. 1–15.

[4] R. Gupta, D. Berson, and J. Fang, "Path profile guided partial dead code elimination using predication," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, November 1997, pp. 102–115.

[5] R. Bodik and R. Gupta, "Partial dead code elimination using slicing transformations," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, June 1997, pp. 159–170.

[6] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu, "A new algorithm for partial redundancy elimination based on SSA form," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, June 1997, pp. 273–286.

[7] R. Bodik, R. Gupta, and M. L. Soffa, "Complete removal of redundant computation," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, June 1998, pp. 1–14.

[8] R. Gupta, D. Berson, and J. Fang, "Path profile guided partial redundancy elimination using speculation," in *Proceedings of the International Conference on Computer Languages*, May 1998, pp. 230–239.

[9] R. Lo, F. Chow, R. Kennedy, S. Liu, and P. Tu, "Register promotion by sparse partial redundancy elimination of loads and stores," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998, pp. 26–37.

[10] J. Knoop, O. Rüthing, and B. Steffen, "Optimal code motion: Theory and practice," *ACM Transactions on Programming Languages and Operating Systems*, vol. 16, pp. 1117–1155, July 1994.

[11] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual Volume 1: Application Architecture, Document Number 245317-003*, December 2001.

[12] Standard Performance Evaluation Corporation, "SPEC CINT2000 benchmarks," 1999. http://www.spec.org/cpu2000/CINT2000.

[13] R. Bodik, "Path-sensitive, value-flow optimizations of programs," Ph.D. dissertation, University of Pittsburgh, Pittsburgh, PA, 1999.

[14] E. Morel and C. Renvoise, "Global optimization by suppression of partial redundancies," *Communications of the ACM*, vol. 22, pp. 96–103, February 1979.

[15] D. E. Knuth and F. R. Stevenson, "Optimal measurement points for program frequency counts," *BIT*, vol. 13, pp. 313–322, 1973.

[16] R. N. Horspool and H. C. Ho, "Partial redundancy elimination driven by a cost-benefit analysis," in *8th Israeli Conference on Computer Systems and Software Engineering*, June 1997, pp. 111–118.

[17] Q. Cai and J. Xue, "Optimal and efficient speculation-based partial redundancy elimination," in *Proceedings of the 2003 International Symposium on Code Generation and Optimization*, March 2003, pp. 91–102.

[18] L. T. Simpson, "Value-driven redundancy elimination," Ph.D. dissertation, Rice University, Houston, TX, 1996.

[19] R. Bodik and S. Anik, "Path-sensitive value-flow analysis," in *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, January 1998, pp. 237–251.

[20] R. Bodik, R. Gupta, and M. L. Soffa, "Load-reuse analysis: Design and evaluation," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, May 1999, pp. 64–76.

[21] J. Lin, T. Chen, W. Hsu, P. Yew, R. Ju, T. Ngai, and S. Chan, "A compiler framework for speculative analysis and optimizations," in *Proceedings of PLDI 2003*, 2003, pp. 289–299.

[22] L. Feigen, D. Klappholz, R. Cassazza, and X. Xue, "The revival transformation," in *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1994, pp. 421–434.

[23] R. Bodik, R. Gupta, and M. L. Soffa, "Interprocedural conditional branch elimination," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, June 1997, pp. 146–158.

[24] B. Steffen, "Property oriented expansion," in *Proc. Int. Static Analysis Symposium (SAS'96), Aachen (Germany)*, vol. 1145 of *Lecture Notes in Computer Science (LNCS)*, September 1996, pp. 22–41.

[25] D. I. August, W. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 92–103.

[26] P. Briggs and K. D. Cooper, "Effective partial redundancy elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, June 1994, pp. 159–170.

[27] B. C. Cheng and W. W. Hwu, "Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation," in *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000, pp. 57–68.

[28] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," in *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, 1986, pp. 135–146.

[29] B. V. Cherkassky and A. V. Goldberg, "On implementing the push-relabel method for the maximum flow problem," *Algorithmica*, vol. 19, no. 4, pp. 390–410, 1997.

[30] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[31] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, December 1992, pp. 45–54.

[32] R. Gupta, D. Berson, and J. Fang, "Resource-sensitive profile-directed data flow analysis for code optimization," in *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997, pp. 358–368.

[33] R. Gupta and R. Bodik, "Register pressure sensitive redundancy elimination," in *Proceedings of the International Conference on Compiler Construction, LNCS 1575*, 1999, pp. 107–121.

[34] S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann Publishers, 1997.

[35] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998, pp. 227–237.

[36] D. Mosberger and S. Eranian, "Chapter 9: Understanding system performance," in *IA-64 Linux Kernel: Design and Implementation*, Upper Saddle River, NJ: Prentice Hall, 2002, pp. 363–408.

[37] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[38] P. Briggs, K. D. Cooper, and L. Torczon, "Rematerialization," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, June 1992.