Path-Sensitive, Value-Flow Optimizations of Programs

by
Rastislav Bodík
Diploma, Technical University Kosice, Slovakia, 1992
M.S., University of Pittsburgh, 1994

Submitted to the Graduate Faculty of
Arts and Sciences in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

University of Pittsburgh
1999

UNIVERSITY OF PITTSBURGH

FACULTY OF ARTS AND SCIENCES

This thesis was presented

by

Rastislav Bodík

It was defended on

May 31 1999

and approved by

Prof. Rajiv Gupta

Prof. Mary Lou Soffa

Prof. Mark Moir

Dr. Michael Schlansker (HP Labs)

Committee Chairperson

Path-Sensitive, Value-Flow Optimizations of Programs

Rastislav Bodík, Ph.D.

University of Pittsburgh, 1999

Current compiler optimizers are conservative and inflexible. As a result, even "highly optimized" programs execute half of their instructions redundantly, only to recompute previously computed values. Ideally, these values should be remembered and later *reused*, removing recomputations.

Unfortunately, this reuse strategy fails often. The culprit is intermittent reuse—one that exists only along some execution paths leading to the redundant instruction. This path-specific reuse is frequent, but to remove it, the optimizer may need to pay the *exponential* price of optimizing each path separately.

This thesis describes how to defeat this exponential path explosion, in its various forms: how to analyze paths separately only when it matters, via demand analysis; how to generate less path-specific code, via optimal profiling feedback; and how to avoid profiling individual paths, via adding confidence to imprecise profiles. The result is a path-sensitive optimization framework that is powerful enough to remove nearly all redundancies, yet practical enough to permit an industrial-strength implementation.

More specifically, the various forms of path explosion are attacked by dividing the optimizer into three stages, each responsible for one aspect of paths. The *representation* exposes the reuse of values, the *analysis* collects paths with exposed reuse, and the *transformation* exploits the collected reuse by removing the redundancies.

The representation models the flow of a recomputed value. By symbolically naming the value, it reduces the (invisible) value flow into a (visible) data flow. As a result, recomputation is detected essentially as reaching definitions. The representation, called the *Value Name Graph*, names the value on demand, given a set of optimized computations. To obtain path-sensitivity, the names are formed separately for each path, but are analyzed separately only when it matters, i.e., when the value has a different name on a path.

The analysis marks paths with value reuse and weighs the marked paths with a run-time profile to guide the subsequent transfromation. Unfortunately, precise weighing requires frequencies of all paths. To make profile-guiding practical, this thesis develops a family of *estimator* algorithms based on *edge* frequencies, a cheaper but inherently imprecise alternative to path profiles. When weighing the reuse, estimators bound the inherent error, adding confidence to imprecise profiles.

The transformation combines three orthogonal methods, to effect a complete removal of recomputations, while reducing the exponential growth of program size growth caused by generating path-specific code. The expensive *path duplication* is resorted to only when the growth-free *code motion* fails to transform the program, and only when the profile-driven *control speculation* cannot profitably impair some paths to optimize others.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation: programs recompute values (often)

This thesis develops a framework for a broad class of *value-flow* optimizations. Common to these optimizations is removal of program instructions that recompute values that are already known, because they have already been computed (by an execution of the same instruction or some other instruction), or because they can be computed at compile time. When two instructions compute the same value, we say that the *value flows* between them. In the following program, the value of the expression $A[i]$ flows to the expression $A[i-1]$ (computed one iteration later), and to the expression $A[i-2]$ (computed two iterations later).

```
for i=3,N do
   A[i] = A[i-1] + A[i-2]
end for
```

Redundant value recomputations offer a conceptually simple yet powerful optimization: rather than recomputing the new value, the old one is *reused*. The most common way to reuse the value is to keep it in a register until it is needed again. In the example program, the expressions $A[i-1]$ and $A[i-2]$ redundantly recompute the value. The computation (involving address computation and a memory load) can be removed using two registers that carry the value of $A[i]$ for two iterations, as shown below. On the entry to any loop iteration, the registers r_1, r_2 carry the values of $A[i-1]$ and $A[i-2]$, respectively.

```
r1 = A[2]
r2 = A[1]
for i=3,N do
   t = r1 + r2
   r2 = r1
   r1 = A[i] = t
end for
```

Redundant recomputations occur frequently. Based on value-flow redundancy elimination, current compilers remove about 30% of executed arithmetic instructions [BC94] and 10% of executed conditional branches. To estimate the amount of redundancy *present* in the program, we can refer to hardware run-time prediction mechanisms. Based on remembering old values and predicting new ones, these techniques predict correctly as much as 80% of values [SVS96] and 95% of conditional branches [YP91]. Clearly, what can be predicted is in some sense redundant. Even though not all of the predicted values can be removed by a

```
while ...  do                   function compare(x, y)
   if compare(A[i], B[k])          if x.f < y.f
      C[top++] = A[i++]               return 1
   else                            else
      C[top++] = B[k++]               return 0
   end if                          end if
end while                       end compare
```

Figure 1.1: **A fragment from Merge Sort.** The load of $B[k]$ and the *if compare* statement are redundant. Both redundancies are path-specific.

compiler, the high prediction rates suggests that programs are inherently highly redundant, which encourages us to develop techniques that improve the compilers optimizations.

Figure 1.1 shows a fragment from a Merge Sort program. It illustrates why there is so much redundancy, but also why optimizing it away is not trivial, for the programmer or for the compiler. The *while* loop merges two sorted arrays A and B. In each loop iteration, it compares their top elements and move the smaller one to the accumulator array C. The memory access to the top elements is redundant: if $A[i]$ is the smaller element, then $B[k]$ is not moved and so $B[k]$ in the next iteration refers to the same array element. The $B[k]$ expression produces the same value; it is a value-flow redundancy. As in the first example, the optimizer should remember the previous value of $B[k]$ in a register, reuse it and remove the load of $A[i]$.

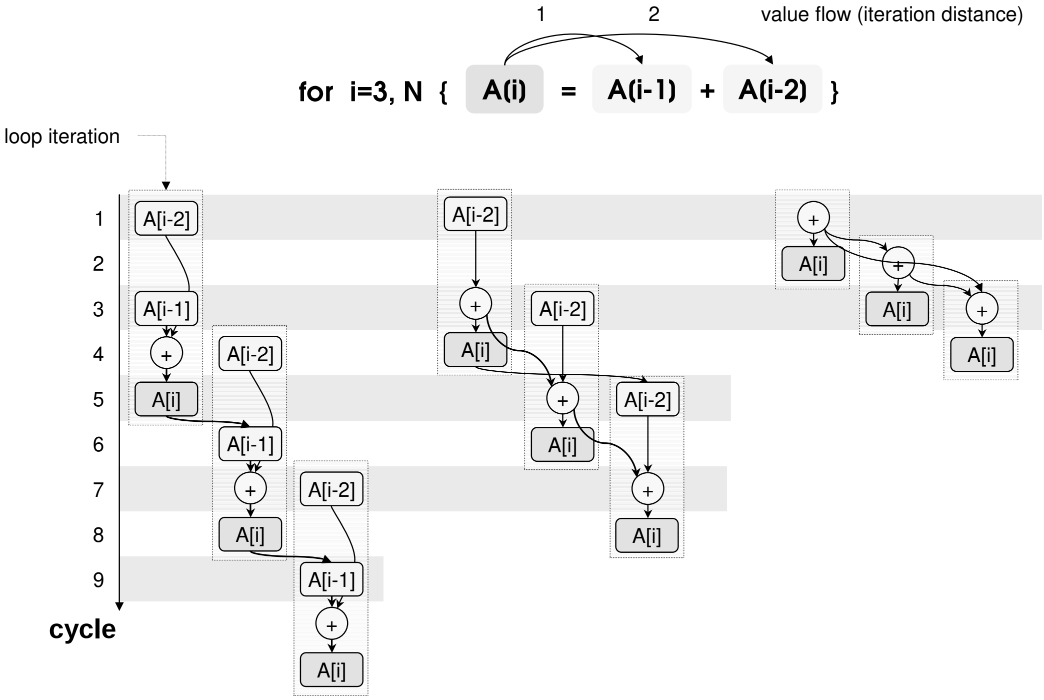
Unfortunately, such an optimization is not directly applicable because $B[k]$ produces the same value only along one path through the loop. Along the other path, k is incremented, after which $B[k]$ will refer to a different array element and hence must be loaded from the memory.

The Merge Sort example exhibits another redundancy. Observe that the *if* statement in the *while* loop always branches in the same direction as the *if* in the function compare. This branch correlation is also a case of value-flow redundancy, because the two conditional expressions always compute the same boolean value. Because we always know the direction of the redundant branch, we would like to bypass and transfer the control from the return points directly to the branches of the *if* statement. Here again, we cannot simply remove the redundant conditional branch. To bypass it, we need to know along which path we arrived at the branch. Compare the (failed) optimizations of $B[k]$ and the *if* statement. While the former is optimizable only along some paths, the latter is optimizable along all paths, but each path offers a different optimization. Another noteworthy difference is that the $B[k]$ optimization removes a *data* dependency, and the *if* redundancy removes a *control* dependency.

In summary, value-flow redundancy exists even in reasonably well-written programs, but is often hard to remove. For the programmer, their removal would require awkward (and error-prone) usage of temporary variables. For the compiler, it would require distinguishing individual program paths, a non-trivial task, as elaborated later in this chapter. In fact, the two redundancies in the Merge Sort above are beyond the power of existing optimizers, but the optimizer presented in this thesis can remove them.

1.2 Optimization benefits

In general, the removal of redundant instructions may speed up the program in two ways:



a) unoptimized execution b) critical path reduced c) HW resources freed

Figure 1.2: **Two benefits of removing value-flow redundancies:** reducing critical path of data dependences; reducing the resource requirements of the program.

1. *Reduce hardware requirements of the program.* When the redundant instruction is removed, there are fewer instructions to execute. As a result, there is less contention for hardware resources (functional units, registers, cache ports), which allows scheduling the (unoptimized) instructions earlier.

2. *Shorten the critical path of data dependences.* Because the reused value is available sooner than the recomputed one, instructions that need the value can be scheduled earlier. Essentially, removing an instruction breaks one path of data dependences among instructions. When this path is critical (i.e., longest), the program may complete sooner.

Note that the instruction schedule is improved regardless of whether it is created statically (by a compiler) or dynamically (by an out-of-order processor). Also observe that, while the resource constraints can be overcome with wider processors, the critical path constraint reflects the data-flow limit of the program and hence require program transformation.[1]

Figure 1.2 illustrates the two optimization benefits on the running example. The cycle-by-cycle diagram compares the schedule of the unoptimized program with the program in which only the redundant $A[i-1]$ was removed (b), and with the program in which both $A[i-1]$ and $A[i-2]$ were removed (c). Each iteration of the loop body depicts only those operations that influence the speed of the loop (the loads, the add, and the store). We measure the loop speed using the iteration initiation rate, i.e., the number of processor cycles after which a new loop iteration can be started; the fewer cycles, the faster the loop execution. To

---

[1] We note that recently proposed hardware techniques attack the data-flow limit [SVS96, SS97, LS97]. In Section 1.3 we argue that, like compiler optimizations, they transform the program, albeit on the fly, and without explicitly rewriting the code.

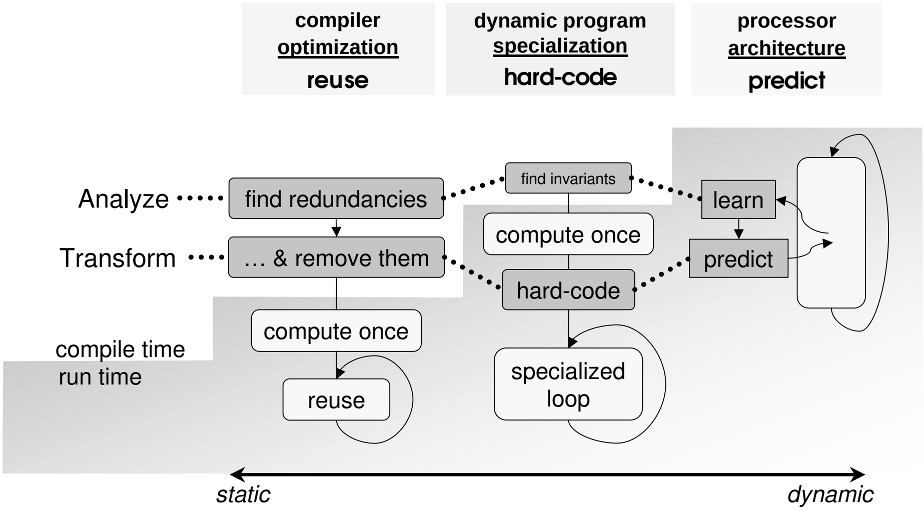


Figure 1.3: **The spectrum of approaches to value-flow optimization.** The spectrum extends from static (compile-time) to dynamic (run-time).

demonstrate the effect of insufficient hardware resources, assume that the processor can issue at most one memory instruction (load or store) per cycle.

In the unoptimized program, the add instructions from subsequent iterations lie on a critical path of data dependences. The iteration-to-iteration length of this path is three cycles, composed of add, store, and load latencies. The critical path does not allow issuing a new iteration faster than every three cycles. After removing the load of $A[i-1]$ (b), the adds communicate the value directly via register. The length of the critical path has been shortened to one cycle. As a result, the next iteration can be issued each two cycles. Note that the optimal issue rate of one cycle has not been achieved (despite the minimized critical path), as only one memory instruction can execute per cycle, and each iteration has two such instructions. After the load of $A[i-2]$ is removed (c), resource requirements are reduced. Each iteration contains only one memory operation, allowing the issue rate of one cycle. In summary, the final optimization benefited from reducing both the critical path and the resource requirements. Both of them resulted from the value-flow optimization.

1.3 Compiler optimization $\subset$ program optimization

The previous sections made three important points:

1. Programs are highly value-flow redundant.
2. Removal of value-flow redundancies has important benefits.
3. Current compilers are not very good at removing the redundancies.

Because of (1) and (2), redundancy elimination is perhaps the most important optimization for instruction-level parallelism. Because of (3), it should not come as a surprise that *compiler optimization* is not the only technology for *program optimization* focused on value flow. Indeed, there are at least two other: *dynamic program specialization* and *processor architecture*. Based on different principles than value reuse, together with compile optimization, they represent a spectrum that places this thesis in a broader research context.

Figure 1.3 contrasts the the three approaches to value-flow optimization. What is common to all of them is that they first analyze the program and then transform it. What differs is how the optimization labor is divided between compile-time and run-time. Consider how they optimize a loop that contains a loop-invariant statement, i.e., a statement that computes the same value in each iteration, and thus is redundant.

In compiler optimization, the analysis finds this redundant statement and then transforms the program so that when it is run, the loop-invariant value is computed only once and then stored in a register from which it can be reused when needed later.

Dynamic program specialization delays the transformation until run-time [$\text{APC}^{+}96$,KEH91,$\text{GMP}^{+}97$, CN96, MCB99]. Rather than reusing the value from a register, the value (once known at run time) is hard-coded into the loop, using run-time code generation. The result is a loop specialized for the given (loop-invariant) value. Delaying the transformation until run-time has an important benefit. Since the known value can be hard-coded into the program, there is no need to occupy a register, a scarce resource. The down side is that the cost of run-time program transformation.

Processor architecture of modern superscalar processors perform a fully run-time optimization, using *value prediction* [SVS96, LS97]. They observe computed values, learn how they change, and predict future ones. When the value is predicted correctly, the program is effectively transformed, because future instructions can execute earlier. However, when the value is mis-predicted, we pay a penalty of having to re-execute mis-predicted instructions. (Processor architecture also uses *instruction reuse*, a non-predicting technique that avoids recomputation of values by memoizing previous arguments of instructions [SS97].)

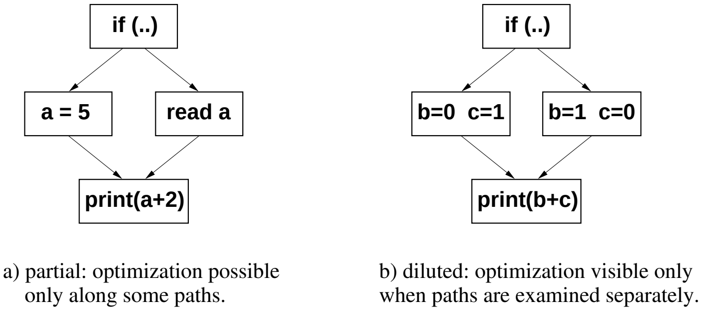
The three technologies form a spectrum, ranging from purely static to purely dynamic. Each of them offers unique advantages. The focus of this thesis is the compiler optimization. Its important advantage is zero run-time cost; there is no run-time code generation or mis-predictions. The disadvantage is that the compiler is blind to run-time values. Hence, one of the driving motivations behind this research is to explore how much redundancy can be removed by analyzing the program text alone. In the long run, such a compiler-centric research may indicate what kind of optimization should be delayed to run-time, and how they it should be performed.

1.4 Path sensitivity

Among the three technologies, compiler optimization is the oldest. In fact, the problem of redundancy removal is as old as the first compilers [Coc70]. One reason why optimizers are still far from saturating their possibilities is that they are not sufficiently *path-sensitive*.

The obstacles posed by paths have two distinct flavors. First, the optimizer may fail when the redundancy is ”partial”, i.e., when the optimization is possible only along some program paths. In Figure 1.4(a), one can replace the computation of $a + 2$ with a simple constant 7 along one incoming path but not the other. There, $a + 2$ must be computed at run-time. We say that $a + 2$ is *partially redundant*.

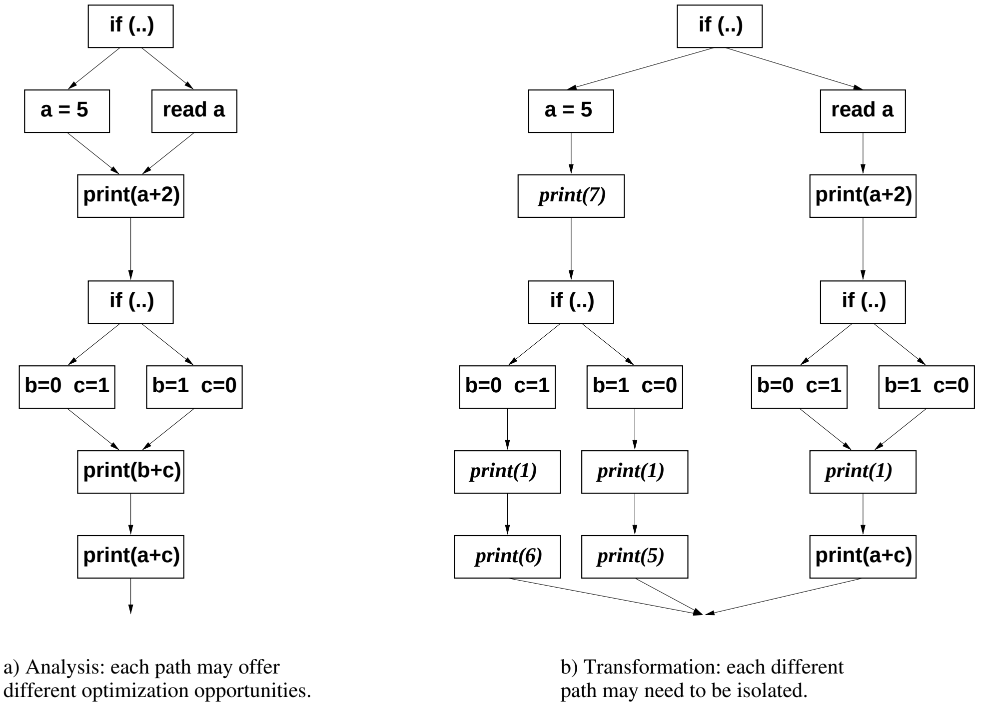
The second flavor is more tricky, as shown in Figure 1.4(b). While $b + c$ equals 1 along all paths, common analyzers would discover this fact only if each path is analyzed separately. When analyzed together, this fact would be diluted, because neither the variable b nor c has a single unique constant value along all paths considered together. The values of b and c are either 0 or 1, and so they are not considered to be a constant. Consequently, their sum is assumed not to be a constant. This problem is know as *non-distributivity* of a dataflow problem. Non-distributive (formulations of) dataflow problems produce imprecise information because they dilute the information about individual control flow paths.



a) partial: optimization possible only along some paths.

b) diluted: optimization visible only when paths are examined separately.

Figure 1.4: **Two flavors of path sensitivity.**



a) Analysis: each path may offer different optimization opportunities.

b) Transformation: each different path may need to be isolated.

Figure 1.5: **Exponential path explosion:** in analysis, and in transformation.

1.5 Challenges: exponential path explosion

The reasons why analyzers do not examine program paths separately is that there is an exponential number of paths, even in program with no loops. To stay practical, analyzers treat paths together, summarizing their results whenever paths meet, diluting optimization opportunties.

Unfortunately, paths explode exponentially not only in analysis, but also in program transformation, when we want to exploit the optimizable paths. To enable the optimizations in Figure 1.5, two paths had to be physically separated via code duplication. This duplication may cause an exponential code growth: we obtained three copies of the `print(1)` node, even though all three copies have the same contents. The result is the undesirable code growth, which may grow exponentially in the number of program nodes.[2]

---

[2] Real programs have loops and thus unbounded number of paths. Even though we optimize cyclic paths, we still pay just an exponential cost.

Given the inevitable exponential cost, the imperative for path-sensitive optimization is to exploit individual program paths only as far as it is practical. At the core of this thesis is a battle against exponential path explosion: because of the adverse effects of code growth, the more we suppress the growth, the more practical the optimization. While some practical path-sensitive optimizers exist, as our experiments will document, there is much room for improvement. This thesis advances the boundary of what is practical in value-flow optimizations. As a result, we can double the benefit over existing optimizers.

1.6 The thesis

The goal of this thesis is to develop compiler optimization algorithms that will tartget value-flow problems. The distinguishing feature of these algorithms will be that they are path-sensitive. More specifically, we want to demonstrate that *path-sensitivity can be made*

1. ***effective:*** *in that it improves optimizer's power,*
2. ***practical:*** *in that the path-sensitive optimizer will be immune to the exponential blow-up,*
3. ***broad:*** *in that path-sensitivity will be applicable to all value-flow optimization problems.*

Chapter 2

PathFinder: the Optimization Framework

The remainder of this thesis presents PathFinder, a framework for deriving *path-sensitive*, *value-flow* compiler optimizations. The term *path-sensitive* denotes *how* the optimization is performed: PathFinder exploits both *diluted* and *partial* path-specific optimization opportunities, i.e., it attacks both flavors of path-sensitivity (see Section 1.4).

The term *value-flow* describes *what* computations are targeted by PathFinder. The PathFinder framework can derive optimizers for the value-flow class of computations, which include standard optimizations such as the removal of common subexpressions, loop invariants, partial redundancies, array bound checks, conditional branches, redundant loads and stores, dead code and also constant propagation (see Section 1.1). PATHFINDER generalizes these optimizations, by providing a uniform framework, and improves thir power, by making them path-sensitive.

2.1 The optimizer stages

When designing the path-sensitive value-flow framework, it was required that

1. handle both path-sensitivity flavors, and
2. can be parameterized and tuned for the various optimization tasks, including targeting arithmetic expressions, memory accesses operations, conditional branches, etc.

The two goals were accomplished by separating the optimizer into three stages, shown in Figure 2.1. In this optimizer architecture,

- *value-flow representation* exposes the reuse of values,
- *dataflow analysis* collects paths with exposed reuse, and the
- *program transformation* exploits the collected reuse by removing the redundancies.

In finer detail, the representation builds a program model—a graph that connects equivalent computations. The novel representation, called the *Value Name Graph*, answers two questions: which computations are value equivalent, and along which path the equivalence holds. The representation is responsible for avoiding *dilution* of path-specific value flow (recall the second flavor of path-sensitivity shown in Figure 1.4(b)). The dilution is avoided by naming the value as it flows through the program, as if each path was analyzed separately. This naming technique improves the optimization also when paths need not be considered separately, through the use of symbolic names that exposes non-trivial relationships among instructions.

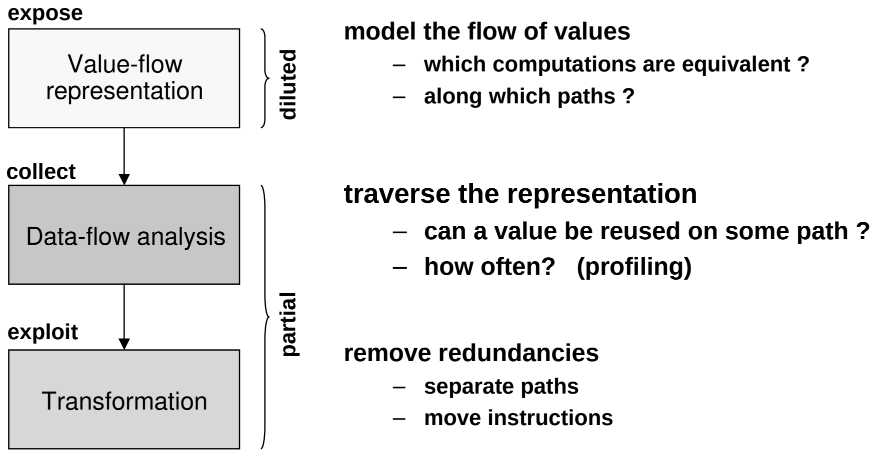


Figure 2.1: **The PathFinder stages and their function.**

The dataflow analysis stage traverses the representation and collects value reuse opportunities, by marking optimizable paths. The analysis answers the question ”is there a path along which this computation could reuse a value?” Furthermore, using run-time program profiling, it determines how frequently this reuse occurs at run time. Effectively, the analysis separates optimizable paths from the rest of program paths, which corresponds to the *partial* path sensitivity.

The transformation takes the redundancies collected by the analysis and removes them, using either physical path separation (as shown in Figure 1.5) or by moving the redundant instructions, which is a desirable, more economical alternative that does not duplicate program structure. Like the analysis stage, the transformation takes care of the partial path-sensitivity. However, while the analysis marks the optimizable paths and weighs them with a profile, the transformation decides how to transform the paths.

The most important consequence of staging is separation of the various optimizer’s goals and the various forms of exponential path cost:

Representation goals:

- avoid dilution of path-specific opportunities, but
- do not model separately all possible paths.

Analysis goals:

- collect all reuse exposed by the representation, but
- do not incur too much cost. The cost has two components:

  Analysis cost: because the representation models some paths separately, it may be larger than the original program.

  Profiling cost: for path-specific profile-guidance, we need the frequencies of optimizable paths; there may be too many paths requiring profiling.

Transformation goals:

- remove all redundancies collected by the analysis (completeness), but

  - ◦ do not cause too much code growth, due to physically duplicating paths with distinct optimization opportunities (code growth).

After the trade-offs between goals and costs are divided into stages, they become manageable. In fact, the overall contribution of this thesis is that we can achieve the various (potentially exponentially exploding) goals of the optimizer, while keeping the cost at a practical level.

2.2 Background and Related Work

One component of this thesis is defining the class of value-flow optimizations and identifying its members. Another component is characterizing optimization algorithms from this class by isolating orthogonal issues inherent in the various optimization algorithms. The goal of this section is to first describe common value-flow optimizations and their shared properties and then to characterize them by analyzing the approaches underlying these optimizations.

The class of value-flow optimizations is comprised of optimizations that analyze the flow of values in a program in order to remove operations that compute a value that is redundant because: a) it was previously computed, or b) there is a less expensive way to compute the value, or c) it will not be used in the remainder of the program.

Partial Redundancy Elimination (PRE) is a generalization of global common subexpressions and loop-invariant code motion optimization [MR79]. The aim of PRE is to delete expression computations that recompute a value produced previously by other expressions; that is, an expression is redundant if its value flows to it from a previous identical computation (which can be its own previous dynamic instance). Both *Scalar Replacement* [CCK90] and *Register Promotion* [LC97] remove memory operations that access memory cells previously loaded by another load operation. Also called load/store elimination, these two optimizations, rather than detecting the flow of expression values, are concerned with value flow of (identical) addresses for the memory operations. *Conditional Branch Elimination* eliminates conditional branches that are redundant because their outcome is known along some incoming paths [MW95a, BGS97a]. In this optimization, the value of interest is the value of the branch condition. Array bound check optimization is a special case of this optimization. *Constant Propagation* removes expressions producing values known at compile-time. Conceptually, constant expressions are redundant because there is a compile-time expression from which the value flows to the (redundant) constant expression executed at run time. *Strength Reduction* is another value-flow optimization [DI80, KRS93]. Rather than reusing the flowing value directly, it is used to find a cheaper way to compute the result of the expression. *Partial Dead Code Elimination* [KRS94c] eliminates statements that compute a value that will not be used in the remainder of the program.

While all these optimizations share the paradigm of exploiting value-flow redundancies, the various techniques developed to perform the optimizations differ in how they analyze value-flow patterns and how they modify the program to remove redundancies that exist only on some paths. To summarize existing approaches, three components of the value-flow optimizer have been identified: value-flow representation, program analysis, and program transformation. Each existing technique uses some form of value representation that prescribes algebraic rules for modeling the value flow. Then, the analysis traverses the value representation to connect redundant computations, i.e., to identify value-flow patterns. Finally, the program transformation component modifies the program to remove redundant computations.

This thesis divides the process of value-flow optimization into three components in order to separate issues that are independent. Such separation leads to a better understanding of underlying problems, cleaner algorithms, and more general parameterization of the framework. Although few existing techniques explicitly separate these three components, the following three subsections distill the contribution of prior work with respect to the representation, analysis, and transformation. Rather than elaborating on the value-flow *optimizations* described above, the following discussions analyzes the principles behind some important *algorithms* implementing the optimizations.

2.2.1 Value-Flow Representation

The simplest value representation uses *lexical expressions:* two computations are value-equivalent only if they have the same name *and* the name is not invalidated (killed) along the path. For example, the assignments `x:=a+b` and `y:=a+b` evaluate expressions with an identical lexical name `a+b`; if neither `a` nor `b` is redefined between the two assignments, the two assignments compute the same value under the lexical model. Note that the actual verification of the rule is left for the program analysis component (Section 2.2.2) which traverses all paths between the two assignments. The lexical value representation is used in many basic and advanced optimizations: subexpression elimination, loop-invariant code motion [ASU86], and partial redundancy elimination [MR79, Dha91, KRS92].

The lexical model is overly restrictive. It fails to uncover value equivalence when two different variables carry the same value. For example, following an assignment `x:=y`, expressions `2*x` and `2*y` are value-equivalent, although lexically different. *Value numbering* is a method for finding equivalences of lexical expressions in the presence of such copy assignments [CS69]. The value numbering model relaxes the lexical identity to the identity of syntax tree representations of expressions: `2*x` may be equal to `2*y` (but not to `2+x`). The validity rule is also relaxed: `2*x` is equivalent to `2*y` if `x` was assigned to `y` or vice versa, or if their assignments computed expressions that are (transitively) equivalent in the value numbering model.

While value numbering traces value flow across copy assignments, it fails when the assignment has a more complex right-hand side. For example, array expressions $A[i+1]$ and $A[i]$ are identical if i is updated between the two computations with the assignment $i := i + 1$. The value equivalence of $A[i + 1]$ and $A[i]$ can be determined using *symbolic manipulation* and simple algebraic rules. Various methods for symbolic value models have been proposed. Rau develops a conceptual framework [Rau91] in the spirit of abstract interpretation [CC77] in which repeated back-substitution of names along loop back-edges can detect loop-carried value equivalences on a path-per-path basis. The framework concentrates on presenting the problems that arise in naming and comparing symbolic expressions originating in different loop iterations; it does not develop practical solutions to these problems.

Reif and Lewis provide a formalism for using back-substituted symbolic expression names on the program CFG for determining constant values [RL77]. More recently, a symbolic back-substitution technique based on the Gated Single Static Assignment (GSA) representation [BMO90] was presented by Tu and Padua [TP95]. To name values in a path-sensitive fashion, they assign path predicates to symbolic expressions using GSA gating functions. This approach can be effective in answering queries on pairs of symbolic expressions, especially when the resulting symbolic expressions have simple gating functions. In order to use this representation for data-flow analysis, a powerful boolean symbolic evaluation system may be needed. Johnson and Schlansker describe how such a system can be constructed and utilized in solving predicated flow problems [JS96].

Briggs and Cooper propose a simple symbolic *reassociation* that improves value numbering by reordering nodes in the abstract syntax trees on which value numbers are computed [BC94]. For example, by transforming $e_1 = (a + c) + b$ into the canonical $(a + b) + c$, it can be found equivalent with another $e_2 = (a + b) + c$.

Finally, there are data-flow frameworks for array value-flow analysis, intended primarily for load/store elimination [BG96a, DGS93a]. By focusing their application domains to single loops with affine functions of loop induction variables, they model flow between recurrent array elements, such as $A[i]$ ad $A[i + 1]$.

This thesis presents a value-flow representation, called *Value Name Graph*, that combines the capabilities of syntax-based value numbering and symbolic evaluation methods.

2.2.2 Data-Flow Analysis

Data-flow analysis is a bridge between the value-flow representation and the program transformation. By traversing each path between potentially equivalent computations, the analysis verifies the algebraic rules posed by the representation, for example, whether a variable has been redefined. Being a tool for summarizing global program properties, data-flow analysis identifies value flow patterns (whether a value is computed on incoming or outgoing paths), which is then used to navigate the program transformation phase of value-flow optimizations. The following discussion shows that each existing technique that is not restricted to individual basic blocks uses global data-flow information, directly or indirectly.

Elimination of redundancies usually requires computation of a few data-flow problems to guide the transformation. Global common subexpression elimination computes availability of lexical expressions and removes the statement if the value it produces is available along all incoming paths [ASU86]. Partial redundancy elimination (PRE) based on code motion of redundant statements is formulated as a bidirectional data-flow problem [MR79]. Modern PRE formulations decompose the bidirectional problem into two unidirectional problems: availability and anticipability (also called very busy expressions) [KRS94a].

To determine which redundant statements can be removed, the approach in [RWZ88] uses the notion of dominators: if a computation is dominated by a value-equivalent computation, then it can be removed because its value has been computed on all incoming execution paths. Data-flow analysis is used here to calculate the dominator relation. Global value numbering [AWZ88] verifies the value model using the Static Single Assignment form (SSA) [CFR$^+$91]. Because SSA encompasses information on whether a variable is redefined between two program points, it can connect definitions and uses of the same variable. The SSA is computed from the dominator relation and hence contains the data-flow component, too.

Program analysis can navigate the transformation process better if, besides proving equality of computations, it also gives an estimate of the benefit gained by the optimization. Such benefit can be derived from a program profile, traditionally represented in the form of execution frequencies of CFG edges. Ramalingam developed a data-flow analysis framework that computes the probability of a fact occurring, rather than only boolean existence information [Ram96]. Unfortunately, the framework does not explain how to apply frequency-based problems to maximize optimization profitability. This thesis provides such a methodology.

Because advanced program analysis is costly in time, prior research has developed methods for reducing its cost. Hank *et al.* propose a region-based compiler architecture in which procedures are partitioned into disjoint segments that are analyzed and optimized separately [HHR95]. By selecting appropriate region sizes, the usually quadratic complexity of optimization algorithms will be prevented from exploding beyond practicality. Duesterwald *et al.* developed an orthogonal approach which, instead of limiting the analysis scope, examines *on demand* only those program statements that may affect the result [DGS97]. This thesis uses demand-driven approach for performing inter-procedural value-flow analysis.

2.2.3 Program Transformation

In the long history of research and implementation of value-flow optimizations, four distinct transformations for removing value-flow redundancy have been be identified: code deletion, code motion, control flow restructuring, and control speculation. In an interprocedural domain, inlining and cloning have also been applied to enable exploitation of opportunities that exist on interprocedural paths.

Code *deletion* is the simplest form of removing a redundant statement: if the value of the statement is previously computed along each incoming path, then the statement can simply be removed. To verify that the redundancy exists along all paths, the optimization can be restricted to basic blocks, as in subexpression elimination [ASU86] or value numbering [CS69]. For applying deletion globally (across multiple basic blocks), data-flow analysis is applied to confirm that value is available along all paths, as in global subexpression elimination [ASU86]. Alternatively, dominators can be used, as in global value numbering [RWZ88].

Deletion is impossible when a statement is redundant along a strict subset of all incoming paths. Code motion is a technique that hoists the partially redundant statement so that it is removed from paths on which it is redundant. Effectively, hoisting introduces compensation code on non-redundant paths, changing partial redundancy into full redundancy, which enables deleting the statement from its original position. Loop-invariant code motion is the simplest form of such motion transformation. Morel and Renviose generalized it to arbitrary control flow graphs by formulating the code motion problem as a bi-directional data-flow analysis problem [MR79]. Knoop *et al.* found a uni-directional formulation for this problem [KRS94a].

The necessary code motion may be blocked when it would change program semantics or impair the program for certain inputs. When code motion fails to eliminate all partial redundancies, control flow restructuring can be applied. In value-flow optimization, restructuring is based on separating the optimizable paths from the unoptimizable paths, which is acomplished by duplicating all statements along the path that needs to be separated. A simple form of restructuring is tail duplication [HMC$^+$92] which separates frequently executed paths to improve scheduling by separating control flow merge points. Restructuring is also necessary when redundant operations are unhoistable, such as unconditional branches [MW92a] or conditional branches [MW95a, BGS97a].

Gupta *et al.* apply control speculation, which is a transformation that inserts computations onto paths that did not compute these computations in the unoptimized program. As a result, some paths are optimized and some are impaired. To control the impairment, a run-time program profile is used [GBF98]. A speculative PRE was also explored in [HH97, CKL$^+$98].

Other kinds of value-flow optimizations are also built on one or more of these four transformations. Elimination of partially dead value presented in [KRS94c] maximizes optimization that is possible with code motion alone. Dead values that cannot be removed with code motion must be eliminated through restructuring, as shown in [BG97a]. Strength reduction (an extension of PRE) has so far been presented as code-motion optimization [DI80, KRS93].

Clearly, the four transformations differ in their power and cost [BGS98a]. Deletion is only applicable on fully redundant operations, hence not suitable for path-sensitive optimizations. While code motion is economical in that it does not increase code size, it is less powerful than restructuring, which can eliminate all redundancies but may incur significant code explosion. Control speculation does not remove all redundancies and it impairs some paths, but it introduces no duplication. It is the goal of the proposed research to integrate the transformations so that the more economical transformations are used whenever possible, and the costly one is resorted to only when its result would be expected to result in a significant run-time speed-up.

2.2.4 Path-Sensitive Optimization Frameworks

A few path-sensitive value-flow optimizations developed to date are either highly specific or amorphous enough to escape the three-stage classification (representation–analysis–transformation). Consequently, we present them as frameworks.

Holley and Rosen [HR81] were first to recognize the benefit of maintaining program assertions that are specific to individual execution paths during the analysis. In their *qualified* data-flow framework, control flow paths on which a variable has different constants are separated through (virtual) code duplication. Path duplication results in exposing path-specific contexts, which are then used not only to remove conditional branches whose outcomes are known in a given context, but primarily to improve def-use computation. Because removing redundant branches also removes some paths that would never be executed, def-use pairs realizable exclusively along infeasible paths are not spuriously collected on the expanded control flow graph. Originally developed to improve analysis on programs with IBM register-indirect jumps, the analysis does not scale to general branch elimination where many variables must be analyzed. A practical solution to multi-variable analysis is offered in [BGS97c] by means of demand branch correlation analysis, which reduces the cost of finding redundant value-flow patterns.

Rather than associating path-specific data-flow facts with execution paths in the graph-theoretic sense, Johnson and Schlansker represent the execution paths as boolean expressions derived from predicates of conditional branches on those paths [JS96]. Intended for optimization of programs with predicated execution [MLC$^+$92], their analysis is primarily applicable for removing spurious dependencies that prohibit operation reordering or register allocation [ED95]. However, relaxing or strengthening the predicate condition guarding the execution of an operation corresponds to moving the operation in the control flow graph, and hence can achieve optimization.

Steffen [Ste96] presents an extension of the Holley-Rosen approach. In his *property-oriented expansion* framework, path separation is driven not only based on constants of variables but also on particular assignments being or not being previously computed on a path. This allows elimination of branches (as in [HR81]) as well as of entire assignments. Steffen also observes that some code duplication may be unnecessary and re-merges back unnecessarily split paths using finite state automaton minimization.

Ammons and Larus [AL98] extend Holley's and Rosen's qualified analysis by separating the paths not on variable assertions but instead on frequently executed paths which are separated from each other to maintain the *hot* path-specific context. Profile-directed graph expansion is a practical alternative to Holley-Rosen or Steffen property-oriented expansion approaches: while some path contexts are sacrificed, those along dynamically important paths are preserved. The framework is presented as a constant propagation optimization. After the analysis, the algorithm recombines separated paths that present no useful opportunities.

In a global view, preliminary research conducted as part of this research has identified four main issues with *path-sensitive* value-flow optimizations: a) solving *non-distributive* problems without conservative approximation (e.g., non-linear constant propagation), b) collecting *path-specific opportunities* (e.g., variable has a different constant along each path), c) *exploiting* specific opportunities (e.g., enabling folding of path-dependent constants through restructuring), and d) directing the analysis effort towards hot paths.

The approach of Ammons and Larus attacks all four issues uniformly by separation of hot paths, their subsequent individual analysis, and recombination. Instead, the approach proposed by ths thesis is to reserve restructuring for the actual transformation. This implies a different overall strategy: a) non-distributive problems are solved precisely along *all* paths by customizing the data-flow *name space* [BA98], b) distinct path-specific opportunities are collected through demand-driven analysis as in branch elimination [BGS97a],

c) all profitable opportunities are exploited with economical transformations (code motion and speculation), and d) infrequent program regions will be avoided using the profile-navigated demand-driven analysis.

Gupta *et al.* [GBF97b, GBF98, GBF97c] also use path profiles to carry out path-sensitive optimization. Similar to the method of Ammons and Larus, path profiles are used to maintain path-specific context along each hot path. The results of the analysis are then used to integrate code-motion with control speculation. In contrast to the method of Ammons and Larus, the transformation is delayed until after the results of the analysis are known.

2.3 Contributions and the structure of the thesis

In each of the stages of the PATHFINDER framework, this thesis develops new techniques. Next, we summarize these contributions, on a per chapter basis.

Chapter 3 *Representation.* This chapter develops the *Value Name Graph (VNG)*, a novel program representation that models the flow of a recomputed value. By symbolically naming the value, the VNG reduces the (hidden) value flow into the (exposed) data flow. As a result, recomputation is detected essentially as reaching definitions. The representation, called *Value Name Graph*, names the value on demand, for a set of optimized computations. Although the names are formed separately for each path, obtaining path-sensitivity, the paths are analyzed separately only when it matters, i.e., when the value has a different name.

Chapter 4 *Dataflow analysis.* This chapter develops a dataflow analysis that marks optimizable paths. This marking is used to guide the subsequent transformation. Rather than enumerating all optimizable paths, the analysis encodes them with polynomial complexity in a way from which the transformation can recover them. The solution is based on a lattice that informs whether, at a given point, the value flow exists along all, some, or no paths. This chapter also shows how to perform value-flow analysis inter-procedurally.

Chapter 5 *Dataflow analysis (with profiling).* To navigate the transformation trade-offs, the analysis weighs the marked paths with a run-time profile. Unfortunately, precise weighing requires frequencies of all paths. To make profile-guiding practical, this thesis develops a family of *estimator* algorithms based on *edge* frequencies, a cheaper but inherently imprecise alternative. When weighing the reuse, estimators bound the inherent error, adding confidence to imprecise profiles.

Chapter 6 *Transformation (intra-procedural).* This chapter develops a transformation that (nearly) completely removes all program redundant recomputations, at almost zero code growth. The transformation combines three orthogonal methods. We resort to the expensive *path duplication* only when the growth-free *code motion* fails to transform the program, and only when the profile-driven *control speculation* cannot profitably impair some paths to optimize others. The spectrum of algorithms is based on a single abstraction, a *Code-Motion-Preventing (CMP) Region*, which contains identifies adverse effects of the control flow on the desired optimization.

Chapter 7 *Transformation (inter-procedural).* This chapter develops a transformation that completely removes redundancies that are interprocedural, i.e., those where the value flows across procedure boundaries. Rather than resorting to (expensive) inlining, we separate optimizable paths by generating multiple procedure entry points and multiple exit points (which may return to different points in the caller). Thanks to entry/exit splitting, paths can be separated across procedure boundaries, even when the call site invokes one of many procedures, as in virtual procedure calls.

Chapter 8 *Empirical evaluation.* This chapter compares the power of the framework with that of an ideal value-flow optimizer, using the optimization of redundant loads. We developed a run-time program monitoring algorithm that exposed the amount of value reuse present in the program. This ideal amount was compared with the amount detected by our analysis. We observed that we captured at least 80% of the reuse present in the program.

Overall, the techniques presented in this thesis improve not only the effectiveness but also the efficiency of the optimizer.
The thesis is emphasizes the following principles:

- it develops a *generic framework* from which new optimizations can be derived,
- the power of the representation and transformations algorithms each results from combining techniques with *orthogonal strengths*,
- the trade-offs of the program transformation are *profile-directed*, and
- to evaluate the power of the framework, we performed a limit study which served as an ideal reference point.

2.4 Preliminaries

This chapter presents the necessary defintions. The reader is assumed to have some background in dataflow analysis; the rest of the thesis is developed from first principles.

We use a control flow graph as our representatrion because it is a common program intermediate representation. Because it directly exposes the control flow paths, it enables an intutive formualtion of code motion.

Definition 2.1 (Directed graph) $G = (N, E), E \subseteq N \times N$.

Predecessor and successors are defined with the maps *pred* and *succ*: $\mathit{pred}(n) = \{m | (m, n) \in E\}, \mathit{succ}(n) = \{m | (n, m) \in E\}$.

A *path* p from n to m on a graph G is a sequence of k nodes $n_{i_1}, \ldots, n_{i_k}$, $n = n_{i_1}$, $m = n_{i_k}$. When the path p includes the nodes n and m, we write $p = [n, m]$. When the path excludes the nodes n and m, we write $p = \langle n, m \rangle$. □

Definition 2.2 (Control flow graph) A control flow graph $G = (N, E, \mathit{start}, \mathit{end})$ is a directed graph with two distinguished nodes *start* and *end*, which have no predecessors, and no successors, respectively. We assume that each node $n \in N$ lies on a path $p = [\mathit{start}, \mathit{end}]$. □

Chapter 3

Value-Flow Program Representation

The value-flow representation is a value-centric model of the program; it exposes value-equivalent instructions. Specifically, it finds control flow paths along which a *value flows* between instructions computing the value.

Typically, when analyzing programs for value recomputation, each value is identified with its lexical name. When two expressions match the name, they compute the same value. But what name should be used when the value flows between equivalent expressions that have different names? The program representation presented in this chapter overcomes the naming problem by synthesizing names that fully trace the flow of a value and by performing data-flow analysis on this synthesized name space.

The PATHFINDER analysis integrates three orthogonal techniques: symbolic interpretation, value numbering, and data-flow analysis. Symbolic interpretation first creates all necessary names and value numbering then determines which names are synonymous. The result is expressed in a new program representation, called the *Value Name Graph* (VNG). Once the VNG is constructed, any conventional data-flow analysis can answer two fundamental optimization questions: which computations are value-equivalent, and along which control flow paths?

The VNG is path-sensitive: value reuse is detected even when each path requires different names, for example due to conditionally incremented loop induction variables. The VNG can be parameterized for redundancy optimizations such as common subexpression elimination, constant propagation, or load/store elimination. By phrasing these optimizations on the VNG, we obtain greater optimization power and broader applicability.

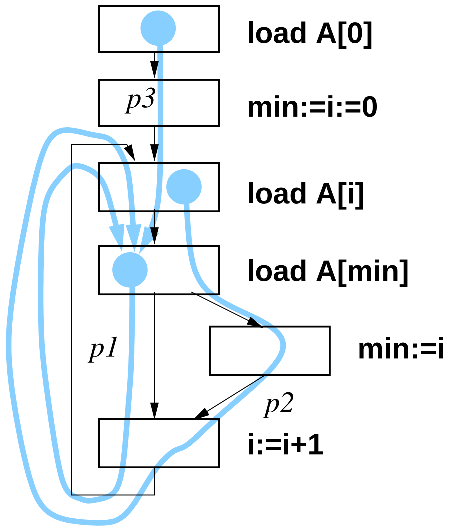
This chapter begins by presenting two goals of the representation in Section 3.1. Next, Section 3.2 contrasts the flow of data with the flow of values, and defines value flow formally. Section 3.3 briefly describes the most common approaches for detecting value flow, focusing on their strengths and weaknesses. Section 3.4 presents the construction of the VNG, and explains why the VNG is a synergy of three orthogonal existing approaches. Finally, Section 3.7 compares the VNG with other value-flow representations and Section 3.8 experimentally evaluates its scalability, and thus also its practicality.

3.1 The goals

Value-flow representation exposes the recomputation of a value in the program, to facilitate value-flow analyses and optimizations. In the PATHFINDER path-sensitive value-flow optimizer, the representation has two distinct goals:

1. Which instructions compute the same value? (a *value-flow* goal)

```
val := A[0]
min := i := 0
do
    if (A[i] < A[min])
        min := i
    i := i+1
while (i<N)
```



a) the source program.

b) the value flow of $A[min]$.

Figure 3.1: **The FIRSTMIN program.**

2. Along which paths do they compute the same value? (a *path-sensitive* goal)

Both goals are important to perform the optimization, as illustrated by the program in Figure 3.1(a). The program, FIRSTMIN, traverses the array A and computes into variable *min* the index of the smallest element in the array. A close examination of the program reveals that the load of $A[min]$ is redundant, due to the fact that the value of $A[min]$ has been loaded from the memory by some other instruction along each path leading to the redundant load. The redundancy can be shown on a per-path basis as follows. Along path p_1 (taken when $A[i] \not< A[min]$), the expression $A[min]$ in the current and the following iterations refers to the same memory location; hence, the later load is redundant. Along path p_2 (taken when $A[i] < A[min]$), the variable *min* is redefined and $A[min]$ no longer refers to the same array element. However, along this path, $A[i]$ in the current iteration equals $A[min]$ in the following iteration. Hence, path p_2 offers a reuse between $A[i]$ and $A[min]$. Finally, along path p_3 (taken when the loop is entered), the value of $A[min]$ equals $A[0]$ loaded before the loop. In conclusion, because paths p_1, p_2, p_3 "cover" all possible paths leading to the load of $A[min]$, whichever path is taken to $A[min]$, its value has already been loaded previously and can be reused.

This detected redundancy enables the removal of the load of $A[min]$, which in turn enables scheduling the loop better, at twice the iteration issue rate.[1] The 100% speed-up is due to reducing the critical path of data and control dependences in the loop, as explained in Section 1.2. Figure 3.2 shows the schedule for the critical path of the FIRSTMIN loop, before and after the removal of load of $A[min]$.

The moral of the FIRSTMIN example is that a traditional, path-*in*sensitive analysis fails to discover the redundancy that leads to the optimization. The reason for its failure is that when paths p_1 and p_2 are considered together, it is impossible to detect that each path has a different source of value reuse (the source on p_1 is $A[min]$, and the source on p_2 is $A[i]$). In other words, while both paths are optimizable, they are optimizable in a different way. Without path sensitivity, these optimization opportunities are diluted, as at most one of the two sources can be discovered by the path-insensitive analysis.

In conclusion, both goals of the representation are necessary for a powerful optimization. It is not enough to detect value flow. It must be detected a path-sensitive way.

---

[1] Assuming the *if* statement is compiled into a conditional move instruction [Sit93, Dul98].

the source program:

```
val := A[0]
min := 0
i := 0
do
    if (A[i] < A[min])
        min := i
    i := i+1
while (i<N)
```

```
r := val := A[0]
min := 0
i := 0
do
     if (A[i] < r)
         min := i
         r := A[i]
     i := i+1
while (i<N)
```

the schedule (per cycle):

```
1  adr := A + min
2  r := load adr
3  if r < A[i]
4     min := i
```

before optimization

```
1  if r < A[i]
2     min := i; r := A[i]
```

after optimization

Figure 3.2: **The benefits of optimizing FIRSTMIN.** The instruction schedule of the loop, before and after the removal of load $A[min]$. The schedule shows only the instructions on the critical path of data and control dependences.

3.2 What is value flow?

This section explains why detecting value flow is hard. It distills some properties of value flow, show why it is not explicitly exposed by the program text, and contrast value flow with data flow (which is explicitly exposed by the program). Overall, this section attempts to outline the properties that a value-flow representation needs in order to reach the two goals stated in the previous section. These properties are relied upon in the following section, to explain the insufficiency of existing value-flow representations. It concludes with a list of practical analysis problems that can be phrased as value-flow problems and thus solved in the PATHFINDER framework.

3.2.1 Value flow versus data flow

Procedure $f(a)$ in Figure 3.3(a) exhibits two instances of value reuse. Regardless of the procedure parameter a, two pairs of statements always compute the same value: S_2, S_5 and S_3, S_4. Their value-equivalence becomes visible when the value they compute is expressed in terms of the procedure parameter a, as shown in Figure 3.3(b).

Although contrived, the example illustrates why analyzing *value flow* is harder than analyzing *data flow*. The distinction between the two is typically not acknowledged, perhaps because a simple approximation of value flow can be computed as the well-known dataflow problem of *available expressions*, as described later in Section 3.3.1. However, we argue that the two flows are qualitatively different.

Dataflow analysis problems typically examine properties of *variables*. For example, the problem of reaching definitions is to compute, for each variable v and each program node n, the set of definitions of v that may reach n. Similarly, the liveness problem is to compute whether the data in the variable v will be used in some instruction on some path from n to *end*. Dataflow analysis is well-suited for such variable-centric problems because the definitions and uses of variables are obvious from the program text. In our example, the fact that x is live on the exit of the statement S_2 is evident from the fact that x is defined in S_2 and used in S_3.

On the other hand, value flow problems are concerned with properties of *values*; we want to know which instructions compute the same value. Compared to variables, values are "invisible" entities: while each

```
      f (a) {
$S_1$:   b := -2*a
$S_2$:   x := b - a
$S_3$:   a := 2*b - x
$S_4$:   x := b - a
$S_5$:   z := a + x
      }
```

a)

$$S_2 = S_5 = -3a$$

$$S_3 = S_4 = -a$$

b)

Figure 3.3: **Data flow versus value flow.** (a) The procedure with two instance of value reuse, between S_2 and S_5, and between S_3 and S_4. (b) Values computed by the two pairs of equivalent statements, expressed as a function of the procedure parameter a.

variable has a unique lexical name, a value may a) require multiple names and b) it may not even be stored in any variable as it *flows* between the value-equivalent statements. Consider the value flow between S_2 and S_5 in Figure 3.3: a) the value has a different lexical name at the two statements (x versus z); and b) between S_4 and S_5, no variable stores the value—the value will be "resurrected" only when S_5 executes.

In summary, while the flow of data is lexically exposed, by the definitions and uses of variables, the program text does not directly identify value-equivalent statements. This qualitative difference is illustrated in Figure 3.4, where data-flow related statements are connected with def-use edges (left), and value-equivalent statements are connected with "value-reuse" edges (right). The value-reuse edges are not obvious from the program text. Instead, they require some form of algebraic manipulation.

To turn the above discussion into desirable properties of the path-sensitive value-flow representation, we can state the following:

- The representation must add edges to the program that will connect the equivalent statements, reflecting their algebraic equivalence. The edges need not be added physically; they may be represented by labeling the equivalent computations.
- The representation must allow placing the edges along path, as shown in Figure 3.1(b), where the paths p_1 and p_2 are the path-specific edges.

The edges can be viewed as def-use edges. With such a view, the flow of values will appear as flow of data. Thus, the representation reduces value flow problems to data flow problems. This is precisely the approach taken by the Value Name Graph.

3.2.2 A formal definition of value flow

Our framework optimizes any problem from a *value-flow class*. We define an analysis problem to fall into the value-flow class if the analyzed entities (values) can be related using *path back-substitution functions*, as follows.

Definition 3.1 (Symbolic Back-Substitution) Let $p = [n, m]$ be a CFG path. Let e_n and e_m be two names of a value computed at node m. The name e_n is a name valid at node n, and the name e_m is valid at node m. The names e_n and e_m are *some* representations of the computed value, perhaps symbolic expressions or abstract syntax trees inductively built from the names of program variables. Function $F(p)$ is a *back-substitution function* is it maps a name e_m and the path p into the name e_n. We say that F *back-substitutes* e_m along path p, yielding e_n.

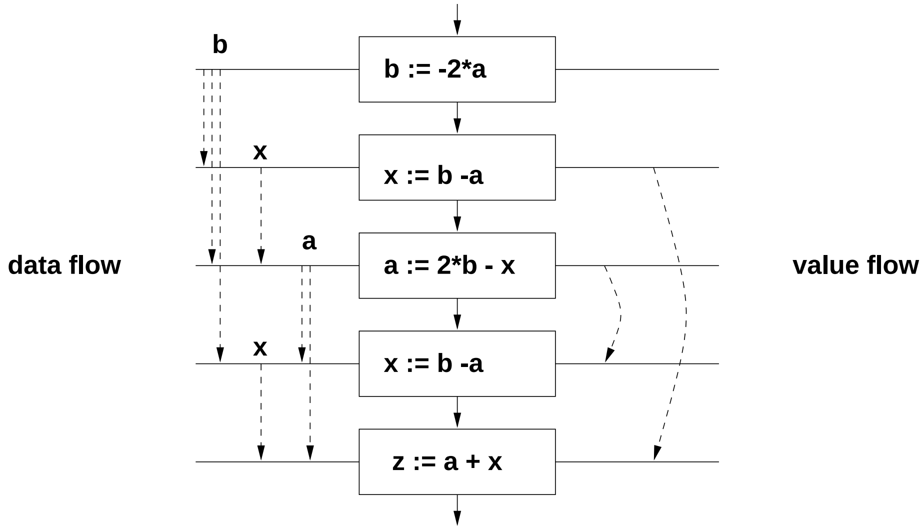


Figure 3.4: **Compared to data flow edges, value flow edges are "intangible."** While def-use edges can be identified *lexically*, i.e., from the text of the program, the value-reuse edges require some form of *symbolic* manipulation of the program, to expose the algebraic equivalences.

Definition 3.2 (Value flow) Let $p = [k, n, m]$ be a CFG path with subpath $p_1 = [k, n]$ and $p_2 = \langle n, m]$, such that $p = p_1.p_2$. We say that a value flows from n to m if $F(p_1.p_2)(e_m) = F(p_1)(e_n)$.

Example 3.1 In Figure 3.3, the value flows from S_3 to S_4 because $F([S_1, S_3].\langle S_3, S_4])(x) = F([S_1, S_3])(a) = -a$. The function $F(p)(x)$ performs the back-substitution of x along path p.

3.2.3 What everything is value flow?

Let us list next a few common problems that fall into the value flow class and can thus be solved in a value-flow framework.

- value recomputation (expression, loads),
- branch correlation (and hence array bound checking),
- constant propagation,
- some type inference problems.

3.3 Existing techniques for value-flow detection

This section reviews three existing techniques for detecting value flow: dataflow analysis, value numbering, and symbolic evaluation. These three techniques are orthogonal because each can detect a different kind of value-equivalent statements. The (incomparable) strengths of these techniques are described here primarily because our value-flow representation integrates the three techniques in a way that preserves their advantages and removes their limitations, as will be described in Section 3.4.

3.3.1 Dataflow analysis

Dataflow analysis, while data-flow in nature, is the most commonly used technique for detecting (an approximation of) value-flow [KU77]. The well-known dataflow problem of *available expressions* computes an approximation of the value-flow problem of availability of values. In available expressions, each value is identified with its lexical name, typically the right-hand-side of an assignment. When two lexical names match, the addresses compute identical values (unless there is kill of the name along the path). Because the analysis uses only one name per value, it cannot detect computations that are equivalent but have different names. For example, the equivalence of $A[i]$ and $A[\mathit{min}]$ along path p_2 in Figure 3.1 will not be detected by available expressions. On the other hand, when dataflow analysis finds equivalence, it does so in a path-sensitive fashion. For instance, in the same example, the equivalence of $A[\mathit{min}]$ along p_1 is detected and, as a result, the path p_1 can be "marked" by the dataflow analysis as a path along which the expression $A[\mathit{min}]$ is available. In summary, dataflow analysis is path-sensitive but not value-sensitive.

3.3.2 Value numbering

Value numbering partially overcomes the naming problem of dataflow analysis [Cocke,Schwartz]. For each value, it builds an abstract syntax tree, which serves as its "name." Two names are equivalent if their syntax trees match: they have identical shape, including operators at inner nodes, and identical variables at leaves of the tree. Value numbering is local (works in basic blocks) and therefore not path-sensitive. The Global Value Numbering extends value numbering along all paths, but it still requires that the equivalence holds along all incoming paths [AWZ88]. In summary, value numbering is suited for discovering which names refer to the same value (are synonymous), but is not path-sensitive (i.e., the equivalent computations must be synonymous along all paths connecting them). Furthermore, value numbering is not able to perform (arbitrary) algebraic manipulations of value names.

3.3.3 Symbolic evaluation

There are various ways how to symbolically execute a computation. Here we are alluding to methods that perform algebraic simplifications of expressions. Constant propagation algorithms are such methods. They may fold constant into expressions and evaluate them, essentially simplifying the expressions. Another method is to perform a symbolic backward substitution of program expressions. Compared to constant propagation, the algebraic simplification is symbolic, rather than arithmetic. Such symbolic simplification does not easily extend to handling path (on which a different simplification may be needed). However, Tu and Padua were able to make symbolic manipulation global (i.e., working on all paths) using a version of SSA form [TP95]. In summary, symbolic manipulation of programs is suited for creating the names for a value of an expression, and for performing simplification on the symbolic names. Such manipulations, however, are not path-sensitive.

3.3.4 Summary: a need for integration

The three techniques are orthogonal—each can be used to detect a certain class of value redundancies [ARZ92]. Their strengths, summarized in Figure 3.5, are integrated in our representation, as follows. We use symbolic evaluation to name the value as it flows through the program. Such names will sufficiently describe the value, even when it is not resident in any variable. Value numbering will compact the symbolic

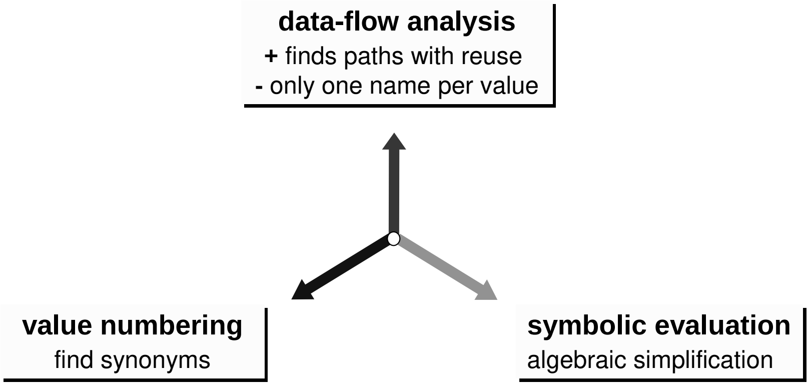


Figure 3.5: **Three orthogonal value-flow detection techniques.**

names. Together, the two will build a representation that will reduce value flow into the data flow. Finally, dataflow analysis will be used to mark paths along which reuse exists, obtaining path sensitivity.

3.4 Value Name Graph

The preceding sections have set the stage for the presentation of the program representation. Section 3.1 distilled two goals of the program representation that are needed for an effective optimization, Section 3.2 explained why value flow is not exposed by the traditional program representation, and Section 3.3 showed that, while existing approaches do not offer a sufficient solution, they offer complementary qualities whose integration could form the basis of the desired representation. The *Value Name Graph (VNG)* is exactly such an integration. This section first intuitively explains the VNG and then presents the three steps of its construction.

Our representation is called a Value Name Graph because it exposes the value flow by properly naming the (recomputed) value that flows between equivalent computations. The central idea is to create sufficient names so that a value can be identified even when it flows outside the scope of the lexical name under which it is originally computed. Where the original name is not valid, an equivalent symbolic name is used. The symbolic names form *value threads*, which conceptually represent the value-flow arrows in Figure 3.1(b). The VNG is the collection of these value threads. Propagating dataflow facts along these threads achieves

- *symbolic analysis*, because the threads represent symbolic value names, and
- a *path-sensitive value-flow analysis*, because a thread is formed for each individual program path.

The synthesized names are created using symbolic substitutions, using a backwards propagation of the lexical name of the value, as follows. Clearly, when a value is computed by an instruction, it can be identified with its “lexical” name. Consider the value computed in the statement S_5 below; the value is identified with the name $b + x$. We want to propagate the name backwards and modify it whenever it becomes invalid. While this (lexical) name is valid at the entry of S_5, it is not valid at the entry of S_4, because the value of x before S_4 is different than before S_5—we say that S_4 invalidates the name $b + x$.

S_1: `b := -2*a` $\leftarrow \texttt{name}_1(S_5) = -3a$
S_2: `x := b - a` $\leftarrow \texttt{name}_2(S_5) = b - a$
S_3: `a := 2*b - x` $\leftarrow \texttt{name}_3(S_5) = x$
S_4: `x := b - a` $\leftarrow \texttt{name}_4(S_5) = 2b - a$
S_5: `z := b + x` $\leftarrow \texttt{name}_5(S_5) = b + x$

Whenever the propagated value name is invalidated by an assignment to a variable in the name, another (symbolic) name for the value is synthesized using back-substitution. Across S_4, the name $b + x$ changes to $2b - a$, because x is substituted with $b - a$. As long as the symbolic name can be expressed in a selected language of symbolic names, this process identifies the value along its entire flow. The sequence of created names forms a value thread. The central property of the VNG is that the computations that lie on the same thread compute the same value. In the example, S_2 and S_5 lie on the same thread.

While the value threads identify identical statements, they contain too many symbolic names to be useful as a practical representation. The final Value Name Graph is compact; it encodes the value threads using scalar variables. Each computation is rewritten to refer to a scalar variable C_i. When two computations refer to the same C_i, they are on the same value thread. The VNG for the above example is given below; equivalent computations refer to the same scalar variable (see Figure 3.3). Technically, the VNG below is not a graph. Instead, it is a program *form* that encodes the graph. We will use the graph and the program form interchangeably.

S_1: C_1
S_2: C_2
S_3: C_3
S_4: C_3
S_5: C_2

Effectively, the scalar variables C_i are "compacted" symbolic names. Sometimes, a value thread requires multiple names, even after compaction. The multiple scalars are connected using ϕ-assignments, which switch between the compacted names, forming value (compacted) threads. The ϕ-assignments impose the SSA property of the scalar names [CFR$^+$89]. Figure 3.6 shows the VNG for the FIRSTNAME program. Each access to array A is renamed to refer to a scalar variables C_i, and the equivalent computations are connected across ϕ-assignments. For example, the value flow between $A[i]$ and $A[min]$ along p_2 in Figure 3.1(b) is manifested as a def-use chain: C_1, $C_3 := \phi(C_2, C_1)$, $C_2 := \phi(C_0, C_3)$. The def-use chain identifies the CFG path along which the computations are equivalent. In particular, the phi-assignments specify that the chain follows only the path p_2. Note that the VNG not only connects equivalent instructions but also does it along the appropriate paths.

Thus, the general rule is that two computations produce the same value when they are connected with a def-use chain across the ϕ-assignments. The flow of values appears as flow of data, that is, as dependences between definitions and uses of the scalar variable C_i and can be encoded in dataflow transfer functions.

In effect, the VNG converts the problem of *availability of values* into the problem of reaching definitions (transitive across ϕ-assignments). In summary, the design of the Value Name Graph was influenced by the two goals from Section 3.1. The result are the following salient properties:

1. *The value-flow goal.* The VNG elevates the (invisible) value flow to the (visible) data flow level. By transforming value flow to data flow, the VNG exposes value recomputations in the form of references

```
$S_1$:val := A[0]
$S_2$:min := 0
$S_3$:i := 0
$S_4$:do
$S_5$:    if (A[i] < A[min])
$S_6$:        min := i
$S_7$:    i := i+1
$S_8$:while (i<N)
```

the original program

$$\begin{array}{rcl} A[0] & \rightarrow & C_0 \\ A[i] & \rightarrow & C_1 \\ A[min] & \rightarrow & C_2 \end{array}$$

the transformation

```
..   := $C_0$
$C_2 := \phi(C_0, C_3)$
do
    if ($C_1 < C_2$)
        ...
    $C_3 := \phi(C_2, C_1)$
while ( )
```

its Value Name Graph

Figure 3.6: The Value Name Graph for FIRSTMIN program (the SSA form is not a graph!).

to (the same) scalar variables, allowing us to answer the question "which instructions compute the same value?"

2. *The path-sensitive goal.* The VNG supports dataflow analysis. Being a path-sensitive technique, dataflow analysis can mark the paths along which a value is recomputed.

Additionally, because the VNG is a sparse representation similar to the SSA form, it can be implanted into existing SSA-based PRE implementations, improving their precision [LCK$^+$98].

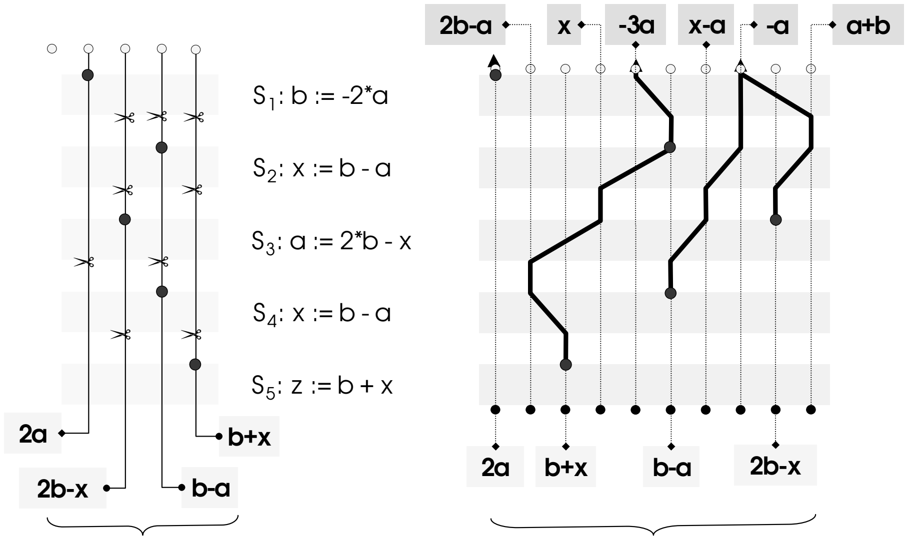
3.4.1 Constructing the VNG

Let us start with an overview of the VNG construction. The construction has three steps, each corresponding to one of the three underlying approaches. First, the *symbolic evaluation* places the value thread, by synthesizing the names that are necessary to trace the value flow. Second, *value numbering* compacts the value threads, by determining which symbolic names are synonyms for the same value. The result of the first two steps is the VNG. The third step forms the dataflow transfer functions using the scalar names C_i.

Step 1: placing the value threads. The symbolic names are created by propagating the lexical name of the analyzed computation backwards. At each assignment that invalidates the current name, the assignment's right-hand-side expression is substituted into the current name and algebraically simplified. The propagation effectively creates a "symbolic" slice of the original computation.

Figure 3.7 above example shows only the value thread for the value computed by S_5. Figure 3.7 completes the example. It shows, in a graph form, the value threads for all computations in the program. The graph shown is an intermediate form the VNG that is never explicitly constructed. At this intermediate stage, VNG nodes are a cross product of CFG nodes with the synthesized symbolic names. The VNG edges show how the value name changes, forming the value threads. The highlighted nodes represent the analyzed computations.

Step 2: collapsing the value threads. The first step formed the threads, which exposed the recomputation of S_2 by S_5. This recomputation was detected because S_5 and S_2 lie on the same value thread. Note, however, that while S_3 and S_4 are equivalent, they are not on the same value thread; there is no chain of data flow dependences between them and hence dataflow analysis fails to find them equivalent. The lack of a connecting value thread is explained by the fact that the lexical name of S_3 ($2b - x$) did not match with name of S_4 propagated to S_3 by back-substitution ($x - b$). These two names are however *synonyms* for the same value, as can be seen when they are expressed as a function of the procedure formal parameter a; both



traditional: lexical name space **VNG: symbolic name space**

Figure 3.7: **The VNG after Step 1 (right).** The thick lines are *value threads* that connect equivalent computations. In contrast, traditional dataflow analysis (left) builds threads using only the lexical name. This names is killed (the scissors), preventing threads from connecting equivalent computations.

are equal to $-a$. The problem is that the back-substitution process matched the names of S_3 and S_4 only at a predecessor of S_3.

This delayed match is inherent in the symbolic back-substitution process. It is corrected by a value numbering step, which collapses threads whose names are synonymous. Two names are synonymous if the back-substitution reduces them to the same symbolic name. The collapsed threads are shown in Figure 3.8. After the threads are collapsed, the equivalent computations S_3 and S_4 lie on the same thread and dataflow analysis can find the recomputation. Note the change in the value names. While the first step used the synthesized symbolic names, the second step names the value with the scalar names C_i. The scalar names directly correspond to the congruence classes formed during value numbering.

Step 3: dataflow analysis. Once the graph is constructed, recomputations of the same value are placed on the same (collapsed) threads. Computations are converted to accesses to scalar names and the dataflow analysis determines availability of a value as a GEN/KILL analysis. The analysis is a bitvector analysis, except for ϕ-assignments, where the dataflow facts are propagated between bits corresponding to the ϕ-ed names.

Figure 3.9 summarizes the construction of the VNG using the three orthogonal techniques. The following subsections describe the construction of the VNG in detail.

3.4.1.1 Initial parameters

The VNG is parameterizable. It can be constructed to expose value flow among arithmetic expressions, memory load operations, conditional branches, etc. The VNG can be tailored for each kind of recomputation by specifying the symbolic language that generates the names of the values during back-substitution. While the arithmetic recomputations in the preceding examples required a name that was an expression composed of variable names and operators $+, -, \times, /$, the analysis of conditional branches may

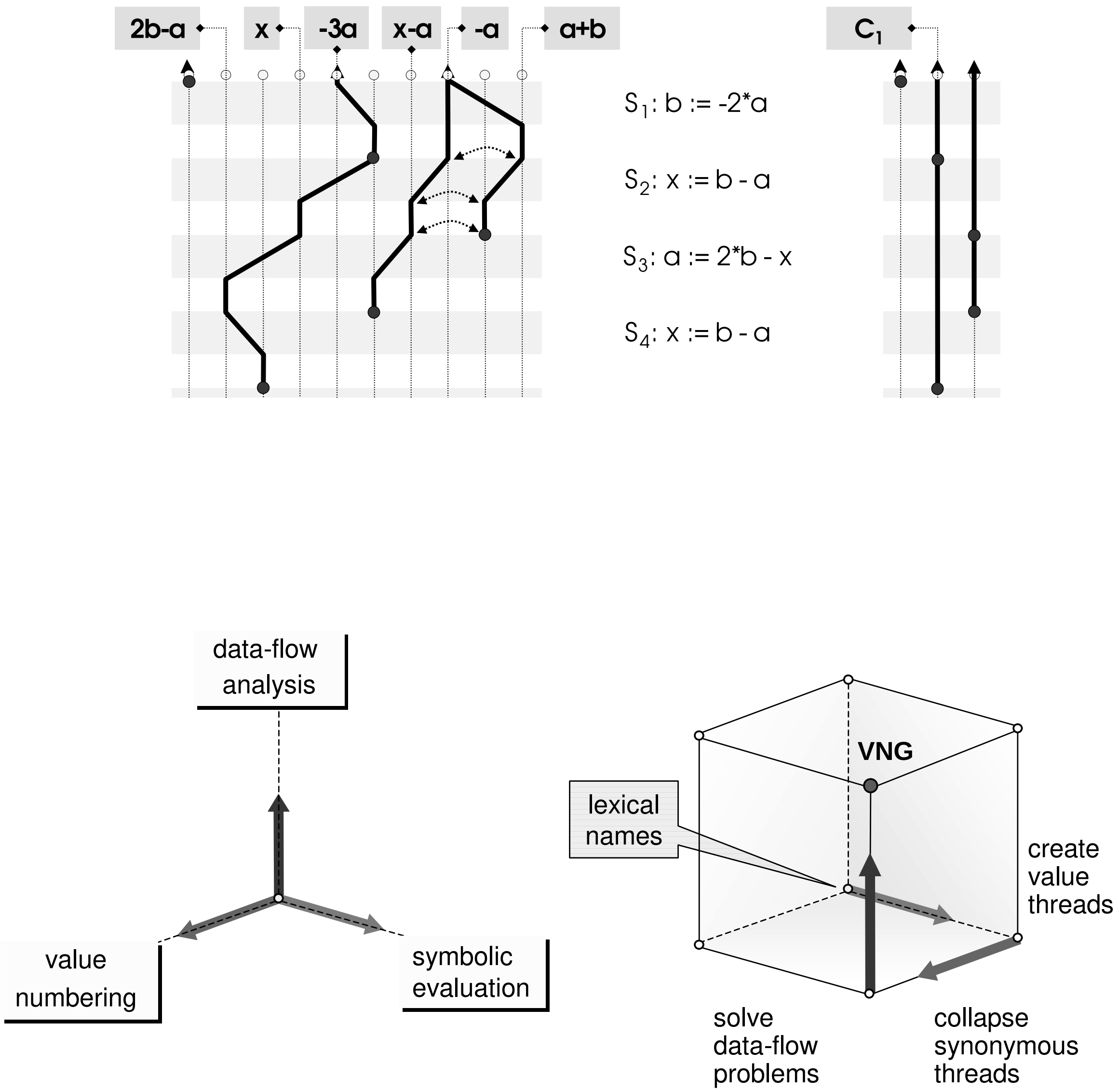


Figure 3.9: **The three steps of VNG construction.**

require a name with relational operators, for example $x < 0$. Yet another language of names is needed for analysis of repeated memory accesses to pointer-based data structures.

The seed set of computations of interest. The first VNG parameter is the set of computations whose recomputation is the focus of the analysis. These analyzed computations serve to seed the back-substitution process; the value threads will start unrolling backwards from them. The computations of interest form a *seed set* $\boldsymbol{O}$ of pairs $(n, \boldsymbol{e})$, where the name $\boldsymbol{e}$ denotes a computation of interest at the CFG node n. (The language of symbolic names S will be defined shortly.)

$$\boldsymbol{O} = \{(n, \boldsymbol{e}) \mid n \in N, \boldsymbol{e} \in \boldsymbol{S}\}$$

Note that while $\boldsymbol{e}$ is a lexical name, we assume it belongs to the language of symbolic names.

Example 3.2 The seed set $\boldsymbol{O}$ for the example program in Figure 3.3 consists of the entire right-hand-side expressions, for each node of the program:

$$\boldsymbol{O} = \{(S_1, -2 * a), (S_2, b - a), (S_3, 2 * b - x), (S_4, b - a), (S_5, a + x)\}$$

The seed set for the program in Figure 3.6 consists of the addresses for the array accesses (assuming $\mathit{sizeof}(A[i]) = 4$):

$$\boldsymbol{O} = \{(S_1, \mathit{adr}(A)), (S_5, \mathit{adr}(A) + 4 \times i), (S_5, \mathit{adr}(A) + 4 \times \mathit{min})\}$$

□

The language of symbolic names Given a seed set $\boldsymbol{O}$, we have to decide upon a symbolic language P from which we will draw the symbolic names for creating the value threads. For example, the pattern P may restrict the symbolic names to the form $a \times v + b$, where a and b are program literals and v is a scalar variable. Selection of P is mainly an issue of the implementation, where one desires a suitable trade-off among a few issues that influence the accuracy and complexity of the analysis:

suitability: The format of a symbolic name should suit the kind of computations in the seed set, as mentioned above; arithmetic expressions will need arithmetic operators to describe their values, while memory accesses have to describe indirect memory accesses. One could certainly permit all operators and an arbitrarily deep nesting of expressions; however, such a freedom of expression may create too many names during the back-substitution process, which may prove impractical. In practice, we observed that simple names are expressive enough for a given kind of seed computations.

semantics: How is the symbolic name evaluated? Does $x + y$ mean the usual addition of values? Does $x + y$ equal $y + x$. Is there a canonical form such that two equal expressions have "always" the same representation? For example, the canonical form for $x+z+y$ may be the alphabetically sorted $x+y+z$, which enables matching it with $z + x + y$.

interpretation: Which program statements modify the name and how? In other words, what happens when the back-substitution "inserts" a name into a symbolic name: what are the substitution and simplification rules? For example, are we willing to substitute only "copy" assignments $x := y$ or also more complex ones, such as $x := y + z$. In the latter case, the complexity of the synthesized names may grow beyond polynomial.

termination: When the symbolic language is infinite, the back-substitution may not terminate, as new names continue to be created. To force termination, let us introduce a parameter w, which stops the back-substitution after all control flow paths with w or fewer cycles have been examined.

implementation: The format of the name must permit efficient implementation. For example, can two symbolic names be compared in constant time? Can simplification be performed efficiently?

We assume the language P has a special name $\perp_{\text{name}}$ that denotes all values whose names are outside P.

```
Input:
    control flow graph $G = (N, E, start, end)$.
    seed set $\boldsymbol{O}$,
    substitution function substitute$(n, \boldsymbol{e}) : N \times P \to P$,
    loop iteration window $w$.
Output:
    value name graph $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{E}, \textbf{start}, \boldsymbol{vngend})$,
    set of synthesized names $\boldsymbol{S} \subseteq P$.

begin
    Step 1: route the value threads (Figure 3.11)
    Step 2: collapse the threads (Figure 3.14)
        convert threads to SSA form
        collapse threads using global value numbering
    Step 3: dataflow analysis
        define dataflow transfer functions (Section 3.4.1.4)
        solve dataflow problem (Chapter 4)
end
```

Figure 3.10: The algorithm for constructing the Value Name Graph.

Example 1. Consider the problem of determining redundant address computations for accesses to one-dimensional arrays. Choosing $P = c_0 + \sum_{v_i \in V} c_i v_i$ is sufficient to represent address expressions for array indexes such as $A[5 * i + j + 3]$ and even multi-dimensional but not $A[i * j]$. The set $\boldsymbol{O}$ contains address operands from all load and store nodes because they are the eventual consumers of address computations.

To accommodate indirect addressing, the symbolic language of value names is enriched with a pointer dereferencing operator $*$ and back-substitution rules for loads and stores. Loads increase the indirection level: when a name $t + 1$ is propagated backwards across $t :=$ load L, it will change to $*L + 1$. Stores may reduce the indirection: across store L, t, the name $*L + 1$ will change to $t + 1$.

The memory address is represented with a symbolic name $E = c_0 + c_1 v_1 + \ldots + c_n v_n + *(E')$, where c_i are literals, v_i are program variables, and $E' = E \mid \epsilon$. The term E' adds addressing indirection. In the actual implementation, one may want to set a maximum number of indirection levels, in order to limit the number of symbolic names created during back-substitution. In our experiments, the level used was level 0 (no $*$ operator in the address name) and level 1 (one $*$ operator in the address). □

3.4.1.2 Step 1: placing the value threads

The first step synthesizes the names for the analyzed values. The output of the first step is the set of symbolic names $\boldsymbol{S_1} \subset P$ and the value transfer set *VT* describing how the value threads alternate between symbolic names. The *VT* is defined as follows: $(n, \boldsymbol{e}, \boldsymbol{e'}) \in \mathit{VT}$ if the name $\boldsymbol{e}$ was back-substituted into e' at CFG node n.

The *VT* set represents the value threads without explicitly building the VNG nodes and edges shown in Figure 3.7(b). Instead, the VNG is represented by specifying only VNG edges that connect different names (where backsubstitution modified the propagated name). This is accomplished by representing each symbolic name $\boldsymbol{e} \in \boldsymbol{S_1}$ with a "symbolic" variable $[\boldsymbol{e}]$. The alternations between names are represented as assignments between those variables, as shown in Figure 3.12(c). We say that $(\boldsymbol{e} :=_n \boldsymbol{e'})$ if $(n, \boldsymbol{e}, \boldsymbol{e'}) \in \mathit{VT}$.

Note that the second step of the construction will transform the symbolic names $[e]$ assignments into the SSA form and collapse them using value numbering (see Section 3.4.1.3).

The algorithm, shown in Figure 3.11, is a sequence of backward traversal of the CFG. Starting from each of the seed computations, the traversal propagates their original (lexical) names and updates them accordingly across each affecting node by performing back-substitution. Each traversal places value threads along all acyclic paths; it stops at loop back-edges. The following traversal continues from the loop back-edges, extending the threads across loop iteration boundaries. The multiple traversals thus create threads that model value flow along all possible paths across w loop iterations. Setting w to 3 will discover equivalences among address computations that occur within any three consecutive loop iterations.

More specifically, the algorithm in Figure 3.11 first initializes the traversal worklist with the seed computations (line 2). Note that the worklist maintains names on the *exit* of a node; therefore, the traversal starts from the predecessors of the seed computations. Then, $w + 1$ traversals are performed, each using a separate worklist (lines 4–18). This termination is similar to *widening* [CC77]. The VNG thus models the flow of a value along any w consecutive loop iterations, which provides sufficient scope for optimizations in the instruction-level parallelism domain.

Line 8 forces the back-substitution to fail at the CFG start node *start* as a way of asserting the most conservative assumptions when the program is entered. The function *substitute*(n, e) determines how name e changes across node n (line 9). When the name changes across n, line 10 outputs the assignment denoting that the thread changes names at n. Lines 11–16 continues the traversal onto predecessors, passing names that cross backedges into the next traversal (line 13), as discussed above. The *backedges*(E) is defined to be the set of CFG edges that are backedges in *some* depth-first traversal of the CFG. Such a definition marks all backedges of irreducible loops. The visited flag marks that the name e has visited the exit of node n (line 7). The flag prevents propagation of a name twice (line 11). Finally, lines 19–21 terminate the threads that are left "in the air" along the back-edges after the traversal sequence was forcefully terminated after $w + 1$ traversals. These unfinished threads are grounded to $\bot_{\text{name}}$, asserting the same conservative assumptions as on the start node *start* at line 8.

Figure 3.12(c) shows the Step-1 VNG for the program in Figure 3.12(a). Figure 3.12(b) shows the VNG in the (conceptual) graph form. We illustrate the back-propagation using p_4, the address operand of the load in node 9. When propagating p_4 across the assignment $p_4 := p_3 + 1$ in node 7, the right-hand side $p_3 + 1$ is substituted into the current name p_4. We obtain $p_3 + 1$, which becomes another name for the analyzed address of the load (9). After crossing $p_3 := \textsf{load}\ L_p$ in node 6, $*L_p$ is substituted for p_3 and $*L_p + 1$ becomes yet another name for the address (L_p is the address of the global variable p). The name will be further changed at nodes 4, 3, and 1. (Note that the Figure 3.12(b) is showing the VNG construction only along the *then* path.) The address operands of remaining memory operations will also undergo this back-propagation. The process of name creation is demand-driven, as only the necessary names are created.

3.4.1.3 Step 2: collapsing the value threads

The second step collapses threads placed by the first step. Collapsing can be viewed as post-processing of the back-substitution. It places on the same thread those computations that back-substitution found equivalent but did not place on the same thread. These are computations that were symbolically reduced to the same name, but the reduction occurred "upstream" from the two computations, as is the case for S_3 and S_4 in Figure 3.7. Technically, the collapsing is performed by finding, at each node n, which symbolic names reduce to the same symbolic name. These names are synonyms for the same value and are placed into

```
Input:
    control flow graph $G = (N, E, start, end)$.
    seed set $\boldsymbol{O}$,
    substitution function $substitute(n, \boldsymbol{e}) : N \times P \rightarrow P$,
    loop iteration window $w$.
Output:
    synthesized symbolic names $\boldsymbol{S_1} \subset P$
    value transfers $A = \{(\boldsymbol{e_i} :=_n \boldsymbol{e_j}) \mid n \in N, \boldsymbol{e_i}, \boldsymbol{e_j} \in \boldsymbol{S_1}\}$.
begin
    $A := \emptyset$
    $worklist_0 := \{(m, e) \mid (n, e) \in \boldsymbol{O}.m \in pred(n)\}$
    for $i = 1$ to $w + 1$ do $worklist_i := \emptyset$
    for $i = 0$ to $w$ do
        while $worklist_i$ not empty do
            remove a pair $(n, e)$ from $worklist_i$
            $\text{visited}[n, e] := true$
            if $n = start$ then $e' := \bot_{\text{name}}$
            else $e' := substitute(n, e)$
            if $e' \neq e$ then add $(\boldsymbol{e} :=_n \boldsymbol{e'})$ to $A$, add $\boldsymbol{e'}$ to $\boldsymbol{S_1}$
            for each $m \in pred(n)$ such that $\text{visited}[m, e'] = false$ do
                if $(m, n) \in backedges(E)$ then
                    add $(m, e')$ to $worklist_{i+1}$
                else
                    add $(m, e')$ to $worklist_i$
            end for
        end while
    end for
    for each $(e, n) \in worklist_{w+1}$ do
        add $(\boldsymbol{e} :=_n \bot_{\text{name}})$ to $A$
    end for
end
```

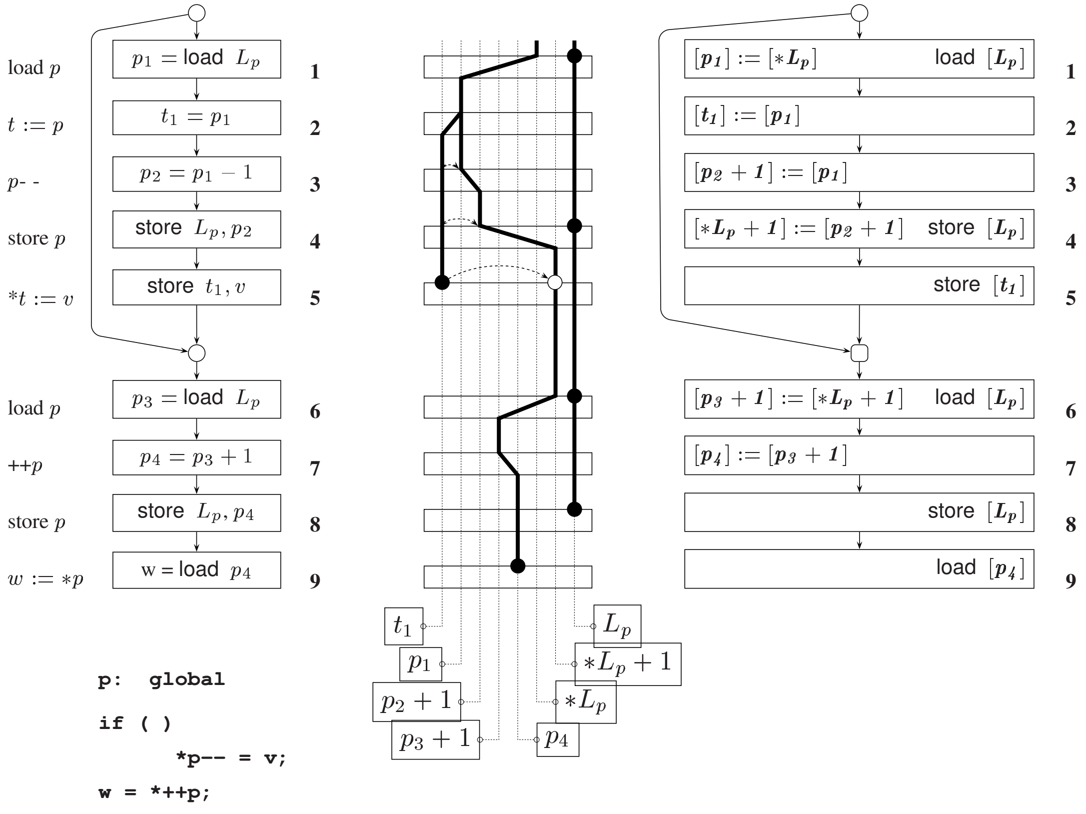
Figure 3.11: Step 1 of VNG construction: symbolic back-substitution.

an equivalence class, which effectively collapses all threads placed into the class. The names of the classes serve as the symbolic names for the final VNG.

Thread collapsing is thus expressed as a partitioning of the symbolic name space into equivalence classes, at each CFG node. Therefore, the second step is called *Symbolic Value Numbering (SVN)*, for its similarity with the standard Global Value Numbering (GVN) that also discovers which names refer to the same value [AWZ88]. To compare our SVN with the GVN, we observe that, like the GVN,

- the SVN computes which value names are synonymous, representing synonyms as partitions;
- the SVN computes synonyms globally, accounting for all program paths leading to a given CFG node;
- the SVN uses a compact SSA encoding of names, to maintain a single, global partitioning, rather than a separate one at each CFG node.

Unlike GVN,



```
p:  global
if ( )
        *p-- = v;
w = *++p;
```

(a) the source program. **(b)** the (conceptual) VNG. **(c)** the VNG after Step 1.

Figure 3.12: **Step 1 of the VNG construction: example**

- the SVN detects synonyms based on their symbolic (algebraic) equivalence, rather than from their structural (syntactic) equivalence.
- the SVN uses a more precise (but also more expensive) partitioning algorithm, as explained below.

The symbolic value numbering has two sub-steps. First, the Step-1 VNG is transformed into the Static Single Assignment (SSA) form [AWZ88]. Second, the SSA-ed symbolic names are globally partitioned. Recall that after Step 1, the VNG is represented as assignments between $[e]$ variables, denoted $(e :=_n e')$. When converted to SSA form, each $[e]$ variable will be assigned exactly once.[2] The SSA form is achieved by creating multiple copies of each variable, one for each assignment, and by connecting the copies with ϕ-assignments inserted at control flow merge points that can be reached by two different assignments of the same variable. The property desired from the SSA transformation is that the insertion of ϕ-assignments effectively breaks threads into the *largest* snippets on which the synonyms relation remains the same. As a result, the collapsing need not be expressed at each node, but on the snippets, in single global partitioning. Because the $[e]$ variables are ordinary scalar variables, any existing SSA construction algorithm can be used [CFR$^+$91, SG95]. The SSA transformation changes the symbolic name space from $\boldsymbol{S_1}$ into an SSA-ed name space denoted $\boldsymbol{S_2}$.

---

[2] Except for $[\bot_{\text{name}}]$, which is never assigned, because it is never back-substituted into.

```
$S_1$: $x_0$ := $\phi$(..,..)
$S_2$: if ( )
$S_3$:     $y_0$ := $x_0$
$S_4$: else
$S_5$:     $y_1$ := $x_0$
$S_6$: end if
$S_7$: $y_2$ := $\phi$($y_0$,$y_1$)
$S_8$: $x_0$ ...  $y_2$
```

Figure 3.13: **GVN fails to find the equivalence of x_0 and y_2, In contrast, the SVN succeeds.**

To partition the SSA-ed names, one could employ the GVN algorithm [AWZ88]. However, GVN computes an imprecise solution in cases that may occur often on the VNG. Consider Figure 3.13. In this program, GVN fails to find the equivalence of x_0 and y_2, but SVN succeeds.

Our partitioning algorithm represents the synonyms as a graph, in which edges connect equivalent names. The algorithm is optimistic. It starts with all names assumed equal and then removes edges that cannot safely be assumed synonymous. An edge $([\boldsymbol{e_i}], [\boldsymbol{e_j}])$ is removed when there is control flow path on which $[\boldsymbol{e_i}]$ is reduced to $\perp_{\text{name}}$ (back-substitution failed) before $[\boldsymbol{e_i}]$ and $[\boldsymbol{e_j}]$ reduced to the same non-$\perp_{\text{name}}$ name. After the fixed point is reached, the $O(|\boldsymbol{S_2}|^2)$-sized graph is converted to $O(|\boldsymbol{S_2}|)$-sized equivalence-class partitioning, to save memory. The more economical class partitioning cannot be used before the fixed point is reached because the graph does not necessarily represent an equivalence relation. Therefore, equivalences must be represented at the granularity of edges, not partitions.

The details of the algorithm are in Figure 3.14. The input is the equality graph. The graph is optimistically initialized; edges connect each two SSA variables whose live ranges overlap. Thus, the graph is a live-range interference graph. It is built during the SSA construction. The algorithm starts by identifying pairs of names that are definitely not synonymous and then propagates their inequality through the def-use chain of the SSA variables, until no more edges can be removed. Line 1 finds all names $[\boldsymbol{e_i}]$ that failed to be back-substituted, at any node n. Line 2 then finds all names $[\boldsymbol{e_j}]$ that are live at that node. Names $[\boldsymbol{e_i}]$ and $[\boldsymbol{e_j}]$ must be different and are inserted into the worklist. Taking edges from the worklist, the algorithm removes an edge (line 9) if the edge has not been removed yet (line 8). For each removed edge $([\boldsymbol{e_i}], [\boldsymbol{e_j}])$, their inequality is propagated to all uses of $[\boldsymbol{e_i}]$ and $[\boldsymbol{e_j}]$, in a forward direction along the def-use chain, across ϕ-assignments and the back-substitution assignments $(\boldsymbol{e_i} :=_n \boldsymbol{e_j})$.

Returning to the running example, Figure 3.15(a) shows the SSA form of the Step-1 VNG shown in Figure 3.12(c). In our example, only $[*\boldsymbol{L_p} + \boldsymbol{1}]$ needs an SSA subscript and a ϕ-node; the live ranges of other $[\boldsymbol{e}]$ variables do not cross the control-flow merge point. The SVN algorithm partitions the $[\boldsymbol{e}]$ names into congruence classes as shown in the figure. The names of the classes are the symbolic names used by the final VNG, which is shown in Figure 3.15(b). Note that the optimized computations have been rewritten to refer to the names of the equivalence classes.

Notice that the Step-1 VNG cannot find reuse between the equivalent nodes 5 and 9, because they are not on the same value thread (see Figure 3.12(b)). After the threads have been collapsed, the two nodes refer to the same thread, which is represented as a chain of data dependences from C_2 to C_3 (see Figure 3.15(b)).

Input:

$IG = (IN, IE)$, interference graph for the SSA'ed name variables:

$IN = \boldsymbol{S_2}$,

$IE \subseteq (IN \times IN)$: $([e_i],[e_j]) \in IE$ iff $[e_i]$ and $[e_j]$ are both live at some CFG node n.

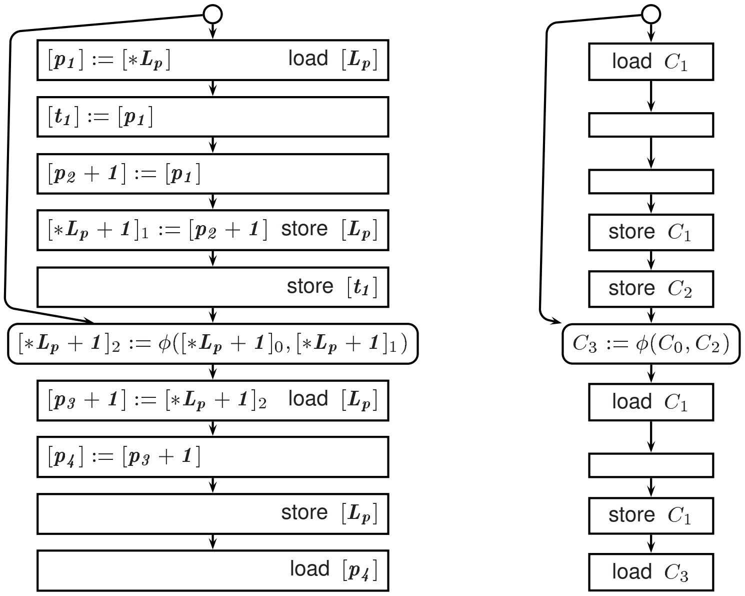
Output:

$\boldsymbol{S}$, the symbolic name space of the final VNG

$\boldsymbol{S_2} \rightarrow \boldsymbol{S}$, partitioning of SSA'ed name space into equivalence classes.

```
begin
    for each $(e_i :=_n \bot_{\text{name}})$ do
        for each $[e_j]$ that is live at node $n$ do
            add $(e_i, e_j)$ to $worklist$
        end for
    end for
    while $worklist$ not empty do
        remove edge $([e_i],[e_j])$ from $worklist$
        if $([e_i],[e_j]) \in IE$ then
            remove edge $([e_i],[e_j])$ from $IE$
            for each assignment $([e_k] :=_n [e_i])$ or $[e_k] :=_n \phi(\ldots, [e_i], \ldots)$ do
                add $([e_k],[e_j])$ to $worklist$
            end for
            for each assignment $([e_k] :=_n [e_j])$ or $[e_k] :=_n \phi(\ldots, [e_j], \ldots)$ do
                add $([e_i],[e_k])$ to $worklist$
            end for
        end if
    end while
    partition $IN$ into classes: each connected subgraph of $IG$ is one class $C_i$
end
```

Figure 3.14: Step 2: collapse value threads using value numbering.



Congruence classes:

C_0: $\{[*\boldsymbol{L_p} + \boldsymbol{1}]_0\}$
C_1: $\{[\boldsymbol{L_p}]\}$
C_2: $\{[*\boldsymbol{L_p}], [\boldsymbol{p_1}], [\boldsymbol{t_1}], [\boldsymbol{p_2} + \boldsymbol{1}], [*\boldsymbol{L_p} + \boldsymbol{1}]_1\}$
C_3: $\{[*\boldsymbol{L_p} + \boldsymbol{1}]_2, [\boldsymbol{p_3} + \boldsymbol{1}], [\boldsymbol{p_4}]\}$

(a) the SSA form of the step-1 VNG. **(b)** the final VNG.

Figure 3.15: **Step 2 of the VNG construction: example**

3.4.1.4 Step 3: dataflow transfer functions

The previous two steps constructed the nodes and edges of the VNG. To solve dataflow problems on the VNG, it remains to construct the dataflow transfer functions and the equation system for computing dataflow problems. In this section, we show how to compute availability of values, a forward problem. The availability computation presented here is path-*in*sensitive, because it only determines whether the value is available along *all* paths. A solution to this problem is sufficient for performing global common subexpression elimination (Global CSE) on the VNG, i.e., for eliminating fully redundant computations. Chapter 4 presents a path-sensitive formulation of dataflow analysis on the VNG, which will be used to perform partial redundancy elimination.

First, we define the VNG constructed by the first two steps.

Definition 3.3 (Value Name Graph) Let $G = (N, E, \textit{start}, \textit{end})$ be a control flow graph and $\boldsymbol{S}$ be the name space. The names in $\boldsymbol{S}$ are the names of of congruent class names created in Step 2, together with the names $\perp_{\text{name}}$.

The Value Name Graph (VNG) is a graph $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{E}, \textbf{start}, \textbf{end})$, where $\boldsymbol{N}$ is the set of nodes, $\boldsymbol{N} = N \times \boldsymbol{S}$. $\boldsymbol{E}$ is the set of edges, $\boldsymbol{E} \subseteq \boldsymbol{N} \times \boldsymbol{N}$ such that $((m, \boldsymbol{e}'), (n, \boldsymbol{e})) \in \boldsymbol{E}$ iff $m \in \textit{pred}(n)$ and

- if $\boldsymbol{e}$ is defined (by a ϕ-node) in n, then $\boldsymbol{e}'$ is the (m, n)'s argument of the ϕ, or
- if $\boldsymbol{e}$ is not defined in n, then $\boldsymbol{e} = \boldsymbol{e}'$.

The start node **start** is connected to all nodes $(\mathit{start}, \boldsymbol{e})$, for all $\boldsymbol{e} \in \boldsymbol{S}$. Similarly, **end** terminates all CFG nodes, by connecting all $(\mathit{end}, = \boldsymbol{e})$ VNG nodes. □

The *pred* and *succ* functions are defined on VNG as usual.

Definition 3.4 (Thread) A *thread* is a path in the VNG graph. In contrast, a *path* is a path on the CFG graph. Note the relationship between threads and paths. For each thread $\boldsymbol{p}$, there is exactly one *sibling* path p, which can be obtained with a function *path*: $p = \mathit{path}(\boldsymbol{p})$. □

Clearly, while each thread has one sibling paths, along each path there are multiple threads, each for a different value.

Each VNG node $\boldsymbol{n} = (n, \boldsymbol{e})$ corresponds to a value $\boldsymbol{e}$ at a CFG node n. We can now compute the availability of these values. Let us assume that the value is computed by all VNG nodes in the seed set $\boldsymbol{O}$.

$$\boxed{\begin{array}{rcl} \mathit{AVAIL}_{\mathit{out}}[\textbf{start}] & = & \bot \\ \mathit{AVAIL}_{\mathit{out}}[\boldsymbol{n}] & = & f_{\boldsymbol{n}}^{\mathit{AVAIL}}(\mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}]) \\ \mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] & = & \bigwedge_{\boldsymbol{m} \in \mathit{pred}(\boldsymbol{n})} \mathit{AVAIL}_{\mathit{out}}[\boldsymbol{m}] \\ f(\mathit{AVAIL})_{\boldsymbol{n}}(x) & =_{\mathit{df}} & \begin{cases} \top & \text{if } \boldsymbol{n} \in \boldsymbol{O}, \quad \text{(GEN)} \\ x & \text{otherwise.} \end{cases} \end{array}}$$

A computation $\boldsymbol{n} = (n, \boldsymbol{e})$ is fully redundant and can be removed if $\mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] = \top$.

Killing stores. Dataflow analysis on the VNG is closely coupled with the back-substitution step. Recall that the VNG threads detect reuse aggressively—because back-substitution may be defined to ignore may-aliasing, the threads extend uninterrupted across potentially killing stores. As a result, the VNG detects instructions that always read from the same location but it does not reflect that a store *may* change the contents of this location between these two reads. Treating may-aliasing kills separately is an intentional design decision, because some hardware mechanism enable reuse exploitation even in the presence of (infrequent) kills, using a data-speculative load [KSR94, GKKG98, RCT$^+$98].

May-aliasing can be easily accounted for in dataflow analysis on the VNG. Using our running example, assume that p_4 may equal L_p. Because $[\boldsymbol{p_4}]$ belongs to congruent class C_3 and $[\boldsymbol{L_p}]$ belongs to C_1, each store to C_1 must kill reuse in class C_3 and vice versa. Therefore, in Figure 3.15(a), the store in node 8 would kill the reuse for the load in node 9. To account for killing statements, we modify the transfer function f^{AVAIL}:

$$\boxed{f(\mathit{AVAIL})_{(n,e)}(x) =_{\mathit{df}} \begin{cases} \top & \text{if } \boldsymbol{n} \in \boldsymbol{O}, \quad \text{(GEN)} \\ \bot & \text{if } \boldsymbol{e} \text{ may be aliased with } n\text{'s value, (KILL)} \\ x & \text{otherwise.} \end{cases}}$$

Depending on the optimizer, this kill may entirely destruct the reuse, preventing register promotion, or may mark only the reuse as unsafe.

3.5 Separable VNG

This section introduces a restricted case of the value name graph. The purpose is to simplify the presentation of the profiling and transformation algorithms in the following chapters. The restricted

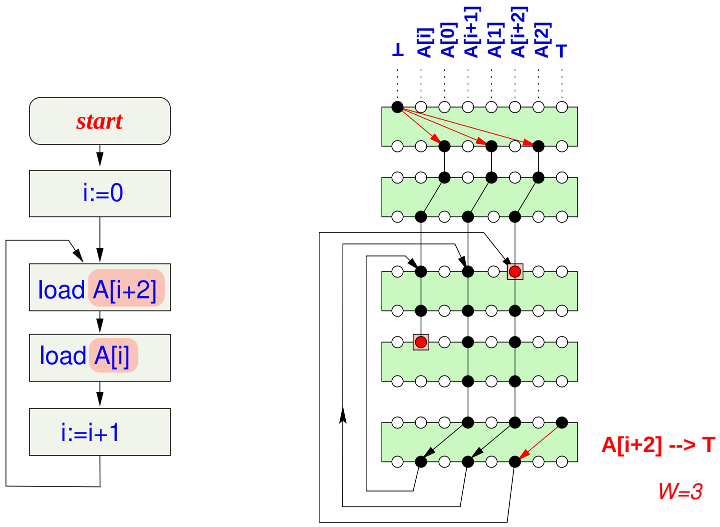


Figure 3.16: **The VNG can detect recurrent array accesses.**

version, called *separable*, behaves just like a CFG with lexical (Morel-Renviose-style) redundancies. This simplifications allows us to present the algorithms first at an intuitive level, and then extend them for the *general* VNG.

Definition 3.5 (Separable VNG) A value name graph $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{E}, \textbf{start}, \textbf{end})$ is called *separable* if all symbolic names can be analyzed separately, i.e., their threads do not interfere. Formally, for each edge $\boldsymbol{e} = ((n, \boldsymbol{e}), (m, \boldsymbol{e}')) \in \boldsymbol{E}$, either $\boldsymbol{e} = \boldsymbol{e}'$ or $\boldsymbol{e} = \bot_{\text{name}}$ or $\boldsymbol{e}' = \bot_{\text{name}}$. ☐

3.6 Applications of the VNG

This section shows how the VNG can be parameterized to various value-flow analysis problems. It presents the detection of recurrent array accesses and a version of constant propagation.

3.6.1 Recurrent array accesses

Because the back-substitution traverse a few consecutive loop iterations, the VNG naturally exposes the redundancy of recurrent array accesses. In fact, the VNG is a generalization of the *stretched loop* [BG96b]. Figure 3.16(a) contains two such accesses. The VNG for the loop is in Figure 3.16(b). In the VNG, the equivalent array accesses are connected with a thread that extends across three loop iterations. For the analysis and transformation stages of the PathFinder optimizer, such inter-iteration threads are no different than “acyclic” ones, and thus are handled uniformly.

3.6.2 Distributive non-linear constant propagation

The traditional formulation of Constant Propagation *(CP)* does not distribute across the meet operator; therefore, algorithms for this data-flow problem need to trade precision for effectiveness [CCKT86, GT93]. Recently, a distributive formulation of a *linear* version of CP was presented [SRH96]. The linear version is restricted in that only assignments with at most one variable in the right-hand side (e.g., `x:=2*y+3`)

```
if (..)
   a := 5
   b := 3
else
   a := 3
   b := 5
end if
x := a * b
```

a) is x a constant?

```
if (..)
    $([\boldsymbol{3} \times \boldsymbol{a}] := [\boldsymbol{15}])$
    $([\boldsymbol{a} \times \boldsymbol{b}] := [\boldsymbol{3} \times \boldsymbol{a}])$
else
    $([\boldsymbol{5} \times \boldsymbol{a}] := [\boldsymbol{15}])$
    $([\boldsymbol{a} \times \boldsymbol{b}] := [\boldsymbol{5} \times \boldsymbol{a}])$
end if
$([\boldsymbol{x}] := [\boldsymbol{a} \times \boldsymbol{b}])$
```

b) Step-1 VNG

```
if (..)
    $([\boldsymbol{3} \times \boldsymbol{a}]_{\mathit{0}} := [\boldsymbol{15}])$
    $([\boldsymbol{a} \times \boldsymbol{b}]_{\mathit{0}} := [\boldsymbol{3} \times \boldsymbol{a}]_{\mathit{0}})$
else
    $([\boldsymbol{5} \times \boldsymbol{a}]_{\mathit{0}} := [\boldsymbol{15}])$
    $([\boldsymbol{a} \times \boldsymbol{b}]_{\mathit{1}} := [\boldsymbol{5} \times \boldsymbol{a}]_{\mathit{0}})$
end if
$[\boldsymbol{a} \times \boldsymbol{b}]_2 := \phi([\boldsymbol{a} \times \boldsymbol{b}]_0, [\boldsymbol{a} \times \boldsymbol{b}]_1)$
$([\boldsymbol{x}]_{\mathit{0}} := [\boldsymbol{a} \times \boldsymbol{b}]_{\mathit{2}})$
```

$$
\begin{aligned}
&C_0 = \{[\boldsymbol{15}],\\
&[\boldsymbol{3} \times \boldsymbol{a}]_0, [\boldsymbol{5} \times \boldsymbol{a}]_0,\\
&[\boldsymbol{a} \times \boldsymbol{b}]_0, [\boldsymbol{a} \times \boldsymbol{b}]_1,\\
&[\boldsymbol{a} \times \boldsymbol{b}]_2, [\boldsymbol{x}]_0\\
&\}
\end{aligned}
$$

c) SSA form of Step-1 VNG

d) after value numbering

Figure 3.17: **Constant propagation using the VNG.**

are interpreted. Using VNG, one is able to formulate a distributive, non-linear CP algorithm which can handle arbitrary assignments (within the domain of the symbolic pattern P). The authors are not aware of any other such algorithm.

The VNG-based constant propagation (VNGCP) is performed as a by-product of the VNG construction by grouping names into equivalence classes together with program constants: if any name is back-substituted to a constant, then there must be a path along which an operand from the seed set $\boldsymbol{O}$ has a constant value. The subsequent value numbering step verifies: if a name $\boldsymbol{e}$ is in a class with a constant d, the value of e equals d along all program paths. No data-flow analysis (Step 3) is required because the value numbering already verified that the name is a constant along all incoming paths. Figure 3.17 presents an example. After the value numbering step of the VNG construction, the congruent class contains the symbolic name $[\boldsymbol{15}]$, a constant, and also the symbolic name $[\boldsymbol{x}]_0$. These two names are synonymous, and hence x is the constant 15.

It should be noted that the distributivity comes at cost of exponential worst-case complexity, due to the number of symbolic names that the VNG may contain. However, the symbolic approach to constant propagation has two advantages, compared to existing CP algorithms. First, because it is based on *symbolic* names, the VNGCP can find constants whose value is known only at link-time. For example, x may be found to be $A + 4$, where A is the address of an array. Second, the freedom to choose P permits discovery of constants across assignments with arbitrary right-hand side expressions; adjusting P allows one to tune the tradeoff between the cost and the power of the analysis.

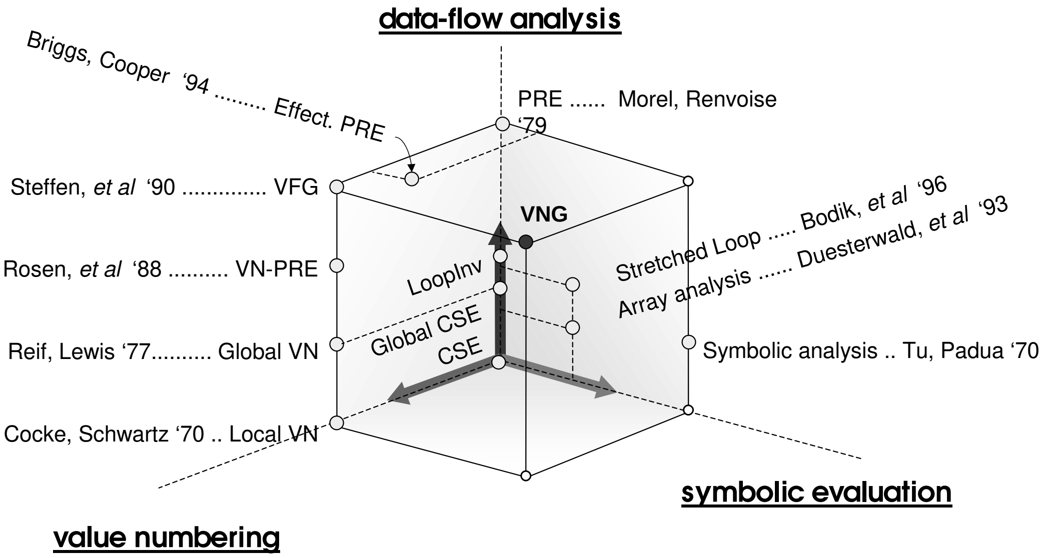


Figure 3.18: **Related work.** Existing techniques for value-flow detection can be compared on the basis of which of the three orthogonal mechanisms they employ and to what extent they exploit their power.

3.7 Related work

A number of approaches for detecting equality among values have been developed. The prior work can be best compared with our representation on the basis of what subset of the three underlying mechanisms it employs and to what degree their power is exploited. The three-dimensional cube serves well to relate the existing techniques (Figure 3.18).

Let us start with techniques that are based on the lexical names of values. To improve their power, they add a degree of dataflow analysis.

Local common subexpression elimination (CSE). When none of the three methods is used, equivalent computations are identified only using their lexical names. Without dataflow analysis, one is restricted to basic blocks. This path-insensitive (local), name-insensitive (lexical) technique is in the origin of the three-dimensional cube.

Global common subexpression elimination (Global CSE). By adding some dataflow analysis power, one can verify if subexpressions are equal along *all* paths, but still is path-insensitive [Coc70].

Loop-invariant code motion (LoopInv). By even smarter global dataflow analysis, one can find loop-invariant computations. Since loop-invariants are redundant only along *some* paths, this technique is partially path-sensitive.

Partial redundancy elimination (PRE). PRE uses dataflow analysis to its full extent—it identifies computations that are equivalent only along an arbitrary subset of paths [MR79].

The techniques based on value numbering follow an analogical historical improvement, through adding the path-sensitive power of dataflow analysis.

Local value numbering (Local VN). Restricted to a basic block, local value numbering does not consider any program paths.

Global value numbering (GVN). This method detects syntactic equivalences that hold along *all* program paths. While the commonly used algorithm for GVN [AWZ88] do not employ dataflow analysis to verify the all-path equivalence, they use the SSA form, which essentially encodes a result of dataflow analysis (def-use chains) in a compact way. The same is true about [RL77], which uses a precursor of the SSA form.

Value-numbering-driven PRE (VN-PRE). Rosen *et al.* present a PRE algorithm that is driven by syntactic equivalences, rather than lexical ones [RWZ88].

Instead of using lexical names, the redundancy elimination approach in [Cli95] solves data-flow problems on the name space of global value numbers [AWZ88].

Value Flow Graph (VFG). Steffen *et al.* make value numbering fully path-sensitive. Their graph representation is similar to the VNG. In fact, it can, too, be viewed as placing value threads. The difference is that their threads are based on equivalence of syntactic terms, whereas our are based on algebraic equivalence.

While adding path-sensitivity to lexical and syntactic (value numbering) names was systematic, there seem to be only a few techniques for performing symbolic evaluation along paths.

Symbolic analysis. Tu and Padua use a *gated* SSA form to reason about symbolic expressions along all program paths simultaneously. Rather than projecting symbolic expressions onto the CFG points, they assign path predicates to symbolic expressions using the gating functions of GSA. This approach can be effective in answering queries on pairs of symbolic expressions, especially when the resulting symbolic expressions have simple gating functions. In order to use this representation for data-flow analysis, a powerful boolean symbolic evaluation system may be needed. Johnson and Schlansker describe how such a system can be constructed and utilized in solving predicated flow problems [JS96].

Array Analysis. Duesterwald, Gupta, and Soffa encoded the algebraic information about array index expressions into dataflow transfer functions. As a result, loop-carried (both recurrent and loop-invariant) array accesses are detected [DGS93b].

Stretched loop. Bodik and Gupta detect loop-carried array accesses with more path-sensitivity. The stretched loop is a (virtually) unrolled loop that reduces array analysis (on affine indices) into the scalar variable domain. In the simpler domain, a PRE-style dataflow analysis can be applied [BG96b]. The application domains of both [DGS93b] and [BG96b] are limited to single loops in which loop indices are incremented unconditionally and address expressions of interest for load/store operations are affine functions of such loop indices. The VNG can analyze nested loops with arbitrary control flow and conditionally incremented induction variables.

Effective PRE. Briggs and Cooper perform PRE dataflow equations on a name space that uses a limited form of symbolic evaluation (reassociation) and a form of value numbering. Because VNG does not use reassociation, it is not strictly better than the Effective PRE technique [BC94]. Expression reassociation is orthogonal to the methodology behind the VNG. By using VNG on a re-associated program, one can benefit from the combined power of the two approaches.

In contrast to existing methods, the VNG combines the power of all three techniques. The authors are not aware of any existing technique that would combine all three approaches into a systematic analysis tool.

The VNG was inspired by the conceptual framework described by Rau [Rau91] in the spirit of abstract interpretation [CC77]. He describes how repeated back-substitution of names along loop back-edges can detect loop-carried value equivalences on a path-per-path basis. The paper presents the problems that arise in naming and comparing symbolic expression originating in different loop iterations. The PATHFINDER framework offers a practical solution to these problems.

3.8 Experiments

The practicality of the VNG representation depends largely on the size of the symbolic name space $\boldsymbol{S}$. When the language of symbolic names P is infinite, the worst-case size of the name space $\boldsymbol{S}$ is exponential in the number of CFG nodes, even when the number of back-substitution iterations w is a constant. This would translate into exponential worst case time and space complexity of our analysis. Fortunately, our experiments show that the size of $\boldsymbol{S}$ is in practice moderate. In fact, for the value-flow analysis of array accesses, it is smaller than the the number of program variables.

Rather than reporting the size of the final name space $\boldsymbol{S}$, we measure the size of $\boldsymbol{S_1}$, the name space created in the first step of the construction. Because $\boldsymbol{S_1}$ is larger than $\boldsymbol{S}$, its size safely estimates the complexity of the analysis on the VNG, but also the complexity of the VNG construction, namely the SSA transformation and the value numbering step.

The experiments evaluated the size of $\boldsymbol{S_1}$ and other VNG properties on a VNG tailored for an elimination of redundant loads and stores. The seed set $\boldsymbol{O}$ contained all source operands of all load and store instructions. The language P was chosen to express address expressions for one-dimensional array accesses, hence $P = c_0 + \sum_{v_i \in V} c_i v_i$. The VNG was implemented in the HP Labs VLIW backend compiler *elcor*.

To determine the rate at which the name space $\boldsymbol{S_1}$ grows as a function of w, we built the VNG for various values of w. The column 0 in Table 3.1 is the initial size of $\boldsymbol{S_1}$, when $\boldsymbol{S_1} = \boldsymbol{O}$. The column labeled i shows the size of $\boldsymbol{S_1}$ after i iterations of back-substitution. The VNG was built for 151 procedures of 12 programs and report those procedures with the steepest growth rate of $\boldsymbol{S_1}$. The first three benchmarks come from spec92, `grep` is a Unix utility, and the `mm`'s are simple matrix multiplies, with static and dynamic memory allocation, respectively. Before the VNG analysis was applied, all classical optimizations [ASU86] were applied on these procedures.

In a majority of procedures in Table 3.1, the growth rate of $\boldsymbol{S_1}$ is linear; in a few, the rate only slightly exceeds a linear curve. For an exponential growth to be observed, a large portion of variables involved in the analysis would have to be back-substituted into a different expression along each branch of an if-statement in a loop body. The address expression computation in the programs we considered did not have such property.

The column V shows the number of variables (virtual registers) referenced in the procedure. The following two columns give the size ratio of $\boldsymbol{S_1}$ and V, for $w = 2$ and $w = 3$. These two values have been selected because they are most likely to be used in practice; there are few opportunities for loop carried value reuse beyond an iteration distance of two [GKT91]. Surprisingly, the name space is much smaller that the set V. While the growth rate of $\boldsymbol{S_1}$ may be non-linear and its actual size may be larger for optimizations other than load/store elimination, we expect the size of $\boldsymbol{S_1}$ to be comparable to V, which enables efficient analysis. If some symbolic patterns P permit dramatic name space growth in practice, the size of $\boldsymbol{S_1}$ can be restricted during its demand-driven construction by terminating the back-substitution as soon as a prede-

Benchmarks		back-substitution iterations w								N	bs	V	S/V	S/V
program	procedure	0	1	2	3	4	5	6	7		[%]		$w = 2$	$w = 3$
alvinn	initialize	9	17	27	41	59	81	107	137	181	6.6	106	0.25	0.39
	input_hidden	8	14	21	28	35	42	49	56	52	17.3	48	0.43	0.58
	hidden_output	6	10	17	24	31	38	45	52	47	14.9	46	0.37	0.52
	output_hidden	16	28	56	89	127	170	218	271	100	26.0	70	0.83	1.27
	hidden_input	4	7	13	19	25	31	37	43	36	16.7	27	0.48	0.70
compress	main	154	210	221	232	243	254	265	276	1045	12.9	633	0.35	0.37
	decompress	47	76	90	101	112	123	134	145	241	20.3	166	0.54	0.61
	prratio	7	12	12	12	12	12	12	12	65	7.7	52	0.23	0.23
	output	61	89	91	93	95	97	99	101	252	23.0	197	0.46	0.47
ear	InitCorrelation	37	51	52	53	54	55	56	57	158	18.9	122	0.43	0.43
	SendInputToCorr	18	27	32	37	42	47	52	57	65	27.7	55	0.58	0.67
	FFTCorrelation	64	105	130	157	188	221	256	293	338	19.5	260	0.50	0.60
	HartleyCorr	42	72	91	111	133	156	180	205	232	19.0	177	0.51	0.63
	StretchDisplay	38	58	73	90	109	130	153	178	187	20.9	141	0.52	0.64
	main	86	131	157	183	209	235	261	287	668	14.1	455	0.34	0.40
	EARSTEP	87	121	141	161	181	201	221	241	337	22.8	246	0.57	0.65
grep	execute	156	301	479	659	839	1019	1199	1379	2193	15.4	1081	0.44	0.61
mm	main	16	29	53	86	128	179	239	308	183	17.0	124	0.43	0.69
mm_dyn	main	25	36	44	52	60	68	76	84	213	15.9	145	0.30	0.36
									Average	347	17.7	270	0.45	0.52

Table 3.1: The size of name space S as a function of W, and other characteristics of the VNG relevant to analysis efficiency.

termined number of names has been created. Such approach was successfully used in the demand-driven analysis in [BGS97a].

To provide some intuition why the symbolic names grow moderately, Table 3.1 presents also the percentage of CFG nodes on which back-substitution was performed, for at least one symbolic name (column *bs*). The column N gives the number of CFG nodes in each procedure. Each node contains a single intermediate statement. The experiments show that more than 80% of nodes does not influence the name of the analyzed address expressions for array accesses.

Chapter 4

Path-Sensitive Dataflow Analysis

Section 3.4.1.4 presented a dataflow analysis on the VNG. That analysis was path-insensitive; it merely verified whether a property (value availability) holds along all paths or not. This chapter presents a dataflow analysis that is path-sensitive. It finds (and marks) *optimizable* program paths—i.e., path along which a value can be reused. The results of the analysis navigate the remaining stages of the optimizer.

Because there are exponentially many optimizable paths, the central issue is how to represent them compactly, with only polynomial cost. A compact marking is accomplished with a lattice that distinguishes whether all, some, or none of the paths through a given node are optimizable. From such a compact encoding, the transformation stage of PathFinder can recover individual optimizable paths.

This chapter starts by defining the optimization property (value reuse) that should be marked on the VNG. Next, Section 4.2 defines two dataflow problems that, when solved, identify whether a path has the optimization property. Section 4.3 defines a lattice that allows computing the two dataflow problems in a path-sensitive way and Section 4.4 gives the dataflow equations for computing the problems. Section 4.5 discusses the solvers that can compute the fixed-point solution, and also deals with some issues of the implementation.

4.1 Value reuse (the analyzed property)

The VNG representation exposes the *value flow*—it shows how the name of the value changes as it flows through the program, via building the value threads. For the purpose of optimization, we are interested in subthreads along which the value flows *between* computations. We call these subthreads *reuse threads*. They correspond to optimizable paths: we want to identify them during the analysis, and separate them during the optimization.

Reuse threads are defined using three kinds of VNG nodes: *generators*, which compute the value; *users*, which both compute the value and can reuse it; *kills*, which kill the value. The reuse thread is a kill-free subthread between a generator node and a user node. The kill nodes account for potential aliases (aggressively) disregarded during-back-substitution, as discussed in Section 3.4.1.4.

Definition 4.1 (Reuse Threads) The set of reuse threads is parameterized on a tuple $(\boldsymbol{G}, \boldsymbol{D}, \boldsymbol{U}, \boldsymbol{K})$, where

- $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{E})$ is a VNG,
- $\boldsymbol{D}$ is the set of *generator* VNG nodes, $\boldsymbol{D} \subset \boldsymbol{N}$, that generate the value,
- $\boldsymbol{U}$ is the set of *user* VNG nodes, $\boldsymbol{U} \subseteq \boldsymbol{D}$, a subset of generator node that (generate but also) consume the value,

- $\boldsymbol{K}$ is the set of *kill* VNG nodes, $\boldsymbol{K} \subset \boldsymbol{N}, \boldsymbol{K} \cap \boldsymbol{U} = \emptyset$, that may "modify" the value.

Reuse threads, denoted $\boldsymbol{R}$, is a (potentially infinite) set of finite-length VNG threads from a generator node to a user node such that no thread contains a generator node or a kill node,

$$\boldsymbol{R}(\boldsymbol{G}, \boldsymbol{D}, \boldsymbol{U}, \boldsymbol{K}) = \{\boldsymbol{p}\langle \boldsymbol{n}, \boldsymbol{m}\rangle \mid \boldsymbol{n} \in \boldsymbol{D}, \boldsymbol{m} \in \boldsymbol{U}, |\boldsymbol{p}| > \boldsymbol{0}, \boldsymbol{p} \cap (\boldsymbol{D} \cup \boldsymbol{K}) = \emptyset\}$$

□

Example 4.1 (Value reuse among memory operations) For the problem of redundant loads, generators $\boldsymbol{D}$ are both loads and stores $(n, \boldsymbol{e})$, i.e., all memory operations that "produce" the value of the memory location. Users $\boldsymbol{U}$ are the loads, i.e., the "optimizable" operations that can reuse the value. Kills are all stores that could not be disambiguated. Also note that if a node is both a generator and a user, it cannot generate the value for itself without following a cycle in the VNG (according to the definiton, the path from generator to the user must be of non-zero length). □

4.2 Availability and anticipability (dataflow problems)

Reuse threads are identified via two dataflow problems—availability and anticipability—introduced in [MR79]. A value is available along an incoming path if the value was generated on the path and was not subsequently killed. A value is anticipated along a path when it will be used on that path before it is generated or killed. While availability is a forward problem, anticipability is a backward problem. These two properties are usually defined on the CFG. The two problems are defined on the VNG as follows.

Definition 4.2 (path-based availability of generators) Let $\boldsymbol{p} = [\textbf{start}, \boldsymbol{n}]$ be a thread and $\boldsymbol{n} = (n, \boldsymbol{e})$. The value $\boldsymbol{e}$ is *available* at n along *path*$(\boldsymbol{p})$ iff there is a kill-free suffix $\boldsymbol{r}$ of $\boldsymbol{p}$. Formally,

$$\boldsymbol{r}\langle \boldsymbol{m}, \boldsymbol{n}\rangle \subseteq \boldsymbol{p} \ \wedge \ \boldsymbol{m} \in \boldsymbol{D} \ \wedge \ \boldsymbol{r} \cap \boldsymbol{K} = \emptyset$$

□

Definition 4.3 (path-based anticipability of users) Let $\boldsymbol{p} = \langle \boldsymbol{n}, \textbf{end}\rangle$ be a thread and $\boldsymbol{n} = (n, \boldsymbol{e})$. The value $\boldsymbol{e}$ is *anticipated* at n along *path*$(\boldsymbol{p})$ iff there is a kill-free and generator-free prefix $\boldsymbol{r}$ of $\boldsymbol{p}$. Formally,

$$\boldsymbol{r}\langle \boldsymbol{n}, \boldsymbol{m}\rangle \subseteq \boldsymbol{p} \ \wedge \ \boldsymbol{m} \in \boldsymbol{U} \ \wedge \ \boldsymbol{r} \cap (\boldsymbol{D} \cup \boldsymbol{K}) = \emptyset$$

□

Lemma 4.1 A node $\boldsymbol{n} = (n, \boldsymbol{e})$ lies on a value thread iff the value $\boldsymbol{e}$ is available on n along some path $p\langle \mathit{start}, n\rangle$ *and* the value $\boldsymbol{e}$ is anticipated along some path $r\langle n, \mathit{end}\rangle$. □

4.3 Marking the value reuse (dataflow lattice)

Definitions 4.2 and 4.3 introduced the dataflow problems in a path-based manner; computing them for each path would give us a path-sensitive dataflow solution, at an exponential cost. To mark the reuse threads on the VNG at polynomial cost, we define a node-based characterization of the two problems. This characterization distinguishes whether the value is available at a node along all paths, no paths, or (strictly) some paths.

Our versions of availability and anticipability predicates are thus not Boolean, instead they can take one of three values: *Must*, *No*, and *May*, which mean that a value holds on all paths (must be available at the node), no paths (is not available), or some paths (may be available, depending on the path taken).

Definition 4.4 (node-based availability, anticipability) The availability of e at the entry of n w.r.t. the incoming paths is defined as:

$$AVAIL_{in}[(n, e)] = \begin{cases} \textit{Must} & \\ \textit{No} & \text{if } e \text{ is available along } \begin{matrix} \textit{all} \\ \textit{no} \\ \textit{some} \end{matrix} \text{ paths from } \textit{start} \text{ to } n. \\ \textit{May} & \end{cases}$$

The anticipability of e at the entry of n w.r.t. the incoming paths is defined as:

$$ANTIC_{in}[(n, e)] = \begin{cases} \textit{Must} & \\ \textit{No} & \text{if } e \text{ is anticipated along } \begin{matrix} \textit{all} \\ \textit{no} \\ \textit{some} \end{matrix} \text{ paths from } n \text{ to } \textit{end}. \\ \textit{May} & \end{cases}$$

where "some paths" means strictly some paths. □

Note that *May* is not a "don't know" answer. Instead it signifies that node n is on paths both with and without the property. The three values are sufficient for the transformation stage to recover the path-specific information.

To compute the three-values dataflow solutions, we define a lattice that gives the partial order of the *Must*, *May*, *No*, and a $\top$ element, which is used a "safe guess" when computing the solution with an iterative dataflow solver.

Definition 4.5 (Path-Sensitive Lattice) Lattice L is a tuple $L = (P, \wedge, \vee)$, where $P = \{\top, \textit{Must}, \textit{No}, \textit{May}, \}$ is a partially ordered set, $\wedge$ is the "meet" operator, and $\vee$ is the "union" operator. The $\leq$ relationship is shown below.

$$\begin{tikzcd}
& T \arrow[dl] \arrow[dr] & \\
\textit{No} \arrow[dr] & & \textit{Must} \arrow[dl] \\
& \textit{May} &
\end{tikzcd}$$

□

4.4 The equation system (transfer functions)

Forward problems The forward problem of availability is computed using a transfer function f^{AVAIL}. The equation system is given below.

$$\boxed{\begin{aligned}
AVAIL_{out}[\boldsymbol{n}] \quad &= \quad f^{AVAIL}_{\boldsymbol{n}}(AVAIL_{in}[\boldsymbol{n}]) \\
AVAIL_{in}[\boldsymbol{n}] \quad &= \quad \bigwedge_{\boldsymbol{m} \in pred(\boldsymbol{n})} AVAIL_{out}[\boldsymbol{m}] \\
f(AVAIL)_{\boldsymbol{n}}(x) \quad &=_{df} \quad \begin{cases} \textit{Must} & \text{if } \boldsymbol{n} \in \boldsymbol{D}, \\ \textit{No} & \text{if } \boldsymbol{n} \in \boldsymbol{K}, \\ x & \text{otherwise.} \end{cases}
\end{aligned}}$$

Backward problems On a *separable* VNG, backward problems are set up analogously to the forward ones (separable VNG is defined in Definition 3.5, page 37). For general VNGs, however, dataflow facts may merge not only when control-flow paths merge, but also when threads merge (which can happen at the arguments of a ϕ-node). This thread merge has a different effect than the control-flow one, which creates *May* when *Must* meets with *No*, combining paths in a path-sensitive way. On the other hand, the thread merge combines threads that share the same *sibling* path (see Definition 3.4). It is sufficient that one of the threads has the property. Therefore, when *Must* meets with *No* the result is *Must*.

The thread merge uses the "union" operator $\vee$, which is defined using the following order

$$\top > \mathit{Must} > \mathit{May} > \mathit{No}.$$

Example 4.2 (Backward Dataflow Analysis) Consider the example below. The threads for $a+b$ and $a+c$ meet on the CFG edge between S_2 and S_5 (draw this example in the graph form!). This merge influences the computation of anticipability. The statements in S_6 and S_7 are anticipated at S_2. Note that the solution of anticipability does not change if S_6 or S_7 was removed. ☐

```
$S_1$: if ( ) then
$S_2$:    c := b
$S_3$: else
$S_4$:    ...
$S_5$: end if
$S_6$: ..  := a + b
$S_7$: ..  := a + c
```

the original program

$a+b \;\to\; \boldsymbol{c_3}$
$a+c \;\to\; \boldsymbol{c_5}$

```
$S_1$: if ( ) then
$S_2$:     $\boldsymbol{c_1}$ anticipated here
$S_3$: else
$S_4$:     ...
       $\boldsymbol{c_3} := \phi(\boldsymbol{c_1}, \boldsymbol{c_2})$
       $\boldsymbol{c_5} := \phi(\boldsymbol{c_1}, \boldsymbol{c_4})$
$S_5$: end if
$S_6$: $\boldsymbol{c_3}$
$S_7$: $\boldsymbol{c_5}$
```

the Value Name Graph

Equipped with the union operator $\vee$, we can set up the equation system.

$$\begin{array}{rcl}
\mathit{ANTIC}_{in}[\boldsymbol{n}] & = & f_{\boldsymbol{n}}^{\mathit{ANTIC}}(\mathit{ANTIC}_{out}[\boldsymbol{n}]) \\
\mathit{ANTIC}_{out}[(n,\boldsymbol{e})] & = & \displaystyle\bigwedge_{m \in \mathit{succ}(n)} \; \bigvee_{(m,e') \in \mathit{succ}((n,e))} \mathit{ANTIC}_{in}[(m,\boldsymbol{e}')] \\
f_{\boldsymbol{n}}^{\mathit{ANTIC}}(x) & =_{df} & \begin{cases} \mathit{Must} & \text{if } \boldsymbol{n} \in \boldsymbol{U}, \\ \mathit{No} & \text{if } \boldsymbol{n} \in (\boldsymbol{K} \cup (\boldsymbol{D} \setminus \boldsymbol{U}), \\ x & \text{otherwise.} \end{cases}
\end{array}$$

4.5 Computing dataflow solution (the solver)

The maximal fixed-point of the equation systems can be found with an iterative dataflow solver. An effective alternative to an iterative algorithm can be a demand-driven analyzer [DGS97, SRH96].

The four lattice values are efficiently encoded using two bits. Each value $\boldsymbol{v}$ is represented as a pair $(v_{\text{all}}, v_{\text{some}})$, where the first bit is true of the property holds on all paths and the second if the property holds on some (possibly all) paths.

$\boldsymbol{v}$	v_{all}	v_{some}
$\top$	1	0
Must	1	1
No	0	0
May	0	1

Given this encoding, the “meet” operator $\wedge$ performs a logical *and* ($\cap$) on v_{all}, and *or* ($\cup$) on v_{some}.

$$\begin{aligned}\boldsymbol{v} \wedge \boldsymbol{w} &= (v_{\text{all}}, v_{\text{some}}) \wedge (w_{\text{all}}, w_{\text{some}}) \\ &= (v_{\text{all}} \cap w_{\text{all}}, v_{\text{some}} \cup w_{\text{some}}).\end{aligned}$$

Chapter 5

Estimators: High-Fidelity Profiling using Low-Cost Profiles

When an optimizing compiler has knowledge of the run-time behavior of the program (e.g., the execution frequencies of statements), it can avoid making transformation that would result in a small run-time benefit. The most common types of program profiles are *edge* profiles and *path* profile. They measure the frequencies of edges and paths, respectively.

A typical profile-directed optimizer works in two steps. First, a profiler measures the run-time behavior of the program. Second, the program transformation consults the generated profile, to *estimate* the run-time benefit of alternative transformation choices. In a path-sensitive setting, these two steps are currently incompatible: *edge* profiles are inexpensive to collect but their estimates are imprecise, path-insensitive. *Path* profiles produce more precise, path-sensitive estimates, but they are expensive to collect and non-trivial to consult and incorporate into dataflow analysis.

This chapter bridges profiling and optimization. It presents a technique that uses edge profiles, an inexpensive, path-insensitive measurement of program's control-flow behavior, and produces a profile-weighted dataflow information—an information that is a) informative: compatible with the value-flow optimizer and b) practical: inexpensive, yet precise.

Informativeness: Dataflow analysis on the Value Name Graph answers essentially Boolean questions: "given a computation at node n, does a value flow along some path to n?" When this question is answered in relative terms, using the expected execution frequency of the value flow, program transformations can perform a cheaper program optimization, as will be shown in Chapter 6. This chapter presents a set of *estimator* algorithms for computing such *frequency information* by combining Boolean dataflow information with a program profile. In this dissertation, the frequency information has two uses: it navigates the program transformation (Chapter 6) and evaluates the performance of the entire framework by quantifying the amount of value-flow it *exposed* by the static analysis (Chapter 8).

Practicality: Practicality was the primary design goal behind the PATHFINDER estimators and was was achieved through two novel contributions. First, estimators were designed to work from edge profiles, which have low profiling cost and are widely supported. Unfortunately, edge profiles are inherently imprecise, because they cannot reconstruct frequencies of individual program paths. To add confidence to profile-directed optimization, the estimators bound the inherent imprecision which surfaces in the computed frequency. The bound is achieved by assuming an optimistic and a pessimistic scenario of the profiling error.

The second contribution is that the estimators form a hierarchy of increasing precision. The hierarchy provides a practical solution to profile-weighted analysis: when a simple estimator returns bounds that are too loose, the compiler can invoke a better estimator. The better estimator will obtain bounds guaranteed to be no worse, while reusing (some) information already computed by the simpler estimator, decreasing both the compile time and the implementation effort. The tunable precision is obtained via two algorithmic principles: a) localizing program regions vulnerable to the edge-profile error, and b) reducing the error through various notions of control flow reachability.

Another contribution are the experiments with edge profiles. While our estimators cannot eliminate the inherent edge-profile error, by computing its bounds they restrict it, and also quantify the fundamental limitations of edge profiles. The experiments show that the inherent error is small (at least for load reuse). With good algorithms, the error can be considerably narrowed down: our second best estimator was able to bound the error down to 5%, a 4-fold improvement over our simplest estimator. Hence, when used properly, edge profiles seem to provide sufficient precision and a simple implementation.

This chapter starts by stating the problem, motivating an ideal estimator, and reviewing the related work. Section 5.2 intuitively introduces the principles behind our estimator algorithms. The following two sections present the estimator algorithms, first for separable VNGs (Section 5.4), and then for general VNGs (Section 5.5). Section 5.6 proves correctness of the algorithms and Section 5.7 empirically evaluates the precision of our estimators. Edge-profile precision is limited; for absolute precision, a more detailed profile is needed. Section 5.8 sketches a novel profiling algorithm that is fully path-sensitive, yet does not require profiling complete program paths. Section 5.9 concludes by summarizing our results and providing recommendations for deployment of estimators in a production compiler.

5.1 Motivation and related work

Before describing the estimator algorithms, this section states the problem of estimating, from a profile, the run-time benefit of an optimization. This section also highlights the inherent issues and describe how they are tackled by existing techniques.

5.1.1 The problem statement

Recall that, on the VNG, value reuse is manifested as a *reuse thread*, a kill-free VNG path connecting a generator and a user of the same value. Each execution of a reuse thread corresponds to exactly one reuse opportunity. Note that each program *path* may execute multiple reuse *threads* running along this paths, giving rise to multiple opportunities (see Definition 3.4 on page 36).

Informally, the *estimation problem* is to determine how many reuse threads are executed by a given program input. Notice that, for the moment, we are not restricting ourselves to a specific profile format to measure the behavior of the program on the given input. In fact, selecting (or designing) the profile is an inherent part of solving the estimation problem. To this end, Section 5.1.4 evaluates existing profile formats, justifying our selection of edge profiles. On the other hand, Section 5.8 *designs* a new profile format.

Figure 5.1 introduces the running example. The estimation problem is illustrated on a program fragment in the VNG form (see Chapter 3), in which the three loads have been detected to refer always to the same memory location, denoted with a symbolic name $[\boldsymbol{x}]$. Node A, C, E are both generators and users of the value. The nodes B and D kill the value, as they may write to $[\boldsymbol{x}]$, according to some alias analysis. Whenever the program executes two loads of $\boldsymbol{x}$ without intercepting the killing store, the later load is redundant and can be eliminated.

The estimation problem is to determine how many times the program input executes a reuse thread, which in the figure are $[A, f, C]$, $[C, j, E]$, $[A, f, h, i, j, E]$, and $[A, f, h, i, [k, l]^+, E]$, where '$^+$' denotes the

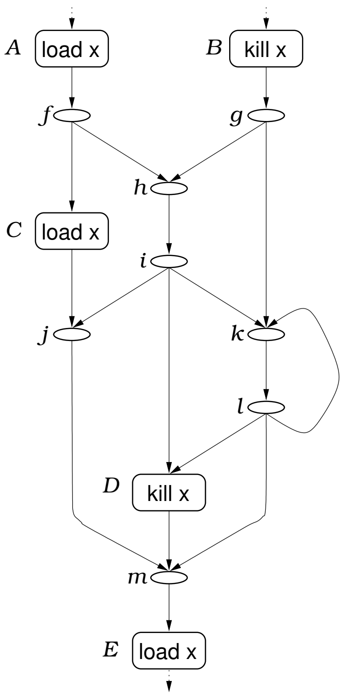


Figure 5.1: **The estimation problem statement:** What is the amount of reuse among the loads of $[\boldsymbol{x}]$? Constituent sub-problems: what kind of program profile should be collected at run-time? How to combine the collected profile with the static analysis?

usual non-zero repetition. Each time any of these threads is taken, exactly one load of x can be removed. Note that because the VNG is separable and has only one symbolic name ($[\boldsymbol{x}]$), threads can be referred to as paths.

5.1.2 The applications of estimators.

The algorithm that solves the estimation problem is called an *estimator*. In this dissertation, estimators have two-fold application:

Profile-directed transformation. Chapter 6 develops a PRE transformation algorithm that overcomes the limitations of traditional PRE based on code motion via a) CFG *restructuring* (called PRE(MR): expensive but achieving optimal removal of redundancies) or b) control *speculation* (called PRE(MS): inexpensive but sub-optimal). To decide between the two options, their expected run-time benefit must be known.

Because restructuring removes all redundancies, the run-time benefit of restructuring equals the amount of reuse in the program, which is exactly the precise estimate bounded by the estimator. For speculation, it turns out that, for each estimator, one can determine how to carry out the speculation transformation in a way that its benefit equals exactly the lower bound of the estimate. Thus, by comparing the lower and the upper bounds, estimators identify when speculation is too sub-optimal, facilitating trade-offs between restructuring and speculation.

Evaluation of the VNG. Chapter 8 uses the estimators to measure the accuracy of the VNG at finding value recomputation. By incorporating the profile into dataflow analysis, an estimate computes, in dynamic terms, the amount of reuse *exposed* by the representation and subsequently *collected* by the dataflow analysis (see Section 2.1). Chapter 8 compares such an estimate with an (ideal) amount of value flow *present* in the program, obtained using *reuse profiling*, done on the same input. The algorithm for reuse profiling is also developed in Chapter 8. Comparing the exposed/collected amount with the ideal amount shows what fraction of the simulator-detected reuse was found by the static analysis, and therefore indicates the precision of the framework.

5.1.3 An ideal estimator.

The process of designing the estimator involves designing or selecting a profile format and developing an algorithm that combines the profile with the result of dataflow analysis. An ideal estimator algorithm should have the following properties.

- *Accuracy:* ideally, the dynamic optimization opportunities counted by the estimator should equal the opportunities that occurred in the given execution of the program, unaffected by profile imperfections and estimator's approximations.
- *Dataflow independence:* the profile should collect measurements that are independent from what dataflow facts will be counted by the estimator (reuse of loads, or constants). Dataflow independence enables reusing the profile for other optimizations, and eliminates the need to perform the profiling step (repeatedly) between the analysis and estimation stages.
- *Low-cost profiling:* the estimator should use a profile format that requires simple program instrumentation and incurs low execution overhead during profiling.
- *Low-cost profile processing:* the information needed by the estimator should be simple and inexpensive to extract from the profile.
- *Sharability:* rather than requiring specialized algorithms, the estimator should rely on algorithms that are commonly implemented in optimizing compilers, to reduce implementation effort. Also, the profile information should be general enough to be shared by other parts of the compiler.

5.1.4 Program profiles.

The first step in developing an estimator is to select (or design) a run-time program profile. Let us review next the most commonly used profile formats, describe the information they collect and highlight their properties.

The use of profiles and profile types Program profiling was originally developed to enhance *control-flow* optimizations. Such optimizations use the profile to learn about control-flow behavior of the program, i.e., about the frequency with which its procedures or statements are executed. Control-flow profiles are used for i) performing compile-time branch prediction, ii) identifying frequently executed (*hot*) program regions, with the goal of enlarging the scope of traditional optimizations, via inlining or trace scheduling, or iii) focusing the effort of static analysis to hot program regions [AdJPS98, AL98, BGS98b].

In contrast to control-flow optimizations, estimators focus on the *value-flow* behavior of the program. Instead of determining the frequency of control-flow paths, they compute the frequency of value-flow

profile type	information collected	profiling cost overhead	profile size	accuracy	processing
edge profile	CFG edges	1-3%	$\Theta(E)$	low	low
path profile	acyclic CFG paths	30%	not reported	medium	medium
whole-program path profile	control trace	$> 100\%$	1–10MB	full	$--$

Table 5.1: **Control-flow profiles.**

paths (called threads on the VNG representation), which are control-flow paths along which reuse exists. What these paths are is determined by the (static) value-flow analysis.

Accordingly, this section divides profiles into control-flow profiles and value-flow profiles. Control-flow profiles received much more attention. There are three profile formats relevant to solving the estimation problem: *edge* profiles, *path* profiles, and *whole-program path* profiles. Value-flow profiles observe values computed by the program. The most common kinds include *address profiling* and *value* profiling. The former are used to assist in static instruction scheduling, the latter in compiler-supported value-prediction [RCT$^+$98]. Because value-flow profiles are more expensive than control-flow profiles, and because they are not independent of the dataflow information to be estimated, we made a deliberate decision to base our estimators upon a control-flow profiles.

Next, let us discuss the control-flow profiles, through the optimizations for which they were developed. The various profile types were developed because, depending on the optimization, a different program region is profiled. Profile-based static branch prediction attempts to predict the direction of a branch from previous program runs [MH86, FF92]. Such optimization requires the counts of nodes or edges in the control flow graph. The profile capturing such information is called an *edge profile*. Edge profile is also used by procedure inlining [CMCH92, ASG97], which requires execution frequencies of each call site.

More sophisticated static branch predictors measure the correlation among multiple branches, requiring profiling of control flow paths [YS94]. Such *path profiles* are also useful in forming an instruction scheduling region, although most schedulers approximate the information with edge profiles [LFK$^+$93, HMC$^+$92, MLC$^+$92].

The above path profiles collect statistics on short [YS94] or acyclic [BL96a] CFG paths. Recently, Larus [Lar99] developed *whole-program path* profiling that captures the entire control flow trace of the program. His profiling algorithms compresses the trace by forming a grammar whose (only) string is the executed trace.

Profiling techniques. The control-flow profiles are summarized in Table 5.1. Next, let us briefly review the algorithms for collecting the control-flow profiles. The most common technique used is instrumenting the program with instructions that increment counters. The algorithms differ in how these counters are incremented and maintained. For edge profiling, the most efficient technique is *sampling* the execution of the program, which is a more efficient but less precise technique than instrumentation. Of the three profiles, sampling was used only for edge profiling.

Edge profiles. An edge profile with about 16% overhead [BL94]. With a hardware-based sampling approach, edge profiles cost only 1–3% overhead [ABD$^+$97]. Recently, a software-based sampling approach was developed, via transient (removable) instrumentation [TS99], or transient interpretation of native instructions [BDB99]. Their cost is comparable to that of the hardware-based approach (a few per cent)..

Path profiles. Path profiles can be collected relatively efficiently, even when compared to the low cost of edge profiling. A path profile can be obtained with about 30% overhead [BL96a].

Whole-program path profiles. Larus's compression technique is the best complete profiling technique. It achieves low memory consumption by compressing the trace (on-line) into a grammar, on which path frequencies can be determined without decompressing the trace. Still, compared to edge and path profiles, the cost of collecting whole-program paths is relatively high (a small integer slowdown). The compressed profile size is also considerable (24MB for 126.gcc).

5.1.5 Related work: existing estimators.

Let us review next the existing estimators. The related work is divided based on the kind of profile information used. A common property all of them is that they are based on control-flow profiles. Therefore, they are dataflow independent: they profile the program before performing any value-flow analysis and estimation.

Edge profiles Ramalingam [Ram96] offers the only existing systematic method for estimating a dataflow solution. Using edge profiles, his *frequency analysis* derives the probability of a data-flow fact holding at a CFG node. The probability replaces the less informative Boolean data-flow lattice. The frequency analysis can be directly applied on the VNG representation: after value flow has been converted into data flow, it is represented in a domain that that frequency analysis can handle.

- *Accuracy.* Unlike our estimators, frequency analysis does not bound the inherent edge-profile error; without quantifying the error, it is not clear how close the frequency analysis is to the actual estimate.
- *Sharability.* While frequency analysis requires an elimination-style dataflow analysis solver (not commonly implemented in existing compilers), our estimators rely on control flow reachability or network flow algorithms, which are easier to implement.
- *Processing cost.* Operating on real numbers, rather than on bit-vectors, frequency analysis are expected to be rather slow. Our estimators delay the floating-point computation until the value-flow patterns are abstracted and summarized. Our estimators offer an alternative to frequency dataflow analysis; they are not as general (not all our estimators can compute the per-node estimate) but are cheaper and fit the need of PRE transformations.

Path profiles Gupta et al [GBF97a, GBF97b, GBF98] present various versions of profile-directed PRE, for exploiting various hardware features. Common to all these algorithms, all the algorithms are navigated using Ball-Larus path profiles [BL96a, BMS98]. While the algorithms do not explicitly compute an estimate, the estimate is computed implicitly, "on-the-fly" during the dataflow analysis.

- *Accuracy.* Unfortunately, even path profiles remedy the correlation problem only partially, as they measure execution frequencies of only acyclic program paths. As a result, estimating value flow along cyclic paths incurs the same branch correlation error, as the measured paths may not fully overlap with the detected reuse paths, thus capturing only part of the correlation needed to reconstruct the frequency of the value-flow path.
- *Processing cost.* the algorithms maintain dataflow information independently for each executed path in the program, hence the slowdown factor due to making the optimization profile-directed equals roughly the number of acyclic paths with non-zero execution frequency. Experiments in [GBF98] show that

1.4% of procedures in a subset of Spec95 executed more than 100 paths, and that 35% of procedures executed at least 5 paths.

- *Sharing.* the algorithms are phrased mostly as bit-vector problems and do not require special dataflow solvers. Collected path profiles can be reused in the instruction scheduler and procedure inliner.

Whole-program path profiles The Larus whole-program path profiler [Lar99] can be used as an estimator. He provides an algorithm for determining the frequency of a subpath, from the compressed profile.

- *Accuracy.* whole path profiles completely eliminate the branch correlation error, they represent an ideal profile for value-flow optimizations.
- *Processing cost.* Unfortunately, there are currently no algorithms for determining the frequency of an set of sub-paths, represented as a regular expression, like the paths $[A, f, h, i, [k, l]^+, E]$ in Figure 5.1. Such cyclic paths commonly carry value reuse and must be estimated by the optimizer.

A hope for an ideal estimator? To summarize the above discussion of existing work, no practical, accurate estimator exists. Two accurate estimator approaches can be immediately suggested, but they are both impractical. The first is to is to follow the dataflow analysis with a *complete* redundancy elimination (Chapter 6), which is followed by re-profiling, to obtain the optimized dynamic computation count. Because the complete redundancy removal is based on restructuring the CFG, such an approach is impractical and—due to the code growth incurred during restructuring—potentially infeasible. The second approach is to enumerate all value-reuse paths detected by value-flow analysis and look up their frequencies in the trace produced by the whole-program path profile. Because dataflow analysis may detects infinitely many reuse threads (due to loops), we need to (somehow) stop their enumeration when it is certain that the further threads will not be contained in the thread.

5.2 The hierarchy of estimators

This section outlines our estimators and defines concepts common to all our estimator algorithms. We describe the properties of our estimators and show how they achieve (nearly) all the goals of the ideal estimator.

The estimators follow three main design choices, each necessitating the following one.

1. *Use edge profiles.* For pragmatic reasons, our estimators are based on edge profiles. Because edge profiles are inexpensive to collect and store, we achieve *low-profiling cost*. Because they measure only control-flow behavior of the program, we achieve *dataflow independence* of profiling. Because edge profiles are widely used, e.g., for procedure inlining and instruction scheduling, we can justify (and amortize) the profiling cost, achieving *sharability*.
2. *Compute error bounds.* Unfortunately, edge profiles contain an inherent profiling error. Because they do not capture branch correlation, they cannot reconstruct path frequencies accurately to the actual execution. Consider the running example annotated with an edge profile, shown in Figure 5.2. One of the paths with value-reuse is $[A, f, h, i, j, E]$. The edge profile cannot precisely determine its frequency; according to the edge profile, the path frequency can range from 30 to 40, which is a large profiling error.

While edge profiles prevent our estimators from achieving true *accuracy*, the estimators presented here gain confidence in edge-profiles by computing not a single (imprecise) estimate, but instead a lower and an upper bound of the accurate estimate, by assuming a pessimistic and an optimistic control flow scenario. Because the bounds limit the inherent error, the accurate estimate lies somewhere between them. The tighter the bounds are to each other, the more precise estimate we obtain.

3. *Hierarchy of estimators.* The amount of the profiling error that appears in the resulting estimate depends on:

   (a) *The inherent edge profile error.* Even though the edge profile is imprecise, when the control-flow behavior of is skewed highly towards a small number of paths, the edge profile may describe these dominant paths very precisely. Consider an extreme example, in which each CFG node is executed at most once. In such a program, the edge profile is as precise as the complete trace.

   (b) *The control-flow complexity of value-flow threads.* Value-flow threads do not always require more than the edge profile. Consider node D in Figure 5.2. If D did not kill the reuse of $[\boldsymbol{x}]$, there would be more value-flow paths, but their overall pattern would be simpler to quantify using the edge profile: the load in E would be redundant always except when it was preceded by B, so the estimate for the load is its frequency (150) less the frequency of B (50).

   Because the amount of error induced in the estimate varies due to these two factors, the full power of our algorithm may not always be needed. Therefore, estimators provide a scalable solution: while one cannot influence the factor (a), one can focus on (b) and tighten—with a scalable effort—the estimator bounds in those places of the program where the threads have a complex control-flow pattern.

   The scalability is achieved via five estimator algorithms that differ in their error-bounding precision and run-time complexity, as shown in Figure 5.3. The practical reason for developing a hierarchy of increasingly better estimators is that when a simpler (and faster) estimator yields loose bounds, one can run the next better (but slower) estimator, with the guarantee that the new bounds will not be worse. Resorting to a stronger algorithm results in *low processing cost*. Furthermore, our estimators share a common paradigm, and rely on information or algorithms used also by other PathFinder stages, thus providing *sharability*.

Having established our design choices (computing *error bounds* of *edge profiles* in *hierarchical* manner), we can formulate the *edge-profile* estimation problem. From now on, the term profile refers to the edge profile. Let $t[\mathit{start}, \mathit{end}]$ be a control trace (i.e., the sequence of executed nodes) and $\mathit{occurs}(p, t)$ the number of times a path p occurs in the trace t, that is, the execution frequency of p in the execution t. Similarly, we define $\mathit{occurs}(\boldsymbol{p}, t)$, for a value-flow thread $\boldsymbol{p}$: $\mathit{occurs}(\boldsymbol{p}, t) = \mathit{occurs}(\mathit{path}(\boldsymbol{p}), t)$.

Definition 5.1 (Edge Profile) Given a CFG $G = (N, E, \mathit{start}, \mathit{end})$ and a profile $\mathit{freq} : E \to \mathbb{Z}$, a trace t *induces* the profile *freq*, denoted t/freq, iff for each $e = (n, m) \in E$ such that $\mathit{freq}(e) = \mathit{occurs}([n, m], t)$. Conversely, a trace t is *permitted* by a profile *freq* if t induces *freq*. ☐

Recall the Definition 4.1 of value reuse (page 43), which are VNG sub-threads, originating at a *generator* VNG node $\boldsymbol{n} \in \boldsymbol{D}$, sinking onto a *user* VNG node $\boldsymbol{m} \in \boldsymbol{U}$, without crossing a *kill* VNG node $\boldsymbol{k} \in \boldsymbol{K}$.

The trace determines the dynamic amount of value-flow in the program. It precisely determines the estimate, i.e., how often a user node is executed such that it is preceded by a generator node without crossing a kill node. The estimation problem is then to bound the estimate. The bounds are computed by considering a pessimistic (and an optimistic) control flow scenario permitted by the profile. Namely, the goal is to find the smallest (and the largest) amount of value flow, among all possible traces permitted by the edge profile.

Definition 5.2 (The Estimation Problem) The *estimation problem* S is a tuple $S = (\boldsymbol{G}, \boldsymbol{R}, \mathit{freq})$, where

- *freq* is an edge profile on a control flow graph $G = (N, E, \mathit{start}, \mathit{end})$, $\mathit{freq} : E \to \mathbb{Z}$,
- $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{E}, \mathbf{start}, vngend)$ is a value name graph on G.
- $\boldsymbol{R} = (\boldsymbol{G}, \boldsymbol{D}, \boldsymbol{U}, \boldsymbol{K})$ be the set of reuse thread (see Definition 4.1)

$$\boldsymbol{R} = \{\boldsymbol{p}(\boldsymbol{n}, \boldsymbol{m}] \mid \boldsymbol{n} \in \boldsymbol{D}, \boldsymbol{m} \in \boldsymbol{U}, |\boldsymbol{p}| > \boldsymbol{0}, \boldsymbol{p} \cap \boldsymbol{U} = \emptyset, \boldsymbol{p} \cap \boldsymbol{K} = \emptyset\}.$$

The *estimation problem* $S = (\boldsymbol{G}, \boldsymbol{R}, \mathit{freq})$ is to compute the lower and upper bounds L and U on the amount of reuse permitted by the edge profile *freq*:

$$\begin{aligned} L &= \min_{t/\mathit{freq}} \sum_{\boldsymbol{p} \in R} \mathit{occurs}(\boldsymbol{p}, t) \\ U &= \max_{t/\mathit{freq}} \sum_{\boldsymbol{p} \in R} \mathit{occurs}(\boldsymbol{p}, t). \end{aligned}$$

□

Example 5.1 Figure 5.2 illustrates the estimation problem $S = (\boldsymbol{G}, \boldsymbol{R}, \mathit{freq})$. Shown is the VNG $\boldsymbol{G}$ from Figure 5.1, annotated with edge profile *freq*. The reuse threads $\boldsymbol{R}$ corespond to the optimization opportunities for the removal of redundant load operations. The value flow to be estimated is specified using the generator (loads/stores), user (loads), and kill (stores) sets:

$$\begin{aligned} \boldsymbol{D} &= \{A, C, E\} \\ \boldsymbol{U} &= \{A, C, E\} \\ \boldsymbol{K} &= \{D\} \end{aligned}$$

The problem is to compute the minimum and the maximum number of reuse opportunities on the three loads of $\boldsymbol{x}$, as permitted by the annotated profile? □

According to Definition 5.2, the bounds L and U are tight; i.e., the relative error $(U - L)/L$ can be entirely attributed to the imperfections of edge profile (although, the error size depends also on the shape of value-flow threads, as explained in bullet 3b on page 55). The best estimator is tight only on a separable VNG with a single symbolic value, like the VNG in Figure 5.2. On a general VNG, the estimation problem may be NP-hard, and our estimators compute its approximation. While the estimators do not guarantee a competitive ratio for their approximations, their precision can be measured using the relative error $(U-L)/L$. (In practice, the relative error of the second best estimator was about 5%).

5.3 Overview of estimators.

Computing the estimate as prescribed by Definition 5.2 would involve iterating across a) all traces permitted by the profile, and b) all possible reuse threads from $\boldsymbol{R}$. Such a direct approach is impractical, as

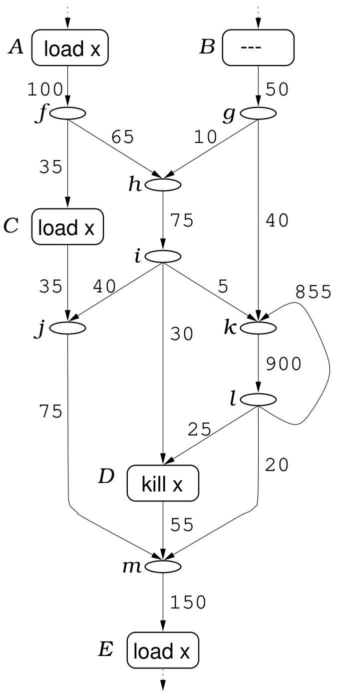


Figure 5.2: **The running example annotated with edge profile.**

there may be too (although finitely) many permitted traces and infinitely many reuse threads, such as those denoted by $[A, f, h, i, [k, l]^+, E]$ in Figure 5.2.

Rather than dealing with individual traces and threads, our estimators find *summary program points* that "summarize" groups of paths with identical value-flow properties. The properties are: a) value is generated along incoming paths, b) value is generated along no incoming paths, and c) value can be reused along all outgoing paths.

Definition 5.3 (Summary Points) *Summary points* are subsets of VNG edges, defined as follows:

1. *Producers.* For each $\boldsymbol{e} \in Pr$, the reuse is generated along all incoming control flow paths by a generator from $\boldsymbol{D}$.
2. *Stealers.* For each $\boldsymbol{e} \in St$ the value from $\boldsymbol{D}$ is available along no incoming control flow paths. □
3. *Consumers.* For each $\boldsymbol{e} \in Co$, the value will be used by exactly one user from $\boldsymbol{U}$ along each outgoing control flow path.

To compute the actual value of the upper bound, our estimators determine how much reuse can flow between generators and the set of consumers points. To arrive at the lower bound, they determine how much (reuse-free) flow can reach from stealers to the consumers, *stealing* [BMS98] the reuse flowing from the sources.

Note the contrast between generators/users/kills (Definition 4.1) and producers/consumers/stealers (Definition 5.3). While the first triple *states* the problem by specifying the value flow patterns in the VNG, the latter triple *represents* the problem statement to enable an efficient and accurate computation of estimate bounds. Producers summarize generators, consumers summarize users, and stealers summarize kills and other reuse-free path.

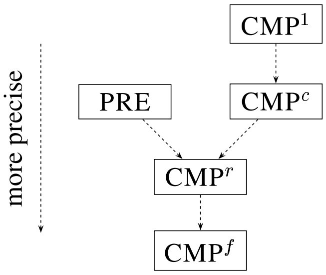


Figure 5.3: **The estimators and their precision ordering.**

An overview of estimators. The estimators differ in how they compute these three sets and how precisely they account for the possible value flow among them. That is, how much error (in addition to the inherent error) they allow when constructing their pessimistic/optimistic reuse scenarios. Next, we present a brief overview of the individual estimators, starting from the conceptually simplest one. The estimator hierarchy is given in Figure 5.3; the estimator names will be explained below.

PRE is the simplest estimator. Mirroring closely the Partial Redundancy Elimination transformation (see Section 6.2), producers are taken to be those instructions that generate the reuse, that is, the instructions from $\boldsymbol{D}$; consumers are the partially redundant computations, that is, the user nodes from $\boldsymbol{U}$; and stealers are the points where an operation must be inserted by PRE to compensate partial redundancy. To determine which generators (or stealers) may produce (or steal) reuse for each consumer computation, PRE uses graph reachability.

The PRE estimator uses a trivial placement of summary points. A smarter placement can improve the bounds by implementing an observation that some reuse threads can be estimated precisely even from the edge profile. For example, if a producer is post-dominated (followed on each control flow path) by a consumer (with no intervening kill), then all reuse from the producer must reach the consumer, and no reuse can be stolen. Consequently, the producer frequency represents a precise estimate for all threads that connect the producer and the consumer, considering them all in concert. For such threads, *any* edge profile is precise; they can be safely excluded from worst-case control flow scenarios.

Thus, the central idea behind the remaining estimators is to isolate the reuse threads for which *any* edge profile is precise and focus on reducing the error in error containing threads. This (effective) factoring of threads is accomplished by placing the summary points as close to each other as possible: starting from the PRE's position of summary points, producers and stealers are delayed (i.e., moved forward) and the consumers are hoisted (i.e., moved backward). When these points meet, then all threads represented by them can be precisely summarized by the frequency of the meeting edge. Conversely, when these points do not meet, the error remains, but we at least minimized the lengths of considered threads, minimizing the number of branches on these threads, and thus reducing the branch-correlation error induced in the solution.

It is interesting to note that the closest placement of summary nodes coincides with the *CMP region*,[1] described in detail in Chapter 6, where it is used to identify obstacles to the complete PRE transformation. The CMP region is the smallest multiple-entry, multiple-exit region in which the entry edges can be

---

[1] *CMP* region stands for *Code-Motion-Preventing* region. By using the term *CMP* before it is introduced later in Chapter 6, we are getting ahead of ourselves. Chapter 6 must follow this chapter because it relies on the estimators presented here to develop its transformation algorithms. The name CMP reflects the transformation and not the estimation for historical reasons: the transformation algorithms were developed earlier [BGS98b] than the estimators [BGS98c].

divided between producers and stealers, and the exits between consumers and (strict) non-consumers. Being the smallest such region, it finds the desired closest placement of summary points.

The CMP region precisely divides (the reuse in) the VNG into uncertain and definite. Each node in the CMP has an error, as the reuse is produced only along *some* incoming paths *and* can be consumed only along *some* outgoing paths. Consequently, without the knowledge of branch correlation in the CMP, it is not possible to determine how much incoming reuse actually flowed to consumers in the profiled program execution. Conversely, each node outside the CMP region is error-free, as the value is *either* produced (or stolen) along *all* incoming paths *or* is consumed along *all* outgoing paths; in either case, edge profile is sufficient.

To summarize, the CMP region contains all the branch-correlation error inherent in the edge profile. Therefore, the four CMP estimators focus on reducing the error contained in the CMP region, with different variations on the optimistic/pessimistic approach, as follows:

CMP^1 estimator conservatively assumes that there is a single CMP (hence the **1** in the name), in which all entries and exits are mutually reachable. This false reachability may connect consumers to spurious producers and stealers, yielding loose bounds.

CMP^c attacks false reachability by partitioning the CMP region into **c**onnected CMP subregions, using graph reachability between CMP entries and exits. The individual connected CMPs are treated with the CMP^1 estimator.

CMP^r exploits entry-exit **r**eachability further. Compared to CMP^c, it removes false reachability even within each connected CMP by computing the amount of value flow as a network flow problem.

CMP^f exposes to the network **f**low computation of all the CFG edges in the CMP, not just the summary entry-exit reachability information, thus exploiting a refined notion of reachability that accounts for how much reuse can flow between CMP entries and exits, and not just whether they are reachable.

The following two sections present the detailed estimator algorithms, Section 5.4 for separable VNGs, to simplify the presentation, and Section 5.5 for general VNGs. Finally, Section 5.6 proves correctness and the precision ordering of the estimators.

5.4 Estimators for separable VNGs

This section assumes a *separable* VNG, i.e., a VNG without ϕ-nodes, and hence without any dataflow transfers between value names. Without transfers, each name can be handled separately as a “slice” of the VNG, with CFG properties. To handle general VNGs, only minor algorithmic extensions are needed, but we delay them to the next section, once the basic principles are clear.

Recall the following notation introduced in the previous section. Given an estimation problem $S = (\boldsymbol{G}, \boldsymbol{R}, \mathit{freq})$, an estimator algorithm $e \in \{\mathrm{PRE}, \mathrm{CMP}^1, \mathrm{CMP}^c, \mathrm{CMP}^r, \mathrm{CMP}^f\}$ returns upper and lower bounds on the accurate estimate, denoted U^e and L^e, respectively. The problem S specifies the value flow using three sets of VNG nodes: generators $\boldsymbol{D}$, users $\boldsymbol{U}$, and kills $\boldsymbol{K}$; the estimate measures the frequency of $\boldsymbol{K}$-free threads between $\boldsymbol{D}$ and $\boldsymbol{U}$. All estimators summarize value flow by finding a placement of summary points: producers Pr, stealers St, and consumers Co. Also, we overload the edge profile by extending its domain to VNG nodes and edges in a straightforward way, $\mathit{freq} : (E \cup N \cup \boldsymbol{N} \cup \boldsymbol{E}) \rightarrow \mathbb{Z}$.

Recall the Definition 4.2 of availability and anticipability, two dataflow properties that describe whether a value flow was computed an incoming path, or can be reused on an outgoing path, respectively. Using the lattice defined in Definition 4.5, the properties have the following meaning.

$$\mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] = \begin{cases} \textsf{\textit{Must}} & \qquad\qquad\qquad\qquad \mathit{all} \\ \textsf{\textit{No}} & \text{if } x \text{ is available along} \quad \mathit{no} \quad \text{paths.} \\ \textsf{\textit{May}} & \qquad\qquad\qquad\qquad \mathit{some} \end{cases}$$

Anticipability (*ANTIC*) is defined analogously.

The placement of summary points will be computed from the dataflow solution of availability and anticipability (see Definition 4.4 on page 45). The following lemma shows that, whatever the particular selection of producers edges, each producer must be *Must*-available. Similar relationships hold for consumers and stealers.

Lemma 5.1

$$(\boldsymbol{n}, \boldsymbol{m}) \in Pr \Rightarrow \mathit{AVAIL}_{\mathit{out}}[n] = \textsf{\textit{Must}}$$
$$(\boldsymbol{n}, \boldsymbol{m}) \in Co \Rightarrow \mathit{ANTIC}_{\mathit{in}}[m] = \textsf{\textit{Must}}$$
$$(\boldsymbol{n}, \boldsymbol{m}) \in St \Rightarrow \mathit{AVAIL}_{\mathit{out}}[n] = \textsf{\textit{No}}$$

Proof. According to Definition 5.3, each producer generates the value each time it is executed. Therefore, the value must be generated on each path leading to the producer. Hence, its availability solution is *Must*. Similar arguments apply for consumers and stealers. □

The PRE estimator. The PRE estimator calculates the estimate independently for each consumer point; given the individual estimates, the total estimate is obtained as their sum. The PRE estimator mirrors the PRE transformation (see Chapter 6):

- consumers Co coincide with the optimizable (redundant) statements, i.e., they are the user computations, $Co = \boldsymbol{U}$.
- producers Pr are the sources of redundancy, $Pr = \boldsymbol{D}$. Computed independently for each consumer point $\boldsymbol{c} \in Pr$, $Pr(\boldsymbol{c}) \subseteq Pr$ are those generators whose value may reach $\boldsymbol{c}$ without crossing a kill, i.e., those generators that are backwards reachable from $\boldsymbol{c}$ along some (kill-free) thread.
- stealers $St(c)$ are those VNG edges onto which a computation must be inserted to make (the partially redundant) $\boldsymbol{c}$ fully redundant. Stealers are also computed using graph reachability.

To compute the upper bound for a consumer $\boldsymbol{c}$, we assume the most optimistic control flow scenario: all produced value that can reach $\boldsymbol{c}$ actually flows to $\boldsymbol{c}$. In such a scenario, the frequency of reuse threads between $Pr(\boldsymbol{c})$ and c equals the lower of $Pr(\boldsymbol{c})$'s and $\boldsymbol{c}$'s frequencies. The lower bound assumes the worst case: all flow from reachable stealers flows to $\boldsymbol{c}$, minimizing the frequency with which $\boldsymbol{c}$ executes with a value flowing from producers, stealing value flowing from producers. The formulae for computing the PRE estimate is shown in Figure 5.4. Note that the $\max$ operator in $L^{\text{PRE}}(\boldsymbol{c})$ serves to make the lower bound non-negative.

$$
\begin{array}{rcl}
Pr & := & \{(\boldsymbol{n},\boldsymbol{m}) \in \boldsymbol{E} \mid \boldsymbol{n} \in \boldsymbol{D}\} \\
Co & := & \{(\boldsymbol{n},\boldsymbol{m}) \in \boldsymbol{E} \mid \boldsymbol{m} \in \boldsymbol{U}\} \\
St & := & \{(\boldsymbol{n},\boldsymbol{m}) \in \boldsymbol{E} \mid \mathit{AVAIL}[\boldsymbol{n}] = \textsf{No} \ \wedge \ \mathit{AVAIL}[\boldsymbol{m}] = \textsf{May}\} \\
Pr(\boldsymbol{c}) & := & \{\boldsymbol{d} \in Pr \mid \exists \boldsymbol{p}(\boldsymbol{d},\boldsymbol{c}] \,.\, \boldsymbol{p} \cap (\boldsymbol{D} \cup \boldsymbol{K} \cup St) = \emptyset\} \\
St(\boldsymbol{c}) & := & \{\boldsymbol{s} \in St \mid \exists \boldsymbol{p}(\boldsymbol{i},\boldsymbol{c}] \,.\, \boldsymbol{p} \cap (\boldsymbol{D} \cup \boldsymbol{K} \cup St) = \emptyset\} \\
L^{\mathrm{PRE}}(\boldsymbol{c}) & := & \max\{0, \mathit{freq}(\boldsymbol{c}) - \displaystyle\sum_{s \in St(c)} \mathit{freq}(\boldsymbol{s})\} \\
U^{\mathrm{PRE}}(\boldsymbol{c}) & := & \min\{\mathit{freq}(\boldsymbol{c}), \displaystyle\sum_{r \in Pr(c)} \mathit{freq}(\boldsymbol{r})\} \\
L^{\mathrm{PRE}} & := & \displaystyle\sum_{c \in Co} L^{\mathrm{PRE}}(\boldsymbol{c}) \\
U^{\mathrm{PRE}} & := & \displaystyle\sum_{c \in Co} U^{\mathrm{PRE}}(\boldsymbol{c})
\end{array}
$$

Figure 5.4: **The PRE estimator.**

Example 5.2 (The PRE estimator) Let us apply the PRE estimator on the VNG in Figure 5.5. (See Example 5.1 for the definition of the problem.) The producers, consumers, and stealers for the PRE estimator are

$$
\begin{aligned}
Pr &= \{(A,f),(C,j),(E,.)\} \\
Co &= \{(.,A),(f,C),(m,E)\} \\
St &= \{(g,h),(g,k),(D,m))\}
\end{aligned}
$$

The bounds for consumers A and C are trivial, as A is not redundant (has no producers) and C is fully redundant (has no stealers):

$$
\begin{aligned}
L^{\mathrm{PRE}}(A) &= U^{\mathrm{PRE}}(A) = 0 \\
L^{\mathrm{PRE}}(C) &= U^{\mathrm{PRE}}(C) = 35.
\end{aligned}
$$

The edge-profile error affects only the (partially redundant) consumer E, whose producers and stealers are

$$
\begin{aligned}
Pr(E) &= \{(A,f),(C,j)\}, \\
St(E) &= \{(g,h),(g,k),(D,m)\},
\end{aligned}
$$

yielding bounds for the consumer E

$$
\begin{aligned}
L^{\mathrm{PRE}}(E) &= \max\{0, \mathit{freq}(\boldsymbol{c}) - \sum_{s \in St(c)} \mathit{freq}(\boldsymbol{s})\} &&= \max\{0, 150 - (10,40,55)\} &&= 45 \\
U^{\mathrm{PRE}}(E) &= \min\{\mathit{freq}(\boldsymbol{c}), \sum_{r \in Pr(c)} \mathit{freq}(\boldsymbol{r})\} &&= \min\{150, 100 + 35\} &&= 135.
\end{aligned}
$$

The total estimate is

$$
\begin{aligned}
L^{\mathrm{PRE}} &= \sum_{c \in \{A,C,E\}} L^{\mathrm{PRE}}(\boldsymbol{c}) = 0 + 35 + 45 \ \ = 80 \\
U^{\mathrm{PRE}} &= \sum_{c \in \{A,C,E\}} U^{\mathrm{PRE}}(\boldsymbol{c}) = 0 + 35 + 135 = 170,
\end{aligned}
$$

which is a $(170 - 80)/80 = 112.5\%$ error. $\square$

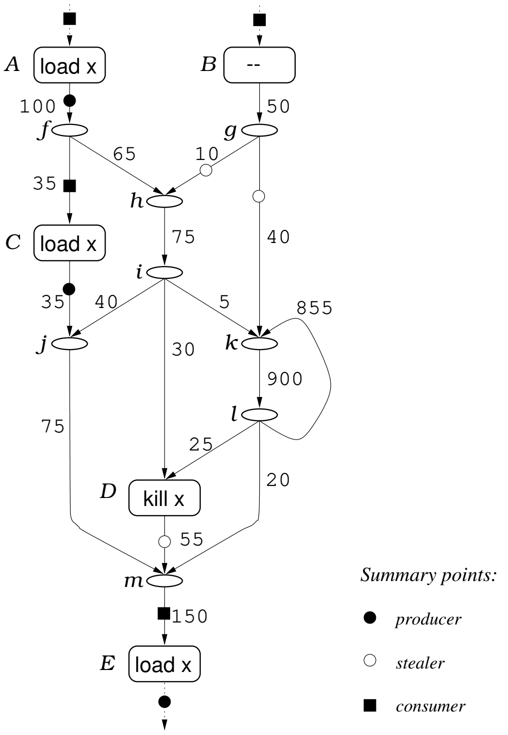
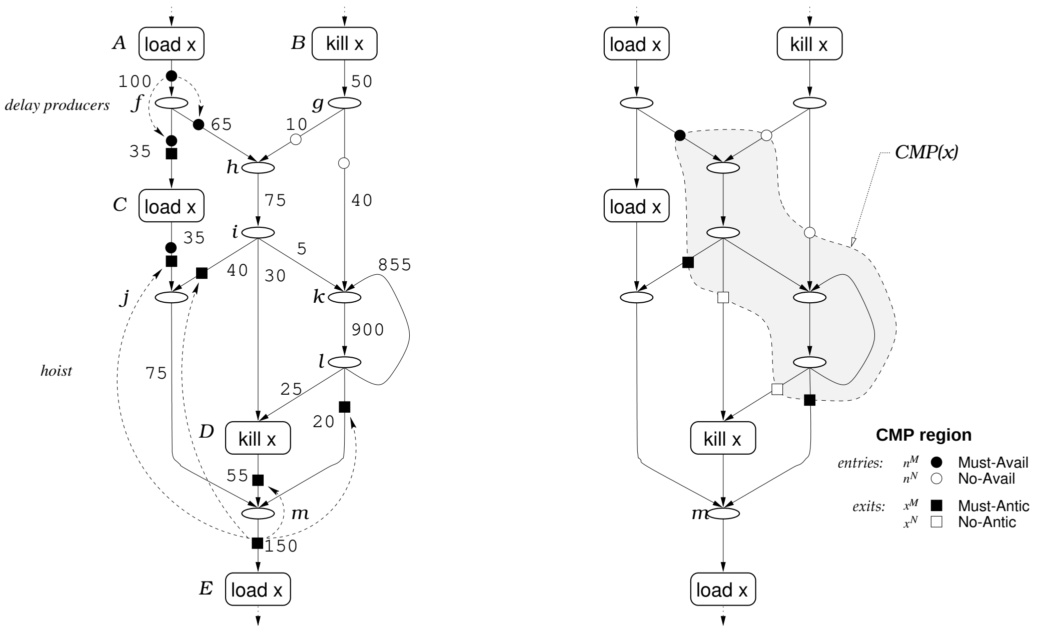


Figure 5.5: **The example of the PRE estimator.**

The CMP estimators. This large PRE's error is not entirely due to the error of the edge profile. Instead, the loose bounds are caused by PRE's crude placement of summary points, which suffers from "overbooking" of a producer to multiple consumers. In the running example, the node A acts as a producer common to consumers C and E, which together consume more value flow than A can produce (while A produces only 100 units, C and E together consume $35 + 100$ units, see Figure 5.5). Overbooking can be removed by dividing the producer's contribution among paths leading from the producer to its different consumers. In the CMP estimators, the contribution is divided by *delaying* [KRS94b] the producer. Delaying moves the producer in forward direction along all paths as far as it remains a producer according to Definition 5.3, i.e., as far as each path through it generates the value. Figure 5.6(a) shows how the producer A is delayed into the edges (f, C) and (f, h), which become the new producers, effectively dividing the contribution of A among consumers C and E.

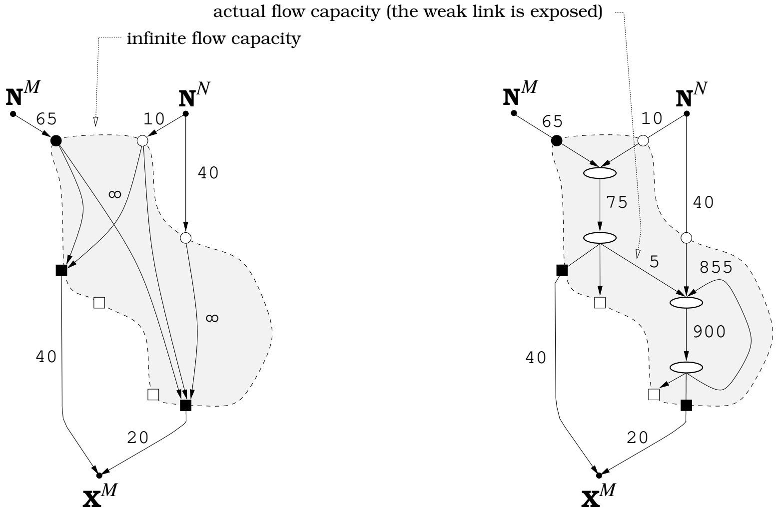
Note that consumers, too, cause PRE's imprecision. In the example, the consumer E claims to be able to consume 150 units (equal to its weight), while only 95 units can reach it, due to the kill node D which blocks 55 units of value flow. This flawed flow accounting is fixed by *hoisting* the consumers. Hoisting moves consumers backwards as far as they remain consumers, much like delaying moves the producers.

After producers are delayed and consumers are hoisted, all summary points are optimally placed. (Note that, unlike its producers and consumers, PRE's placement of stealers was already optimal.) The placements are optimal in the sense the summary points cannot be moved closer to each other without being forced on a path that contradicts their definition (Definition 5.3). As a result, the paths between producers/stealers and consumers are as short as possible, minimizing the number of conditional branches on these paths and thus also the branch-correlation error appearing in the estimates. Indeed, where producers/consumers and stealers meet, creating unit-length paths, edge profile introduces no error. For example, the frequency of the



(a) The source program annotated with an edge profile.

(b) The CMP region for the reuse on memory location x.



(c) CMP^r estimator, based on control-flow reachability.

(d) CMP^f estimator, based on frequency reachability.

Figure 5.6: **Computing the estimates on the running example.**

unit-length producer-consumer path $[(C,j)]$ can be trivially and precisely determined from any edge profile. On the other hand, where the summary points do not meet, error may appear in the estimate.

To put these principles on a solid footing, we observe that the summary points enclose a VNG region. The region has multiple entry edges and multiple exit edges. Its entries can be divided between producers and stealers, and its exits can be divided between consumers or (strict) non-consumers. Because the summary points are optimally placed, the region is the smallest region that contains all the profiling error. It turns out that this region precisely coincides with the CMP region to be presented in Chapter 6. The CMP maximizes the number of paths that can be excluded from the worst-case assumptions about branch correlations: any thread passing outside the CMP region can be estimated precisely from any edge profile. Conversely, any path that crosses the region may have profiling error (for some profiles).

Let us rephrase now the definition of the CMP region (to be stated in Definition 6.4) using the terminology of this chapter. Formally, the CMP is a subgraph of the VNG, delimited by entry end exit edges, such that on each node from the CMP region, its value is generated on some (but not all) incoming threads, and will be used on some (but not all) outgoing threads. The CMP region is identified by solving the problems of anticipability and availability, as defined in Definition 4.2.

Definition 5.4 (CMP Region) Let $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{E}, \textbf{start}, \textbf{end})$ be a VNG. The CMP region is a set of nodes $\boldsymbol{n} \subseteq \boldsymbol{N}$ such that $\mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] = \mathsf{May}$ and $\mathit{ANTIC}_{\mathit{in}}[\boldsymbol{n}] = \mathsf{May}$.

Each CMP region has a set of entry edges and exit edges. Each entry is either

- *Must*-available, denoted $\boldsymbol{n}^{\boldsymbol{M}}$, acting as producers, or
- *No*-available, denoted $\boldsymbol{n}^{\boldsymbol{N}}$, acting as stealers.

Similarly, exits are either

- *Must*-anticipated, denoted $\boldsymbol{x}^{\boldsymbol{M}}$, acting as consumers, or
- *No*-anticipated, denoted $\boldsymbol{x}^{\boldsymbol{N}}$, not participating in the estimator calculations.

☐

The CMP region divides the value flow into *definite* and *uncertain* components. The definite component S_d has no error and equals the sum of frequencies of all definite producers Pr_d, defined in Figure 5.7 below. The salient property of each definite producer is that all value flow it produces will be consumed. The bounds of the uncertain component are computed in the CMP region and are given in Figure 5.8.

$$
\begin{aligned}
Pr_d &:= \{(\boldsymbol{u},\boldsymbol{v}) \mid \mathit{AVAIL}_{\mathit{out}}[\boldsymbol{u}] = \mathsf{Must} \wedge (\mathit{AVAIL}_{\mathit{in}}[\boldsymbol{v}] = \mathsf{May} \vee \boldsymbol{v} \in \boldsymbol{U}) \wedge \mathit{ANTIC}_{\mathit{in}}[\boldsymbol{v}] = \mathsf{Must}\} \\
S_d &:= \sum_{\boldsymbol{n} \in Pr_d} \mathit{freq}(\boldsymbol{n}) \\
U^e &:= S_d + U^e_u \\
L^e &:= S_d + L^e_u
\end{aligned}
$$

Figure 5.7: **The CMP estimators for separable VNGs.** $e \in \{\mathrm{CMP}^1, \mathrm{CMP}^c, \mathrm{CMP}^r, \mathrm{CMP}^f\}$ The formulas for computing the uncertain component of the estimate (L^e and U^e) are given in Figure 5.8.

Example 5.3 (CMP Region, Definite Estimate) The CMP region for the running VNG example is shown in Figure 5.6(b). The CMP region excludes from the worst-case considerations threads $[A,f,C]$ and $[C,j,E]$, because their estimate can be computed from an edge profile precisely, using the *definite* producer points, which are

$$Pr_d = \{(f, C), (C, j)\}$$

Each of these definite producers provides 35 units of reuse. Because these producers are definite, their reuse will be fully consumed, by C and by E, respectively. Thus, the definite reuse is

$$S_d(x) = 70.$$

The definite reuse remains the same for all CMP estimators. □

The CMP estimators differ in how they compute $U_u^{CMP}(x)$ and $L_u^{CMP}(x)$, which are the bounds of the uncertain component of the estimate. Figure 5.8 compares the CMP estimators.

The CMP^1 estimator. This is the simplest CMP-based estimator. CMP^1 identifies CMP entries and exits and, to minimize its computational cost, assumes that each CMP entry-exit pair is mutually reachable. The upper-bound scenario resulting from this assumption is that all $\boldsymbol{n}^{\boldsymbol{M}}$ entries are producers for all $\boldsymbol{x}^{\boldsymbol{M}}$ consumers. The lower bound follows the same conservative assumption that the CMP region is fully connected. The bounds are computed as in the PRE estimator. Figure 5.8 gives the formulas.

CMP^1 is very efficient; it computes only the *ANTIC* and *AVAIL* data-flow solutions. Entries and exits are identified by examining the two data-flow solutions locally at each node. The cost of computing the solutions and the entries and exits is amortized, as they are also needed by the PRE transformation (Chapter 6).

Example 5.4 (CMP^1) For the running example in Figure 5.6(b), CMP^1 yields

$$\begin{aligned} L_u^{\text{CMP}^1} &= \max\{0, (40+20) - (10+40)\} = 10 \\ U_u^{\text{CMP}^1} &= \min\{65, 40+20\} \qquad\qquad\quad = 60. \end{aligned}$$

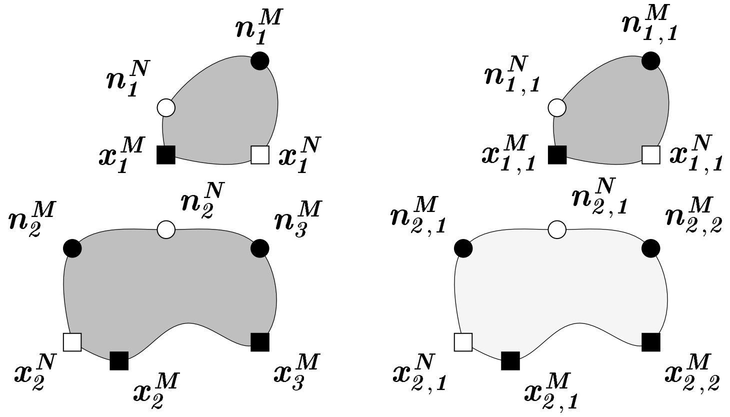
The total estimate is

$$\begin{aligned} L^{\text{CMP}^1} &= S_d + L_u^{\text{CMP}^1} = 70 + 10 = 80 \\ U^{\text{CMP}^1} &= S_d + U_u^{\text{CMP}^1} = 70 + 60 = 130, \end{aligned}$$

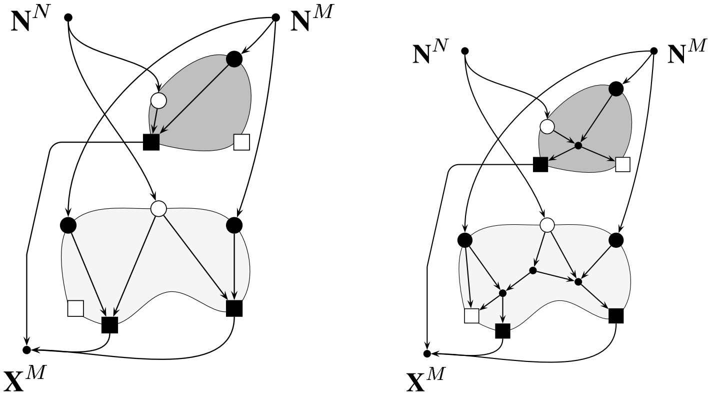
which improves PRE's upper bound by removing overbooking of the producer A, reducing the error from 112.5% to $(130-80)/80 = 62.5\%$. Note that, while CMP^1 is better than PRE in this example, it is not strictly superior in general, as indicated in the hierarchy graph in Figure 5.3. □

The CMP^c estimator. This estimator improves the precision of CMP^1 by eliminating some false entry-exit reachability assumed by CMP^1. To this end, it identifies connected CMP subregions, thus partitioning producer, stealer, and consumer sets. Smaller sets result in less overestimation when considering the worst-case scenarios. The bounds are computed separately for each connected CMP, and then summed. In the experiments, this partitioning of the CMP region produced the highest increase in precision. As a result, CMP^c is the recommended estimator for practical applications, due to its cost-precision balance (see Section 5.7 for further empirical observations).

The CMP^c estimator is more complex than CMP^1. In addition to computing *AVAIL*, *ANTIC* and identifying CMP entries and exits, it must compute 1) reachability of CMP entry-exit pairs, producing a reachability graph, and 2) find connected subgraphs of the reachability graph, to find connected CMP subregions. Fortunately, these two analyses are also needed in the PRE transformation, to guide the profile-directed speculation (see Section 6.3.2.1).



	CMP^1	**CMP**^{c}
U_u	$\min\{\sum_i \boldsymbol{n}_i^M, \sum_i \boldsymbol{x}_i^M\}$	$\sum_i \min\{\sum_j \boldsymbol{n}_{i,j}^M, \sum_j \boldsymbol{x}_{i,j}^M\}$
L_u	$\max\{0, \sum_i \boldsymbol{x}_i^M - \sum_i \boldsymbol{n}_i^N\}$	$\sum_i \max\{0, \sum_j \boldsymbol{x}_{i,j}^M - \sum_j \boldsymbol{n}_{i,j}^N\}$



	CMP^r	**CMP**^{f}
U_u	maxflow($\mathbf{N}^M, \mathbf{X}^M$)	
L_u	$\sum_i \boldsymbol{x}_i^M$ − maxflow($\mathbf{N}^N, \mathbf{X}^M$)	

Figure 5.8: **The CMP-based estimators for separable VNGs:** algorithms for computing the uncertain component of the estimate. In the formulas, n^M, n^N, and x^M (are overloaded to) mean the frequencies of the corresponding CMP entries and exits. maxflow(u, v) denotes the maximum flow between vertices u and v in the shown networks. **CMP**^{1} assumes all CMPs are one, i.e., that all entries and exits are mutually reachable. **CMP**^{c} separates connected CMPs, eliminating some false reachability. **CMP**^{r} exploits intra-CMP reachability, using a max-flow computation. **CMP**^{f} exposes to the max-flow all intra-CMP edges, including their actual profile weights.

Example 5.5 (CMP^c) In the running example, the CMP region is connected, hence the CMP^c estimate is identical to that of CMP^1. ☐

The CMP^r estimator. This estimator adds more precise handling of intra-CMP reachability. Each CMP region is represented as a bipartite graph in which *entry* and *exit* nodes are connected if there is a thread connecting them (see Figure 5.8). The bipartite graphs are connected into a network using three super-nodes $\mathbf{N}^M$, $\mathbf{N}^N$, and $\mathbf{X}^M$ that connect all producers, stealers, and consumers, respectively. The flow capacities of edges connecting the super-nodes mirror the frequency of CMP entry and exit edges; the capacity of intra-CMP edges is (conservatively) set as infinite. Equipped with this network, we compute the upper reuse bound as the maximum flow between $\mathbf{N}^M$ and $\mathbf{X}^M$. Similarly, the amount of reuse that can be stolen from consumers is given by the max-flow between $\mathbf{N}^N$ and $\mathbf{X}^M$.

Compared to CMP^c, the CMP^r estimator does not identify connected sub-regions, but instead computes the more costly network-flow. Note that the network construction implicitly partitions the CMP into connected sub-regions.

Example 5.6 (CMP^r) The network for our running example is shown in Figure 5.6(c). Because CMP exit (i, j) is not reachable from CMP entry (g, k), less reuse can be stolen than in CMP^c, which improves its lower bound:

$$\begin{aligned}
L^{\text{CMP}^r} &= S_d + L_u^{\text{CMP}^r} \\
&= S_d + max\{0, \sum_i \boldsymbol{x}_i^{\boldsymbol{M}} - \text{maxflow}(\mathbf{N}^N, \mathbf{X}^M)\} \\
&= 70 + \max\{0, (40 + 20) - 30\} \\
&= 100, \\
U^{\text{CMP}^r} &= S_d + U_u^{\text{CMP}^r} \\
&= S_d + \text{maxflow}(\mathbf{N}^M, \mathbf{X}^M) \\
&= 70 + 60 \\
&= 130,
\end{aligned}$$

which is a $(130 - 100)/100 = 30\%$ error. ☐

The CMP^f estimator. While an entry-exit pair may be graph-reachable (i.e., reachable along a thread), it may not be sufficiently *frequency*-reachable. In Figure 5.6(b), such a pair is the CMP entry (f, h) and the CMP exit (l, E). The only path connecting them contains a *weak link*—the edge (i, k) with a low frequency of 5. Even though there is enough value flow on the entry, the weak link prevents this flow from saturating the exit (l, E)—only 5 units of reuse can be exploited. To account for weak links, it suffices to expose to the max-flow computation the inside structure of the CMP at the edge level, including edge frequencies, as shown in Figure 5.6(d).

Example 5.7 (CMP^f) After the weak link is accounted for, the upper bound of the previous estimator is improved:

$$\begin{aligned}
L^{\text{CMP}^r} &= S_d + L_u^{\text{CMP}^r} \\
&= S_d + max\{0, \sum_i \boldsymbol{x}_i^{\boldsymbol{M}} - \text{maxflow}(\mathbf{N}^N, \mathbf{X}^M)\} \\
&= 70 + \max\{0, (40 + 20) - 30\} \\
&= 100, \\
U^{\text{CMP}^r} &= S_d + U_u^{\text{CMP}^r} \\
&= S_d + \text{maxflow}(\mathbf{N}^M, \mathbf{X}^M) \\
&= 70 + 45 \\
&= 115,
\end{aligned}$$

which is a $(115 - 100)/100 = 15\%$ error. Note that, for this example, CMP^f estimator produced tight bounds; that is, there exist control traces t_1, t_2 that induce the edge profile P and t_1's "estimate" equals L^{CMP^r}, and t_2's "estimate" equals U^{CMP^r}. □

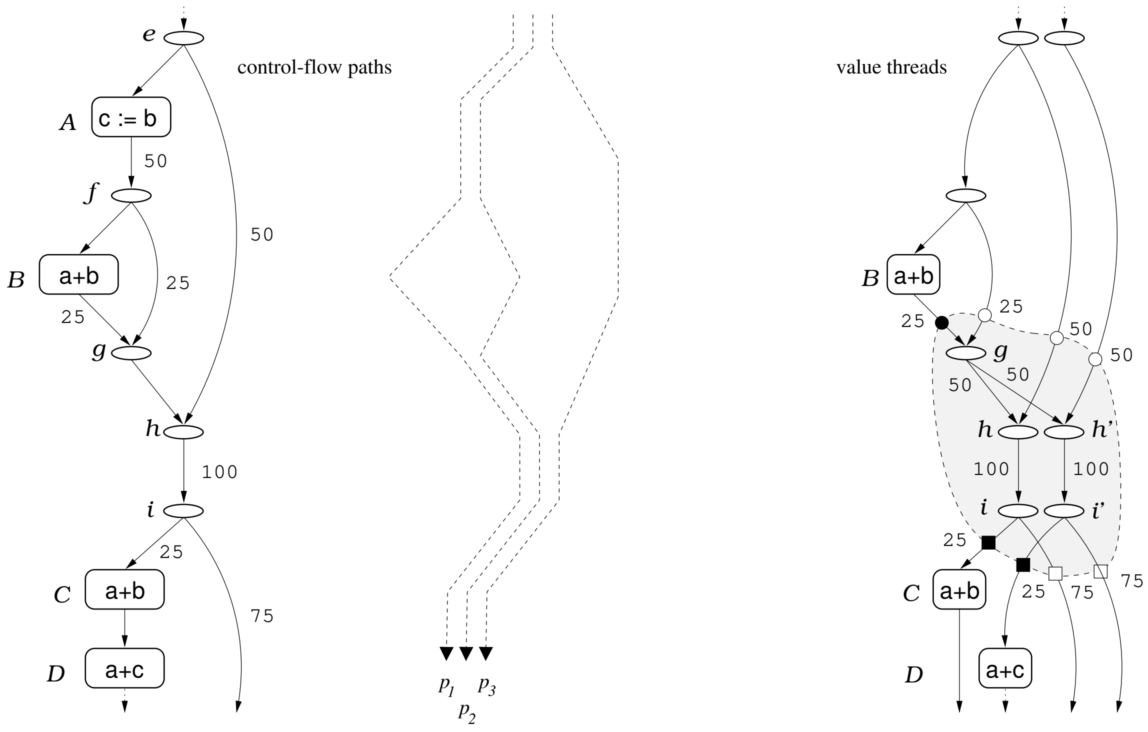
5.5 Estimators for a general VNG

The previous section presented estimators for a *separable* VNG. On a separable VNG, threads can be considered in isolation, like bits in a separable bit-vector problem. In a *general* VNG, threads may be split (even when underlying control flow paths do not split), signifying that a value is identified under multiple, synonymous names. Such thread splitting qualitatively changes the VNG properties. In particular, what is considered "overbooking" of a generator by the consumers on a separable VNG may be considered legal on the general VNG. This section extends our five estimators to handle the complications of the general VNGs.

The example in Figure 5.9 illustrates the issues raised by general VNGs. Consider the control flow paths p_1 and p_2 shown in Figure 5.9(b). Along these paths, the nodes C and D are equivalent, yet in the VNG they are (correctly) not placed on the same thread. The VNG is shown in Figure 5.9(c). Nodes C and D are not the same thread because they are not synonymous along all incoming paths. They differ along path p_3; as a result, the thread going across g is *split* into h and h'.

The consequence of such thread splits is that a generator node can "provide" more value flow than its own execution frequency. Consider the node B in Figure 5.9. Node B is a generator with frequency 25, yet it provides enough value flow for both C and D, which are consumers with a frequency of 25 *each*. The reason why B can "feed" both C and D is that its value flow is duplicated when it crosses node g, where the thread is split. As a result of the split, the incoming value flow is *duplicated* onto the outgoing threads (i.e., onto nodes h and h'). A general VNG has thus two kinds of thread splits. First, the split at i is induced by control flow, just like in the separable VNG. At control-flow split, the frequency of value flow is *divided* among outgoing threads. Second, the split at g is induced by synonyms. At a synonym split, the frequency is *duplicated*.

On a general VNG, the flow entering the region need not equal the flow leaving the region, which was the case on a separable VNG. The reason is that value flow may be duplicated in the region. Consider the CMP region in Figure 5.9(c). The sum of entry edge frequencies is 150, but the sum of exit edge frequencies is 200. The difference between exit and entry frequencies equals the amount of "flow duplication" in the CMP region. The amount of this duplication is an important value; it will be used to generalize our estimators.



a) The CFG of the program.

b) The paths through the CFG.

c) The VNG of the program and its CMP region.

Figure 5.9: **Computing the estimate on a general VNG.**

The PRE estimator. The formulas for computing the PRE estimate are the same as for the separable VNGs shown in Figure 5.4. The algorithm remains the same as before because each consumer still finds all its producers and stealers. Compared to the separable setting, the only difference is that some of the overbooking is actually legal value flow, due to flow duplication, as discussed above.

The *CMP* estimators. Two extensions are needed to handle merging threads. First, when hoisting the consumer summary points, it must be guaranteed that they are placed at points from which exactly one user is reachable along each outgoing control flow path, even in the presence of thread merging. Second, it must be accounted for duplication of flow in the region. Because the frequency of CMP exits may be greater than the frequency of CMP entries, the entries alone do not tell us how much flow can be produced or stolen.

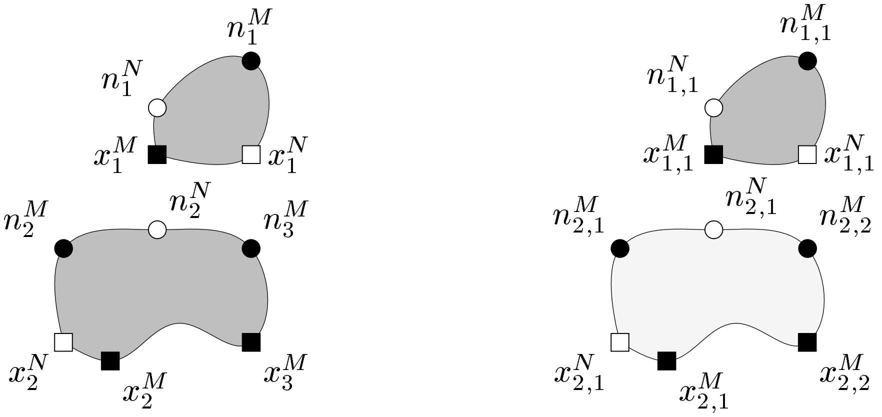
1. Placement of consumers. When hoisting the consumer points, it must be guaranteed that consumers are placed at points from which exactly one user can is reachable a) along each outgoing control flow *path*, rather than b) along each value name *thread*, a condition used in the separable setting, in which a) and b) coincide. The justification for the refinement is that, using b), multiple users can be reached along a single control flow path. Therefore, the frequency of the consumer point under-represents the amount of users reachable from the consumer. In Figure 5.9(c), the consumers cannot be hoisted across node g because the consumers would be merged into one consumer whose frequency would bar less than those of the summarized users. (Use a better example, in which the edges emanating from thread-merge node are *Must*-anticipated.)

2. Flow duplication in the CMP region. Let r be a CMP region, f_N be the sum of r's entry edge frequencies, $f_N = \sum_{\boldsymbol{n}_i} freq(\boldsymbol{n}_i)$, and f_X be the sum of r's exit edge frequencies, $f_X = \sum_{\boldsymbol{x}_i} freq(\boldsymbol{x}_i)$. Then, $\Delta = f_X - f_N$ gives the amount of flow duplicated in the CMP region. Because the duplication happens inside the region, a fraction of Δ carries the value reuse and a fraction is reuse-free, depending on whether the flow originated at producer entries or stealer entries. In CMP^1, CMP^c, and CMP^r, these fractions cannot be determined (because the inside of the CMP region is not examined) and hence these estimators (conservatively) increase by Δ both the producers frequencies and the stealer frequencies. In contrast, because the CMP^f estimator exposes the individual inner edges of the region, it can account for how much of the duplicated flow is from producers versus from stealers. Instead of computing the Δ, CMP^f reduces the estimate into a generalized version of the network flow problem, defined below.

For the *CMP* estimators, the formulas in Figure 5.7 remain valid in the generalized setting. The formulas for the uncertain component of *CMP* estimators change, as shown Figure 5.10. The CMP^1 estimator computes Δ for all entries and exits together, assuming all belong to the same connected CMP region. The CMP^c estimator finds the connected sub-regions and computes Δ_i for each ith sub-region. The CMP^r estimator accounts for the duplicated flow by adding a node δ to the network that reflects the intra-CMP reachability. The added node, denoted δ, increases the amount of flow that can reach the consumer exits, both from the producer super-node $\mathbf{N}^M$ and the stealer super-node $\mathbf{N}^N$.

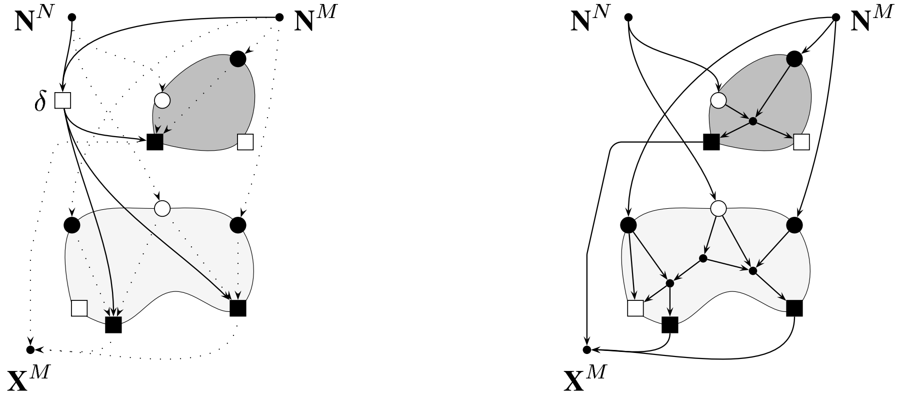
The CMP^f estimator The CMP^f estimator exposes to the max-flow computation all inner CMP edges. To account for flow duplication, let us introduce a generalized max-flow problem, called *max-dup-flow*, in which some nodes in the network rather than preserve the incoming flow onto the outgoing edges (in the usual way), while others can *duplicate* it.

Definition 5.5 (Duplicating Flow Network) A *duplicating flow network* $G = (N_p, N_d, E)$ is a directed graph whose vertices N are divided into *flow-preserving* vertices N_p and *flow-duplicating* vertices N_d, $N = N_p \cup N_d, N_p \cap N_d = \emptyset$. Each edge $(n, m) \in E, E \subseteq N \times N$ has a non-negative *capacity* $c(n, m) \geq 0$. For



CMP^1 **CMP**^{c}

	$\Delta = \sum_i x_i^M + \sum_i x_i^N - \sum_i n_i^M - \sum_i n_i^N$	$\Delta_i = \sum_j x_{i,j}^M + \sum_j x_{i,j}^N - \sum_j n_{i,j}^M - \sum_j n_{i,j}^N$
U_u	$\min\{\Delta + \sum_i n_i^M, \sum_i x_i^M\}$	$\sum_i \min\{\Delta_i + \sum_j n_{i,j}^M, \sum_j x_{i,j}^M\}$
L_u	$\max\{0, \sum_i x_i^M - \Delta - \sum_i n_i^N\}$	$\sum_i \max\{0, \sum_j x_{i,j}^M - \Delta_i - \sum_j n_{i,j}^N\}$



CMP^r **CMP**^{f}

	$\Delta = \sum_i x_i^M + \sum_i x_i^N - \sum_i n_i^M - \sum_i n_i^N$	
U_u	maxflow($\mathbf{N}^M, \mathbf{X}^M$)	
L_u	$\sum_i x_i^M$ − maxflow($\mathbf{N}^N, \mathbf{X}^M$)	

Figure 5.10: **The CMP-based estimators for bi-distributive VNGs:** algorithms for computing the uncertain component of estimates. The algorithms generalize the algorithms for separable VNGs (see Figure 5.8). **CMP**^{1} adds the amount of flow duplicated in the CMP region, denoted Δ to both the produced flow and the stolen flow. **CMP**^{c} is similar, except Δ is computed for each connected sub-region. **CMP**^{r} adds more flow to the consumers by adding a "channel" between the super-nodes and the consumers. The flow capacity of the added node δ is Δ. Dotted lines mark edges inherited from the separable CMP^r. **CMP**^{f} computes the bounds using a generalized version of the max-flow problem, in which some nodes duplicate, rather than distribute the incoming flow.

convenience, we assign the capacity of 0 to all nonexistent edges. We distinguish two special nodes $s, t \in N_s$, which are the *source* and *sink* of the network, respectively.

Definition 5.6 (Max-Dup-Flow Problem) Let $G = (N_p, N_d, E)$ be a duplicating flow network (with an implied capacity function c, source s and sink t). A *flow* in G is an integer-valued function $f : E \to \mathbb{Z}$ that satisfies the following four properties:

1. *Capacity constraint:*

$$\forall (n, m) \in E \ . \ 0 \leq f(n, m) \leq c(n, m)$$

2. *Flow preservation (at flow-preservation nodes N_s):*

$$\forall n \in N_s - \{s, t\} \ . \ \sum_m f(m, n) = \sum_k f(n, k)$$

3. *Flow duplication (at flow-duplication nodes N_d):*

$$\forall n \in N_d \ . \ \sum_m f(m, n) = \max_k f(n, k)$$

Reducing the estimation problem to the max-dup-flow problem. The reduction of the estimation problem to the max-dup-flow problem is straightforward. For each VNG node $\boldsymbol{n}$ that is a synonym split node (like g in Figure 5.9(c)), a flow-duplicating node is created. For all other VNG nodes, we create a flow-preserving node.

Computing the Max-Dup-Flow problem. While we do not have an algorithm for computing the max-dup-flow precisely. Instead, an approximation is computed by reducing the max-dup-flow problem to the standard max-flow. The reduction is similar to that used in CMP^r. The reduction adds Δ units of flow to the CMP region but, compared to CMP^r does it in a more fine-grained fashion. Rather than connecting to the node δ the exis of the CMP region, it connects to δ the flow-duplicating nodes from the CMP region. This accounts for (some) weak links in the CMP region.

5.6 Correctness and complexity

Theorem 5.1 (Correctness)

$$L^{\mathrm{CMP}^f} \leq L, \ U^{\mathrm{CMP}^f} \geq U$$

Theorem 5.2 (Precision Order)

$$\begin{aligned} L^{\mathrm{CMP}^1} &\leq L^{\mathrm{CMP}^c} \geq L^{\mathrm{CMP}^r} \geq L^{\mathrm{CMP}^f} \\ U^{\mathrm{CMP}^1} &\geq U^{\mathrm{CMP}^c} \geq U^{\mathrm{CMP}^r} \geq U^{\mathrm{CMP}^f} \end{aligned}$$

Computational complexity ToDo.

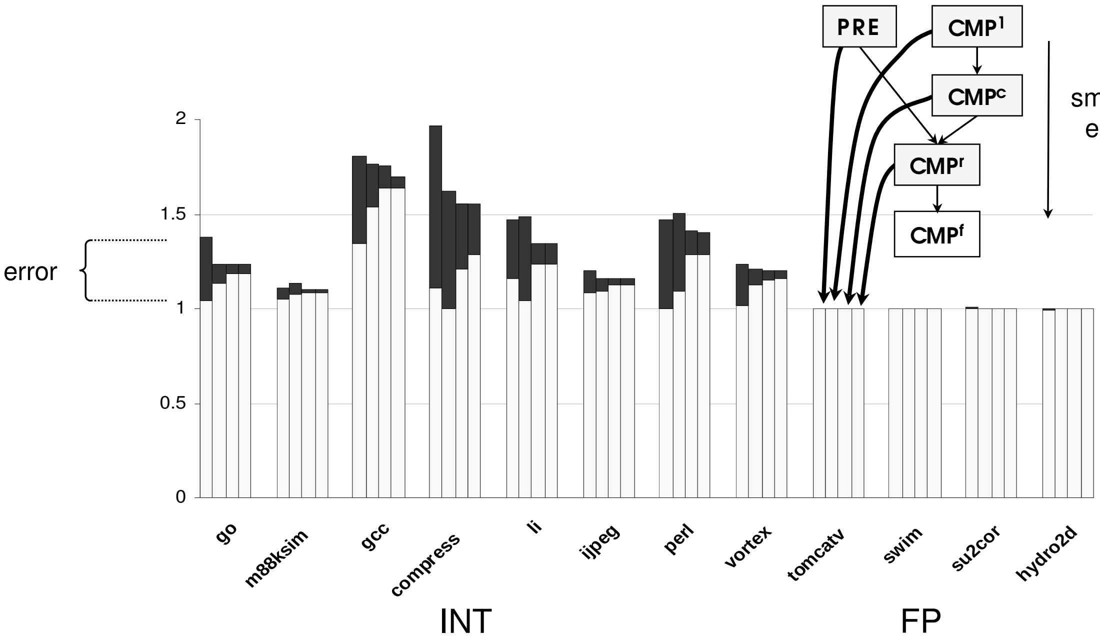


Figure 5.11: **An experimental comparison of estimator precisions.** The CMP^f estimator was not implemented.

5.7 Experiments

Figure 5.11 compares the precision of the estimators. For each benchmark, the figure plots the weighted reuse obtained by four estimators (CMP^f was not implemented). The reuse is broken up into four parts; the left two bars together represent the definite reuse component R_d, on which all benchmarks are normalized. The third and fourth bars are the lower and the upper bounds on the uncertain reuse. The floating-point benchmarks (the lower four) have nearly no uncertain reuse, due to simple control flow. On the other hand, the reuse in integer benchmarks has a significant uncertain component. It can be observed that with good algorithms, the profiling error can be greatly reduced. Note that while, in theory, CMP^c is not strictly more precise than PRE (as the precision ordering shows), it performs much better in practice. In fact, CMP^r is appreciably better than CMP^c only on `gcc`. Hence, due to its simplicity, CMP^c may be the estimator of choice. Overall, the average error was 15% for PRE and 5% for CMP^r.

An important observation was that the estimator precision is strongly dependent on the pointer aliasing information. By interrupting some reuse paths, the killing stores induce more CMP regions, with more entries and exits, increasing the amount of uncertain reuse. For the comparison in Figure 5.11, we selected the configuration of load-reuse analysis that caused the largest estimator errors (kill set = each array and pointer store, and each procedure call; see Section 5.7).

5.8 Correlation profiling

The CMP-based Correlation Profiling estimator is not based on edge profiles. Instead, it assumes profile information that correlates CMP entries and exits sufficiently to avoid the profiling error. We present it to show what profile information may fully eliminate the profiling error.

Using the CMP region, we can specify what information from a profiler would enable computing the reuse with no branch-correlation error. Coming back to Figure 5.6(b), we can observe that the precise amount of uncertain reuse equals the number of times a producer entry n^M is followed by a consumer exit x^M. Therefore, measuring the pair-wise correlation between CMP entries and exits captures all branch correlation that affects the amount of reuse. After the data-flow analysis identifies the CMP regions, the profiler can instrument the program to collect this pair-wise information. Whether such a pair-wise profiling can be (efficiently) performed prior to knowing the shapes of CMP regions in the profiled program is an open question.

5.9 Conclusion

The estimators presented in this chapter compute, given an edge profile, the (bounds of) cumulative frequencies of the reuse threads detected by the value flow analysis. The estimators are conceptually simple: they do not reconstruct the path frequencies but instead find program points that summarize the paths. The points are value-flow-dependent; if value flow-patterns are simple (e.g., value-flow holds on all paths, or can be handled with code-motion PRE), the edge profile errors may not appear, for any given edge profile.

The experimental results suggest that imprecise, data-flow-independent profiling followed by smart estimation algorithms is a very successful strategy for profile-directed optimization. If one desires to use a more precise, data-flow-dependent profiling, then the correlation profiling described in Section 5.8 seems to be the alternative of choice.

Mor especifically, the experiments suggest that the CMP^c estimator offers an ideal balance between precision and computational cost. On average, CMP^c was able to bound the inherent profiling error down to about 6.5%, a sufficient precision for profile-directed optimization. Compared to the more complex estimators, CMP^c has linear asymptotic complexity.

By bounding the branch correlation error, the estimators expose the inherent precision of edge profiles. Our experiments suggest that:

- Edge profiles are precise for load-reuse optimization. If this result extends to other value-flow problems, we do not need path-profiles, which are one of the exponential factors in value-flow optimizations.
- Considering that the inherent edge-profile error is small, as suggested by our experiments, the maximum amount of error in the result of Ramalingam's frequency analysis will be correspondingly small (the result of his analysis always falls between our lower and upper bounds).

Chapter 6

Intra-procedural Removal of Redundancies

This chapter describes the last stage of the PATHFINDER optimizer—program transformation—whose task is to perform the actual optimization of the program. Given the redundant computations detected by the previous stages, a program transformation stage modifies the program so that values are reused, rather than redundantly recomputed. Such a transformation is known as Partial Redundancy Elimination (PRE) [MR79], because it removes redundancies that are only *partial* (i.e., path-specific). Because partial redundancies include common subexpressions and loop-invariant computations, PRE has become the most important component of global optimizers.

Ideally, the transformation should remove all computations detected as redundant. While such a *complete* transformation is possible (decidable), it may require a different code transformation along each optimizable program path, which requires a separation of individual optimizable paths, which may in turn incur prohibitive growth in the program code size. To avoid code growth, practical PRE algorithms restrict their toolset to *code motion*, a method that moves redundant instructions but does not separate paths. The price of practicality is, however, the failure to remove the redundancies completely. Experimental observations show that the penalty is severe. In static terms, 73% of loop-invariant statements cannot be eliminated from loops by code motion alone. In dynamic terms, the traditional (code-motion) PRE eliminates only half of redundancies that are strictly partial.

This chapter focuses on achieving a (nearly) complete PRE while incurring an acceptable code growth. This goal is achieved by combining the strengths of three transformation methods—*code motion*, control flow *restructuring*, and control *speculation*—integrated in a parameterizable way that induces a family of transformation algorithms, all built on the same abstraction. The algorithms are characterized either as profile-independent or profile-guided.

The main profile-independent algorithm integrates the economical code motion with the more powerful restructuring. In contrast to existing complete techniques, the algorithm resorts to restructuring merely to remove obstacles to code motion, rather than to carry out the entire optimization. For a large class of problems, this algorithm achieves minimal code growth.

When a program profile is available, additional code growth reduction is possible by sacrificing completeness where it is dynamically insignificant. Based on the profile-based estimators from Chapter 5, the main profile-guided algorithm combines code motion with control speculation. Speculation overcomes the obstacle to code motion not by separating paths, but by inserting computations on program paths that did not execute them in the original program. Estimators ensure that after this (potentially counterproductive) transformation, the program is improved more than it is impaired. In fact, estimators can maximize the optimization benefit of speculation.

The last algorithm balances all three transformations: when the economical code motion fails and the unsafe speculation impairs the program, restructuring is used. In practice, the algorithms presented in this chapter can achieve a near-complete redundancy removal with very little code growth.

This chapter is concerned with the reuse of *intra*-procedural value flow. *Inter*-procedural redundancies will be attacked in the following chapter.

6.1 Overview

The PATHFINDER stages presented so far *detected* the reuse of values: in turn, the representation exposed value flow using threads, the analysis marked value reuse on the threads, and the estimators weighted the reuse threads with profile information. The task of the last stage is to *exploit* the value reuse, by transforming the original program so that values are reused rather than redundantly recomputed. This transformation, known as Partial Redundancy Elimination (PRE), was first proposed by Morel and Renviose [MR79].

By removing computations that are partially redundant, PRE is responsible for handling the *partial* flavor of path-sensitivity, i.e., the reuse that is available only along some paths (see Section 1.4). Formally, partially redundant computations are VNG *user* nodes for which reuse was detected along some (but not necessarily all) paths. See Definition 4.1 for the description of reuse threads.

Definition 6.1 (Partial Redundancy) A VNG node $\boldsymbol{n} \in \boldsymbol{U} \subseteq \boldsymbol{N}$ is *partially redundant* if there is a reuse thread $\boldsymbol{p}[\boldsymbol{m}, \boldsymbol{n}) \in R$.

It can be shown that a computation $\boldsymbol{n}$ is partially redundant when

$$AVAIL_{in}[\boldsymbol{n}] \in \{\mathsf{Must} \vee \mathsf{May}\}$$

(see Definition 4.4.)

PRE is attractive because, by targeting computations that are redundant only along some program paths, it subsumes and generalizes two important value-reuse optimization: global common subexpression elimination and loop-invariant code motion. Furthermore, because the VNG uniformly represents all kinds of value flow (e.g., value recomputation, repeated loads, correlated branches), PRE serves in our value-flow framework as a unified program transformation technique.

The ideal optimization goal is to remove all computations detected as redundant. While such a program transformation is possible, it may require isolation of optimizable program paths, which may incur prohibitive code growth, due to the exponential path explosion, as shown in Section 1.5. Therefore, practical PRE algorithms are based on *code motion*, an economical transformation that reorders instructions but does not change the shape of the control flow graph, prohibiting the expensive isolation of individual paths [BC94, CCK$^+$97, DRZ92, Dha91, DS88, DS93, KRS94b, MR79]. The price of the restriction to code motion, however, is the failure to remove all detected redundancies. In theory, even the optimal code-motion algorithm [KRS94b] breaks down on loop invariants in while-loops, unless preceded by do-until conversion (which is based on path separation). In practice, one half of (dynamic) computations that are strictly partially redundant (i.e., *not* redundant along some paths) are left unoptimized due to code-motion obstacles, according to our experiments.

This chapter achieves a (nearly) complete PRE by first conceptually analyzing the Morel-Renviose code-motion algorithm, and then by systematically addressing its limitations. The result is a family of PRE algorithms that combines strengths of three transformation methods: *code motion* (economical), control flow *restructuring* (powerful), and control *speculation* (potentially harmful). The three methods are abbreviated with M, R, S, respectively.

As the first step towards a complete PRE with affordable code growth, we present an algorithm that integrates the code motion with restructuring, denoted PRE(MR). The algorithm allows a complete removal of redundant expressions while minimizing code duplication. In contrast to existing complete techniques [Weg75b, Weg75a, Ste96], it resorts to restructuring merely to remove obstacles to code motion, rather than to carry out the entire optimization, thus eliminating unnecessary code duplication. The resulting code growth is minimal for a large class of problems. On SPEC95, we found the code growth to be three times smaller than that of the pure restructuring approaches (denoted PRE(R)).

No prior work systematically treated the integration of code motion and transformation. The PRE(MR) algorithm controls the extent of code duplication by restricting it to a *code-motion preventing* (CMP) region, which localizes adverse effects of control flow on the desired code motion. Figure 6.2(a–c) illustrates our first algorithm, PRE(MR), through optimization of the loop in Figure 6.1.

```
while (true) {
   if (O)
      c + d
   else if (P)
      break
   if (Q)
      c + d
   else
      R
   S
   a + b
}
```

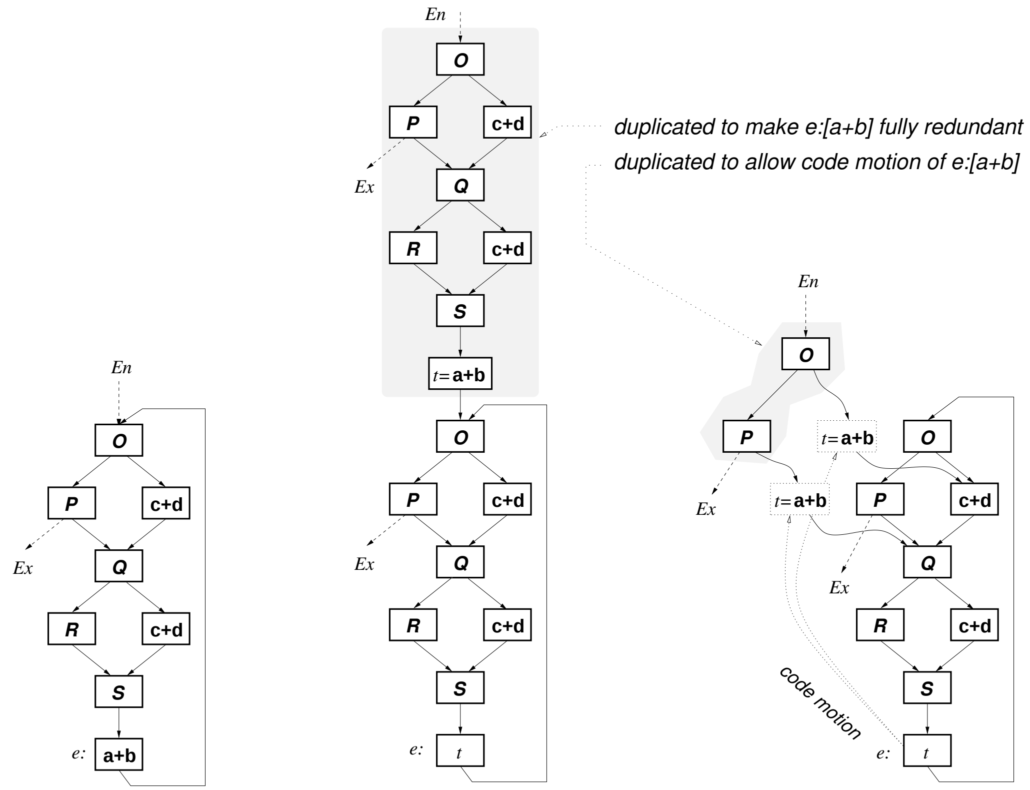
Figure 6.1: **The example loop.**

Let us assume that no statement in the loop in Figure 6.2 defines variables a, b, c, or d. Hence, the computations $a + b$ and $c + d$ are loop invariant.[1] Although $\boldsymbol{a + b}$ is loop-invariant, it cannot be removed from the loop with code motion alone because it would be executed on the path $[En, O, P, Ex]$, which does not execute $\boldsymbol{a + b}$ in the original program. If the frequency of this path is higher than that of paths that execute $\boldsymbol{a + b}$, the optimization could slow down the program. To avoid counterproductive transformations, the traditional PRE disallows such *unsafe* code motion.

The desired optimization is possible only if the CFG is restructured. The pure-restructuring PRE duplicates each node on which the value of $\boldsymbol{a + b}$ is available strictly partially (i.e., not along all paths). Such a duplication splits partial redundancy into full redundancy and no redundancy [Weg75b, Weg75a, Ste96]. The resulting program is shown in Figure 6.2(b). While $\boldsymbol{a + b}$ is fully optimized, restructuring unnecessarily peeled off the entire loop body.

---

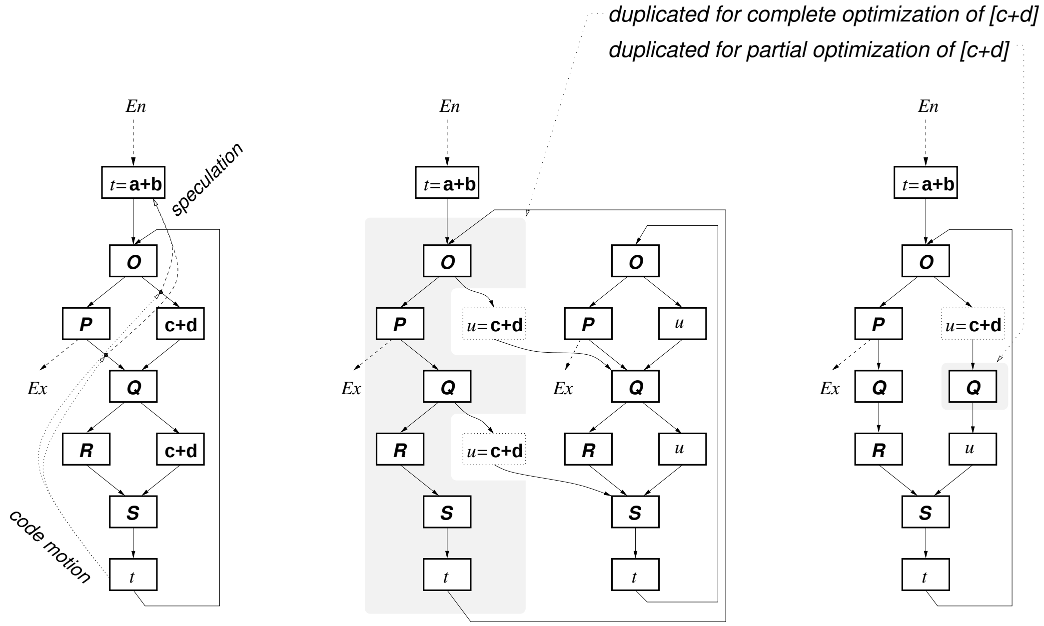
[1] The program in Figure 6.2(a) induces a separable VNG; it can thus be viewed both as a CFG with expressions $a + b$, $c + d$, and as a VNG with computations named $\boldsymbol{a + b}$, $\boldsymbol{c + d}$.



a) the source program.

b) optimization of [a+b]. via restructuring.

c) our optimization of [a+b], via code motion + restructuring.



d) optimization of [a+b], via code motion + speculation.

e) optimization of [c+d], via code motion + restructuring.

f) a trade-off variant of e).

Figure 6.2: PRE through integration of code motion, control flow restructuring, and control speculation.

In contrast, our PRE(MR) algorithm achieves completeness with much smaller duplication scope, see Figure 6.2(c). We apply the more economical code-motion transformation to its full extent, resorting to restructuring merely to enable the desired code motion—in this case, the hoisting of $\boldsymbol{a} + \boldsymbol{b}$ out of the loop. To hoist $\boldsymbol{a} + \boldsymbol{b}$, it is sufficient to isolate the offending path $[En, O, P, Ex]$. The necessary scope of duplication is computed as the CMP region, short for code-motion-preventing region, which is highlighted in Figure 6.2(c). The restructuring is achieved by duplicating the region, after which hoisting can be performed without impairing the offending path. As no opportunities for value reuse of $\boldsymbol{a} + \boldsymbol{b}$ remain, the resulting optimization of $\boldsymbol{a} + \boldsymbol{b}$ is complete. Yet, in contrast to Figure 6.2(b), only two basic blocks have been duplicated.

Profile-guided PRE Using the dynamic count of eliminated computations as the measure of optimization benefit, our profile-guided algorithms trade (some) optimization for (much) code duplication. To reduce code growth, we use profiling, in two different ways:

- To identify infrequently executed paths *with* reuse. Reuse along these paths is not worth the restructuring cost.
- To identify infrequently executed program paths *without* reuse. These paths may be effectively be disregarded when hoisting computations. (Hoisting into these paths constitutes control speculation).

The first profile-guided algorithm, denoted PRE(MS), combines motion with speculation. It does not use restructuring; instead, when code motion fails to compensate partial redundancy into full redundancy, the compensation is done speculatively, by inserting computations onto program paths that did not contain them in the original program.[2] Such insertion is speculative because when these paths are taken, the cost of the insertion will not be amortized by a removed computation.

With speculation, some paths are impaired so that others can be optimized. The net benefit of speculation depends on the difference of frequencies of the improved and the impaired paths, computed from the profile. In our example, if the profile reveals that the offending (impaired) path $[En, O, P, Ex]$ is less frequent than the paths that execute $\boldsymbol{a} + \boldsymbol{b}$, then $\boldsymbol{a} + \boldsymbol{b}$ will be speculatively hoisted from the loop, as shown in Figure 6.2(d).

The next profile-guided PRE algorithm makes the complete PRE(MR) more practical by limiting its code growth. The algorithm, denoted PRE(Mr), restructures selectively. It sacrifices those value-reuse opportunities that are infrequent but require significant code duplication. The PRE(Mr) algorithm is illustrated on the optimization of $\boldsymbol{c} + \boldsymbol{d}$; Figure 6.2(d) serves as the starting point. Figure 6.2(e) shows the (complete) optimization of $\boldsymbol{c} + \boldsymbol{d}$, performed by PRE(MR) by duplicating the shaded CMP region and subsequently performing the code motion of $\boldsymbol{c} + \boldsymbol{d}$. Because the program control flow structure affects the optimization of $\boldsymbol{c} + \boldsymbol{d}$ more adversely that it affects $\boldsymbol{a} + \boldsymbol{b}$ (i.e., the CMP region for $\boldsymbol{c} + \boldsymbol{d}$ is larger than that of $\boldsymbol{a} + \boldsymbol{b}$), more code is duplicated. If the size of duplication outweighs the run-time gains, according to some utility function, the PRE(MR) algorithm can be scaled back to select a smaller set of nodes to duplicate, yielding the PRE(Mr) algorithm. An example of such an incomplete PRE is in Figure 6.2(f), where the size of basic block S is assumed to be greater than would be justified by the frequency of value reuse flowing through S.

Finally, this chapter presents the PRE(Msr) algorithm, which balances all three methods. PRE(Msr) uses restructuring only when speculation cannot be done beneficially. Experiments show that PRE(Msr) is near-complete PRE with very little code growth.

---

[2] Control speculation can be viewed as an *unsafe* version of code motion.

Minimal Do-Until Conversion Do-Until Conversion (DUC) is a common pre-processing transformation for enabling code motion of loop invariants out of while-do loops that would otherwise prohibit the desired code motion [Muc97]. DUC is based on path separation; it converts while-do loops into do-until loops by duplicating a part of the loop body (the loop exit condition). DUC enables optimization of some (but not all) loop invariants. Additionally, its duplication scope is larger than necessary.

It is interesting to note that the PRE(MR) algorithm subsumes and (fully) generalizes DUC. Consider the loop in Figure 6.2(a). A commonly used DUC will fail. A smarter DUC will peel off (unnecessarily) the entire loop iteration, just like the PRE(R)algorithm shown in Figure 6.2(b). No known DUC can enable the hoisting of $c + d$.

In contrast, PRE(MR) produces the necessary do-until conversion: Figure 6.2(c) shows the conversion necessary to hoist $a + b$, and Figure 6.2(e) shows the conversion required by $c + d$. The PRE(MR) conversion is minimal: with any less duplication, the statements could not be hoisted. While Figures 6.2(c,e) show separate conversions for $a + b$ and $c + d$, the two conversions can be naturally composed, as described later in this chapter.

However, we note that, while PRE(MR) is successful in reducing code growth, the profile-guided speculation-based PRE(MS) works much better: our experiments show that it removes nearly all redundancies, without any code growth. Furthermore, PRE(MR) may generate irreducible programs, like the one in Figure 6.2(c). Irreducibility may be produced by other restructuring algorithms, including PRE(R), although it did not manifest itself in Figure 6.2(b). Irreducibility can be corrected; Section 6.5.1 presents a simple, reducibility-preserving version of PRE(MR).

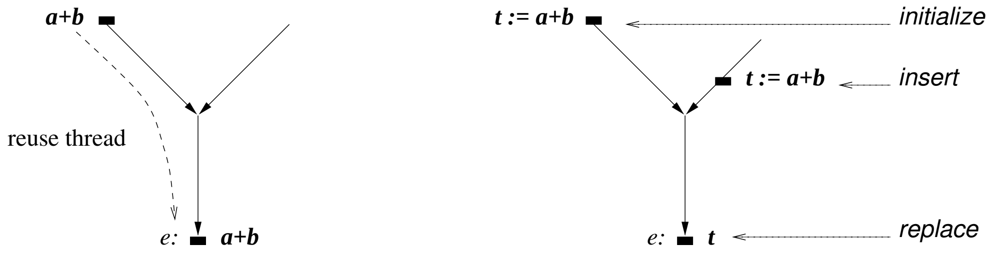
Organization of this chapter Section 6.2 establishes the groundwork by analyzing the limitations of the standard, code-motion PRE and by motivating our solution. The PRE algorithms are presented next. As in Chapter 5, the presentation is simplified by first assuming a separable VNG (Section 6.3), and then a general VNG (Section 6.4). Within each section, the algorithms are divided into profile-independent and profile-guided. Section 6.5 handles various practical issues, such as irreducible graphs and hardware support. Section 6.6 experimentally evaluates the algorithms and Section 6.7 concludes with a summary of the results and discussion of related work.

6.2 Analysis of the Morel-Renviose algorithm

To motivate the approach taken here, this section intuitively describes the principle of the code-motion PRE (denoted PRE(M)), conceptually analyzes when and why it fails, and finally explains the approach for overcoming these limitations.

Figure 6.3 illustrates the principle of PRE(M). The computation $a + b$ is partially redundant because there is a reuse thread leading to it. In CFG terms, there is a control flow path on which $a + b$ is computed without being killed. PRE(M) optimizes $a + b$ by hoisting it away from the reuse thread. Hoisting inserts a computation on incoming non-reuse threads to turn the partial redundancy of $a + b$ into full redundancy. The transformation is completed by initializing a fresh temporary variable t to carry the reused value and replacing the original computation with a reference to t.

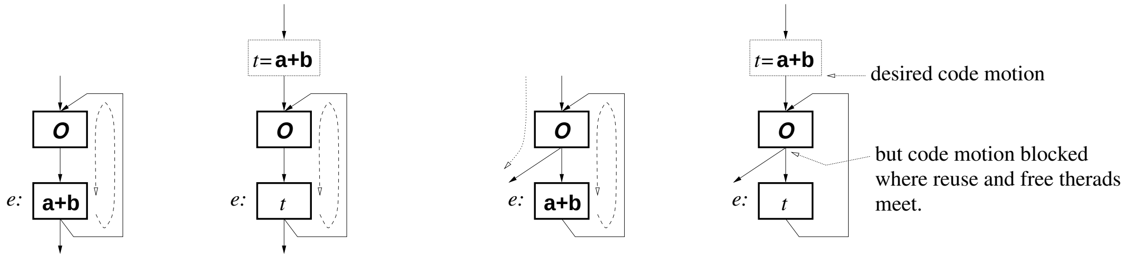
Figure 6.4 illustrates when and why code-motion fails. Figure 6.4(a) shows partially redundant $a + b$ and its reuse thread. In this program, $a + b$ can be hoisted out of the reuse thread, as shown in Figure 6.4(b). However, a slight modification of the program causes the code motion to fail. When the



a) a+b is partially redundant.

b) the optimization of a+b.

Figure 6.3: **The principle of the code-motion PRE transformation.**



a) source program with a reuse thread (dashed arrow).

b) after PRE(M), a+b hoisted from the reuse thread.

c) a version of a), with reuse and free thread.

d) PRE(M) fails, as a+b is inserted onto the free thread.

Figure 6.4: **The reasons for the failure of the code-motion PRE.**

do-until loop is changed into a while-do loop shown in Figure 6.4(c), the desired hoisting fails. Hoisting fails because $a + b$ would have to be placed on a *free* path—one that does not compute the value of $a + b$, and thus cannot reuse its value and amortize the cost of the insertion. To avoid impairment of the free path, PRE(M) disables such *unsafe* code motion; the code motion is blocked before it can enter the free path, as shown in Figure 6.4(d).

This figure also shows the consequences of the unsafe hoisting of $a + b$. The insertion of $a + b$ is a case of *control speculation*, as $a + b$ is executed even when the program will not execute the original computation—we are *speculating* that the *control* will reach the original computation, which would amortize the cost of the speculative insertion. Because PRE(M) is profile-insensitive, it cannot safely and beneficially use speculation.

To summarize the above discussion, PRE based on code motion has two orthogonal deficiencies:

1. *The safe optimization model is too conservative.* To guarantee that the program "never runs slower," the optimization allows improving reuse paths but only when other paths are not impaired.

2. *The program transformation used is not aggressive enough.* Code motion is the only program transformation technique applied; when it is blocked, the optimization opportunity is missed.

Our approach attacks both deficiencies:

1. *Relax the optimization model.* We allow control speculation. To guarantee[3] that the program is improved more than it is impaired, the speculation is profile-driven.

---

[3] As natural for profile-directed optimization, we guarantee that the program is not impaired only for the program input(s) used to generate the profile.

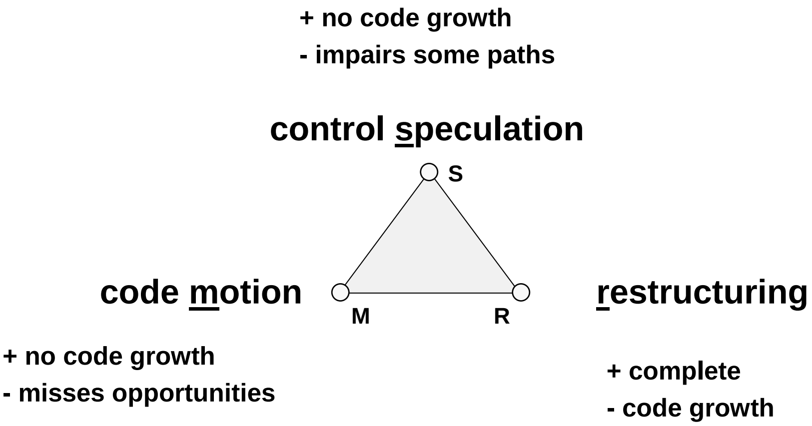


Figure 6.5: **The design space for our PRE algorithm.** The algorithm can use any (combination) of the three program transformation techniques. The algorithm can lie anywhere in the design triangle. It resulting properties depend on how biased it to a constituent technique.

2. *Enhance the program transformation.* We allow the use of control restructuring. Restructuring is less economical than code motion, but we use it only when code motion fails.

Our PRE algorithm thus uses three transformation techniques. These techniques are orthogonal; each is useful in a different situation.

M) *Code motion* does not cause any code growth, but can be blocked before it can fully remove the redundancy.

R) *Control flow restructuring* can alone remove all redundancies but the cost is high code growth.

S) *Control speculation* does not cause any code growth, but it impairs some paths and hence can be counter-productive.

Our PRE algorithms always try[4] to optimize with code motion as much as possible (it is both safe and economical). When code motion fails, then—depending on i) the optimized program and ii) the desired optimization properties—our PRE can employ restructuring or speculation (or both). The optimized program influences PRE with the shape of its reuse threads and its profile. The properties of the optimization depend on which transformations are applied. Figure 6.5 shows the design space of our PRE algorithms, as formed by the three methods. The resulting algorithm can lie anywhere in the design triangle. Its resulting properties depend on how biased it is to the particular transformation methods.

We present five variants of the transformation algorithm, all characterized in Figure 6.6. Two algorithms are profile-independent and three are profile-guided. We also formulate the (profile-independent) PRE(R) algorithm, in order to facilitate an empirical comparison of PRE(MR) with a pure-restructuring PRE approach.

The two goals of a PRE algorithm are as follows:

1. *Completeness:* it is desirable to exploit all reuse detected on the VNG. When the used transformation methods do not permit completeness (as in the case of code-motion), the goal is to maximize the removal of redundancies. Such a "best" optimization is defined differently for profile-independent and for profile-guided algorithms:

---

[4] On a separable VNG, code motion is used to the *maximum* degree possible.

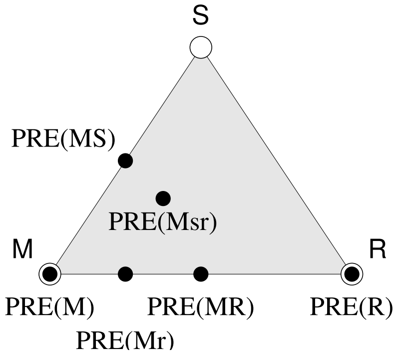


Figure 6.6: **The various variants of PRE algorithms.**

Definition 6.2 (Best static PRE (profile-independent)) A program P' is the *best static optimization* of a program P if no other transformation P'' of P exists (using the allowed methods) such that a path p executes less computations in P'' than in P'. $\square$

Because the measure of optimization quality is each individual program path, the best program may not always exist.

Definition 6.3 (Best dynamic PRE (Profile-guided)) A program P' is the *best dynamic optimization* of a program P if no other transformation P'' of P exists (using the allowed methods) such that the dynamic number of user computations is P'' is less than in P'. $\square$

Because a single measure of quality is used for the entire program, the best transformation must always exist.

2. *Shortest-live ranges:* it is desirable to insert the compensating computations at program points such that the live ranges of the inserted temporaries are as short as possible. More precisely, given the set of *best* optimizations, the live-range optimal is one in which each live range is no longer than the corresponding range in any other best program.

For each PRE algorithm developed in this chapter, a different optimality goal is apropriate, depending on the transformation methods applied (M, R, S), and depending whether profiling is used.

PRE(M): Find best *static* optimization a) without changing the shape of the CFG and b) by moving only the optimized computations (not any other computations in the program).

PRE(MR): Find a complete optimization that minimizes the number of duplicated CFG nodes.

PRE(R): Same optimality goal as PRE(MR).

PRE(MS): Same optimality goal as PRE(M), but in terms of best *dynamic* optimization.

PRE(Mr): Find a dynamic optimization that maximizes a utility function. The utility function reflects a ratio of code-growth cost with the dynamic optimization benefit.

PRE(Msr): Same optimality goal as PRE(Mr), but allow speculation.

6.3 PRE for a separable VNG

As in the previous chapter, the algorithms are presented first for separable VNGs and then for general VNGs. On a separable VNG, each value thread uses only one name throughout the entire program. Such simple VNGs require simpler algorithms and achieve optimal optimizations, typically at polynomial-time cost. In contrast, for general VNGs, we either do not have an optimal transformation algorithm, or the transformation is intractable. This section is devoted to the separable VNGs. Section 6.4 presents extensions needed for generality, focusing on the PRE(MS) algorithm, which appears to be the most useful in practice.

6.3.1 Profile-independent PRE

We will first consider the profile-independent variants of PRE. These algorithms guarantee that, whatever the path frequencies, the optimized program will never execute more computations than the unoptimized program. Thus, these algorithms operate within the *safe optimization model* in which no program path can be impaired. To achieve completeness without impairing any path, code motion obstacles must be overcome with restructuring. Our goal, then, is to minimize the restructuring cost, i.e., the code growth measured as the number of duplicated CFG nodes.

In turn, this section presents PRE(MR), PRE(M), and PRE(R). The first algorithm is the complete PRE that minimizes code duplication. The second algorithm uses our abstractions to derive an intuitive formulation of the optimal code-motion PRE [KRS94b]. The last algorithm is the restructuring-only PRE, presented to serve as a reference point for our experiments.

6.3.1.1 PRE(MR): Code motion + restructuring

PRE(MR) integrates code motion and control flow restructuring. To reduce code growth, restructuring is only a secondary transformation exploited by PRE(MR). It is used merely to enable hoisting across CFG nodes that prevent the desired code motion. The central idea of PRE(MR) algorithm is to identify the smallest set of motion-blocking nodes and duplicate them, restructuring the CFG. After the restructuring, the motion obstacles disappear.

To identify the offending nodes, we determine a predicate *Prevented*$(\boldsymbol{n})$ that characterizes whether a VNG node $\boldsymbol{n}$ blocks the desired code motion. The predicate is based on the solutions to the standard dataflow problems of anticipability and availability (see Definition 4.2). The problems are computed on the *Must-May-No* lattice defined in Section 4.5.

The *Prevented* predicate is derived as follows. A computation is partially redundant if its value is computed on some incoming control flow path by a previous computation. Code motion eliminates the (partial) redundancy by hoisting the redundant computation along all paths until it reaches an edge where the reused value is redundant along either *all* paths or *no* paths. In the former case, the computation is removed; in the latter, it is inserted, to make the original computation fully redundant (recall Figure 6.3). Unfortunately, code motion may be blocked before such edges are reached by nodes that prevent the code motion. These nodes are characterized by the following set of conditions:

1. Hoisting of a computation of $\boldsymbol{e}$ from the exit to the entry of a VNG node $\boldsymbol{n} = (n, \boldsymbol{e})$ is *desired* when an opprtunity for value-reuse exists, which is true when both of the following conditions hold:
   a) The value $\boldsymbol{e}$ is computed on some, but not all, control flow paths leading to n. In such a case, hoisting must continue across n to move the computation to CFG edges where it is either always

or never redundant. Such a situation is true iff

$$\mathit{AVAIL}_{\mathit{out}}[\boldsymbol{n}] = \mathsf{May}.$$

b) The value $\boldsymbol{e}$ is consumed by a user node on some control flow path p emanating from n, i.e., there is a computation to be optimized (and hoisted from) the path p. Such a situation exists iff

$$\mathit{ANTIC}_{\mathit{out}}[\boldsymbol{n}] \neq \mathsf{No}.$$

Hence,

$$\mathit{Desired}[\boldsymbol{n}] \Leftrightarrow \mathit{AVAIL}_{\mathit{out}}[\boldsymbol{n}] = \mathsf{May} \wedge \mathit{ANTIC}_{\mathit{out}}[\boldsymbol{n}] \neq \mathsf{No}.$$

2. Hoisting of a computation of $\boldsymbol{e}$ across $\boldsymbol{n}$ is *disabled* when the computation would impair some control flow path, because the path cannot amortize the computation, as the following two conditions elaborate:

c) The hoisted computation is not fully redundant and hence there is an incoming path from which it cannot be removed, which is true iff

$$\mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] \neq \mathsf{Must}, \text{ and}$$

d) The hoisted computation cannot make other computation redundant, on some path leaving n,

$$\mathit{ANTIC}_{\mathit{in}}[\boldsymbol{n}] \neq \mathsf{Must}.$$

Hence,

$$\mathit{Disabled}[\boldsymbol{n}] \Leftrightarrow \mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] \neq \mathsf{Must} \wedge \mathit{ANTIC}_{\mathit{in}}[\boldsymbol{n}] \neq \mathsf{Must}.$$

A node n prevents the code motion for e when the motion is both desired and disabled. By way of conjunction, we get the code motion-preventing condition:

$$\begin{aligned}\mathit{Prevented}[\boldsymbol{n}] &\Leftrightarrow \mathit{Desired}[\boldsymbol{n}] \wedge \mathit{Disabled}[\boldsymbol{n}]\\ &\Leftrightarrow \mathit{AVAIL}_{\mathit{out}}[\boldsymbol{n}] = \mathsf{May} \wedge \mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] \neq \mathsf{Must} \;\wedge\\ &\qquad \mathit{ANTIC}_{\mathit{out}}[\boldsymbol{n}] \neq \mathsf{No} \wedge \mathit{ANTIC}_{\mathit{in}}[\boldsymbol{n}] \neq \mathsf{Must}\end{aligned}$$

Because $\mathit{AVAIL}_{\mathit{out}}[\boldsymbol{n}] = \mathsf{May}$, node $\boldsymbol{n}$ is neither a generator nor a kill node ($\boldsymbol{n} \notin \boldsymbol{D}, \boldsymbol{n} \notin \boldsymbol{K}$, see Definition 4.1), from which one can show that

$$\mathit{Prevented}[\boldsymbol{n}] \quad \Leftrightarrow \quad \mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] = \mathsf{May} \wedge \mathit{ANTIC}_{\mathit{in}}[\boldsymbol{n}] = \mathsf{May}.$$

The predicate $\mathit{Prevented}[\boldsymbol{n}]$ characterizes the smallest set of nodes that block the code motion. For the desired code motion to be enabled, the blocking condition of these nodes must be "removed" via restructuring. The *Code-Motion-Preventing (CMP) Region* is the set of such nodes.

Definition 6.4 (CMP Region) Let $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{E}, \textbf{start}, \textbf{end})$ be a value name graph. The *CMP region* of $\boldsymbol{G}$, denoted $\boldsymbol{G}^{\boxtimes}$, is a subgraph of $\boldsymbol{G}$ such that $\boldsymbol{G}^{\boxtimes} = (\boldsymbol{N}^{\boxtimes}, \boldsymbol{E}^{\boxtimes}, \boldsymbol{I}, \boldsymbol{O})$, where

$$\begin{aligned}\boldsymbol{N}^{\boxtimes} &=_{df} \{\boldsymbol{n} \in \boldsymbol{N} \mid \mathit{Prevented}[\boldsymbol{n}]\} && \text{nodes}\\ \boldsymbol{E}^{\boxtimes} &=_{df} \boldsymbol{E} \cap (\boldsymbol{N}^{\boxtimes} \times \boldsymbol{N}^{\boxtimes}) && \text{edges}\\ \boldsymbol{I} &=_{df} \{(\boldsymbol{n}, \boldsymbol{m}) \in \boldsymbol{E} \mid \boldsymbol{n} \notin \boldsymbol{N}^{\boxtimes} \wedge \boldsymbol{m} \in \boldsymbol{N}^{\boxtimes}\} && \text{entry edges}\\ \boldsymbol{O} &=_{df} \{(\boldsymbol{n}, \boldsymbol{m}) \in \boldsymbol{E} \mid \boldsymbol{n} \in \boldsymbol{N}^{\boxtimes} \wedge \boldsymbol{m} \notin \boldsymbol{N}^{\boxtimes}\} && \text{exit edges.} \quad \square\end{aligned}$$

The entry and exit edges are used to attach a copy of the CMP region to the rest of the program during restructuring. The set of CMP *entry* edges, $\boldsymbol{I}$, can be factored into two sets: entry edges on which the value is available along all incoming paths or no incoming paths. These two sets are denoted $\boldsymbol{I}^{\mathbf{M}}$ and $\boldsymbol{I}^{\mathbf{N}}$:

$$\forall(\boldsymbol{m}, \boldsymbol{n}) \in \boldsymbol{I}^{\mathrm{M}} \;.\; \mathit{AVAIL}_{\mathit{out}}[\boldsymbol{m}] = \textsf{\textit{Must}} \qquad \text{called } \textsf{\textit{Must}} \text{ entry edges,}$$
$$\forall(\boldsymbol{m}, \boldsymbol{n}) \in \boldsymbol{I}^{\mathrm{N}} \;.\; \mathit{AVAIL}_{\mathit{out}}[\boldsymbol{m}] = \textsf{\textit{No}} \qquad \text{called } \textsf{\textit{No}} \text{ entry edges.}$$

Similarly, *exit* edges can be factored into those on which the value is consumed (anticipated) along either all outgoing paths or no outgoing paths. These two sets are denoted $\boldsymbol{O}^{\mathbf{M}}$ and $\boldsymbol{O}^{\mathbf{N}}$:

$$\forall(\boldsymbol{m}, \boldsymbol{n}) \in \boldsymbol{O}^{\mathrm{M}} \;.\; \mathit{ANTIC}_{\mathit{in}}[\boldsymbol{n}] = \textsf{\textit{Must}} \qquad \text{called } \textsf{\textit{Must}} \text{ exit edges,}$$
$$\forall(\boldsymbol{m}, \boldsymbol{n}) \in \boldsymbol{O}^{\mathrm{N}} \;.\; \mathit{ANTIC}_{\mathit{in}}[\boldsymbol{n}] = \textsf{\textit{No}} \qquad \text{called } \textsf{\textit{No}} \text{ exit edges.}$$

Observe that each *reuse* thread must enter the CMP region through a *Must* entry and leave through a *Must* exit. Similarly, a *free* thread must enter through a *No* entry and leave through a *No* exit.

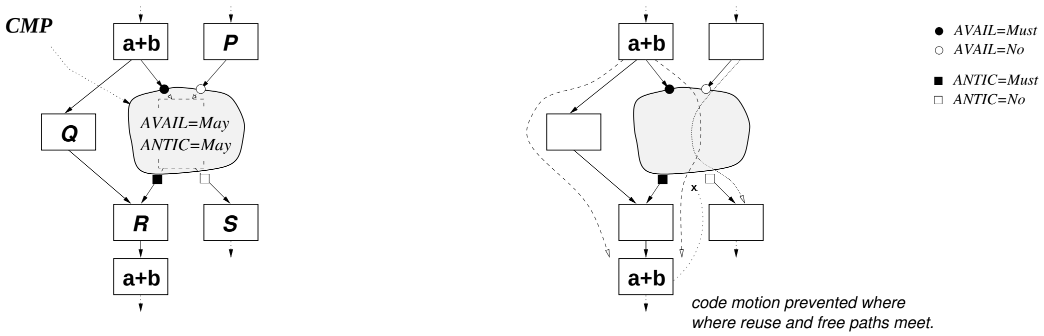
To explain how PRE(MR) removes obstacles presented by the CMP region, let us assume for simplicity that the VNG contains only one name, $\boldsymbol{a} + \boldsymbol{b}$, which allows us to view the VNG as a CFG (i.e., threads coincide with paths). We will deal with multiple names once the central idea is clear. In PRE(MR), the obstacles of the CMP region are removed by duplicating the entire region, as illustrated in Figure 6.7. The goal of duplicating the CMP region is to factor the *May*-availability that holds in the original region into *Must*-availability and *No*-availability, each holding in one copy of the region. After *May*-availability no longer holds in the region, the paths with *Must*- and *No*-availability have been separated.

To see why *May*-availability can be split into *Must*- and *No*-availability in the two respective region copies, observe that a) no region entry edge is *May*-available, and b) the solution of availability within the region depends solely on solutions at entry edges, because the value is neither computed nor killed within the region. Hence, the desired factoring can be carried out by attaching to each region copy the subset of either *Must* or *No* entry edges, which separates the offending paths, as shown in Figure 6.7(b). The exit edges are duplicated and attached to both copies of the region. After the CMP region is duplicated, the condition *Prevented* is false on each node, enabling the desired code motion, as shown in Figure 6.7(c).

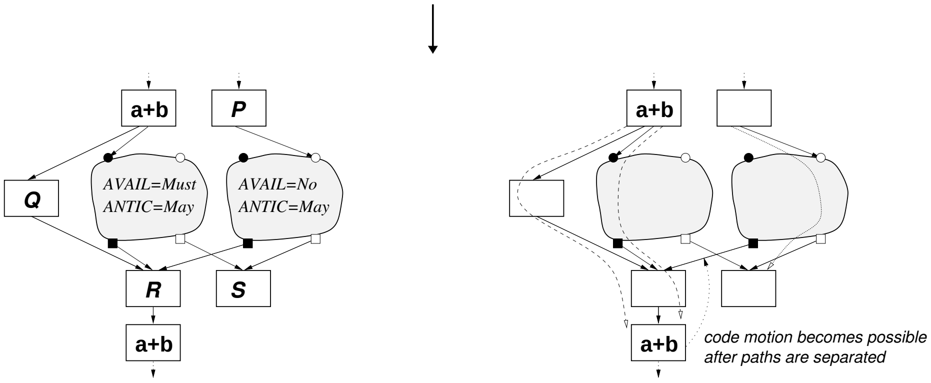
Let us remove the restriction that the VNG has a single value name. It is now possible that a CFG node n prevents code motion of multiple names, denoted $\boldsymbol{e_1}, \ldots, \boldsymbol{e_k}$, where $k > 1$. Such a node (and its duplicates) may need to be duplicated up to k-times, producing up to 2^k copies of n. The first two copies of n, denoted $n_{\textsf{\textit{Must}}}$ and $n_{\textsf{\textit{No}}}$, separate the reuse/free paths for $\boldsymbol{e_1}$. After enabling the hoisting of $\boldsymbol{e}_1$, the paths for the remaining names may still meet at $n_{\textsf{\textit{Must}}}$ and/or $n_{\textsf{\textit{No}}}$. Further duplication of these copies separates both $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$ paths, producing nodes $n_{\textsf{\textit{Must}},\textsf{\textit{Must}}}, n_{\textsf{\textit{Must}},\textsf{\textit{No}}}, n_{\textsf{\textit{No}},\textsf{\textit{Must}}}, n_{\textsf{\textit{No}},\textsf{\textit{No}}}$. This process continues until there is a copy of n for each subset of $\boldsymbol{e_1}, \ldots, \boldsymbol{e_k}$.

Some of the node duplicates may not be necessary. For example, if reuse paths for $\boldsymbol{e_1}$ match those of $\boldsymbol{e_2}$, then nodes $n_{\textsf{\textit{Must}},\textsf{\textit{No}}}, n_{\textsf{\textit{No}},\textsf{\textit{Must}}}$ need not be created. There are two alternatives how to restructure the program and create only the needed nodes. The first is to restructure along name $\boldsymbol{e}_1$ and then recompute *AVAIL*. If paths for some other name $\boldsymbol{e}_i$ have been split in the process, its *Prevented* condition will disappear. The restructuring then continues with names whose *Prevented* predicate still holds.

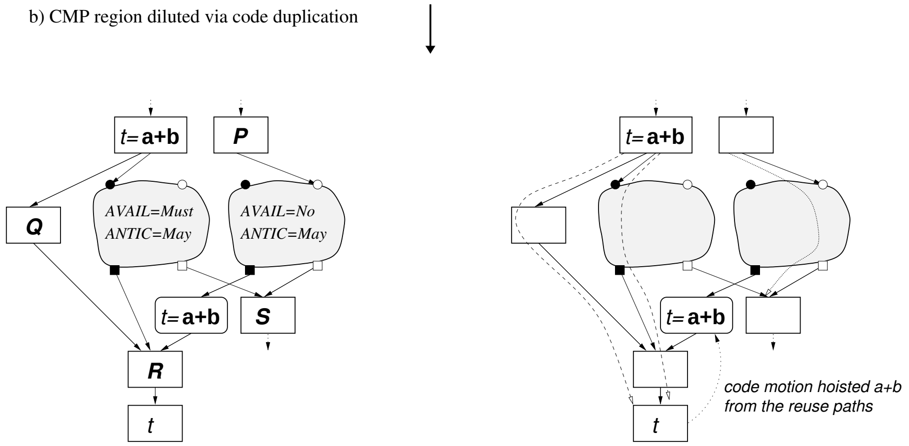
The PRE(MR) algorithm presented uses the second alternative: to avoid recomputing the *AVAIL* solution, it creates all 2^k copies of each node, to perform restructuring for all names simultaneously. Spurious copies will be manifested as unreachable CFG nodes and will be removed in a cleanup phase. The PRE(MR) algorithm has the following three steps:



a) code motion prevented by CMP region



b) CMP region diluted via code duplication



c) complete PRE of [a+b]

Figure 6.7: **Removing obstacles to code motion via restructuring.**

1. *Compute availability of generators and anticipability of users.* The computation of the two problems is described in Chapter 4.
2. ***R:*** *Remove the CMP region via control flow restructuring.* The algorithm, shown in Figure 6.10, has three parts: node duplication (lines 1–12), edge adjustment (lines 13-17), and removal of unreachable nodes (lines 18–24). The algorithm, as presented, builds a restructured CFG G^r from the original CFG G. However, the algorithm can be easily altered to modify G in-place.

   The function *CMP*(n) maps each CFG node into the set of value names that belong to the CMP region at n. This function directs how many copies of each node are created at line 3: one copy of each CFG node is created for each combination of *Must* or *No* paths that must be separated at node n, that is, one copy for each subset of *CMP*(n). Note that if *CMP*(n) is an empty set, only one copy is created, denoted $n_\emptyset$, which is the case for the *start* and *end* CFG nodes. Each node is duplicated together with its dataflow solutions *AVAIL* and *ANTIC*. Lines 6 and 9 effectively separate the reuse and free paths, by adjusting the *AVAIL* dataflow solution.

   Given the duplicated nodes with adjusted *AVAIL* solutions, line 15 places edges between nodes with compatible *AVAIL* solutions, creating the restructured CFG. Line 14 tests whether two CFG nodes have compatible solutions. Essentially, the test prevents a *Must*-available node to be connected with a *No*-available node, ensuring that *Must* and *No* paths are separated. Note that $AVAIL_{out}[m_i]$ is the vector of *AVAIL* solutions for all symbolic names from $\boldsymbol{S}$. The lattice partial order $\leq$ was defined in Definition 4.5. The function $\phi^{-1} : (E \times 2^{\boldsymbol{S}}) \rightarrow 2^{\boldsymbol{S}}$ permutes the names in the dataflow vector, to reflect how the value names change at ϕ nodes. At separable VNGs, this function is an identity.

   Finally, unreachable nodes are detected using a work list algorithm, by propagating control flow from the $start_\emptyset$, the start node of G^r.

Figure 6.8: **The PRE(MR) algorithm.**

Theorem 6.1 (Completeness of PRE(MR)) PRE(MR) is optimal in that it minimizes the number of computations on each path.
Proof. First, each original computation is replaced with a temporary (Eq. 6.1). Second, no computation is inserted where its value is available along any incoming path. Hence, no user computation in the optimized program is partially redundant. □

Within the domain of the Morel and Renviose code-motion transformation, where PRE is accomplished by hoisting optimization candidates (but not hoisting any other statements) [MR79], PRE(MR) achieves minimum code growth.[5] This follows from the fact that after CMP restructuring, no node is unreachable and no program node can be removed or merged with some other node without destroying any value reuse.

---

[5] Outside this domain, further code growth reduction is possible, for example, by moving some instructions out of the CMP prior to its duplication.

3. ***M:*** *Optimize the program via code motion.* The code motion transformation is carried out by replacing each generator computation $\boldsymbol{n} = (n, \boldsymbol{e}) \in D$ with a temporary variable t_e (eq. 6.1). The temporary is initialized with a computation, which can be inserted at three different points:

   Eq. 6.2: at each *No*-available edge that sinks onto a *May*-available node. This insertion will compensate the partial redundancy into the full redundancy. The insertion edge must also be *Must*-anticipated to verify that an optimized computation was hoisted to the insertion edge.

   Eq. 6.3: before each user computation that is not partially redundant.

   Eq. 6.4: before each *generator* computation that is not a *user* computation (e.g., a store instruction in redundant load optimization). Recall that such computations cannot reuse the value although they can generate it.

   The last two cases initialize temporaries for computations that have been replaced but have not been hoisted.

$$\mathit{Replace}(n, \boldsymbol{e}) \quad \Leftrightarrow \quad (n, \boldsymbol{e}) \in \boldsymbol{D} \tag{6.1}$$

$$\mathit{Insert}((n, m)) \quad \Leftrightarrow \quad \mathit{AVAIL}_{\mathit{out}}[\boldsymbol{n}] = \mathsf{No} \wedge \mathit{AVAIL}_{\mathit{in}}[\boldsymbol{m}] = \mathsf{May} \wedge \mathit{ANTIC}_{\mathit{in}}[\boldsymbol{m}] = \mathsf{Must} \tag{6.2}$$

$$\vee\, \mathit{AVAIL}_{\mathit{in}}[\boldsymbol{m}] = \mathsf{No} \wedge \boldsymbol{m} \in \boldsymbol{U} \tag{6.3}$$

$$\vee\, \boldsymbol{m} \in \boldsymbol{D} \setminus \boldsymbol{U} \tag{6.4}$$

Figure 6.9: **The PRE(MR) algorithm, continued.**

Algorithm complexity. The cost of restructuring (Step 2) dominates the dataflow analysis (Step 1) and the code motion (Step 3). Due to duplication, the size of N^r may be $O(2^{\boldsymbol{S}})$ times larger than N, where $\boldsymbol{S}$ is the number of VNG symbolic names. In practice, the algorithm was significantly slowed down only on some very large procedures (of size more than 1000 nodes). This explosion is to be expected, due to the exponential number of possible program paths. Yet, PRE(MR) is very successful in reducing code growth. As will be shown in Section 6.3.1.4, its code growth is less than half of that caused by PRE(R).

6.3.1.2 PRE(M): Code motion

Besides enabling an efficient complete PRE, the abstraction of the CMP region also facilitates an intuitive formulation of an optimal code-motion PRE. Recall from Section 6.2 that PRE(M) is optimal when

a) it achieves *best static* optimization, i.e., it removes all redundancies that can be optimized with code motion alone, and

b) it is live-range optimal, i.e., the live ranges of inserted temporary variables are as short as possible.

Existing optimal algorithms [DS93, KRS94b, CCK$^+$97] work in two phases.

```
Input:
    control flow graph $G = (N, E, start, end)$
    value name graph $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{E}, \boldsymbol{S})$
    CMP region $\boldsymbol{G}^{\boxtimes} = (\boldsymbol{N}^{\boxtimes}, \boldsymbol{E}^{\boxtimes}, IN, OUT)$
Output:
    CFG $G^r = (N^r, E^r, start_\emptyset, end_\emptyset)$
Auxiliaries:
    $CMP : N \to 2^{\boldsymbol{S}}, CMP(n) = \{\boldsymbol{e} | (n, \boldsymbol{e}) \in \boldsymbol{N}^{\boxtimes}\}$

    Create $N^r$: duplicate CMP nodes and adjust their dataflow solutions
    for each $n \in N$ do
        for each $\boldsymbol{C} \in 2^{CMP(n)}$ do
            copy the CFG node, including its dataflow solutions AVAIL, ANTIC
            make a copy of $n$, denoted $n'$
            add $n'$ to $N^r$
            for each $\boldsymbol{e} \in \boldsymbol{C}$ do
                $AVAIL_{in}[(n', \boldsymbol{e})] := AVAIL_{out}[(n', \boldsymbol{e})] := Must$
            end do
            for each $\boldsymbol{e} \in CMP(n) \setminus \boldsymbol{C}$ do
                $AVAIL_{in}[(n', \boldsymbol{e})] := AVAIL_{out}[(n', \boldsymbol{e})] := No$
            end do
        end do
    end do

    Create $E^r$: connect the (new) nodes in $N^r$
    for each pair $(m_i, n_j)$ such that $(m, n) \in E$ do
        if $AVAIL_{out}[m_i] \leq^{\boldsymbol{S}} \phi^{-1}_{(m,n)}(AVAIL_{in}[n_j])$ then
            add $(m_i, n_j)$ to $E^r$
        end if
    end for

    Remove from $G^r$ nodes unreachable from $start_\emptyset$, the start node of $G^r$.
    add $start_\emptyset$ to $Reachable$; add $start_\emptyset$ to $worklist$
    while $worklist$ is not empty do
        remove a node $n$ from $worklist$
        $A := \{m | (n, m) \in E^r \wedge m \notin Reachable\}$
        $Reachable := Reachable \cup A; worklist := worklist \cup A$
    end while
    $N^r := Reachable; E^r := E^r \cap (Reachable \times Reachable)$
```

Figure 6.10: **The R phase of PRE(MR):** remove the CMP region via control flow restructuring.

1. Partially redundant computations are hoisted as high as permitted by code-motion obstacles. Such a transformation produces the best static optimization.

2. Hoisted computations are rolled back as low as possible without undoing the optimization. Such a transformation results in the best static optimization *with* the shortest live ranges of temporaries.

In [KRS92, KRS94b], the two phases are called *busy* code motion and *lazy* code motion, respectively.

The two-phase approach complicates the comprehension of the algorithm, for two reasons. First, being an artifact of the dataflow equations, the busy code motion looks different (and hence counter-intuitive) than the desired final transformation. Second, although the lazy code motion is a separate, corrective step, it has to be considered in concert with the first phase to be understood.

In contrast, our PRE(MR) algorithm operates in a single phase. It retains the single-phase property even when its restructuring component is disabled, which makes it a PRE(M) algorithm. Such a constraint provides a natural formulation of an optimal code motion. Most importantly, PRE(M) is derived from the same framework as our other PRE algorithms (i.e., it is based on the CMP region and the *Must*-*May*-*No* lattice), and hence it can be inexpensively implemented as a special case of a more general PRE algorithm. We also note that the computational cost of our PRE(M) is equal to that of the two-phase algorithms.

First, we explain why PRE(MR) is a single-phase algorithm. Next, we show how to restrict PRE(MR) to PRE(M) without losing the intuitive single-phase property. The two-phase PRE hoists aggressively until code motion is blocked. In other words, hoisting continues upwards even after the hoisted computation has been hoisted from its reuse path. Such a blind hoisting overshoots, and hence is corrected by the second phase, which rolls back (delays) the hoisted computations .

In contrast, the PRE(MR) algorithm hoists only as far as necessary. The hoisting stops as soon as the hoisted computation leaves its reuse path. The hoisting is stopped *early*, thanks to our more expressive lattice: hoisting continues while *AVAIL* is *May* (still on a reuse path), but stops as soon as *AVAIL* turns to *No* (no longer on a reuse path) or *Must* (on a reuse path but fully redundant). Stopping early results in shortest live ranges, avoiding the need for roll-back of insertions.

To derive PRE(M) from PRE(MR), one needs to do more than disable restructuring of the CMP region. Our PRE(M) is based on the observation that the reuse flowing across the CMP region is exactly the reuse that code motion cannot exploit. The critical issue thus is how to prevent hoisting into paths that emanate from exits of a CMP region (such hoisting is precisely what would have to be rolled back). It is sufficient to ensure that *AVAIL* on a CMP exit is always *No*, indicating that no reuse is available to code motion across the region. This is the only modification of PRE(MR) needed to derive PRE(M).

The PRE(M) algorithm uses the dataflow property $AVAIL^M$, which holds when a value exploitable by code motion is available at a node. The *ANTIC* property is as in PRE(MR).

Definition 6.5 (Availability of Generators Accessible to Code Motion) The M-availability of e at the entry of n w.r.t. the incoming paths is defined as:

$$AVAIL_{in}^{M}[(n, e)] = \begin{cases} \textit{Must} & \text{if } e \text{ is available along } \textit{all} \text{ paths from } \textit{start} \text{ to } n, \\ \textit{May} & \text{if } e \text{ is available along } \textit{some} \text{ paths from } \textit{start} \text{ to } n, \\ \textit{No} & \text{if } e \text{ is available only along paths that contain a node from a CMP region.} \end{cases}$$

□

It can be shown that $AVAIL_{in}^{M}[n] \in \{\textit{Must}, \textit{May}\}$ iff all reuse flowing to n can be removed with code motion. To summarize, PRE(M) has two steps: dataflow analysis and code motion.

1. *Compute availability of generators, anticipability of users.* We compute the standard anticipability, as in PRE(MR). Availability accounts for the killing effect of CMP nodes, which are treated as kill nodes. The flow function for the *AVAIL*^{M} problem follows. The third line kills reuse if $\boldsymbol{n}$ is a CMP node.

$$f_n(x) \;=\; \begin{cases} \textsf{\textit{Must}} & \text{if } \boldsymbol{n} \in \boldsymbol{D}, \qquad \text{(value generated)} \\ \textsf{\textit{No}} & \text{if } \boldsymbol{n} \in \boldsymbol{K}, \qquad \text{(value killed)} \\ \textsf{\textit{No}} & \text{if } \textit{Prevented}[\boldsymbol{n}], \qquad \text{(CMP region)} \\ x & \text{otherwise.} \end{cases}$$

2. ***M)*** *Optimize the program via code motion.* Given a maximal fixed point solution to *ANTIC* and *AVAIL*^{M}, the transformation proceeds as in PRE(MR), except that *AVAIL*^{M} replaces *AVAIL*.

Figure 6.11: **The PRE(M) algorithm.**

Theorem 6.2 (Computational optimality of PRE(M)) Given the restriction of immutable shape of the control flow graph, PRE(M) achieves optimization that is *best* under code motion.

Proof outline. The proof is based on showing that any reuse that remains after PRE(M) requires crossing the CMP region and hence code motion would be blocked. □

To understand the concept behind the PRE(M) algorithm, we need only to understand the definition of *AVAIL*^{M} and its flow function.

While PRE(M) is easy to understand, it requires computation of three dataflow problems (*ANTIC*, *AVAIL*, and *AVAIL*^{M}), each requiring two bits in dataflow vectors per value name. In contrast, equivalent two-phase algorithms compute three problems of one bit each (availability, anticipability, delayability). Next, we show how to compute PRE(M) with the same efficiency. The efficiency is improved by computing not *AVAIL*^{M} but a "weaker" predicate that requires only one bit.

To compute *AVAIL*^{M}, we need the solution of *AVAIL* and *ANTIC*, which are required by the predicate *Prevented*. To avoid computing both *AVAIL* and *AVAIL*^{M}, it is tempting to combine the detection of CMP nodes and their killing effects into a single dataflow problem. The following transfer function for *AVAIL*^{M} does exactly that; the *Prevented* predicate does not use the value of *AVAIL*, but instead it uses the value of *AVAIL*^{M} that is being computed (*AVAIL*^{M} appears as the value x, in the third line).

$$f'_n(x) \;=\; \begin{cases} \textsf{\textit{Must}} & \text{if } \boldsymbol{n} \in \boldsymbol{D}, \\ \textsf{\textit{No}} & \text{if } \boldsymbol{n} \in \boldsymbol{K}, \\ \textsf{\textit{No}} & \text{if } x = \textsf{\textit{May}} \wedge \textit{ANTIC}_{in}[\boldsymbol{n}] = \textsf{\textit{May}}, \qquad \textit{Prevented}[\boldsymbol{n}] \\ x & \text{otherwise.} \end{cases}$$

It can be shown that the maximal fixed point solutions for f_n and f'_n are both equal to *AVAIL*^{M}. Unfortunately, f'_n is not monotone: given $\textit{ANTIC}_{in}[\boldsymbol{n}] = \textsf{\textit{May}}, x_1 = \textsf{\textit{Must}}, x_2 = \textsf{\textit{May}}$, we have $x_1 \sqsupseteq x_2$ but $f_n(x_1) = \textsf{\textit{Must}} \not\sqsupseteq f_n(x_2) = \textsf{\textit{No}}$. Therefore, an iterative dataflow solver may produce (conservatively) imprecise values of *AVAIL*^{M} [KU77], i.e., we may obtain the solution *May* instead of *No*.

Instead, we define another availability property, denoted AVAIL_w^M, which is "weaker" than AVAIL^M but sufficient for computing the *Insert* predicate correctly. AVAIL_w^M is defined as follows:

$$\mathit{AVAIL}_w^M = \textsf{Must} \quad \Rightarrow_{df} \quad \mathit{AVAIL}^M = \textsf{Must} \tag{6.5}$$

$$\mathit{AVAIL}_w^M = \textsf{May} \quad \Rightarrow_{df} \quad \mathit{AVAIL}^M = \textsf{May} \tag{6.6}$$

$$\mathit{AVAIL}_w^M = \textsf{No} \quad \Rightarrow_{df} \quad \mathit{AVAIL}^M = \textsf{No} \vee \mathit{ANTIC} = \textsf{No} \tag{6.7}$$

It can be shown that AVAIL_w^M can be used instead of *AVAIL* in the predicate *Insert*, without changing its value.

The reason for the condition 6.7 is that AVAIL_w^M can be computed efficiently using three Boolean (i.e., one-bit) problems: $\mathbf{AN}_{all}$, $\mathbf{AV}_{all}$, and $\mathbf{AV}_{some}$, defined as follows:

$$\begin{aligned} \mathbf{AN}_{all} \quad &\Leftrightarrow_{df} \quad \mathit{ANTIC} = \textsf{Must} \\ \mathbf{AV}_{all} \quad &\Leftrightarrow_{df} \quad \mathit{AVAIL}_w^M = \textsf{Must} \\ \mathbf{AV}_{some} \quad &\Leftrightarrow_{df} \quad \mathit{AVAIL}_w^M \neq \textsf{No} \end{aligned}$$

Clearly, the pair $(\mathbf{AV}_{all}, \mathbf{AV}_{some})$ expresses the solution of AVAIL^M. The true and false values are denoted $\top$ and $\bot$, respectively.

$$\begin{aligned} \mathit{AVAIL}^M = \textsf{Must} \quad &\Leftrightarrow \quad (\mathbf{AV}_{all}, \mathbf{AV}_{some}) = (\top, \top) \\ \mathit{AVAIL}^M = \textsf{May} \quad &\Leftrightarrow \quad (\mathbf{AV}_{all}, \mathbf{AV}_{some}) = (\bot, \top) \\ \mathit{AVAIL}^M = \textsf{No} \quad &\Leftrightarrow \quad (\mathbf{AV}_{all}, \mathbf{AV}_{some}) = (\bot, \bot) \end{aligned}$$

To compute $\mathbf{AN}_{all}$, $\mathbf{AV}_{all}$, and $\mathbf{AV}_{some}$, we observe that $\mathbf{AN}_{all}$ is the well-known (Boolean) must-anticipability property, which holds when the value is anticipated along all outgoing paths. Similarly, $\mathbf{AV}_{all}$ is the must-availability, which follows from $\mathit{AVAIL}_w^M = \textsf{Must} \Leftrightarrow \mathit{AVAIL} = \textsf{Must}$, which in turn follows from the fact that no *Must*-available node is in the CMP region.

Once $\mathbf{AN}_{all}$ and $\mathbf{AV}_{all}$ are solved, $\mathbf{AV}_{some}$ is computed with the following transfer function.

$$f_{\mathbf{AV}_{some}}(\boldsymbol{n})(x) \quad = \quad \begin{cases} \top & \text{if } \boldsymbol{n} \in \boldsymbol{D}, \\ \bot & \text{if } \boldsymbol{n} \in \boldsymbol{K}, \\ \bot & \text{if } \mathbf{AV}_{all} = \bot \wedge \mathbf{AN}_{all} = \bot, \qquad\qquad \mathit{Prevented}_{\boldsymbol{w}}[\boldsymbol{n}] \\ x & \text{otherwise.} \end{cases}$$

It can be shown that $f_{\mathbf{AV}_{some}}$ indeed computes the maximal fixed point for $\mathbf{AV}_{some}$ ($f_{\mathbf{AV}_{some}}$ is false iff all incoming paths either have no reuse or are not *Must*-anticipated, or they kill the reuse with $\boldsymbol{K}$ or a CMP node).

Intuitively, the condition $\mathit{Prevented}_w$ is weaker than *Prevented* (i.e., it is true more often)

$$\begin{aligned} \mathit{Prevented}_w \quad &\Leftrightarrow_{df} \quad \mathbf{AV}_{all} = \bot \wedge \mathbf{AN}_{all} = \bot \\ &\Leftrightarrow_{df} \quad \mathit{AVAIL} \neq \textsf{Must} \wedge \mathit{ANTIC} \neq \textsf{Must} \\ \mathit{Prevented} \quad &\Leftrightarrow \quad \mathit{AVAIL} = \textsf{May} \wedge \mathit{ANTIC} = \textsf{May} \end{aligned}$$

Therefore, the weaker condition kills reuse at CMP nodes and some other nodes as well. However, both are equivalent for our purpose, as it is safe to kill when there is no reuse ($\mathit{AVAIL} = \textsf{No}$) or when there is no hoisting ($\mathit{ANTIC} = \textsf{No}$). The weaker predicate $\mathit{Prevented}_w$ is beneficial because computing and testing non-*Must* requires one bit, while two bits are required to test *May*. As a result, we obtain the same implementation complexity as the algorithms in [DS93, KRS94b]: three data-flow problems must be solved, each requiring one bit of solution per expression.

In conclusion, the CMP region is a convenient abstraction for terminating hoisting when it would unnecessarily extend the live ranges. It also provides an intuitive way of explaining the shortest-live-range solution without applying the corrective step based on delayability [KRS94b]. Furthermore, the CMP-based, motion-only solution can be implemented as efficiently as existing shortest-live-range algorithms.

6.3.1.3 PRE(R): Restructuring

PRE(R) removes all redundancies, relying exclusively on path separation. As mentioned above, PRE(R)'s aggressiveness is undesirable because it duplicates code when the more economical code motion could be used instead. We present PRE(R) here in order to explain more formally why some duplication is unnecessary, and also because it is used in this subsection as a reference point for our experiment that compares the code growth of PRE(M), PRE(MR), and PRE(R).

PRE(R) turns partial redundancy into full redundancy not by compensating the partial redundancy, but by separating out reuse paths. Compared to PRE(MR), however, reuse paths are separated not only where they meet a *free* path—a requirement for enabling code motion—but all the way to the partially redundant computation, which causes more code growth than PRE(MR). Formally, while PRE(MR) duplicates when

$$\mathit{Prevented}[\boldsymbol{n}] = \mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] = \mathsf{May} \wedge \mathit{ANTIC}_{\mathit{in}}[\boldsymbol{n}] = \mathsf{May},$$

PRE(R) duplicates when a reuse is only partial

$$\begin{aligned}\mathit{PartialReuse}[\boldsymbol{n}] &= \mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] = \mathsf{May} \wedge \mathit{ANTIC}_{\mathit{in}}[\boldsymbol{n}] \neq \mathsf{No}\\ &= \mathit{Prevented}[\boldsymbol{n}] \vee (\mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] = \mathsf{May} \wedge \mathit{ANTIC}_{\mathit{in}}[\boldsymbol{n}] = \mathsf{Must})\end{aligned}$$

That is, PRE(R) duplicates nodes on which $\mathit{AVAIL}_{\mathit{in}}[\boldsymbol{n}] = \mathsf{May} \wedge \mathit{ANTIC}_{\mathit{in}}[\boldsymbol{n}] = \mathsf{Must}$. On these nodes, the partially redundant computation could instead be hoisted.

PRE(R) can be expressed as a form of PRE(MR), in which the restructuring phase duplicates not only the CMP region, but its superset $\{\boldsymbol{n} \mid \mathit{PartialReuse}[\boldsymbol{n}]\}$.

1. *Dataflow analysis:* As in PRE(MR).
2. ***R)*** *Restructuring:* As in Figure 6.10, except that line 2 duplicates node for which *PartialReuse* is true.
3. *Optimization:* No code motion, as all computation are either fully redundancy on not redundant; merely remove fully redundant computations. Still, predicates *Insert* and *Replace* remain as in PRE(MR).

Figure 6.12: **The PRE(R) algorithm.**

There are at least three algorithms that fit the PRE(R) category [Weg75b, Weg75a, Ste96]. Although not formulated via region duplication, they peel off reuse paths. To decide where the redundancy is only partial, all of them use a form of *AVAIL*.

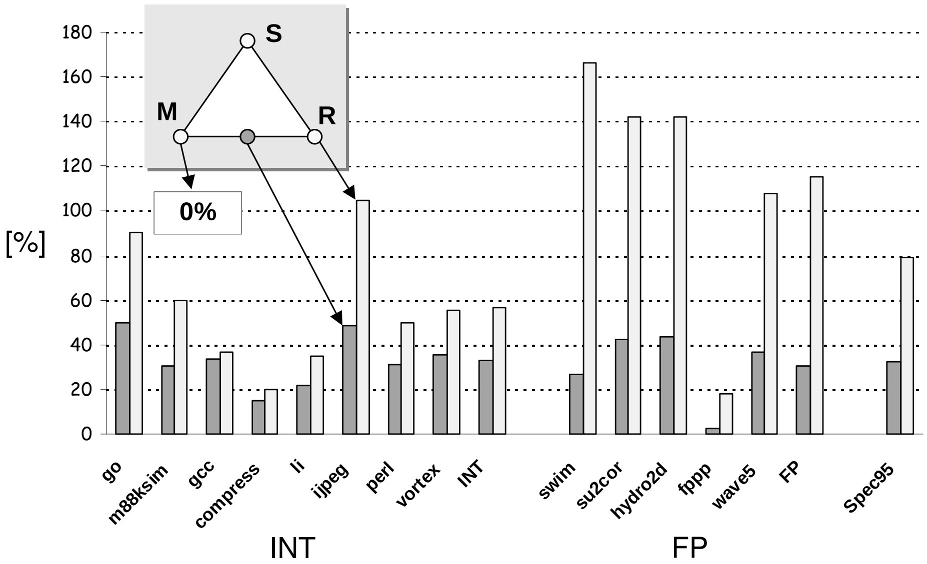


Figure 6.13: **Code growth of the three profile-independent PRE algorithms.**

6.3.1.4 A code growth experiment.

The purpose of the experiment is to evaluate the performance of PRE(MR) in terms of reducing the code-growth cost incurred by PRE(R). Both PRE(R) and PRE(MR) effect a complete removal, but PRE(MR) should duplicate less code in practice. Figure 6.13 compares their code growth. Indeed, combining code motion with restructuring reduces the amount of duplication to less than one half of the pure restructuring approach. On average for our SPEC95 benchmarks, our PRE(MR)algorithm was able to reduce the code growth of PRE(R) from 80% to about 33%.

Compare the ratio of PRE(R)/PRE(MR) code growth on the integer (INT) versus the floating-point (FP) benchmarks. The scientific FP programs exhibit more regular control flow structure than the control flow sensitive INT programs (they have relatively more loops than if statements) and so code motion is more successful on FP, requiring little restructuring. In other words, using pure restructuring on the FP programs was more of an overkill than on the INT programs. In fact, as will be shown in Section 6.6, on FP benchmarks code motion alone was able to remove nearly all redundancies (measured as the dynamic amount).

In our experiment, the code size was measured as the size of basic blocks (i.e., the number of instructions in the basic block). The final code size did not include the amount of inserted statements (see the *Insert* predicate), which may also grow the program code slightly. For this reason, in Figure 6.13, the code growth of PRE(M) is shown to be 0.

The code growth of both PRE(MR) and PRE(R) depends on the amount of redundancies and the shape of their reuse paths. In this experiment, the value-flow representation was very restricted; it detected only the class of lexically identical arithmetic expressions (as is common in traditional PRE). Namely, a three-address instruction $a\ op\ b$ was considered partially redundant when it was preceded on some control flow path by an identical instruction $a\ op\ b$, and a, b were not redefined since that previous computation. The operator *op* was any arithmetic and logic operator in the PlayDoh [KSR94] instruction set. Loads, stores, and conditional branches were not analyzed.

On some large procedures, the code growth exploded beyond practical means. To make the experiment feasible, the algorithm was terminated when the procedure size reached 3,000 instruction. The comparison of the two algorithms was made only on procedures that did not exceed the limit, in either algorithm.

Despite decreasing the cost of PRE(R) to less than a half, PRE(MR)'s growth (33%) is still a very significant code increase. In a production compiler, the allowable growth ranges around 20% [ASG97]. Keep in mind also that the code size is a precious resource, much like processor registers; the code-growth budget needs to be shared among procedure inlining and loop unrolling, which typically have higher payoffs than PRE and hence would be allocated a larger fraction of the budget. Furthermore, recall that our experiment targeted only the lexical redundancies; using the VNG representation, PRE(MR) would grow the code much more than 33%.

In conclusion, because the PRE(MR) algorithm achieves the smallest possible code growth (within the Morel-Renviose domain), we conjecture that, with known program transformation methods, further code growth reduction must be achieved via sacrificing some reuse opportunities. However, in the absence of profile information, code growth cannot be further reduced without impairing the optimization. This observation suggests the necessity of profiling, which selects opportunities to be sacrificed. Profile-guided PRE is developed next.

6.3.2 Profile-guided transformation

While the CMP region is the smallest set of nodes whose duplication enables the desired code motion, its size is often prohibitive in practice. In this section, relying on the profile for estimation of the run-time optimization benefit, PRE is made more practical by avoiding code replication that is

- unprofitable (PRE(Mr): too little benefit for too much duplication) Section 6.3.2.2, or
- can be replaced with careful speculative impairment of free paths, Section 6.3.2.1, or
- both, Section 6.3.2.3.

6.3.2.1 PRE(MS): code motion + speculation

Once PRE is profile-driven, the measure of *best* optimization changes. Rather than improving each path as much as possible, the goal is to minimize the overall number of dynamic optimizations, as measured by the supplied profile (recall Definitions 6.2, 6.3). PRE(MS) is an algorithm that finds such best *dynamic* optimization. While PRE(MR) is optimal in the absence of profile information (in that is minimizes code growth), PRE(MS) is optimal in the *presence* of run-time profile (in that it maximizes the dynamic benefit for a given profile).

Because PRE(MS) is profile-driven, it is also speculative in another sense: since the program is optimized under a specific run-time profile (and therefore also under a specific program input), we provide no guarantee that the transformation will actually not slow the program down for another input. However, our preliminary experiments indicate that PRE(MS) is extremely stable across various profiling inputs.

Next, we describe the principle of combining code motion with speculation. In PRE(M), hoisting of a computation e is blocked whenever it would enter a free path p that does not compute e in the original program. Such *speculative* code motion is disabled because executing e along path p could

1. raise spurious exceptions when computing e (e.g., overflow, page fault), and
2. outweigh the dynamic benefit of removing the original computation of e.

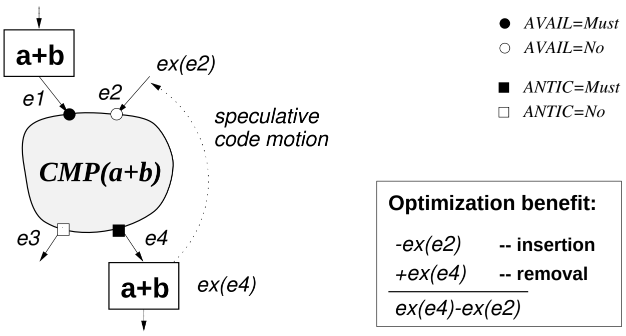


Figure 6.14: **PRE(MS): a simple version of speculation-profitability test.** Optimal speculation is found using estimators from Chapter 5.

Of course, the first restriction does not hold for instruction that cannot cause exceptions, such as additions or memory operations that can be proven to access a valid address. Excepting instructions can be optimized with modern processors, which support *delayed exceptions* [Dul98]. Special versions of excepting instructions are provided which suppress raising the exception until the generated value is actually used [MCB$^+$93]. Using delayed exceptions in PRE is a simple extension described in Section 6.5.2.

The second problem is harder. To guarantee that speculation is profitable, PRE(MS) utilizes the CMP region to determine the positions of speculative insertion points that

- make speculation profitable, and
- minimize live ranges of temporary variables.

Figure 6.14 illustrates the PRE(MS) algorithm. While the version of the algorithm discussed here is not computationally optimal (it does not maximize the benefit), it illustrates well the principle of combining code motion and speculation.

In PRE(MS), instead of duplicating the CMP region, we hoist the expression into all *No*-available entry edges. This makes all exits fully available, enabling complete removal of original computations along the *Must* exits. In the example, $\boldsymbol{a} + \boldsymbol{b}$ is moved into the *No*-available region entry edge e_2. This hoisting is speculative because $\boldsymbol{a} + \boldsymbol{b}$ is now executed on each path going through e_2 and e_3, which previously did not contain the expression.

To determine whether such a speculative hoisting is beneficial, we examine the execution frequencies of entry and exit edges, as follows. After the speculative hoisting, the dynamic amount of computations is decreased by the execution frequency $\mathit{freq}(e_4)$ of the *Must*-anticipated exit edge (following which a computation was removed), and increased by the frequency $\mathit{freq}(e_2)$ of the *No*-available entry edge (into which the computation was inserted). Speculation is profitable if the insertion is less frequent than the removal, i.e., the total execution frequency of *Must*-anticipable exit edges exceeds that of *No*-available entry edges.

The algorithm sketched above is not computationally optimal. Optimality is achieved via two observations. First, it is not necessary to speculate into all *No* entry edges. When a *Must* exit cannot be reached from a *No* entry, the entry need not be speculated to enable the optimization of that exit. If the entry has a high frequency, avoiding its speculation may increase the difference between improvement and impairment. Consider Figure 5.6(c) in the previous chapter. It is more profitable to speculate only into the (g, h) entry (benefit is 30); speculating to all *No* entries yields a benefit of 10.

Second, CMP entries are not the only possible speculation points [Tu99]. Optimal speculation points may lie within the CMP region. Consider Figure 5.6(d), but assume that the *AVAIL* solution on CMP entries is reversed (i.e., (f, h) is *No*-available, and (g, h), (g, k) are *Must*-available). In such a modified program, it is more profitable to speculate into the inner edge (i, k), at the benefit of 15, than into the entry edge (f, h), at the (negative) benefit of -5.

The two observations lead to the main result about PRE(MS):

Theorem 6.3 The maximal benefit of speculation equals the lower bound of the most precise estimator.
Proof outline. Recall that the lower bound of an estimator is computed as the frequency of the *Must*-exits minus the frequency of reuse that can be *stolen* from the *No*-entries. The proof is based on showing that without speculation all the stolen reuse must be "covered" with (speculative) insertions. In other words, all *No*-available paths through the CMP region must be made *Must*-available via (speculative) insertions. □

This constructive proof directly suggests the PRE(MS) algorithm. The central idea is to place the insertions at the least frequent set of CMP edges that ensure that all *Must*-exits are *Must*-available. This set of edges is found using the estimator algorithm. Namely, the optimal speculative insertions are those edges that are saturated in the network flow problem computed by the CMP^f estimator. If a path through the CMP contains multiple such edges, we select the one closest to the exits to obtain shortest live ranges. To summarize, an estimator computes not only the maximum speculation benefit, but also determines the insertion points.

1. *Dataflow analysis.* As in PRE(MR).
2. ***S)** speculation.*
    (a) Compute the CMP^f estimate. (CMP^f computes the tightest estimate for a separable VNG, for a single computation. Hence is lower bound computes the maximum speculation benefit.)
    (b) Find the min-cut on the flow network $(\mathbf{N}^M, \mathbf{X}^M)$, such that the edges in the min-cut are as close to $\mathbf{X}^M$ as possible. These will become the insertion edges.
3. *Recompute dataflow analysis.* Add the new speculative insertion points into $\boldsymbol{D}$ and recompute *AVAIL*.
4. ***M)** code motion.* Same as in PRE(M).

Figure 6.15: **The PRE(MS) algorithm.**

An important consequence of Theorem 6.3 is that the best speculation can be found from the edge profile. The speculative benefit is independent from branch correlation and edge profiles are as precise as path profiles in the case of speculative-motion PRE.

Corollary 6.1 (Edge profile is S-precise) Edge profile is sufficient to find the optimal optimization that uses speculation (but not restructuring).
Proof outline. The proof follows from the fact that PRE(MS), an algorithm driven by edge profile, results in best dynamic PRE. □

6.3.2.2 PRE(Mr): Selective restructuring

PRE(Mr) extends PRE(MR) by *inhibiting restructuring* in response to code duplication *cost* and the expected dynamic *benefit*. The resulting profile-guided algorithm duplicates a CMP region only when the incurred code growth is justified by a corresponding run-time gain from eliminating the redundancies.

We model the profitability of duplicating a CMP region R with a cost-benefit threshold predicate $T(R)$, which holds true if region's optimization benefit exceeds a constant multiple of the region size. Our metric of benefit is the dynamic amount of computations whose elimination will be enabled after R is duplicated, which we denote $\mathit{Rem}(R)$. That is,

$$T(R) \Leftrightarrow_{\mathit{df}} \mathit{Rem}(R) > c \;.\; \mathit{size}(R) \text{ where } c \text{ is a constant parameter.}$$

When $T(R) = \mathit{true}$ for each region R, the algorithm is equivalent to the (complete) PRE(MR). When $T(R) = \mathit{false}$ for each region, the algorithm reduces to the code-motion-only PRE(M).

Obviously, the predicate T determines only a sub-optimal tradeoff between exploiting PRE opportunities and limiting the code growth. In particular, it does not explicitly consider the instruction cache size and the increase in register pressure due to introduced temporary variables. We have chosen this form of T in order to avoid modeling complex interactions among compiler stages. In practice, T is usually supplemented with a code growth budget (e.g., in [ASG97], code is allowed to grow by about 20%).

The benefit $\mathit{Rem}(R)$ of a CMP region is computed using an estimator, which bounds the reuse flowing across a CMP region. In PRE(Mr) we assume that the entire connected region is duplicated (as compared to the possibility of copying some paths from the region, or the entire CMP region). Therefore, the CMP^c estimator is the appropriate one to use for computing (the bounds of) *Rem*(R).

The algorithm PRE(Mr) that duplicates only profitable CMP regions is given below. It is structured as its complete counterpart, PRE(MR): after dataflow analysis, we proceed to eliminate CMP regions, separately for each value name e. While in PRE(MR) it was sufficient to treat all nodes from a single CMP together (all of them were duplicated), selective duplication benefits from dividing the CMP into disconnected subregions, as is done in the CMP^c estimator. After all profitable regions are eliminated, the motion-blocking effect of CMP regions remaining in the program must be captured. All that is needed is to apply the PRE(M) on the improved control flow graph. Hoisting that remains to be prevented by some CMP node after the selective restructuring was performed, will be avoided by recomputing the M-availability (AVAIL_w^M), which forces *No*-availability whenever a CMP is detected.

Our PRE(Mr) algorithm does not address the important problem of which names e should be restructured first. Because each CFG node may be duplicated multiple times, causing its exponential replication, it is desirable to optimize most beneficial reuse paths first, before there are too many copies of each node. One possible heuristic is to order names (in line 1) based on the reuse computed with an estimator. Performing PRE(Mr) optimally (i.e., maximize $\mathit{Rem}(R)$ while guaranteeing that $T(R)$ holds) seems NP-hard, due to the need to consider multiple names simultaneously.

In Chapter 5, edge profiles were used to estimate the benefit *Rem* of duplicating a region. An alternative is to use *path profiles* [ABL97,BL96a], which are convenient for establishing cost-benefit optimization trade-offs when restructuring must be used. To arrive at the value of the region benefit with a path profile, it is sufficient to sum the frequencies of *Must-Must* paths, which are paths that cross any region entry edge that is *Must*-available and any exit edge that is *Must*-anticipated. These are precisely the paths along which value reuse exists but is blocked by the region. While there is an exponential number of profiled acyclic paths, only 5.4% of procedures execute more than 50 distinct paths in SPEC95 [GBF97b]. This number drops to 1.3%

1. *Dataflow analysis.* As in PRE(MR).

2. ***r)*** *Partial restructuring:* remove profitable CMP regions.

```
for each computation $e$, in some benefit order do
    for each connected subregion $R_i$ of CMP$[e]$ do
        determine optimization benefit $Rem(R_i)$
        carry out frequency analysis of AVAIL on $R_i$ using CMP$^c$
        if subregion is profitable, duplicate it
        if $T(R_i)$ then duplicate $R_i$
    end for
end for
```

3. ***M)*** *Code motion.* Perform PRE(M), including the recomputation of *AVAIL*$_w^M$.

Figure 6.16: **The PRE(Mr) algorithm.**

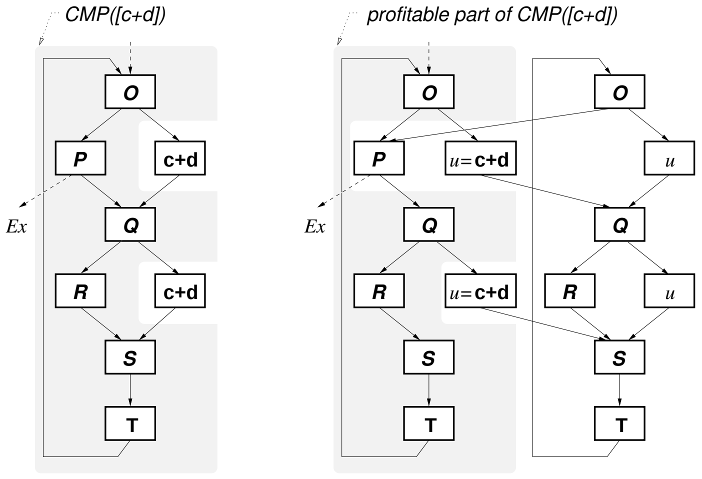
when low-frequency paths accounting for 5% of total frequency are removed. Since we can afford to approximate by disregarding these infrequent paths, summing individual path frequencies constitutes a feasible algorithm for many CMP regions. Furthermore, because they encapsulate branch correlation, path profiles compute the benefit more precisely than frequency analysis based on correlation-insensitive edge profiles.

Sub-CMP restructuring. Moreover, the notion of individual CMP paths leads to a better PRE(Mr). Considering the CMP region as an indivisible duplication unit is overly conservative. While it may not be profitable to restructure the entire region, the region may contain a few paths *Must-Must* paths that are frequently executed and are inexpensive to duplicate. Our goal is to find the largest subset (frequency-wise) of region paths that together pass the threshold test $T(R)$. An example illustrating the PRE(Mr) algorithm is given in Figure 6.17. The table in the figure lists *Must-Must*(i.e., optimizable) paths that have been executed at least once, together with their execution frequencies. The table also lists the size of each basic block and shows which basic blocks must be duplicated to enable optimization of a given path. The task is to select a subset of paths such that the benefit (i.e., the sum of their frequencies) and th epath duplication cost (i.e., the sum of sizes of selected basic blocks) satisfy the predicate $T(R)$ with the constant $c = 1$. Selecting paths 1, 2, and 4 satisifies $T(R)$; the benefit is 50 and the cost of duplicating basic blocks O, Q, R, S, T is 45.

The task of *partial restructuring* is to localize a subgraph of the CMP that has a small size but contains many hot *Must-Must* paths. By duplicating only such a subregion, we are effectively peeling off only hot and short paths. In Figure 6.2(e), only the (presumably hot) path through the node Q was separated. Again, the problem of finding an *optimal* subregion, one whose benefit is maximized but passes the $T(R)$ predicate and is smaller than a constant budget, seems NP-hard. Furthermore, path separation for multiple expressions simultaneously should be considered. However, the empirically very small number of hot paths promises an efficient exhaustive-search algorithm.

6.3.2.3 PRE(Msr): motion + selective restructuring + selective speculation

PRE(Msr) integrates both restructuring and speculation. It can either select a profitable subgraph of the CMP for each, or restructure to enable speculation (similarly to how restructuring enables code motion).



a) source program b) after optimal PgPRE

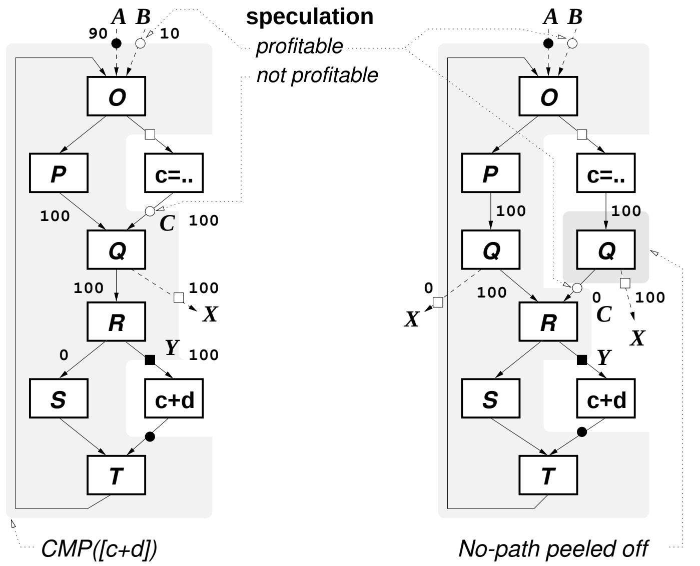
non-zero-freq. *Must-Must* path	path freq.	select	basic block (size)					
			O	P	Q	R	S	T
			25	50	5	5	5	5
1: [3,Q,b]	20	y			x			
2: [4,S,T,O,a]	20	y	x				x	x
3: [4,S,T,O,P,Q,b]	10		x	x	x		x	x
4: [3,Q,R,S,T,O,a]	10	y	x		x	x	x	x
Total selected:	50		45					

Figure 6.17: **An example of PRE(Mr).** Assume $T(R)$ parameterized $c = 1$. Tightening code-growth constant to $c = 0.5$ results in the program in Figure 6.2(e).

This section presents the principles that can lead to an efficient heuristics, based on path profiles, as in the sub-CMP version of PRE(MR).

Integrating partial speculation and restructuring offers additional opportunities for improving the cost-benefit ratio. We are no longer restricted to peeling off hot *Must-Must* paths and/or selecting *No*-entries for speculation. When the high frequency of a *No* entry prevents speculation, we can peel off a hot *No*-available path emanating from the entry, thereby reducing entry edge frequency and allowing the speculation, at the cost of some code duplication.

Figure 6.18(a) shows an example program annotated with an edge profile. Because peeling hot *Must-Must* paths from the highlighted *CMP*$([c+d])$ would duplicate all blocks except S, we try speculation. To eliminate the redundancy at the CMP exit edge Y with frequency $ex(Y) = 100$, a computation must be inserted into *No*-entries B and C. While B is low-frequency (10), C is not (100), hence the speculation is disadvantageous, as $ex(Y) = 100 < ex(B) + ex(C) = 10 + 100$. Now assume that the exit branch in Q is strongly biased and the path C, Q, X has a frequency of 100. That is, after edge C is executed, the execution will always follow to X. We can peel off this *No-No* path, as shown in (b), effectively moving the speculation point C off this path. After peeling, the frequency of C becomes 0 and the speculation is profitable, $ex(Y) = 100 > ex(B) + ex(C) = 10 + 0$.



a) source program b) speculation made profitable

Figure 6.18: **An example of PRE(Msr) optimization.**

To summarize the relationship between the three profile-guided approaches, the overall goal is to enable for code motion a maximum of paths with value reuse (*Must-Must* paths). This can be done by peeling the path from the region, at the cost of duplicating all basic blocks in the path. The duplication cost may be shared among various duplicated paths. Alternatively, partial speculation enables reuse along all *Must-Must* paths sinking into a *Must* exit, by inserting the computation into an appropriate set of *No* entry edges. Finally, the cost of speculating at an entry can be reduced by peeling off a *No-No* path.

It remains to be shown that all PRE algorithms presented in this section are live-range optimal.

Theorem 6.4 (Shortest live ranges). Given the CMP-restructured (or original) control flow graph, PRE(MR) (PRE(M)) is optimal in that it minimizes the live range lengths of inserted temporary variables.

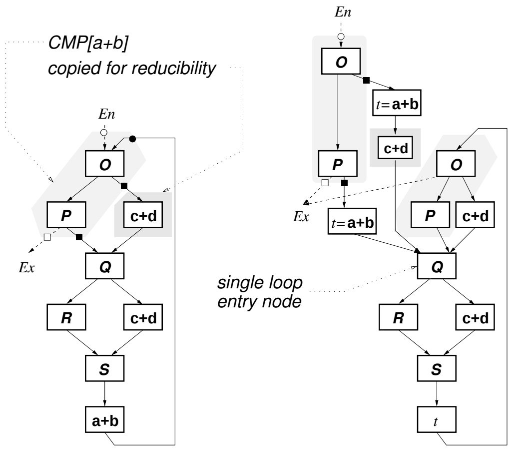
Proof. An initialization point *Insert* cannot be delayed either because it would become partially redundant, destroying completeness, or because its temporary variable is used in the immediate successor. □

6.4 PRE for an arbitrary VNG

We present here the extension of the PRE(MS) algorithm for the general VNG representation. We restrict our attention to this algorithm because our experiments suggests that it is most practical of all. It achieves near complete redundancy removal of redundancies at zero code growth, and it is simple to implement (no restructuring is needed).

6.4.1 Code motion + speculation

PRE(MS) for general VNGs proceeds as PRE(MS) for separable VNGs. It relies on an estimator to determine where to place the speculative insertions in the CMP regions. Unfortunately, even though all five



a) source program b) reducible ComPRE of [a+b]

Figure 6.19: **Reducible restructuring.**

estimators in Chapter 5 were in both the separable and the general form, not all general estimators are suitable for determining the placement of insertions. The reason is that the two best estimators (CMP^r and CMP^f) are based on computing maximum flow on a network whose structure does not directly reflect the CFG of the program, and hence the saturation of the network edges cannot be used as a criterion for the insertion, as it was in the separable PRE(MS).

Fortunately, the CMP^c estimator supports the necessary speculation decisions. As our experiments show, its lower bound is almost as precise as that of CMP^r and hence the reliance on CMP^c does not seem to be much of a sacrifice.

The general PRE(MS) algorithm reflects its underlying CMP^f estimator. Its insertion candidate points are the entries of the CMP region, but not the inner CMP edges. Each connected CMP region is estimated separately. When its lower bound is greater than zero, then the speculation across the connected region is beneficial and the algorithm inserts speculative computation into all CMP *No*-entries.

6.5 Miscellaneous issues

This section covers a two PRE issues that mainly concern the implementation in a production compiler. First, we sketch how to deal with the irreducibility of the CFG that may be introduced by the restructuring-based algorithms. Second, we outline how to adapt speculation-based algorithms for exploiting the control speculation features of the IA-64 processor architecture [Dul98].

6.5.1 Reducible restructuring

Duplicating a CMP region may destroy reducibility of the control flow graph. In Figure 6.2(c), for example, PRE(MR) resulted in a loop with two distinct entry nodes. Even though PRE(R)preserves reducibility on the same loop (Figure 6.2(b)), like other restructuring-based optimizations [AL98, BGS97b, Ste96], it is also plagued by introducing irreducibility. One way to deal with the problem is to perform PRE after all optimizations that require single-entry loops. However, many algorithms for scheduling (which should ideally follow PRE) rely on reducibility.

After PRE(MR), a reducible graph can be obtained with additional code duplication. An effective algorithm for normalizing irreducible programs is given in [JC96]. To suppress an unnecessary invocation of the algorithm, we can employ a simple test of whether irreducibility may be created after a region duplication. The test is based upon examining only the CMP entry and exit edges, rather than the entire program. Assuming we start from a reducible graph, restructuring will make a loop L irreducible only if multiple CMP exit edges sink into L, and at least one region entry is outside L (i.e., is not dominated by L's header node). If such a region is duplicated, target nodes of region exit edges may become the (multiple) loop entry nodes. Consider the loop in Figure 6.19(a). Two of the three exits of $\mathit{CMP}[a + b]$ fall into the loop. After restructuring, they will become loop entries, as shown in Figure 6.2(c).

Rather than applying a global algorithm like [JC96], a straightforward approach to make the affected loop reducible is to peel off a part of its body. The goal is to extend the replication scope so that the region exits sink onto a single loop node, which will then become the new loop entry. Such a node is the closest common postdominator (within the loop) of all the offending region exits and the original loop entry. Figure 6.19(a) highlights node `c+d` whose duplication after CMP restructuring will restore reducibility of the loop. The postdominator of the offending exits is node Q, which becomes the new loop header.

6.5.2 Spurious exceptions

The IA-64 architecture [Dul98] introduces *delayed exceptions*, a mechanism to support control speculation of instructions that may raise exceptions, such as divisions or memory access instructions. In general, when the compiler reorders these instructions in a way that they may be executed on paths that would not execute them in the original program (e.g., they are hoisted above a conditional branch), their exception may be spurious, which changes the semantics of the original program. As was mentioned in Section 6.3.2.1, speculative PRE in particular introduces the problem of spurious exceptions.

To preserve the semantics, IA-64 allows an raised exception to be *delayed:* when the excepting instruction is marked as *delaying*, its exception is suppressed and propagated as a special flag attached to the register that stores the result of the excepting instruction. The delayed exception may be raised later, when some instruction marked as *catching* reads from the flagged register. Such an exception-catching instruction is usually placed in the original program point of the reordered (excepting) instruction, so that the exception is raised exactly when it would be in the original program.

In the PRE context, two steps must be performed to allow speculation of excepting instructions on the IA-64 architecture.

1. Speculative inserted computations that may raise exceptions must be marked as *delaying*. This step is trivial in all our speculation-based algorithms.

2. The delayed exceptions must be *caught* in a way that preserves exception semantics. This can be accomplished by marking as catching all *user* nodes that can consume the value produced by a speculative insertion point. Recall that all user computations are replaced with temporaries, so the exceptions will be caught with copy instructions of the form $x := t$, where t is the temporary.

6.6 Experiments

The experiments evaluating the transformation algorithms were performed using the HP Labs VLIW back-end compiler *elcor*, which was fed SPEC95 benchmarks that were previously compiled, edge-

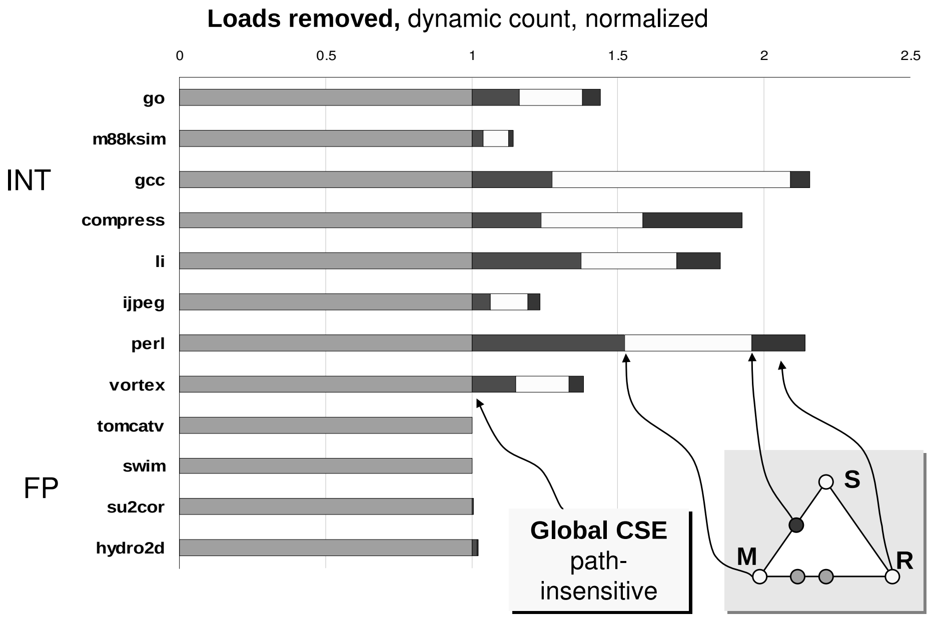


Figure 6.20: **Relative completeness of three PRE algorithms.**

profiled, and inlined (only SPEC95(int)) by the *Impact* compiler. Table 6.1 shows program sizes in the total number of nodes and expressions. Each node corresponds to one intermediate statement. Memory requirements are indicated by the column max space, which gives the largest nodes-expressions product among all procedures. The running time of our rather inefficient implementation behaved quadratically in the number of procedure nodes; for a procedure with 1,000 nodes, the PRE time was about 5 seconds on PA-8000. Typically, the complete PRE ran faster than the subsequent dead code elimination.

The first experiment compares the removal power (i.e., completeness) of three PRE algorithms on removal of redundant load instructions. The VNG served as the value-flow program representation for the experiment. The plot in Figure 6.20 shows the dynamic amount of computations removed by PRE(M), PRE(MS), and PRE(R)(and hence also PRE(MR)). Clearly, the following relationship holds

$$\text{PRE(M)} \leq \text{PRE(MS)} \leq \text{PRE(MR)} = \text{PRE(R)}$$

where $a \leq b$ means a removes no more computations than b, in dynamic terms.[6] The first experiment answers this relationship in quantitative terms.

In the graph, the removal power is normalized on the power of Global CSE, a path-*in*sensitive algorithm for redundancy elimination. The graph exposes three important points.

1. Due to the normalization of the amount of reuse, the (dynamic) reuse below 1.0 is path-insensitive (available along all paths) and all reuse above 1.0 is path-sensitive (available along a strict subset of paths).

   Integer programs contain a lot of strictly partial reuse, for which path-sensitive algorithms are important.

---

[6] We assume here that the profile-guided PRE(MS) is optimized and executed on the same program input. This assumption held in our experiments, too.

benchmark	program size				CM prevented			loop inv		dynamic	
spec95int spec95fp	procedures	nodes (k)	expressions (k)	max space (M)	optimizable (% of *expr*)	prevented-CMP (% of *optim*)	prevented-POE (% of *optim*)	loop invar (% of *optim*)	invar-prevent (% of *LI*)	global CSE (% of all)	complete PRE (% of all)
`099.go`	372	153.6	37.3	5.8	10.2	29.6	45.4	7.1	83.4	9.5	11.7
`124.m88ksim`	252	79.5	17.4	4.2	13.1	32.7	45.4	13.0	78.0	7.6	9.4
`126.gcc`	1661	917.2	158.2	38.0	8.0	34.2	45.0	2.5	69.8	3.7	4.6
`129.compress`	24	3.0	0.8	0.1	13.7	20.4	43.4	9.7	45.5	11.5	14.5
`130.li`	357	37.4	8.4	2.0	11.8	22.4	34.4	10.4	69.9	6.8	8.0
`132.ijpeg`	472	81.8	22.8	1.2	13.9	24.1	45.3	5.1	78.1	4.3	5.1
`134.perl`	276	135.0	25.5	40.4	9.6	39.5	51.8	11.9	93.5	4.8	6.8
`147.vortex`	923	325.9	65.7	5.8	16.6	29.5	36.1	6.3	81.6	11.1	13.0
Avg: spec95int	542.1	216.7	42.0	12.2	12.1	29.1	43.4	8.2	75.0	7.4	9.1
`101.tomcatv`	1	0.8	0.2	0.2	21.4	26.4	50.9	13.2	71.4	10.2	13.3
`102.swim`	7	1.6	0.6	0.1	17.0	29.2	46.2	10.4	100.0	10.7	12.0
`103.su2cor`	37	10.6	3.9	2.5	15.3	29.8	53.8	14.5	43.7	12.8	13.0
`104.hydro2d`	43	8.5	2.4	0.4	16.8	21.7	42.7	5.9	41.7	1.9	6.0
`145.fpppp`	37	13.6	6.7	19.6	14.6	52.2	57.7	43.0	91.9	7.1	7.7
`146.wave5`	110	33.3	12.3	5.3	12.4	34.8	47.8	4.9	66.2	7.1	7.8
Avg: spec95fp	39.2	11.4	4.4	4.7	16.2	32.4	49.8	15.3	69.2	8.3	10.0
Avg: spec95	326.6	128.7	25.9	9.0	13.9	30.5	46.1	11.3	72.5	7.8	9.5

Table 6.1: Experience with PRE based on control flow restructuring.

2. In integer programs, the obstacles to code motion are significant.

   PRE(M) optimizes less than half of all strictly partial redundancies. Therefore, it is worth improving the traditional PRE approaches.

3. The performance of the complete PRE was estimated using the CMP^c estimator. The bottom of rightmost bar in the plot shows the lower bound of the estimate. The top of the bar shows the upper bound. The complete reuse is somewhere between the bounds. Also recall that the lower bound of the estimate equals the benefit of speculation (Theorem 6.3).

   PRE(MS) is very close to the performance of the (complete) PRE(MR) algorithm. Therefore, restructuring may not be necessary, at least for load removal.

The third point is especially good news. If this empirical result holds true for other value flow optimizations, then PRE(MS) is able to remove the third exponential path factor, due to the number of path with various optimization opportunities, as introduced in Section 1.5. Still, some value-flow optimizations are inherently restructuring-based, e.g., branch elimination (see Section 7).

Next, we present other experimental results that show the need to develop PRE that goes beyond the code motion. In contrast to Figure 6.20, these experiments were performed using a *lexical* value-flow program representation.

Disabling effects of CMP regions. The column labeled optimizable gives the percentage of expressions that reuse value along some path; 13.9% of (static) expressions have partially redundant computations. The next column prevented-CMP reports the percentage of optimizable expressions whose complete optimization by code motion is prevented by a CMP region. Code-motion PRE will fail to fully optimize 30.5% of optimizable expressions. For comparison, column prevented-POE reports expressions that will require restructuring in PRE(R).

Loop invariant expressions. Next, we determined what percentage of loop invariant (LI) expressions can be removed from their invariant loops with code motion. The column loop invar shows the percentage of

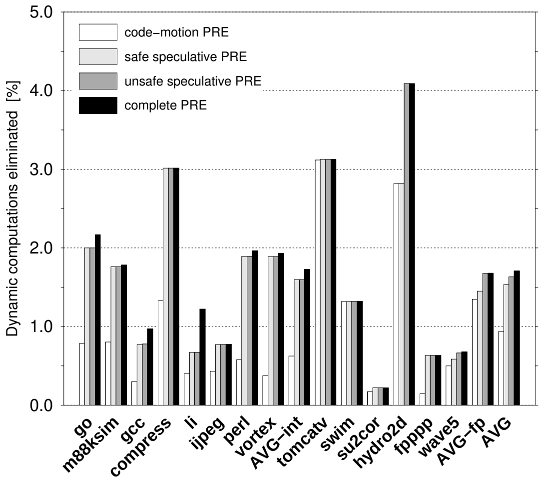


Figure 6.21: **Benefit of various PRE algorithms on a lexical value-flow representation.** : dynamic op-count decrease due to *strictly partial* redundancies. Each algorithm also completely removes full redundancies.

optimizable expressions that pass our test of loop-invariance. The following column gives the percentage of LI expressions that have a CMP region; an average of 72.5% of LI computations cannot be hoisted from all enclosing invariant loops without restructuring.

Eliminated computations. Finally, we report the amount of removed computations. This experiment differs from the one in Figure 6.20 in that PRE is performed on the lexical representation of arithmetic instructions, rather than on the symbolic (VNG) representation of redundant loads. The column global CSE reports the dynamic amount of computations removed by global common subexpression elimination; this corresponds to all full redundancies. The column complete PRE gives the dynamic amount of all partially redundant statements. The fact that strictly partial redundancies contribute only 1.7% (the difference between complete PRE and global CSE) may be due to the style of *Impact*'s intermediate code (e.g., multiple virtual registers for the same variable). We expect a more powerful redundancy analysis to perform better. Figure 6.21 plots the dynamic amount of strictly partial redundancies removed by various PRE techniques. Code-motion PRE yields only about half the benefit of a complete PRE. Furthermore, speculation results in near-complete PRE for most benchmarks, even without special hardware support (i.e., safe speculation). Speculation was carried out on the CMP as whole. Note that the graph accounts for the dynamic impairment caused by speculation. The measurements indicate that an ideal PRE algorithm should integrate both speculation and restructuring. Using restructuring when speculation would waste a large portion of benefit will provide an almost complete PRE with small code growth.

6.7 Conclusion and related work

In summary, this chapter makes the following contributions:

- We present an approach for integrating three orthogonal program transformation methods: code motion, control flow restructuring, and control speculation. We developed a family of PRE algorithms that combine the three methods:

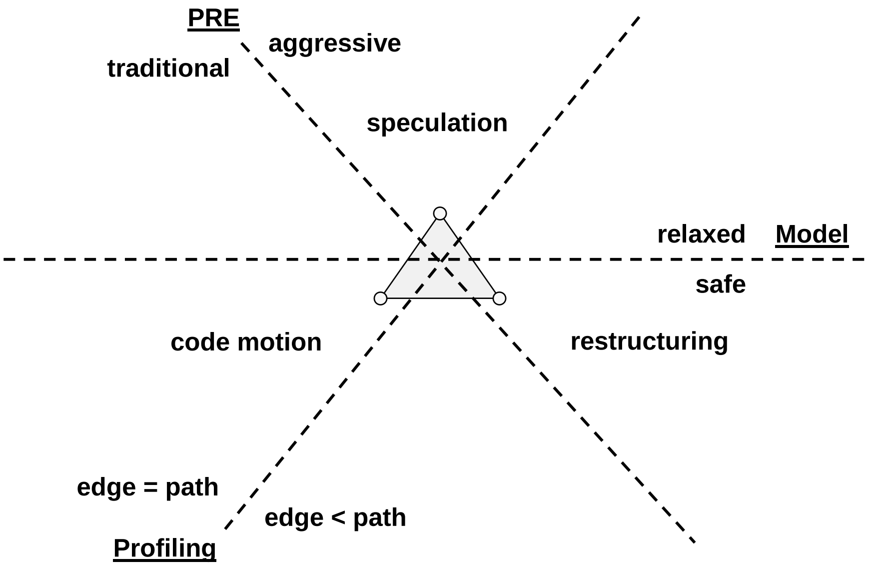


Figure 6.22: **A summary of our results.** *PRE:* We extended the traditional code-motion transformation with two transformation methods, achieving an aggressive PRE. *Model:* We showed how to use code motion and restructuring within the safe optimization model, in which no program path can be impaired. The use of speculation requires a relaxed optimization model, in which path can be impaired. *Profiling:* We showed that, when code motion is combined with speculation, an edge profile is as precise as the path profile. When restructuring is profile-guided, path-profile is more precise than edge profile.

- PRE(MR) is a PRE algorithm that is *complete* (i.e., it exploits all opportunities for value reuse) and greatly reduces the code growth necessary to achieve the desired (complete) code motion. For a large class programs (those with separable VNGs), the code growth is minimal.
- PRE(MS) is profile-guided PRE that integrates code motion and speculation. Because it does not use restructuring, it achieves zero code growth. Our experiments show that its optimization is near complete. The important contribution of the PRE(MS) algorithm is that it determines the speculation points and the benefit (i.e., the difference of improved and impaired path frequencies):
    - ○ such that the benefit is maximized (for separable VNGs),
    - ○ without enumerating the improved and impaired paths,
    - ○ from an edge profile, as precisely as using the execution trace.
- PRE(Msr) balances the three techniques. It resorts to restructuring only when speculation cannot be done (sufficiently) beneficially.
- PRE(M) is a natural restriction of the PRE(MR) algorithm. It produces optimization equivalent to the optimal code-motion PRE [KRS94b], but we believe it is easier to understand.
- Our experiments compare a) the optimization power and b) code growth of PRE(M), PRE(MS), PRE(MR), PRE(R)(the pure-restructuring PRE).

Figure 6.22 summarizes this chapter using three planes that divide the algorithm design space.

The PRE plane: We extended the traditional code-motion transformation with two transformation methods, achieving an aggressive PRE.

The optimization model plane: We showed how to use code motion and restructuring within the safe optimization model, in which no program path can be impaired. The use of speculation requires a relaxed optimization model, in which path can be impaired.

The profiling plane: We showed that, when code motion is combined with speculation, an edge profile is as precise as the path profile. When restructuring is profile-guided, path-profile is more precise than edge profile.

The observation is that edge profile is suficient to find optimal speculation deserves a further comment. Since edge profile is speculation-precise, to exploit the power of path profiles, partial restructuring, rather than (speculative) code motion alone, must be used. This becomes more intuitive once we realize that without control flow restructuring, one is restricted to consider only an individual edge (but not a path) for expression insertion and removal. To compare the CMP-based partial speculation with the speculative PRE in [GBF98], we show how to efficiently compute the benefit by defining the CMP region and how to apply edge profiles with the same precision as path profiles. In acyclic code, we achieve the same precision; in cyclic code, we are more precise in the presence of loop-carried reuse.

In this chapter, we defined the *code-motion-preventing* (CMP) region, which is a VNG subgraph localizing adverse effects of control flow on the desired value reuse. The notion of the CMP is applied to enhance and integrate the three existing PRE transformations in the following ways,

1. Code motion and restructuring are integrated to remove all redundancies at minimal code growth cost (PRE(MR)).
2. Morel and Renviose's original method is expressed as a restricted (motion-only) case of the complete algorithm (PRE(M)).
3. We develop an algorithm whose power adjusts continually between the motion-only and the complete PRE in response to the program profile and the utility function T (PRE(Mr)).
4. We demonstrate that speculation can be navigated precisely by edge profiles alone (PRE(MS)).
5. Path profiles are used to integrate the three transformations and balance their power at the level of CMP paths.

Figure 6.23 summarizes related work and out contributions. The PRE research started with the independent works of Wegman and Wegbreit who developed PRE(R) algorithms [Weg75a, Weg75b]. Later, Steffen created a complete PRE(R) algorithm that removed not only expressions but also conditional branches [Ste96]. Due to the exponential code growth, none of the algorithm is implemented in a production compiler, to the best of our knowledge.

Morel and Renviose created the first practical PRE algorithm [MR79]. To limit the code growth, the algorithm was based on code motion. Their algorithm was later much improved [DRZ92, Dha91, DS88] until the research "stabilized" on lazy code motion, now considered the standard PRE(M) algorithm [KRS92, KRS94a]. Our contribution in the PRE(M) area is an intuitive formulation of an algorithm that produces identical optimization as the lazy code motion.

Between the PRE(M) and PRE(R) algorithm lies our another contribution—the PRE(MR) algorithm that is as complete as the PRE(R) algorithms but minimizes the code growth by performing code motion as much as possible.

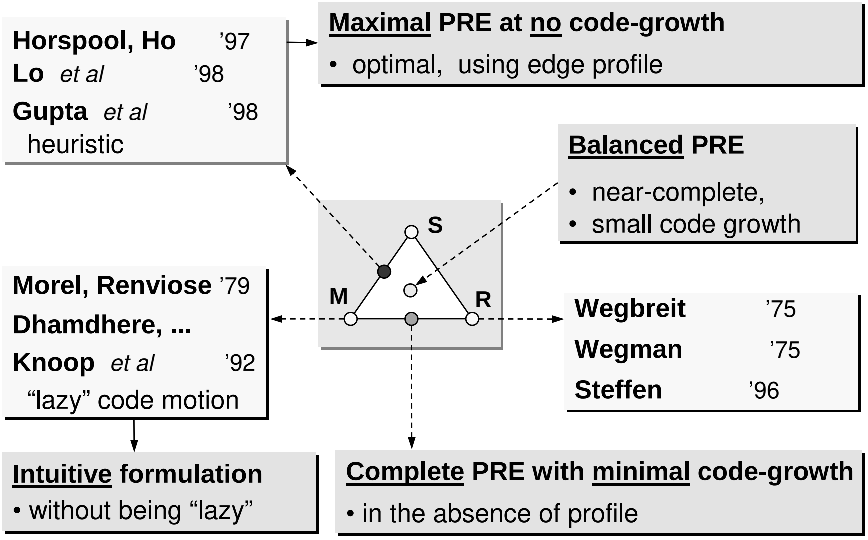


Figure 6.23: **Related work and contributions.**

Recently, various researches extended code-motion PRE with speculations [HH97,GBF98,LCK$^+$98, SJ98]. None of them achieves both optimal speculation and relies on the inexpensive edge profiles.

Finally, our last contribution is the ability to combine all three transformation methods, producing a balanced PRE, in response to the profile and the characteristics of the optimized program. This integrated PRE is enabled by the CMP region, which serves as a single underlying abstraction for all our algorithms.

Chapter 7

Inter-procedural Removal of Redundancies

This chapter is concerned with inter-procedural value flow. Often, a value is computed in one procedure and recomputed in another, as a result of a modular programming style. To exploit such an inter-procedural redundancy, the optimizer requires a interprocedural analysis and transformation.

First, this chapter presents inter-procedural version of dataflow analysis on the VNG. The analysis is distinguished in two respects. First, it is demand-driven. By computing only the dataflow solution required by the transformation stage, it reduces the cost of the analysis, when compared to an exhaustive analysis. Second, the analysis does not require a completely constructed interprocedural VNG. Instead, the VNG is constructed on demand, only the portion that is needed by the demand-driven analysis. The demand-driven construction of the VNG significantly reduces the cost of analysis, but has the drawback that the demand-VNG does not use value numbering. The VNG is built only on back-substitution and dataflow analysis, which are folded together into a single demand-driven pass.

The second part of this chapter deals with inter-procedural transformation. Inter-procedural redundancies cannot be optimized with the techniques presented in Chapter 6, as computations must be moved across procedure boundaries. While inlining can be used to concentrate the reuse into a single procedure, it may be prohibitively expensive in practice due to its code growth. A code-growth-free alternative is inter-procedural code motion [Kno98]. Unfortunately, inter-procedural code motion may fail just like its intra-procedural counterpart. Furthermore, some very attractive optimizations (branch removal, revitalization of calls) cannot be carried out with code motion.

This chapter presents an inter-procedural version of the restructuring-based PRE(R) algorithm. Rather than attempting to combine code motion and transformation (to minimize code growth), we focus on achieving a complete removal of inter-procedural redundancies (without resorting to inlining of procedures). Note that entry/exit splitting of virtual call sites is not restricted to branch elimination, as presented in this chapter. It may be used for any value-flow optimization, as a PRE(R) algorithm.

The goal of an inter-procedural PRE(R) is to separate reuse paths that cross procedure boundaries. To this end, our PRE(R) algorithm performs procedure *entry splitting* and *exit splitting*. The former transformation creates multiple entry points in a procedure; the latter allows a procedure to return to one of several return points in the caller. We show how to use the two transformations in concert, to separate interprocedural paths and convert partial redundancy into full redundancy, which achieves a complete optimization.

As an application of on entry/exit splitting, we develop Inter-procedural Conditional-Branch Elimination (ICBE). Relying on inter-procedural value-flow analysis presented in the first part of this chapter, ICBE removes branches that are correlated with other branches, i.e., branches whose outcomes are known along some execution paths from prior branch outcomes or assignments. Clearly, static branch correlation is a special case of value reuse (of the branch condition value). ICBE eliminates correlated branches along the correlated paths (i.e., reuse paths) by means of code restructuring, which may involve splitting procedure entries and exits. We describe the benefits of our inter-procedural branch elimination optimization and experimentally show that, for the same amount of code growth, the estimated reduction in executed conditional branches is about 2.5 times higher than when only *intra*procedural conditional branch elimination is applied.

7.1 Demand-driven interprocedural dataflow analysis

This section presents an inter-procedural version of dataflow analysis on an inter-procedural VNG. The analysis is distinguished in a few respects.

1. The analysis is *demand-driven*. By computing only the dataflow solution required by the transformation stage, it reduces the cost of the analysis, when compared to an exhaustive analysis. In practice, the optimizer may decide to analyze only the frequently executed *user* nodes.
2. The analysis can be *terminated early*, before the (demanded) solution is completely computed. The analysis is stopped after a budgeted amount of nodes have been visited. The unexplored paths are assumed to have a conservative solution.
3. The VNG is *constructed on the fly* during the dataflow analysis. Only the portion that is needed by the demand-driven analysis is (virtually) constructed, with the goal of avoiding the construction of potentially very large interprocedural VNG. An (undesirable) consequence of this (desirable) delayed VNG construction is that the value numbering is not invoked to collapse the VNG threads, resulting in lower accuracy.

7.1.1 Application: inter-procedural branch correlation

We use the demand VNG analysis to find *statically correlated conditional branches*. A branch is statically correlated (along a path) if the the branch outcome can be determined (along that path) at compile time, from prior statements or branch outcomes. Branch correlation is another name for partial redundancy of branches; the former name reflects that the direction of the redundant branch depends on the direction of some other branch(es). Correlated branches can be eliminated from the optimizable path through code restructuring presented in Chapter 6 and Chapter 7. The former chapter shows how to separate the optimizable paths intraprocedurally. The latter chapter presents inter-procedural separation of paths, by means of on procedure *entry splitting* and *exit splitting*.

Interprocedural Conditional Branch Elimination (ICBE) has a number of benefits, including

- enhancing instruction scheduling and software pipelining,
- improving speculative execution and hardware branch prediction, and
- optimizing C++/Java virtual functions.

Recent research in branch prediction [Kra94, SLM96, YGS95], profiling [BL96b], and the elimination of conditional branches [MW95b] has reported the existence of significant amounts of correlation among conditional branches, presenting opportunities for optimizations. Previous work on conditional branch elimination through static correlation [MW95b] demonstrated substantial performance improvements despite its restricted focus on eliminating conditionals within loops. Experimentally, we show that substantially more static correlation is detected at compile time when programs are analyzed interprocedurally. Using programs from the SPEC95 suite, we discovered that interprocedural detection of correlation enables elimination of 3% to 18% of executed conditionals, which is a factor of about 2.5 improvement over strictly intraprocedural analysis. As illustrated below, this high correlation among branches when procedures are considered is due to the modular fashion in which we write procedures:

- In a procedure, the value returned is often selected by an if-statement. This value may again be checked by the caller. For example, consider a call to a procedure that removes an element from a linked list. The procedure tests whether the list is empty and, if so, returns *nil*. The caller performs an identical test on the return value to determine if *nil* was returned. The later test is fully correlated with the earlier one.
- In order to keep the procedure interface simple by passing few arguments, procedures frequently include checks on the parameters that are also performed by the caller or even by previous calls to the same procedure. For example, procedures from the same library module may be called one after another, propagating values. These procedures often perform correlated tests on the propagated values. With the ICBE optimization, the repeated testing can be eliminated.

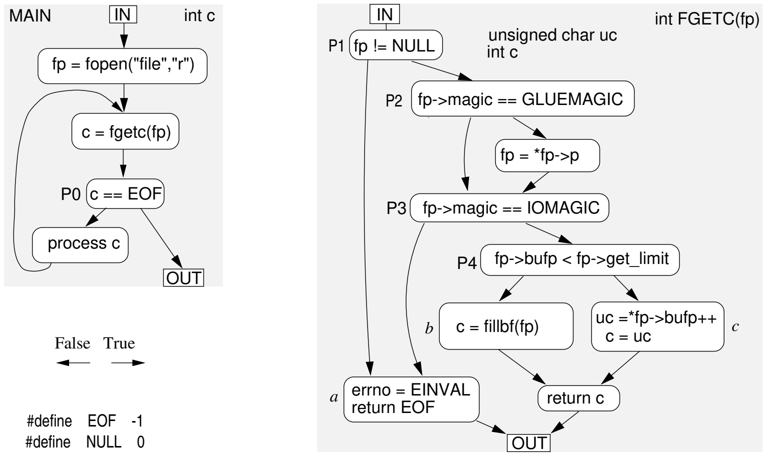
This research implemented the analysis and experimentally investigated the amount of interprocedural correlation detected and the cost of the analysis. Our measurements performed on a subset of SPEC95 programs provide insight into the interprocedural correlation that can be detected statically and its usability for compiler optimizations. It was found that not only the number of conditionals with some correlated paths greatly increases with inter-procedural analysis, but also the effect of branch elimination is more significant because many short, frequently taken interprocedural correlated paths exist. This observation serves as a motivation for performing the VNG analysis *inter-procedurally*.

It was also observed that some correlated branches are correlated along long (usually inter-procedural) program paths. Such correlations require time-demanding analysis. A simple heuristic for controlling the extent of the demand-driven analysis was developed and evaluated: the analysis of each branch was allowed to examine only a few hundred nodes. Paths that were not completely analyzed were (conservatively) assumed to have no value reuse. Such *early termination* reduced the analysis cost by an order of magnitude, while it sacrificed the detection of only a fraction of correlation. Note that the missed correlation was mostly present along long paths and hence it would not be exploitable anyway, due to much duplication required in the transformation stage. This observation serves as a motivation to construct the VNG *on demand*. Since, under the successful early termination heuristic, the value reuse paths are not examined completely, it is desirable to build only the portion of the VNG that *will be* analyzed.

7.1.2 Motivation

We illustrate interprocedural branch correlation on a small application program that uses the stdio GNU C library. The program is shown in Figure 7.1. Function MAIN first opens a text file by a call to fopen and then iterates through each character in the file until EOF is reached. The characters are obtained by a call to fgetc, which returns a character from a buffer that is filled by calling fillbuf.

Consider first the conditional branch P0 in MAIN. This branch is redundant along all incoming paths, hence it can be fully removed. Let us analyze the three paths leading to the branch. Along the path starting at the node a in fgetc, the branch will always exit the main loop, because the node a returns the value EOF. Hence, the *true* outcome of the branch is correlated with the node a. Along the path from the node



(a) The source program.

Figure 7.1: The example program using the GNU C library (version 1.09).

c, the branch will always continue the main loop, because the value uc fetched from the buffer is unsigned and hence always different than the constant EOF, which is -1. Hence, the *false* outcome of the branch is correlated with the node c. The examination of the procedure fillbuf (not shown in the figure) would show that the paths from the node b either return EOF or an unsigned character, just like the nodes a and b.

In summary, the outcome of the branch is redundant along all incoming paths. This section presents an analysis that detects this kind of inter-procedural correlation. While the branch is optimizable along all paths, the optimization differs along the incoming paths (some paths are true-correlated and some a false-correlated). The optimization is possible via interprocedural separation of the paths, which is a program transformation presented in Chapter 7.

Next, consider the conditionals P1, P2, and P3 in the function fgetc. Within the inter-procedural loop created in main, these branches are loop-invariant, because their outcomes are the same in each iteration. The examination of fopen (not shown in the figure) would show that, for each path emanating from fopen, either fp=NULL or fp≠NULL∧fp−>magic=IOMAGIC holds. In either case, all three correlate. As a result, they are not only loop-invariant, but they are also redundant along all paths and thus can be fully removed. Summarizing the example, in the original loop, five conditional branches are executed in each iteration. After the optimization, only one conditional remains.

7.1.3 The demand-driven algorithm

As already mentioned, the demand analysis builds the VNG on the fly, but using only two of its three components—back-substitution and dataflow analysis, which are folded together into (essentially) a single demand-driven pass. Value numbering, which collapses the value threads, is not engaged because it does not fit the demand-driven paradigm. To understand the reasons for it, recall the three steps of the (exhaustive) VNG construction and analysis.

1. Place threads starting from the user (i.e., optimized) computations, using symbolic back-substitution, in a *backward* direction.

2. Collapse threads, using value numbering, in a *forward* direction.

3. Solve dataflow problems, in a *forward* or *backward* direction.

Branch correlation is computed as the problem of availability of branch conditions, a *forward* problem. To compute a forward dataflow problem on demand, the demand-driven analysis proceeds in a backward direction [DGS97]. This reversed direction allows us to fold together the (backward) placing of threads (the first step of the VNG construction) with a demand-driven version of its third step (computing availability). Thus, given a user computation c, the demand-driven VNG analysis performs two steps.

1. Place threads leading to c, using symbolic back-substitution, in a *backward* direction. While placing the threads, look also for computations that generate or kill the value of c, thus *solving* the availability of c.

2. Complete the dataflow analysis of c, by *marking* the threads leading to c with its availability solution. This step proceeds in a *forward* direction.

The unsuitability of value numbering can now be explained. There are two reasons. First, the demand analysis folds the first and the third steps of the exhaustive construction, bypassing the second step, which cannot be combined with them because it proceeds in the opposite direction. Second, because we build threads only for the optimized computations, there are no threads with which they could be merged.

Figure 3.7 on page 27 illustrates which reuse is captured by demand VNG analysis. When S_5 is being analyzed on demand, it is found to recompute the value of S_2, because they lie on the same thread. In contrast, when S_4 is analyzed on demand, its recomputation of S_3's value is missed, because that requires placing S_3's thread (this is the second reason above), and collapsing this thread (this is the first reason above).

One more important issue must be explained. Clearly, the described demand-approach works only for forward dataflow problems. For backward problems, their (reversed) demand-driven direction is not aligned with the backward direction of back-substitution. This deficiency is not significant in some situations. First, for removal of conditional branches, the solution to anticipability (a backward problem) is not needed, as will be explained in Chapter 7. Similarly, some version of speculative PRE do not require anticipability (namely, the PRE estimator in Chapter 5). Second, backward problems can actually be computed on the VNG that is built on demand, albeit imprecisely: once the threads have been placed by the demand analysis, backward problems can be computed on them. Because not all threads have been placed, a (conservatively) imprecise solution may be obtained.

7.1.3.1 Query propagation

Our demand-driven VNG analysis is presented in the context of branch-correlation detection. Our analysis is demand-driven from a given conditional node: only the nodes that may lie on a correlated path are visited and only the relevant data flow information is computed. The analysis is initialized by raising a query at the conditional that corresponds to asking a question “is the outcome of the conditional with the predicate (v relop c) known along some incoming paths?” The form of the raised query is (v relop c), where v is a variable and c a constant. Note that the query format can be made arbitrarily more general, without affecting the algorithm presented here. In fact, the query format corresponds directly to the language of symbolic names P from which we draw the symbolic names for the VNG construction, which we are allowed to pick freely.

The query is then propagated from the conditional backwards along all paths in the ICFG until it can be resolved on these paths. Resolving a query at a node produces one of three answers: TRUE, FALSE, UNDEF. The first two answers are a minor extension of the *Must* value in the path-sensitive lattice (see Definition 4.5). They indicate that the path along which the query reached the node is correlated. TRUE means that the outcome of the conditional along the path is true (i.e., the true outcome must be taken). The answer FALSE means the opposite (i.e., the false outcome must be taken). The UNDEF means that the outcome is unknown because the variable is assigned an unknown value.

For resolving a query, we have identified four sources of static correlation.

- A query is always resolved TRUE or FALSE at a node that assigns a constant to the variable v from the query.
- A conditional branch that involves the variable v. The assertions on variables that exist on the true and false out-edges of the conditional may define the outcome of the predicate in the query. Note that a conditional correlates with itself if there is a path around a loop along which the query variable is not defined.
- A type conversion from an unsigned to signed value, as in the example in Figure 7.2. The result is always non-negative, which may determine the branch predicate outcome.
- After a pointer variable is dereferenced, its value is guaranteed to be non-zero; otherwise a segmentation fault would have occurred.

During the propagation, a copy assignment to the query variable may be encountered, e.g., v := w. When this happens, the query is modified to reflect this assignment before it continues to propagate. This simple form of symbolic back-substitution is essential to capture assignments to and from temporaries, common subexpressions, procedure return values, and parameter passing. As a consequence of this substitution, multiple distinct queries can be raised at a single node. For the query format fixed above, $(v\ \mathsf{relop}\ c)$, at most V number of queries will be raised at each node, where V is the number of program variables. For more general query formats, the w-limiting of back-substitution will have to terminate the query propagation (see Section 3.11 on page 31).

After the analysis terminates, the resolved queries are rolled back along the paths they traversed. The goal is to collect all resolved answers to each query raised at a node. Starting at the successors of nodes where a query was resolved, answers are propagated forward and merged by a set-union operation at control flow merge nodes. At any node, including the conditional itself, each query may have from one to all three possible answers from $\{\mathsf{TRUE}, \mathsf{FALSE}, \mathsf{UNDEF}\}$. For example, if the query raised at the conditional has answers TRUE and FALSE, then there are some correlated paths leading to the conditional where the outcome is true, some correlated paths where it is false, and no paths along which it is unknown. Such a conditional has full correlation.

7.1.3.2 Computing procedure summary nodes

The interprocedural analysis used an *interprocedural control flow graph (ICFG)* that combines CFGs of all program procedures by connecting procedure entries and exits with their call sites, as depicted in Figure 7.3. All edges in the figure define the predecessor-successor relation for nodes. Each procedure

```
Analyze predicate ($v$ relop $c$) in conditional branch node $b$
    initialize $Q[n]$ to $\{\}$ at each node $n$
    form the initial query $q_b = (v, \text{relop}, c, \text{nil})$
    raise_query(pred$(b), q_b)$
    while worklist is not empty do
        remove pair (node $n$, query $q$) from worklist
        case $n$ is entry node of a procedure $p$:
            if $q$ is a summary node query then $A[n, q] :=$ TRANS; add $q$ to $q.sne.$entries$[n]$
            else if $n$ has no predecessors then $A[n, q] :=$ UNDEF
            for each call site node predecessor $m$ of entry node $n$ do
                if $q$ is a summary node query for $j$th exit of $p$ then
                    if $q.sne.q_{in}$ is raised at $j$th exit of $m$ then raise_query$(m, q)$
                else raise_query$(m, q)$
            end for
        case $n$ is call site exit node:
            let $ex$ be the procedure exit predecessor of $n$
            let $m$ be the call site predecessor of $n$ and $en$ the entry node invoked by $m$
            if summary node entry $sne[ex, q]$ does not exist then
                let $q_{sn}$ be a copy of $q$
                $sne[ex, q] := (q_{sn}, ex, \{\})$; $q_{sn}.sne := sne[ex, q]$
                raise_query$(ex, q_{sn})$
            else if $sne[ex, q]$.entries$[en]$ does not exist then
                $sne[ex, q]$.entries$[en] := \{\}$
                raise_query$(ex, sne[ex, q].q_{in})$
            end if
            add $A[ex, sne[ex, q].q_{in}] \setminus \{\text{TRANS}\}$ to $A[n, q]$
            for each query $q_a$ in $sne[ex, q]$.entries$[en]$ do raise_query$(m, q_a)$
        otherwise :
            answer := resolve$(n, q)$
            if answer $\in \{\text{TRUE}, \text{FALSE}, \text{UNDEF}\}$ then $A[n, q] := \{answer\}$
            else for each $m \in pred(n)$ do raise_query$(m,$ substitute$(n, q))$
        end case
    end while

    Procedure raise_query(node $n$, query $q$)
        if $q \notin Q[n]$ then add $q$ to $Q[n]$; add pair $(n, q)$ to worklist
    end
```

Figure 7.2: The interprocedural static correlation analysis.

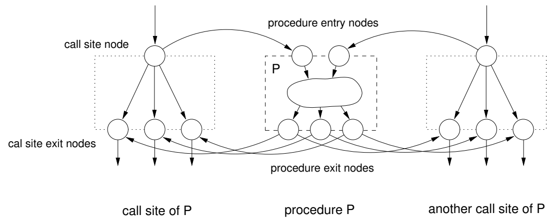
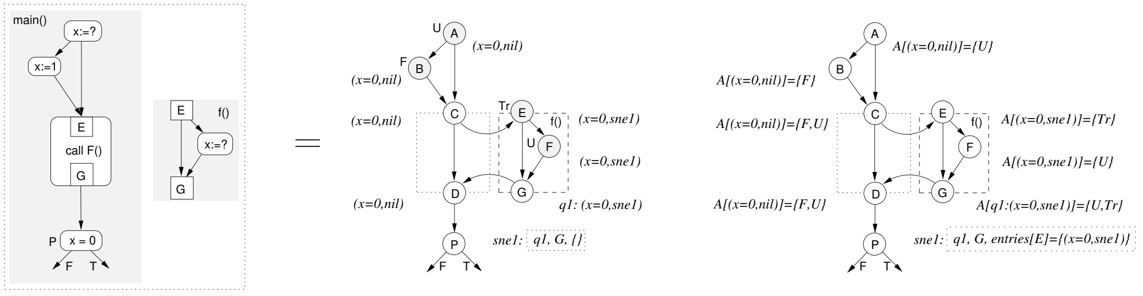


Figure 7.3: Interprocedural CFG in call site normal form.

can have multiple *procedure entry* nodes and multiple *procedure exit* nodes (to support the transformation from Chapter 7). The successors of a *call site* node are the procedure entry node and the associated *call site exit* nodes. The analysis algorithm requires the ICFG to be in the *normal call site form*, where a) each call site node has a single procedure entry successor and b) each call site exit node has exactly one call site predecessor and one procedure exit predecessor. It is assumed that the above nodes are dummy nodes with no program statements.

The computation of summary nodes is motivated by the demand-driven framework of [DGS95], which computes procedure summary nodes also on demand, in order to improve the efficiency of interprocedural analysis. Since in the analysis the queries are propagated through procedures backwards, summary node entries are stored at procedure exit nodes and for each query raised at the exit node we maintain: a) the answers resolved in the procedure, and b) the corresponding queries at the entry of the procedure, if the query propagated all the way to the entry node. All queries raised at procedure exit nodes are used to compute summary nodes and are, therefore, treated specially. When a *summary node query* reaches a procedure entry, it is not propagated to the callers, but resolved with the fourth kind of query answer, TRANS. This answer marks paths through the procedure along which the query was not resolved. The procedure is *transparent* along such paths and the summary node lookup must propagate queries (backward) and collect answers (forward) across call sites of transparent procedures. The analysis handles both call-by-value and call-by-reference parameters.

The analysis algorithm is given in Figure 7.2. The algorithm computes summary nodes without interrupting the analysis. Each query is a tuple $(v, \text{relop}, c, sne)$, where sne is used by summary node queries to keep a pointer to their summary node entries; for non-summary queries, this field is nil. The summary node entry for query q raised at exit node ex is a tuple $sne[ex, q] = (q_{sn}, ex, entries)$, where q_{sn} is the summary node query raised on the procedure exit node ex and *entries*$[en]$ is the set of queries propagated to a particular entry node en. (In this algorithm, a procedure is allowed to have multiple entry nodes, to support entry splitting (Chapter 7)). The analysis is started at line 3 by raising the initial query at the predecessor of the conditional to be analyzed. Line 4 terminates the analysis when no node with an unresolved query remains. Lines 6–13 handle procedure entry nodes. Summary node queries are resolved here to TRANS and are added to the summary node entry as having reached the particular entry node, as described above. The non-summary query is propagated to all call sites of this entry (lines 9 and 12). The summary node query is propagated only when the computation of the summary node was initiated at the exit of the call site (lines 9–11). Lines 14–26 process a call site exit node n. Predecessors of n are determined according to Figure 7.3. If summary node lookup in line 17 fails, a new summary node entry is created and a summary node query q_{sn} is raised. Lines 21–23 update the summary node after a previous split of a procedure entry/exit node. Line 25 resolves the query based on the answers saved in the summary node and line 26 propagates the query across the procedure when a transparent path through the procedure exists. Finally, any other kind of node may be



(a) source program (b) analysis (c) rollback

Figure 7.4: An example of interprocedural correlation analysis.

a source of correlation (lines 27–30). Function **resolve** attempts to resolve a query. If it fails, the query is propagated after it is back-substituted. The algorithm for collecting the analysis answers by propagating the forward can be easily derived from the analysis algorithm.

The analysis is illustrated with an example in Figure 7.4. The four possible query answers are abbreviated in the figure as T, F, U, and Tr, for query answers TRUE, FALSE, UNDEF, and TRANS. The analysis of conditional node P is initiated by raising a query $q : (x = 0, nil)$ at the predecessor of P (Figure7.4(b)). The entry nil signifies that the query does not compute a summary node entry. Since x is a global variable, it cannot be propagated across the intraprocedural edge (C, D). Instead, it is raised at the exit of procedure f, where it initiates computation of a summary node entry sne_1. The summary node entry is computed by raising a summary node query $q_1 : (x = 0, sne_1)$ at the procedure exit node G. The query is resolved at node F to UNDEF because an unknown value is assigned to x. The nodes where a query is resolved are highlighted in the figure. The scope of the summary node is limited to the procedure and hence q_1 is resolved at the procedure entry node to TRANS. Also, the query is recorded in the $entries[\mathsf{G}]$ field of the summary node entry. Whenever a summary node query reaches the procedure entry, a corresponding query is raised at the call site node. In our case, query $q : (x = 0, nil)$ is raised at node C. This query is subsequently resolved at nodes A and B to UNDEF and FALSE, respectively.

The analysis is followed by the rollback phase (Figure 7.4(c)). The answer for a query q is stored in $A[q]$ and consists of all answers for q reaching the node. Note that the UNDEF answer for q at node D was propagated from node C through the TRANS answer of the summary node query. The algorithm fpr performing the roll-back is shown in Figure 7.5.

7.2 Inter-procedural transformation: example and motivation

We illustrate the utility of entry/exit splitting on a small program that calls a library procedure. In this program, many branches are redundant, as their outcomes are determined by prior statements (assignments or other branches). We show that with our inter-procedural restructuring, ICBE eliminates the execution of these branches, even without resorting to inlining. The program, shown in Figure 7.2(a), was used in Section 7.1 to illustrate inter-procedural value-flow analysis. We use it here to demonstrate how our transformation can remove the redundancies detected by that analysis.

The program calls the stdio GNU C library (glibc version 1.09). Function MAIN first opens a text file by a call to fopen and then iterates through each character in the file until EOF is reached. The characters

```
Collect answers to queries raised during the analysis of branch $b$
add $q_b = (v, \text{relop}, c, \text{nil})$, the initial query raised at branch $b$, to $Q[b]$
let worklist be the set of nodes $n$ s.t. a query was resolved at a predecessor of $n$
while worklist is not empty do
    remove a node $n$ from worklist
    for each query $q$ from $Q[n]$ do
        if $n$ is $j$th exit of call site $m$ invoking $i$th entry of procedure $p$ then
            let $e$ be the $j$th exit of procedure $p$
            add $A[e, sne[e, q].q_{in}] \setminus \{\text{TRANS}\}$ to $A[n, q]$
            for each query $q_a$ in $sne[e, q].\textit{entries}[i]$ do add $A[m, q_a]$ to $A[n, q]$
        else if $q$ was not resolved at node $n$ then
            for each $m \in pred(n)$ do add $A[m, \textbf{substitute}(n, q)]$ to $A[n, q]$
        end if
    end for
    if an answer has been added to $A[n, q]$ for any $q$ then add $succ(n)$ to worklist
end while
```

Figure 7.5: **The roll-back algorithm.**

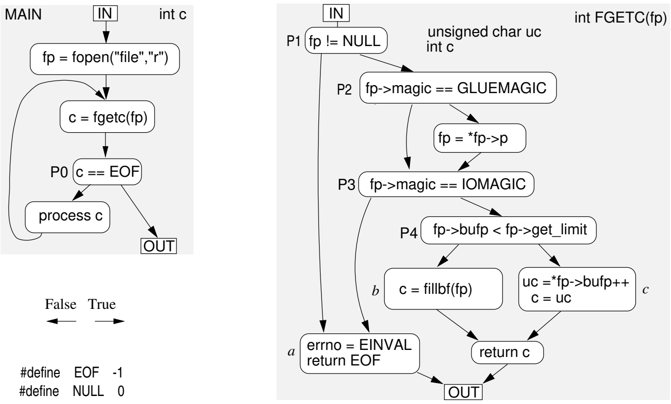
are obtained by a call to fgetc, which returns one character from its buffer until the buffer is empty, when it is refilled by calling fillbuf.

Consider first the conditional P0 in MAIN, which tests the loop-exit condition $c = \text{EOF}$. As was elaborated in detail in Section 7.1, the outcome of P_0 is always true along the path from a to P_0, and it is always false along the path from c to P_0. Assuming for now that the code of function fillbuf is unavailable to the optimizer, nothing can be deduced about its return value, and hence the behavior of P_0 cannot be determined along the path from node b to P_0. In summary, the conditional P_0 is partially redundant along two out of three (sub)paths reaching it. In other words, the analysis discovered a static correlation of P_0: whenever a or c is executed, P_0's outcome is known.

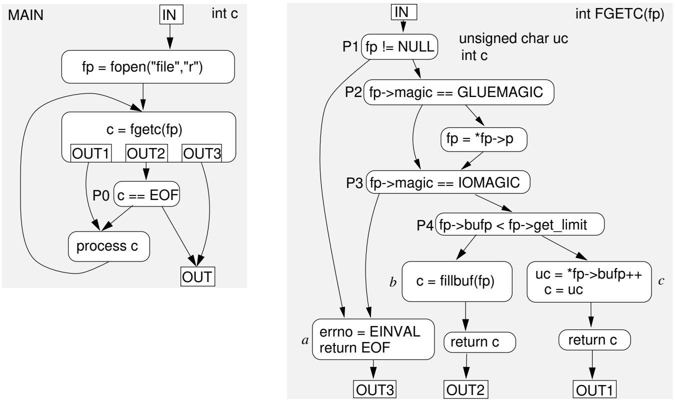
To optimize P_0, the two reuse paths must be separated. The separation peels off the two paths (a) from other paths, to isolate the optimization condition (along the isolated paths, the branch will be bypassed), and (b) from each other, because the branch will jump into a different target along each path. However, procedures pose obstacles to such desired path separation. Procedures are traditionally viewed as inherently single-entry/single-exit regions of code, which means that all paths through the procedure must pass through the unique entry and exit points. To exploit inter-procedural opportunities for conditional branch elimination, the correlated paths crossing procedure entry/exit must be isolated by splitting procedure entry/exit nodes.

To separate the paths across procedure return, we perform *exit splitting*, which creates two exits in the called procedure. After the exit is split, the conditional P_0 is bypassed (i.e., eliminated) each time it is executed, except after the buffer is refilled at node b (see Figure 7.2(b)). Exit splitting can be implemented by passing to the callee additional return addresses; in Section 7.4, we present a more efficient implementation technique whose cost is independent of the number of exits.

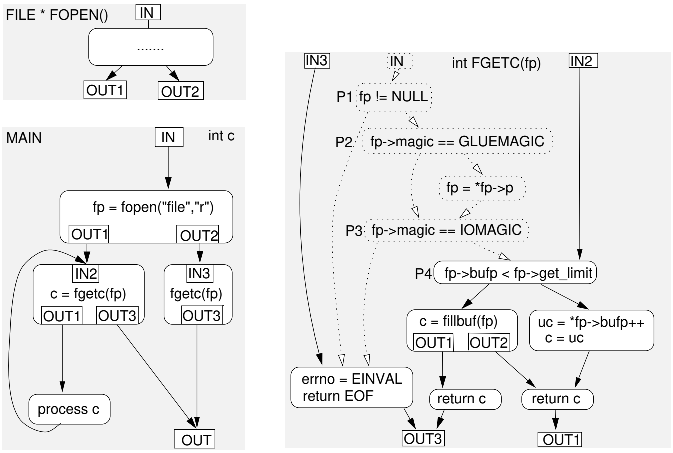
Next, we optimize conditionals P_1, P_2, and P_3 in function fgetc. For all three conditionals, the analysis detected reuse originating in procedure fopen, where either $\text{fp} = \text{NULL}$, or $\text{fp} \neq \text{NULL} \wedge \text{fp->magic} = \text{IOMAGIC}$ holds along any path. In either case, all three conditionals are *fully* redundant (that is, they can be eliminated along all paths, although separation is necessary (recall the reason (b) above). Our



(a) The source program.



(b) After optimization of P0.



(c) Elimination of P1, P2, P3; exit splitting on fillbuf.

Figure 7.6: **The example program using the GNU C library.**

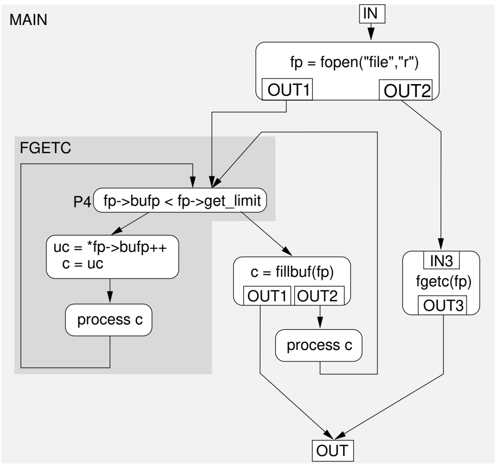


Figure 7.7: **Partial inlining of** fgetc.

optimizer splits the exit of fopen and the entry of fgetc to bypass these conditionals. The result is shown in Figure 7.2. If no other call site of this entry exists, the statements in fgetc that are reachable only from its original entry can be deleted if no other call site of this entry exists.

Let us assume now that the code of fillbuf is available to the optimizer. With this information, the analysis detects that fillbuf returns either EOF or an unsigned value. In either case, the outcome of P_0 is determined. After exit splitting of fillbuf, the conditional P_0 is completely eliminated, as shown in Figure 7.2(c).

To appreciate the power of the inter-procedural transformation, consider the amount of branches removed. In the unoptimized program, each loop iteration executes five conditional branches. After the optimization, only one conditional remains. This optimization cannot be carried out by intra-procedural branch elimination [MW95b], unless inlining is applied. Furthermore, ICBE reduces the code size of procedure fgetc, which enables its (partial) inlining into MAIN, where the resulting (reduced) loop can be efficiently software-pipelined (see Figure 7.7).

7.3 Inter-procedural PRE(R) algorithm

Because this chapter illustrates inter-procedural value-flow optimization by means of redundant branches (and not arithmetic computations, as in the previous chapters), we start by showing that the CMP region naturally supports the *intra*procedural removal of branches. We then proceed to extend the CMP-based restructuring to the inter-procedural setting.

7.3.1 Intra-procedural branch removal

Let us first briefly review the value-flow analysis for conditional branches, which was described in Section 7.1. For each conditional branch, along each path, the value-flow analysis may detect exactly one of three “events:” the branch outcome is always either *true*, or *false*, or not known at compile time. To phrase the three-valued branch redundancy into our two-valued (*Must*/*No*) value-flow framework, we treat the true and the false branch outcomes independently, as two complementary *branch exits*, which either are redundant

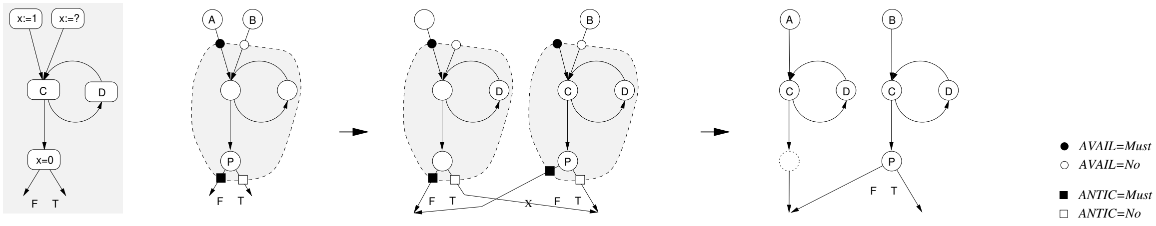


Figure 7.8: **Intra-procedural restructuring.**

along a given path or they are not redundant. For example, the conditional branch

$$P_0 : c = \textsf{EOF}$$

in Figure 7.6 is split into two branch exits

$$P_0^t : c = \textsf{EOF} \text{ and } P_0^f = \neg P_0^t : c \neq \textsf{EOF}.$$

Each branch exit can now be analyzed using the *Must*/*No*/*May* lattice. Given a path p leading to a branch, the solution of availability (*AVAIL*) has the following meaning:

Must: the branch exit is redundant along a path, i.e., the branch exit will be taken whenever the path is taken, and so the branch condition need not be evaluated.

No: the branch is not redundant along the path, i.e., whenever the path is taken, the branch condition must be evaluated to determine whether the branch exit (or the complementary exit) will be taken.

When the *AVAIL* solution for a branch is *May*, then there is a path along which the branch is *Must* (can be bypassed) and also a path along which the branch is *No* (must be evaluated).

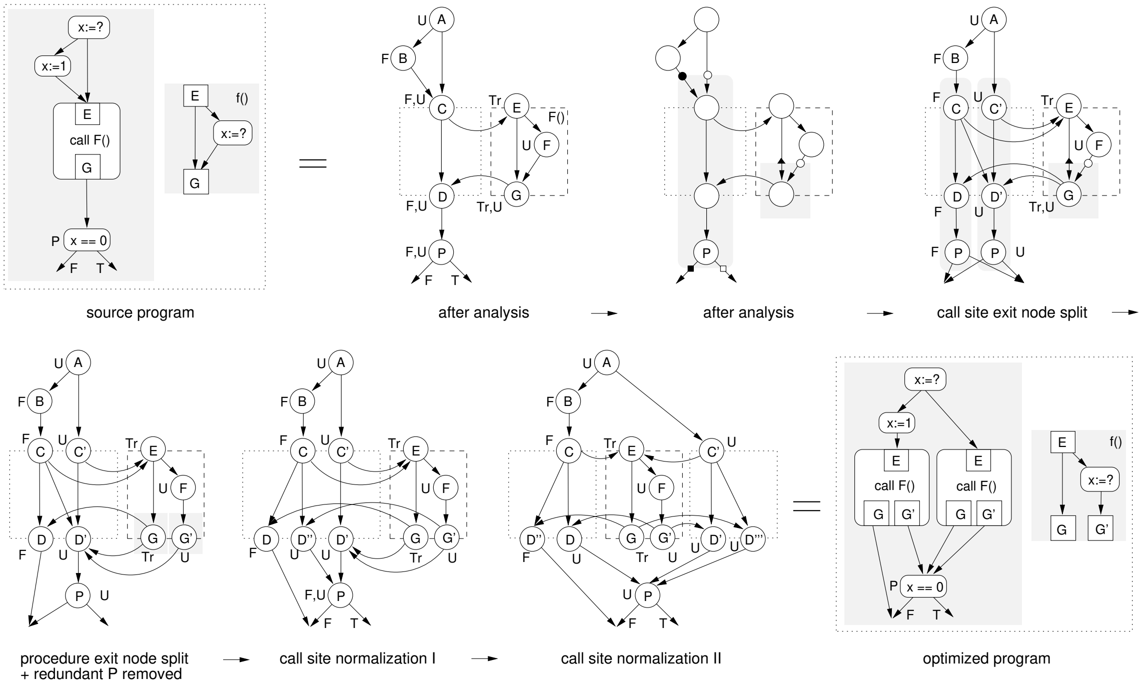
On the VNG, each of the complementary branch exits is associated with the out-edges of the conditional branch node, as shown in Figure 7.8(a). The figure also depicts the CMP region for such a conditional expression. Note that the branch exit coincides with the *Must*-exit edge of the CMP region. The reason for this conicidence is that the branch exit is only *May*-anticipated at the branch node (the branch exit will be computed along one path through the node but not the other). The consequence is that code motion is not applicable for the removal of redundant conditional branches (hoisting will immediately hit the CMP region) and so restructuring must be used instead.

The CMP region for the branch exit indicates how to restructure, just like it does for any other computation. After the CMP region is duplicated, the *No* copy must evaluate the branch. In the *Must* copy of the region the branch outcome is known and hence we can disconnect the complementary branch exit as unreachable and eliminate the branch itself, which concludes the transformation.

The interprocedural CMP region is computed on an interprocedural VNG representation, which is computed on the ICFG (see Figure 7.3) similarly to the way the VNG is computed on the CFG.

7.3.2 Inter-procedural restructuring

Entry splitting occurs when the correlated path is entering the procedure through a procedure entry node. Entry splitting always involves call site splitting. *Exit splitting* occurs when a correlated path crosses a procedure exit node. Exit splitting always involves splitting call site exit nodes.



source program = after analysis → after analysis → call site exit node split →

procedure exit node split + redundant P removed → call site normalization I → call site normalization II = optimized program

Figure 7.9: **Inter-procedural restructuring.** The label F denoes query answer *Must* and the label U denotes query answer *No*.

Thanks to the ICFG representation and the normal call site form, the duplication of the procedure entry and exit nodes require little special handling. All that is required is to consult the procedure summary function when attaching the edges that connect a call site node with its call site exit nodes. This handling is added to lines 13–17 in Figure 6.10. This code also normalizes the call site into the normal form shown in Figure 7.3.

Figure 7.9 illustrates inter-procedural restructuring. Figure 7.9(a) shows the source program and Figure 7.9(b) annotates the nodes of the ICFG with answers to the queries raised in the analysis from Figure 7.4. Figure 7.9(c) marks the CMP region. The triangle denotes a *Trans* entry edge. Note that the node E does not belong to the CMP region, although it lies on both a *Must*-path (reuse path) and on a *No*-path (free path). The node E need not be split because the call site node C will be split, which "sufficiently" splits the two paths. This fact is reflected in the *AVAIL* solution of the node *E*, which contains *Trans*, a single answer. Figure 7.9(d/e) show the ICFG after the CMP region in the caller/callee is duplicated, respectively. Notice how the call-site-related edges are attached. In particular, the edges (C,D) and (C,D') exist because the call site node C may follow in the callee both a *Trans*-path (producing a *Must* solution on the call site exit node) and a *No*-path. Finally, Figure 7.9(f,g) put the restructured call site into the normal form.

7.4 Implementation Details

This section elaborates on some important implementation details of entry/exit splitting. First, we present an efficient implementation of exit splitting. Second, we show how entry/exit splitting can be applied to call sites that invoke one of multiple procedures (e.g., virtual procedures in object-oriented languages).

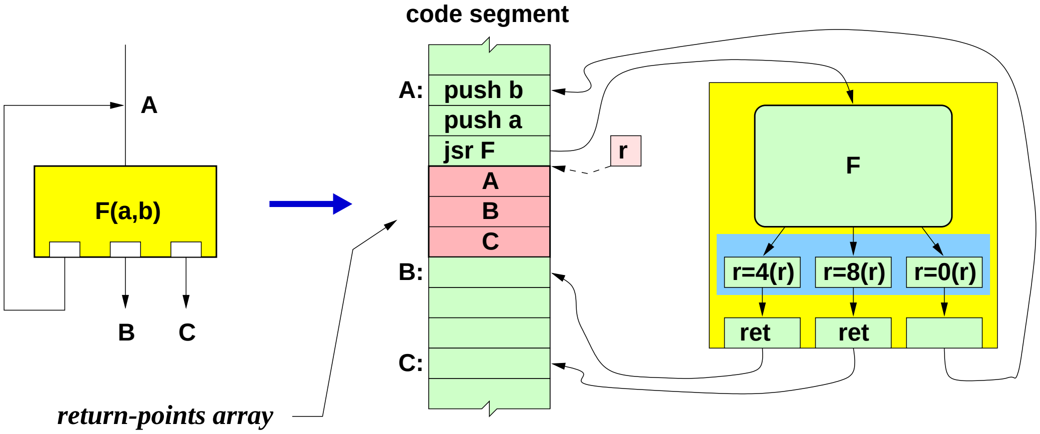


Figure 7.10: **Implementation of Exit Splitting.**

7.4.1 Exit splitting

A naive implementation of exit splitting would pass the alternative return addresses to the callee as additional procedure arguments. The dynamic number of instructions added to implement exit splitting in this style is proportional to the number of procedure exits, a potentially high cost.

Figure 7.10 outlines a method whose cost is independent of the number of exits. Rather than being passed as arguments, the alternative addresses are placed in the *return-points array*, a static memory block initialized at link time and stored in the text segment. The return addresses can be stored in the (immutable) text segment because the addresses remain the same throughout the entire execution, for each call site.

The return-points array starts immediately below the call site (*jsr*) instruction. Such placement has the benefit that the (machine-dependent) return register (denoted r in the figure) points to the beginning of the array, allowing a fast access to alternative return addresses. Given the access to the array, each of the procedure returns fetches its return address using a unique index to the return-points array. The cost of exit splitting is thus a single load operation, executed just before the return operation.

7.4.2 Entry/exit splitting of virtual procedures

Entry/exit splitting enables optimization of multi-target call sites, such as call sites of virtual procedures in object-oriented languages. Such call sites are generally not amenable to procedure inlining, which is a preferred (albeit potentially expensive) transformation for enabling inter-procedural optimizations. Inlining is not always allowed at virtual call sites because they may invoke a different procedure each time they are executed, depending on the type of the receiver object. Even when the call site invokes at most one procedure, the analyzer is not always able to safely confirm this fact.

Virtual call sites invoke the callee indirectly, by fetching its address from a table pointed to by the receiver object. Various organizations of the virtual lookup have been developed, but for our purpose we can assume that the address of the procedure entry is obtained by indexing the lookup table with the type of the receiver object and the method name:

$$ProcEntry = Table[Type, MethodName]$$

Entry splitting is possible even for such indirect calls, by simply indexing the lookup table with the type and the *call site*.[1]

$$\mathit{ProcEntry} = \mathit{Table}[\mathit{Type}, \mathit{CallSite}]$$

With such indexing, when two call sites a, b that call the same method m of type t invoke a different entry of $t :: m$ after entry splitting was performed, the table can distinguish the two entries. Most importantly, this lookup scheme does not require that a and b invoke a different entry of m for all possible receiver types. In particular, when type t' was not optimized, then all call sites will invoke the original entry of the method $t' :: m$. This simply means that $t' :: m$ will forgo the path-specific optimization opportunities created by splitting the paths leading to the original call site (where the entries were split). Conversely, when a (optimized) method $t'' :: m$ cannot take advantage the opportunities, its original entry will be called from all call sites.

The algorithm. To transform a virtual call site, the inter-procedural PRE(R) algorithm need not be modified at all. We only require that the ICFG representation connects each call site nodes with all its possible callees (i.e., the call site node will have multiple successors and the call site exit node will have multiple predecessors). A prerequisite for entry/exit splitting of multi-target call sites is that the number of callees is relatively small. When many procedures can be invoked, it is useful to know which of the callees are frequently executed, to allow a profile-guide selection of methods.

7.5 An application: inter-procedural conditional-branch elimination

This section puts together the demand-driven analysis from Section 7.1 and the transformation presented in this Section 7.3. We develop an optimization for removal of inter-procedurally correlated conditional branches. Because an PRE(R) algorithm is potentially expensive, our approach is profile-guided: we eliminate the conditionals based on their benefit (compute using the PRE estimator) and the code duplication cost.

For each conditional branch considered, the ICBE optimization performs analysis followed by restructuring. First, the conditional is analyzed to detect correlated paths and to determine the amount of code duplication required to eliminate the conditional. If correlation is found and the demands on code growth are acceptable, the program is restructured to create paths along which the conditional and instructions that compute its predicate condition are eliminated.

```
find conditionals matching analyzable pattern ($v$ relop $c$)
select conditionals with high execution count, if profile available
for each selected conditional branch $b$ do
    detect intra and inter-procedural correlation on conditional $b$
    if correlation found and required code duplication is below given limit then
        restructure program
    end if
end for
```

Implementation. The analysis and transformation algorithms were implemented in an interprocedural compiler that is based on the retargetable compiler `lcc` [FH95]. The implementation considered

---

[1] Another alternative is to index the table with the type, method name, and the index of the procedure entry. This will save table space in certain situations.

the correlation of those conditionals that compared a scalar variable (not a structure member) with a constant. Overall, 45% of conditionals in the benchmark programs could be analyzed using this pattern. The implementation included both an intraprocedural correlation analysis, which used MOD and USE [CK88] procedure summary information at call sites, and the inter-procedural analysis, which detected both intra- and inter-procedural correlations. The analysis recognized only two of the four sources of correlation: constant assignments and conditional branches.

Benchmarks. The experiments were performed on the integer SPEC95 suite. Since `lcc` does not generate correct code for the `126.gcc` benchmark, we used `lcc` itself as a compiler benchmark program. The programs are characterized in Table 7.1. The number of procedures, both defined in the program as well as the library procedures called are given in the table. The correlation analysis did not analyze library procedures and thus assumed the worst case behavior at their call sites. Each node in our representation corresponds to a dag of multiple operations and may be viewed as a high-level node. Therefore, the ratio of the number of conditional nodes to the number of all nodes that are executable is higher than usually reported (last 2 columns). Note that the number of all nodes in column 5 includes unexecutable label nodes. The dynamic profile information was collected from the `ref` input set.

Benchmark	source	procedures		nodes		cond/prog [%]	
program	lines	defined	library	all	cond	static	dynamic
099.go	29 246	372	11	38 806	5 304	21.4	29.0
124.m88ksim	19 915	252	35	21 657	2 416	16.5	30.9
129.compress	1 934	24	6	957	89	13.5	20.9
130.li	7 597	357	26	10 718	875	12.9	26.7
132.ijpeg	31 211	467	30	25 420	2 355	12.2	11.7
134.perl	26 871	276	69	50 596	5 623	16.6	29.1
147.vortex	67 202	923	63	104 154	9 646	12.9	28.0
lcc.3.5	26 467	470	21	49 775	5 863	18.1	32.1

Table 7.1: Benchmark programs.

Behavior of statically detectable correlation. First, we conducted experiments to determine the amount of statically detectable correlation for paths restricted to a procedure and for paths that cross procedure boundaries. The top-left graph in Figure 7.11 depicts the number of conditionals that exhibit *some* correlation; that is, those whose outcome is known along some, but not necessarily all, incoming paths. Using the total number of conditionals in a program as a base, the graph shows for each program the percentage of conditionals that were analyzable using our implementation, the percentage of conditionals that were found correlated using intraprocedural analysis and the percentage that were found correlated using interprocedural analysis. The results show that at least twice as many correlated branches are detected using interprocedural analysis than by using intraprocedural analysis. The top-right graph presents the same information weighted by the execution count of each conditional, showing that correlation is detected on conditionals that execute frequently.

The bottom two graphs in Figure 7.11 show the number of conditionals that had *full* correlation. The outcome of such conditionals is known along all paths and hence they can be completely eliminated; however code duplication might be necessary if both TRUE and FALSE correlations are discovered. Here, the benefit of interprocedural analysis is even more evident. If only fully correlated conditionals were to be optimized, the programs would execute between 3% and 19% less conditionals, while intraprocedural analysis enables reduction of only up to 8%. The fact that more useful correlation exists when procedures are

Benchmark	time [sec]		memory [MB]		node-query pairs	
program	overall	analysis	progrep	analysis	total	per cond
099.go	98.4	83.8	50.4	1.7	198 180	120.1
124.m88ksim	56.1	40.0	67.3	1.9	236 252	168.8
129.compress	2.1	0.7	10.4	0.3	6 620	120.4
130.li	9.8	4.6	35.9	0.8	27 201	102.6
132.ijpeg	33.3	19.8	52.7	0.6	32 961	33.5
134.perl	135.2	117.0	49.6	2.6	317 719	197.6
147.vortex	1070.2	1016.9	119.3	3.4	1 378 890	241.5
lcc.3.5	166.4	138.1	60.5	2.4	352 089	217.5

Table 7.2: The cost of correlation analysis.

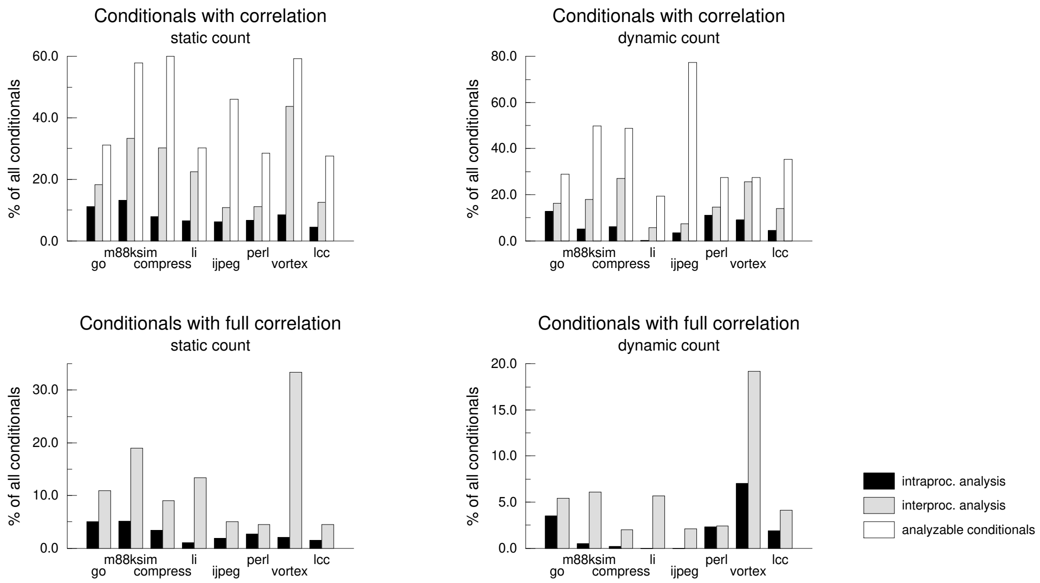


Figure 7.11: **Characteristics of statically detectable branch correlation.**

considered supports our hypothesis that we write procedures in an isolated fashion with repeated computation in the caller and callee.

The branch elimination optimizer replicates code to eliminate conditionals by creating separate paths. Since the amount of code duplication increases with the distance between the correlated branch and the source of the correlation, the extent of code duplication must be estimated before the interprocedural optimization is applied. Figure 7.12 plots the cost-benefit relationship for each correlated conditional. Each point in the graphs represents one conditional with a correlation. The x-coordinate of the point is the number of nodes that are created due to code duplication when the conditional is eliminated. The y-coordinate shows the amount of dynamic instances of conditionals that were avoided by the elimination of this conditional. A comparison with the intraprocedural results reveals that substantially more correlation is detected when procedures are considered, as the full-correlation graphs in Figure 7.11 suggest. But interprocedural correlation also requires more code duplication in many cases because the correlation may span a large part of the call graph. However, the amount of frequently executed correlated conditionals with low duplication needs, positioned in the upper-left quadrant, has increased with interprocedural analysis. These conditionals make ICBE more beneficial than intraprocedural elimination because with less code growth a higher reduction in elimi-

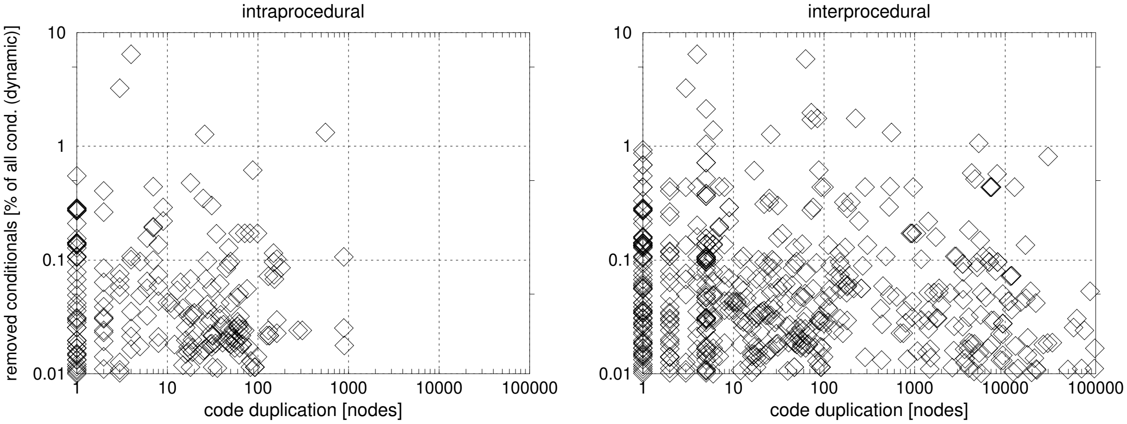


Figure 7.12: Contribution to branch removal vs. code duplication requirements for each correlated conditional.

nated branches can be achieved. The number of eliminated dynamic instances of each optimized conditional was estimated from the execution counts of the nodes where the analysis query was resolved.

Early termination is a heuristic that terminates the query propagation after a certain number of nodes have been visited. The outstanding queries are all resolved to UNDEF. In the experiments, the analysis was terminated after visiting 2000 nodes. As a result, the analysis was a magnitude faster on some benchmarks (vortex), while only a small fraction (within about 15undiscovered. Figure 7.12 provides an explanation: a majority of optimizable nodes require a small amount of duplication (and hence a small amount of query propagation steps). Not analyzing the expensive conditionals loses some opportunities but gains a lot of analysis speed (notice that the x-axis is logarithmic).

Eliminated Branches. The goal of eliminating only conditionals causing reasonable code growth is easily achieved in our approach, for ICBE optimizes conditionals one by one, performing first the analysis and then the restructuring optimization for each conditional. The amount of code growth necessary to optimize the conditional is determined during the analysis phase. The restructuring phase is executed only if the number of new nodes that must be created is less than a predetermined limit. We optimized the benchmarks with various values of the per-conditional duplication limit. Each conditional was optimized only if the number of node duplicates required for its optimization did not increase N, where N ranged from 5 to 200. Figure 7.13 shows the amount of conditionals eliminated and the incurred code growth. Each point in a graph corresponds to one duplication limit value. Note the different y-ranges in the bottom row.

In this experiment, the analysis was terminated after 1000 node-query pairs were examined (see line 5 in Figure 7.2) even though not all queries were resolved. Since in each program there are numerous conditionals that test global variables, early termination of demand-driven analysis avoids far-reaching propagation of their queries and dramatically reduces the analysis time. The early termination is made possible by demand-driven analysis. All queries remaining after the analysis termination limit is reached are conservatively resolved to UNDEF. Terminating the analysis after 1000 nodes is sufficient to find correlated branches that require up to approximately 300 duplicated nodes. Even though not all correlation is detected with early termination, the missed opportunities are likely to be prohibitive due to high code duplication demands. Terminating the analysis early thus seems to be a practical improvement. However, note that for some values of the duplication limit, the inter-procedural analysis may produce worse optimization for the `134.perl` benchmark than its intra-procedural counterpart. The reason is that the analysis termination limit was reached by examining the callees, missing the intra-procedural opportunity. This problem can be alleviated by experimentally increasing the analysis termination limit. Note that the results in Figure 7.11 and Figure 7.12 were computed with an infinite termination limit.

We can conclude that: 1) for a given code increase, ICBE can eliminate significantly more dynamic conditionals than its intra-procedural counterpart; 2) when more code growth can be tolerated, ICBE offers opportunities for additional branch elimination; and 3) the per-conditional limit on code duplication is an effective way to control overall code growth. A better heuristic for deciding whether to apply the optimization would also consider the amount of conditionals eliminated, as opposed to the incurred code growth alone, as was done in our experiments.

7.6 Related Work

7.6.1 Branch elimination

Intraprocedural elimination of conditional branches in loops was developed by Mueller and Whalley [MW95b]. ICBE extends their technique in several respects. First, it can detect and eliminate partial redundancy of branches in loop nests and across procedure boundaries. Second, even in the scope of a single procedure, ICBE is more powerful because it can detect redundancy that is apparent by examining multiple basic blocks along a path, as opposed to a redundancy due to a single basic block detected in their analysis. In addition, in ICBE, the analysis cost and the code growth incurred due to program restructuring can be controlled. Mueller and Whalley [MW92b] also investigated avoiding unconditional jumps by code replication. Krall [Kra94] developed code replication techniques to improve the accuracy of semi-static branch prediction to the accuracy of dynamic prediction.

7.6.2 Other benefits of entry/exit splitting

The primary benefit of ICBE is the reduction in the instruction count (and the schedule length) through the elimination of correlated conditionals and the operations that compute their predicate. In this section we discuss how both the correlation analysis and the inter-procedural restructuring can be applied in other areas of compiler optimization.

Procedure inlining. Most inter-procedurally visible opportunities for branch elimination can be exploited by inlining and subsequent application of intra-procedural elimination of conditionals [MW95b]. However, without the knowledge of correlated paths in the call graph, the pre-pass inlining process must resort to exhaustive inlining, at least in the critical program regions. Short of folding all procedures into a single, flat procedure, there is no guarantee that all statements involved in a correlation will end up within the same procedure, which is necessary to remove the branch. Clearly, pre-pass inlining incurs large code growth.

Inlining becomes more practical when it is directed by our inter-procedural correlation analysis. After correlation of a branch is detected, the procedures involved in the correlation can be merged by post-analysis inlining. Such a solution to ICBE may be desirable in an existing compiler where inlining and intra-procedural branch elimination are already supported. The code growth of post-analysis inlining may be further lowered by performing full ICBE (with inter-procedural restructuring), followed by *partial inlining* [HHR95], in which only frequently executed paths through the optimized procedure are inlined. However, inlining of recursive, virtual, or library procedures may not be feasible. In this case, our inter-procedural restructuring can be applied to carry out ICBE.

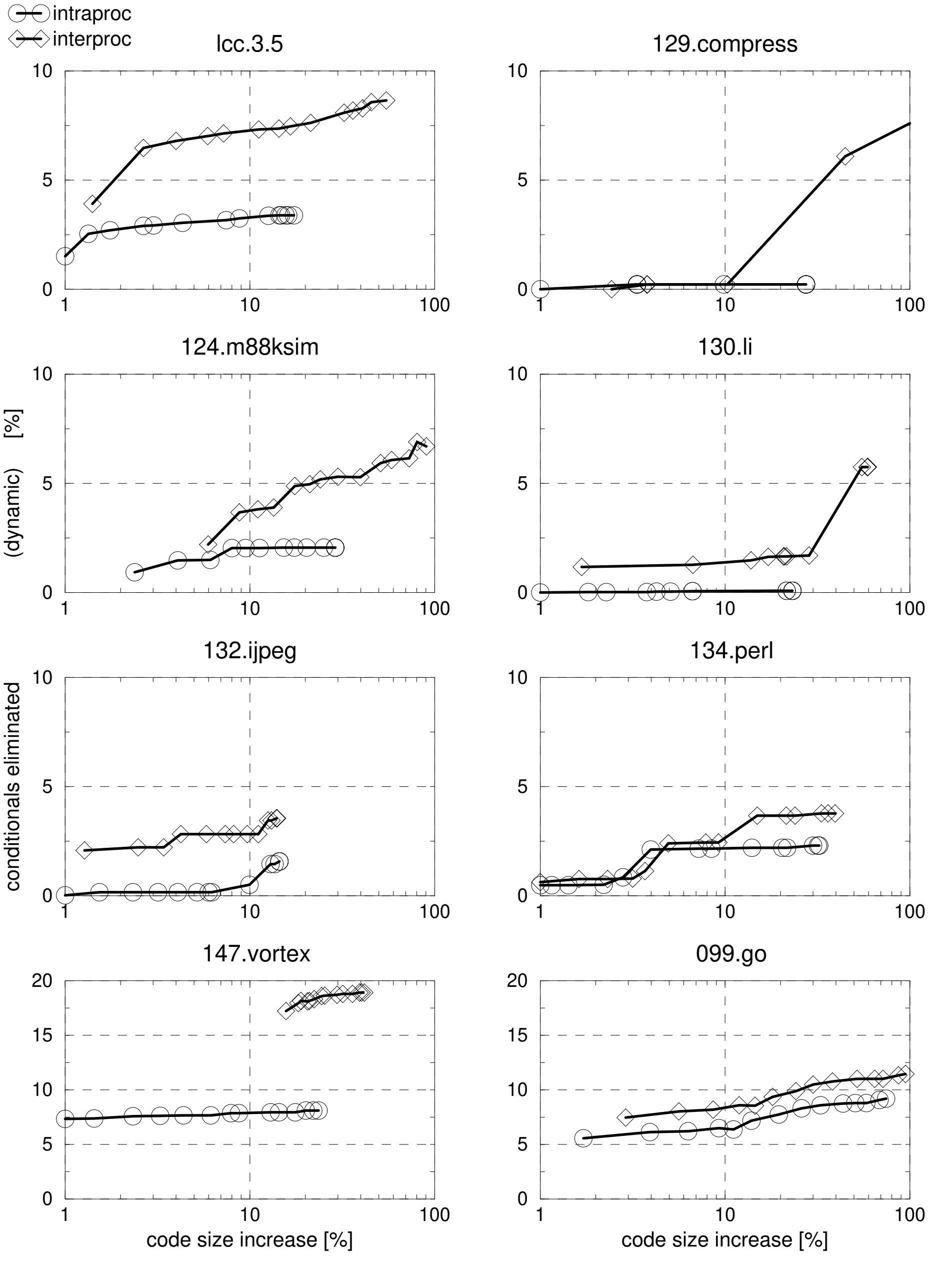


Figure 7.13: Reduction in executed conditional nodes vs. program code growth, for various values of the per-conditional code duplication limit.

Regardless of the exact ICBE scenario, the correlation analysis produces an upper bound on the code growth required to eliminate the conditional and, if profile information is available, provides also a profile-based estimate of the cost-effectiveness of the optimization before it is applied. The inlining algorithm in [AGS97] inlines procedures one by one based on their execution rate until a code growth budget is exhausted. Our correlation analysis can be used in the inliner to give procedures that generate correlation a higher priority so that correlated branches can be removed after inlining [Car95, DH88]. Our restructuring algorithm can be used to eliminate correlated branches after the code growth budget for inlining has been exhausted because its code growth demands are smaller than those of inlining. Richardson and Ganapathi [RG89] observed that the benefit of inlining comes mainly from eliminated procedure call overhead. Our analysis is able to identify procedures whose inlining will create intra-procedural optimization opportunities for branch removal.

Virtual calls sites. Object-oriented languages complicate inter-procedural compilation because call sites invoking member procedures of polymorphic types may transfer control to one of many procedures, depending upon the concrete type of the receiver object. Since such call sites require expensive dynamic dispatching, methods for their elimination through concrete type inference have been developed [AH95, PC94, PC95]. In these methods, demand-driven inter-procedural analysis determines for each call site the set of "reaching concrete types." Subsequent program restructuring separates out paths and clones procedures with the goal of creating call sites reached by a single type of the receiver.

There is an analogy between concrete type inference and our work in that both methods compute at optimizable nodes the set-union of all optimization opportunities and enable optimization by making the opportunities unique through path separation. While ICBE collects values of variables that determine branch outcomes, type inference is interested in the types of receiver objects. With respect to the restructuring algorithms, however, our transformation is more powerful than cloning because exit splitting is able to separate out paths that cross the exit node, which cloning cannot achieve. Section 7.4.2 described how entry/exit splitting is performed at virtual call sites. The entry/exit splitting can prove valuable for object-oriented languages because the cost of passing additional return addresses is small compared to that of a dynamic dispatch.

While concrete type analysis is very successful in enabling inlining at virtual call sites, some call sites will still require dynamic dispatch. In this case, we can use entry/exit splitting, as described in Section 7.4.2.

Fine-grain computer architectures. The elimination of conditional branches is especially important for wide-issue superscalar and VLIW architectures, in which instructions are pre-fetched and executed speculatively across conditional branches based on predictions of their outcomes. With increasing processor parallelism, branch density in the stream of instructions is becoming critical because expensive mechanisms are required to predict and issue multiple conditional branches in a single cycle [Joh91]. Our experiments have shown that between 3% and 18% of executed conditionals can be eliminated by ICBE, reducing branch density.

A mispredicted branch stalls the processor for many cycles and pollutes the instruction cache. Research in correlation-based hardware branch prediction [YGS95] shows that unpredictable branches exhibit correlation with earlier branches. Some unpredictable branches can arguably be eliminated by ICBE. Consider, for example, a procedure that removes an element from a linked list. When the average list length is low, the conditional that tests for an empty list is unpredictable. Nevertheless, the test is correlated with the conditional that tests the return value in the caller. Optimization of unpredictable branches has an especially high payoff.

ICBE can also be used to improve the effectiveness of software pipelining [LH96, RG81] by reducing the number of conditionals and other statements in the loop body, as illustrated by the example in

Figure 7.7. Elimination of branches can significantly speed-up the loop schedule when conditionals that form recurrent cycles of control dependencies are eliminated. Branches testing a flag whose value is assigned within the loop are examples of such conditionals.

Assisting hardware branch prediction. Run-time prediction schemes have been proposed that predict the outcome of a branch using its correlation with the last k branches [SLM96]. Since the exact source of the correlation is not known, all k outcomes are maintained and used for prediction, slowing down the learning process of the predictor. If the correlation is statically detectable, our analysis can provide the prediction hardware with directions about which recent branch(es) should be used for prediction.

Procedure cloning. Our analysis can improve the effectiveness of procedure cloning by performing inter-procedural analysis and applying intra-procedural restructuring. Instead of inter-procedural restructuring, information about the correlation that crosses procedure boundaries is used to clone copies of a procedure. To clone a procedure, the call node is split into multiple nodes, each of which has a call to a clone. Each clone copy can be modified to take advantage of the correlation. In previous work [CHK92], cloning is based upon constants while our approach will take advantage of correlation for cloning.

Library procedures. Even when it is not possible to compile the library procedures together with the application program, we can take advantage of correlation that crosses the application-library boundary. The library procedures can be pre-split by optimization with respect to characteristic application programs and the summary nodes describing the resulting entry/exit splitting can be conveniently stored with the library interface for later lookup during the optimization of the user program. For example, a separate exit from malloc would exist that would be taken when the return value is NULL. Since a large portion of correlation exists across calls to the same or related library procedures, the characteristic program may be as small as the one in Figure 7.6. The original unoptimized procedure entry must be maintained for library procedures. When this entry is invoked, all procedure exits return control to the standard return address so that compilers without ICBE can also use the library.

Inter-procedural optimizations. Because path separation and entry/exit splitting eliminate control flow merge points, conservative merging of data flow information at procedure boundaries is reduced. As a result, other optimizations, such as procedure cloning, partial redundancy and dead code elimination, may be more effective following inter-procedural restructuring. The ICBE optimization can be used to optimize array bounds checks [KW95, Gup90] which typically exhibit correlation. Finally, branch elimination can be used as a component of aggressive program transformations, such as slicing-based partial dead code elimination [BG97b].

Chapter 8

Experimental Evaluation

This chapter experimentally evaluates the power of the PATHFINDER framework. We develop an instance of the framework—specialized for removal of redundant load operations—and measure its various properties. In particular, we focus on comparing PATHFINDER with an ideal optimizer. In an ideal optimizer, value-flow *analysis* detects *all* reuse apparent from the program text, and the value-flow *transformation* removes *all* detected reuse. Our main focus is to measure the completeness of the VNG representation, which decides the success of value-flow analysis of the PATHFINDER framework. Completeness of the transformation stage was evaluated in Chapter 6, where PRE(MS) was shown to be near-complete.

To gauge the completeness of the VNG representation, we need an ideal program analysis, one that exposes all reuse present in the program. Because detecting all reuse is an undecidable problem, such ideal analysis cannot be carried out statically. We obtain the ideal performance via dynamic program analysis. By observing the run-time stream of memory references, we collect all PRE-exploitable reuse and treat it as the ideal analysis performance. To compare the (static) value-flow representation with the (dynamic) ideal value reuse, we use the estimators that compute, given a data-flow solution and a program profile, the dynamic amount of reuse detected by the static analysis. Our experiments show that about 55% of loads executed in SPEC95exhibit reuse. Of those, our analysis exposes about 80%.

This chapter starts by describing register promotion, an optimization that removes redundant load instructions. Next, the suitability of register promotion for evaluating the PATHFINDER framework is justified. Next, a method for measuring the amount of value reuse inherent in a program is presented, along with empirical measurements of the ideal reuse. Finally, this chapter compares the amount of reuse removed by PATHFINDER with the ideal amount.

8.1 Instantiating the framework: register promotion

Without comparison, caches are the best *hardware* defense against the von Neumann memory bottleneck. Capitalizing on data locality, caches win by *reusing* recent memory accesses. How can compilers benefit from these reuse opportunities? In the ideal case, the compiler promotes repeatedly accessed memory locations to registers. Register promotion is the best *compiler* solution for reducing the memory traffic. By removing redundant loads, register promotion decreases the dynamic operation count and shortens instruction schedules. This section describes register promotion and shows how it is derived from PATHFINDER. We also outline our evaluation approach and justify reasons for selecting register promotion as the basis of the evaluation.

Register promotion entails three subproblems. First, *load-reuse analysis* finds loads and stores that access the same address, together with the execution paths along which the reuse exists. In our framework, load-reuse analysis consists of the VNG representation (Chapter 3) and the dataflow analysis carried out on the representation (Chapter 4). Second, *alias analysis* verifies that the reuse detected by the VNG is not disrupted by intervening stores. In our framework, the effects of killing stores are incorporated into the dataflow analysis. Finally, a program *transformation* stores the prior memory access in a register and replaces the redundant load with a register reference. In our framework, transformation is phrased as partial redundancy elimination (Chapter 6).

There are three reasons for using register promotion as the basis for for the evaluation of PATHFINDER.

- *Removing memory accesses is an important optimization.* Memory operations are very expensive to execute in parallel, because they require multiple ports to the hardware cache. In comparison, a register access is, in most cases, a much cheaper operation.

  Additionally, *there are many load-reuse opportunities.* As our experiments show, many load operations are redundant and most of them can be removed.

- *Removing loads is an enabling optimization.* When values are passed between arithmetic and logical instructions via memory—by means of loads and stores—the optimization scopes broken up by the memory accesses is too small for detecting arithmetic value flow.

- *We can develop relatively precise dynamic analysis.* From the point of view of the dynamic (run-time) analysis, there is a value flow between two memory loads if they access the same memory location. In contrast, it appears that a dynamic detection of branch correlation would have a lower precision, because it would involve "mining" for correlation among Boolean values, rather than (unique) memory addresses.

This chapter focuses mainly on the first component of register promotion, load-reuse representation and analysis. Because an optimization is only as powerful as its analysis, improving the precision of the analysis is of high significance. The second component, alias analysis, has a different aim: while load-reuse analysis detects memory references that *must* go to the same location, alias analysis finds those that *may*, thus identifying killing stores. Recent research indicates that, for register promotion, a simple alias analysis may be sufficient [DMM98, LC97]. The third component, PRE transformation, was evaluated in Section 6, where it was shown to near-completely eliminate all *detected* reuse. In this chapter we concentrate on evaluating the *amount* of detected reuse by the VNG representation. We say that an analysis is *PRE-complete* if it detects all reuse that the PRE transformation can exploit.

Typically, optimizations are evaluated by reporting the amount of computations removed. Unfortunately, such absolute measure says little about how much potential remains unexploited. Instead, our evaluation measures the level of PRE-completeness: how far is the analysis from an ideal one? Because detecting load reuse is in general undecidable, we can only hope to find an approximation of the ideal reuse amount. For that purpose, we perform dynamic analysis of the program. Dynamic analysis is a simulation-based limit study: by observing the dynamic stream of memory references, we find all reuse available under a given input and use it as an upper bound of the PRE-exploitable reuse in the program.

In microprocessor-based optimizations, simulation limit studies have long been guiding the research direction and evaluating the designs. As a result, research processors offer impressive solutions to some compiler problems: memory disambiguation [CE98] and redundancy elimination [LS97]. In compiler optimization, too, limit studies can identify untapped potential, point to bottlenecks, and evaluate algorithms.

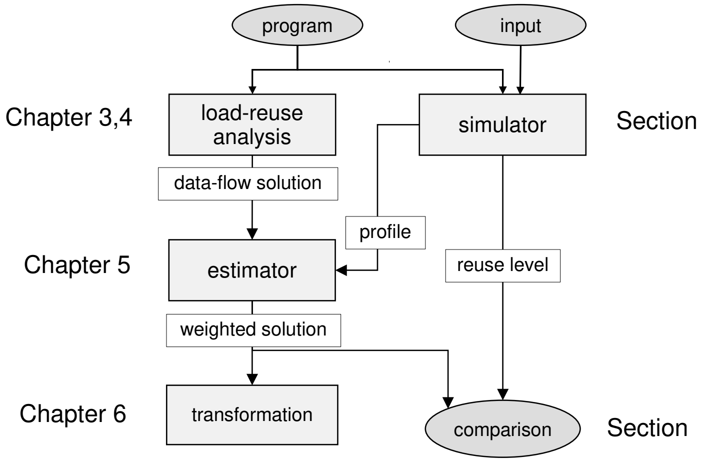


Figure 8.1: **The experimental setup.**

While the (static) load-reuse analysis identifies redundant loads and their reuse *paths*, the (dynamic) limit study yields the *run-time number* of redundantly executed loads. To compare these disparate quantities, we weight the static reuse using the program profile generated by the simulator. The result is the run-time amount of statically detected redundant loads.

Figure 8.1 outlines our experimental setup. Value-flow analysis composed of the VNG and the dataflow analysis carries out a static analysis of the program. The dynamic analyzer composed of a simulator runs the program on some input and outputs two pieces of dynamic information. The first is the reuse level, measured as the percentage of all executed loads that can reuse the value from a prior memory access. The second information is an edge profile collected during the simulation run of the program. The simulation-based dynamic analysis is described in Section 8.2. The profile is combined with the dataflow analysis solution, producing the dynamic amount of reuse detected by the static analysis. The analysis-detected reuse can then be compared with the ideal reuse level. The results are reported in Section 8.3. In PATHFINDER, the profile-weighted reuse is used, besides evaluating the analysis, to navigate the transformation.

8.2 Ideal amount of value flow

This section focuses on load reuse visible at run time. We present a simulation-based limit study that has multiple uses: **a**) measuring the amount of reuse in programs (how large is the optimization potential of register promotion?), **b**) evaluating the load-reuse analysis by providing a reference point (how close is the analysis to its ideal performance?), and **c**) tuning the analysis (which are the redundantly executed load instructions?). In this section, we describe the design of our simulation and show that a large fraction (55%) of loads executed in Spec95 exhibits reuse opportunities.

The primary use of the limit study is to evaluate the precision (completeness) of the load-reuse analysis. The precision is measured as the level of completeness. An analysis is T*-complete* if it detects all reuse that can be removed from the program with a program transformation T. In our context, T is the *partial redundancy elimination (PRE)* presented in Chapter 6. PRE is a code-motion transformation that can exploit reuse even when it exists only on a subset of execution paths incoming to the redundant load. Therefore, PRE has become the basis of modern register promotion techniques [CK94a, BG96b, SJ98, LCK^{+}98].

Unfortunately, detecting load reuse is in general undecidable [Ram94] and so no compile-time PRE-complete load-reuse analysis exists. Therefore, we use an empirical, run-time analysis that measures the reuse in the program as the program executes. In order to provide a close approximation of PRE-completeness, this simulation-based limit study should collect all reuse that PRE can remove, but no reuse that is beyond the power of PRE. The simulation should thus mimic the character of the PRE transformation.

As described in detail in Chapter 6, PRE removes redundancy by hoisting the partially redundant load against all control flow paths until it reaches a memory operation that generates the reuse. At this point, the contents of the promoted memory location is stored in a register that carries the contents to the original load. When PRE is performed on the VNG representation, load reuse is not restricted to acyclic paths; the reused value can be carried (using multiple registers) across a small number of loop iterations. (Recall the parameter w that instructs the back-substitution to build the VNG along w consecutive loop iterations (see Figure 3.10).) In summary, the PRE operational restriction is that the redundant load can reuse a result of some other static instruction (or itself), such that the result is a small number (w) of dynamic instances old.

The simulation algorithm reflects this PRE property. The run-time reuse is detected by remembering for each static memory instruction its *access history:* the dynamic stream of its recent addresses. A dynamic instance of a load is then redundant if a prior load or store accessed the same location without an intervening store. If an intervening store did occur, the load is still redundant; the intervening store becomes the reuse source.

The simulation technique has two contradictory goals. On the one hand, the limit study should yield an *upper bound:* each reuse that can be removed with PRE must be detected. On the other hand, the bound should be *tight:* if a reuse for a given static load is intermittent (e.g., because it is sporadic or input dependent), it should be filtered out as *noise*. In the example below, the reuse between recurrent array accesses (i.e., between the store of $A[i + 2]$ and the load of $A[i]$) is PRE-exploitable by allocating two registers that will carry the value for two iterations [CCK90, CK94a, BG96b]:

```
for (i=0; i<N-2; i++) { A[i+2] = A[i]; }
```

On the other hand, the reuse below is noise. While some consecutive loads from the hash table may access the same location, the reuse is not guaranteed to occur each time the program takes the path across the loop backedge. Therefore, PRE cannot exploit this reuse.

```
while (c=read()) { .. = hashtab[hash(c)]; }
```

To verify the PRE requirement that a path carries its reuse each time it is followed, the simulator would have to do extensive bookkeeping of followed paths. Consequently, we favor a noisier (i.e., less tight) upper bound over an expensive simulation. To reduce the noise, we limit the number of memory cells remembered in the access history of each static load and store. A small number h (1 to 4) of recent accesses is sufficient to capture most loop carried reuses, like the first example above [CK94b]. The simulation parameter h is a counterpart of the VNG parameter w.

PRE is inherently an instruction-level optimization. It is not capable of exploiting loop-level reuse, like the one between loads a and b below. Hoisting b does not work. Instead, the loops must be merged using loop fusion [CMT94], after which PRE can harvest the reuse.

```
for (i=0; i<N; i++) { a:  ..  = A[i]; }
for (i=0; i<N; i++) { b:  ..  = A[i]; }
```

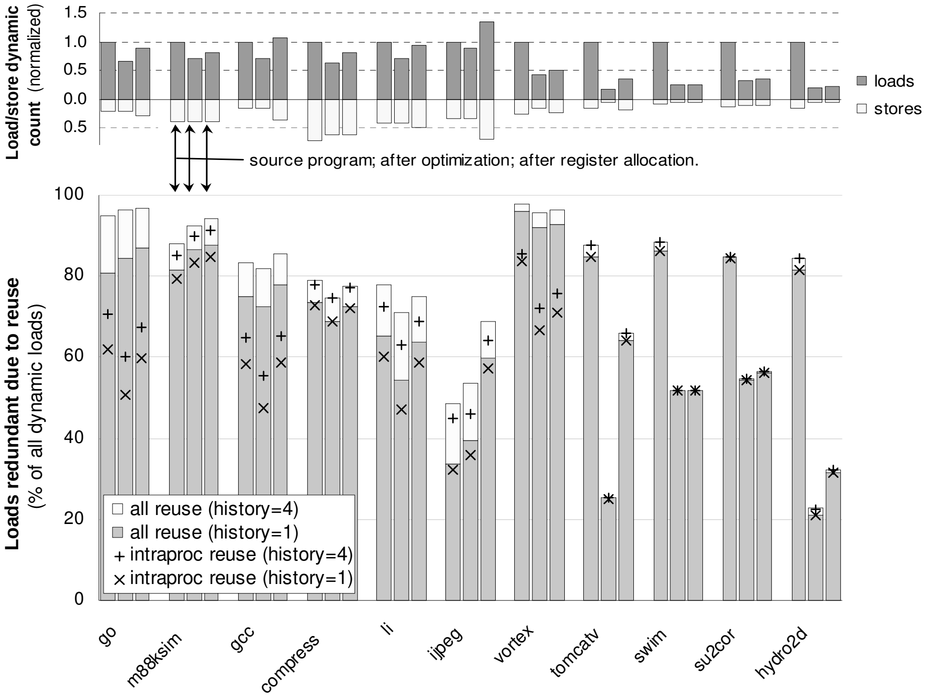


Figure 8.2: **Dynamically detected load reuse.** (Inlining: up to 50% code growth; Spec95 input set: *train*.)

The simulation algorithm will (correctly) not identify the load b to be redundant (unless $\mathtt{N} \leq h$) because the access history remembers only last h accesses made by load a. Hence, the simulation is consistent with the power of PRE.

Reuse Level. Figure 8.2 plots the amount of simulation-observed load reuse. For each benchmark, the experiment was carried out at three points in the compilation: for the original program, after optimizations, and after register allocation. The compiler used was Impact [CMC$^+$91]; the optimizations included the local, global, and loop invariant redundant load elimination, as well as superblock optimizations [HMC$^+$92]. Note that while in the floating-point benchmarks (the four on the right) the removal of many loads was accompanied by the decrease of observable reuse, in the integer benchmarks the optimizer left many redundant loads unoptimized, which suggests that programs with complex control flow require more powerful, path-sensitive optimizations and/or better alias information. Also note the increase in observed reuse after register allocation, which is due to spill-code loads (the target processor was PA-7100).

We show the amount of reuse for the history depth 1 and 4. Increasing the history depth raises the observable reuse much more in integer programs than in the scientific ones (where more recurrent accesses would be expected to be captured with the increased history h). A manual examination of simulation results strongly suggests that the additional reuse collected at the deeper access history is mainly noise, similar to the intermittent reuse in the hash-table example above. Also shown in the graph is the fraction of reuse in which both the generator and the redundant load belong to the same procedure. These reuse patterns are not strictly intra-procedural, as the procedure might have returned and been called during the reuse. However, these "intra-procedural" reuse levels serve as a reference point for our intra-procedural load-reuse analysis (Section 8.3).

Input Variance. Profile-directed *optimization* and simulation-directed *optimization design* are valuable only if the program input exercises input-independent, pervasive program characteristics. How

benchmark	**input**	**reuse** %	**reuse from** %		**l+s**
before opti	*train / test*	$h = 4$	loads	stores	10^6
`m88ksim`	dcrand.big	**87.9**	68.2	48.6	34
	dhry.big	**74.5**	90.4	13.6	135
`compress`	10^4 q 2131	**79.2**	56.1	71.4	13
ref →	5.10^5 e 2231	**71.3**	57.2	64.4	520

Table 8.1: **S** xecuted loads and stc

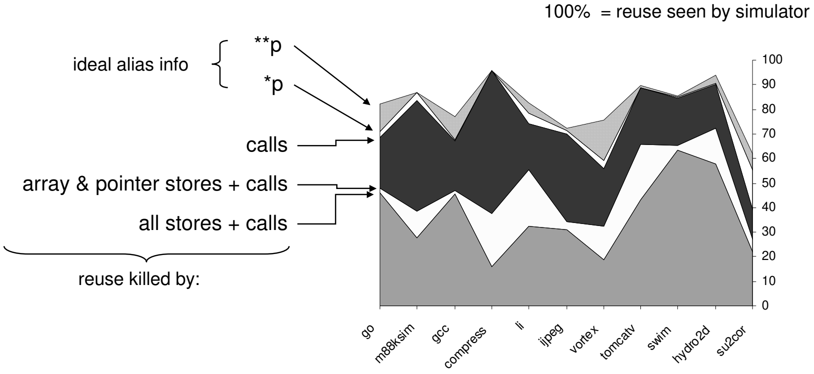


Figure 8.3: **Effects of symbolic language and pointer aliasing on the amount of detected reuse.**

much does reuse vary across different inputs? We modified the inputs on several benchmarks and compared the observed reuse. The results are shown in Table 8.1. The input-based variation of the reuse level is within 18%, which may suggest that reuse is largely input independent. The greatest difference is in `m88ksim`, in which each input directs the execution into different procedures. For the same reason, this benchmark has less reuse generated by stores in the `test` input (fractions add up to more than 100%, as a reuse instance may be generated by multiple instructions, a load and a store). We have manually examined `compress` and discovered that the lower reuse in the larger input is due to fewer noisy loads. Input variance may therefore be useful as a noise reduction mechanism; by taking intersection of reuse detected on different inputs, we may determine regular, statically detectable reuse.

Memory Requirements of the Simulator. While the simulation limit study is considerably more expensive than control flow profiling, it is used once (to design and tune the analysis) unlike the cheaper profiling which is repeated (to optimize each program). Still, the simulation speed was acceptable, at about 9.4 seconds per 1 million loads and stores executed (on PA–8000). The memory required varied greatly. The largest data structures were needed by `swim` (103MB + 32MB hash table) and the smallest by `compress` (4MB + the same hash table).

8.3 Completeness of value-flow analysis

This section experimentally evaluates the *static* value-flow analysis in relation to the *dynamic* limit study from Section 8.2. Because our implementation of the analysis is intra-procedural, the reference point for comparison is the intra-procedurally observed reuse. To minimize noise in the baseline, we use the reuse collected at the access history $h = 1$. We analyzed the unoptimized source programs. In summary, for each benchmark, the baseline for comparison is the “X” mark in the leftmost column in Figure 8.2. Figure 8.3 plots the amount of reuse discovered by the analysis. The plotted amount was computed as the average of the lower and upper bounds returned by the CMP^r estimator.

The load-reuse analysis was carried out under varying assumptions. The two highest lines in Figure 8.3 show the reuse detected at 1-level and 0-level address indirection, respectively. Our implementation considered only indirect loads, not stores, which may explain the lack of indirect reuse in some benchmarks. To determine the reuse-detection power of the analysis, these two lines assumed perfect aliasing under which no stores along a reuse paths would kill the detected reuse. While not all of this aggressively detected reuse can be promoted to registers, it can be exploited with alternative reuse mechanisms, such as data-speculative loads, as noted in Section 8.1. Overall, the comparison with the limit study shows that our analysis is about 80% PRE-complete.

Aliasing. We also studied the killing effects of intervening stores and procedure calls. Because our compiler does not perform alias analysis, we considered three hypothetical levels of pointer aliasing precision, specified as follows: first, we assumed that only procedure calls killed the detected reuse; second, we added to the kill set all stores except for stores to global variables; third, all stores and procedure calls killed the reuse. Due to aggressive inlining, only a small amount of reuse was lost at procedure calls (the white bar segments). However, array and pointer stores remove almost one third of reuse (the dark, middle segments). While this pessimistic hypothetical aliasing gives disappointing results, other researchers showed that even a simple alias analysis may produce memory disambiguation that is near-optimal for purposes of register promotion [DMM98, LC97].

Finally, we experimented with noise-reduction heuristics. We classified as noise all redundant loads whose observed reuse included many dynamically insignificant generators, but no dynamically frequent ones. Even with a conservative noise-reduction criterion, we filtered out about 20% of noisy reuse in `go` and 10% in `ijpeg`, as compared to the baseline in Figure 8.3. This allows us to conclude that our load-reuse analysis is successful; on average, at least 80% of observed reuse is captured.

8.4 Miscellaneous

A powerful load-reuse analysis is beneficial even when the register promotion itself is prevented (due to aliasing or lack of registers). In such a case, the PRE transformation step can employ alternative, albeit less effective, reuse mechanisms. When promotion is unsafe due to interfering stores, the redundant load can be replaced with a *data-speculative load*, which works as a register reference when the kill did not occur, but as a load when it did [KSR94, GKKG98, BG96b, Wu96]. When registers are not available, load reuse can be exploited using *software cache control* [KSR94, GKKG98, RCT$^+$98]. By directing which loaded values remain in the cache and which bypass it, the compiler can improve the sub-optimal hardware cache replacement strategy.

We have experimentally investigated how demanding PRE is with respect to the number of registers consumed. The *register pressure* at a CFG node is the number memory locations whose reuse path crosses that node; each memory location needs one register. We averaged the register pressure over all nodes, weighting each node by its profile frequency. For the 0-level perfect aliasing analysis configuration, the highest average register pressure was 34 registers for `su2cor`. Such an amount of registers will be soon available in general-purpose processors.

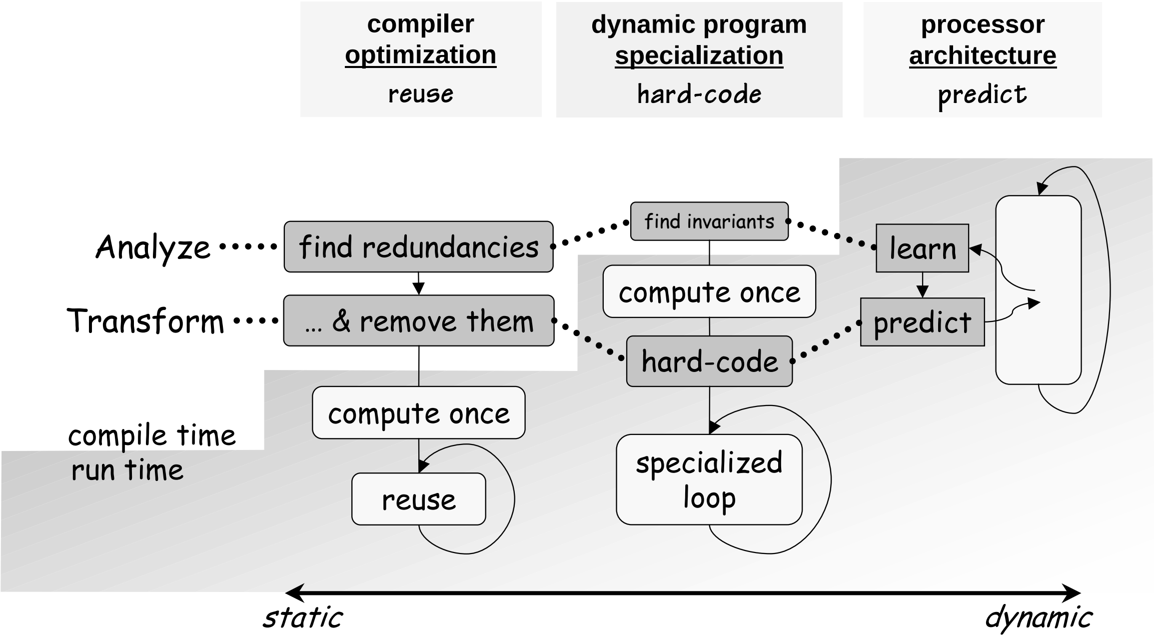


Figure 8.4: **The spectrum of program optimization approaches.** Are the three optimization technologies equipped with unique strengths or can one replace the others?

8.5 Other methods for value-flow optimization

We conclude this chapter by comparing PathFinder (a representative of the compiler optimization technology) with *value prediction* (a representative of architectural approaches to program optimization) [SVS96, SS97, LS97]. Before we describe the experiment, it is helpful to refresh the program optimization spectrum, shown in Figure 8.4. Qualitatively, the three technologies—compiler optimization, dynamic optimization, and processor architecture—operate on different principles. In particular, they differ greatly in how they exploit the program text (static analysis) and how they exploit the values computed by the program (dynamic analysis). The major question that our final experiment attempts to answer is how different these technologies are quantitatively. In particular, what are the redundant computations (and how many are there) that one technology can remove but the other can't?

The experiment compares the dynamic amount of reuse removed by PathFinder with the dynamic amount of values predicted by three different value predictors: the last value predictor [LS97], two-delta stride predictor [SVS96], and context predictor [SVS96]. The first two predictors allocated one element (to store the last value and stride) per static instruction. The size of the context predictor table was 64MB. The prediction and compiler optimization was performed only on (the results of) load instructions. The measurements are plotted in Figure 8.5. The following conclusions and notes can be made:

1. Except for the most expensive (context) predictor, there is a significant number of computations that can be removed only with the compiler.
2. Many of the computations predicted with the stride predictor could probably be removed with loop unrolling or strength reduction. The latter could be considered a form of value-flow optimization: it removed instructions that compute the value of some previous instruction plus some offset.

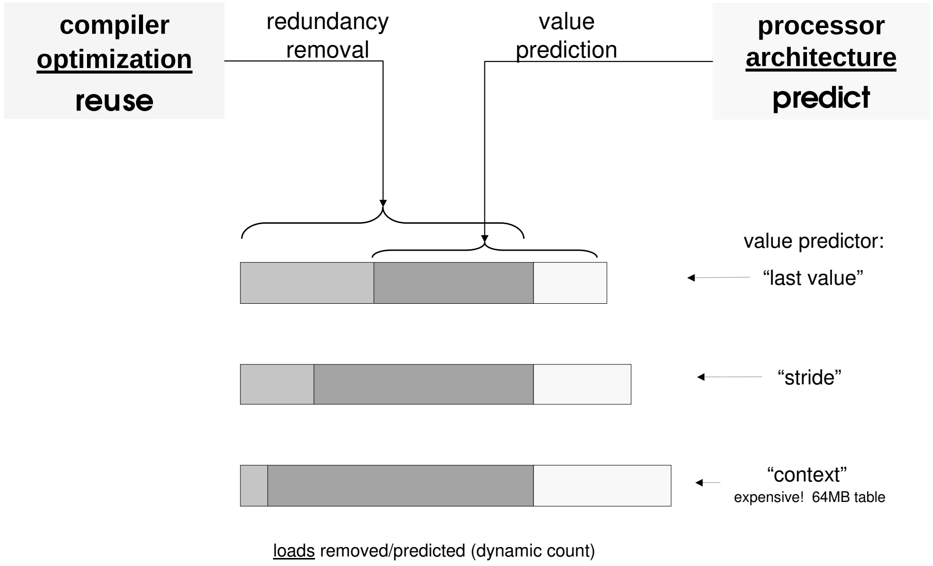


Figure 8.5: **Comparing the power of value-flow analysis and value prediction.**

It would also be interesting to compare PATHFINDER with *instruction reuse* [SS97, SS98], a hardware technique that does not predict the future values (like value prediction), but looks them up in a what we could call a *value-reuse table* (much like compiler-based value-reuse does). The value-reuse table is indexed by the values of the arguments of the operation that is being reused. In that sense, instruction reuse is even more similar to compiler optimization than value prediction.

Chapter 9

Conclusion

Value-flow optimizations are the backbone of compiler techniques for enhancing instruction-level parallelism. As a unifying paradigm, these optimizations exploit the program property that the results of some operations have been previously computed. If value recomputation is detected, the redundant operations can be removed or substituted with cheaper ones. As a result, program code is improved in at least two ways. First, by eliminating useless computations, instruction schedulers can use freed hardware resources to construct shorter schedules. Second, when operations are deleted from the critical path of program dependencies, the optimization achieves faster execution even when resources are not the bottleneck.

Observations of values computed during program execution suggests that compilers miss numerous value-based optimization opportunities: even in highly optimized programs, up to 40% of executed instructions compute the same value as their previous dynamic instance. As shown by our experiments, for most instructions, the source of this redundant recomputation are program expressions that are equivalent only along some—but frequent—execution paths. Because not all executions of an expression are optimizable, this *some-paths* redundancy is beyond conventional optimizers: conservative analyzers fail to expose it, and inflexible transformations fail to remove it.

This thesis unified, generalized, and improved *value-flow* optimizations. Particular emphasis was placed on *path-sensitivity,* with the aim of complete exploitation of opportunities that either may not exist along all execution paths or become visible only when individual paths are examined separately (or both). This goal was accomplished by first defining a class of value-flow problems and then by identifying its member optimizations. The value-flow class is broad and practically significant; a partial list includes loop-invariant code motion, constant propagation, load/store elimination, and branch removal. As the next step, problems shared among optimizations in the value-flow class have been identified, enabling development of an optimization framework in which common optimization issues can be addressed uniformly across the entire class.

The main result of this thesis is an optimization framework for deriving path-sensitive versions of value-flow optimizations. The major obstacle that prevents a construction of a *practical* path-sensitive framework is *exponential* cost incurred when the (exponentially many) *individual* program paths are optimized. This exponential cost comes in three forms:

- *Analysis cost:* each program path may have a *different* optimization opportunity.
- *Transformation cost:* only *some* program paths may offer an optimization opportunity.
- *Profiling cost:* to perform cost-benefit trade-offs, we may require the execution frequency of each program path.

This thesis presents techniques that make these three costs practical, while maintaining the optimization power at nearly optimal level. Specifically, the exponential cost was attacked by dividing the PATHFINDER optimizer into three stages: program representation, dataflow analysis, and program transformation. In each of the stages, this thesis develops new techniques. Their contributions are summarized next.

- *Representation.* We developed the *Value Name Graph (VNG)*, a novel program representation that models the flow of a recomputed value. By symbolically naming the value, the VNG reduces the (hidden) value flow into the (exposed) data flow. The representation, called *Value Name Graph*, names the value on demand, for a set of optimized computations. Although the names are formed separately for each path, obtaining path-sensitivity, the paths are analyzed separately only when it matters, i.e., when the value has a different name.

  By analyzing paths separately only when they offer different optimization opportunities (when the value has a different name), our representation reduces the exponential *analysis cost.*

- *Profile-weighted dataflow analysis.* To navigate the transformation trade-offs, the dataflow analysis must weigh the optimizable paths with a run-time profile. To make profile-guiding practical, this thesis develops a family of *estimator* algorithms based on *edge* frequencies, a cheap but inherently imprecise alternative to path profiles that measure frequencies of individual paths. When weighing the reuse, estimators bound the inherent error of the edge profile, adding confidence to imprecise profiles.

  By using the (linear-size) edge profile, the exponential *profiling cost* is avoided, while achieving estimation precision close to that based on path profiles.

- *Transformation (intra-procedural).* We developed a transformation that (nearly) completely removes all computations detected as redundant. The transformation combines three orthogonal methods. We resort to the expensive *path duplication* only when the growth-free *code motion* fails to transform the program, and only when the profile-driven *control speculation* cannot profitably impair some paths to optimize others. The spectrum of algorithms is based on a single abstraction, a *Code-Motion-Preventing (CMP) Region*, which contains identifies adverse effects of the control flow on the desired optimization. Our experiments show that the version of transformation that combines code motion and speculation is highly successful: it removes nearly all redundancies and, because it does not perform restructuring, achieves zero code growth.

  Thus, by exploiting code motion and control speculation, redundancies are removed without duplicating individual program paths, reducing the *transformation cost.*

- *Transformation (inter-procedural).*

  We developed a transformation that completely removes redundancies that are inter-procedural, i.e., those where the value flows across procedure boundaries. Rather than resorting to (expensive) inlining, we separate optimizable paths by generating multiple procedure entry points and multiple exit points (which may return to different points in the caller). Thanks to entry/exit splitting, paths can be separated across procedure boundaries, even when the call site invokes one of many procedures, as in virtual procedure calls.

  By splitting procedure entries and exits, fewer paths are duplicated than when procedure inlining is used, which reduces the transformation cost, similarly to the intra-procedural transformation.

We compared the power of the framework with that of an ideal value-flow optimizer, using the optimization of redundant loads. We developed a run-time program monitoring algorithm that exposed the amount of value reuse present in the program. This ideal amount was compared with the amount detected by our analysis. We observed that we captured at least 80% of the reuse present in the program.

Besides removal of redundant memory loads, the framework was employed to derive optimizations of arithmetic expressions and an inter-procedural version of removal of conditional branches, with promising results. We believe that the PATHFINDER framework can be used successfully also to derive other optimizations, such as removal of redundant array bounds checks or for constant propagation.

Bibliography

Bibliography

[ABD$^+$97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 85–96, 1997.

[AdJPS98] Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable cross-module optimization. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 301–312, Montreal, Canada, 17–19 June 1998.

[AGS97] Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive inlining. *SIGPLAN Notices*, 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*.

[AH95] Ole Agesen and Urs Hölzle. Type feedback vs. type inference: A comparison of optimization techniques for object-oriented languages. In *OOPSLA'95 Conference Proceedings*, pages 91–107, Austin, TX, 1995.

[AL98] Glenn Ammons and James L. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, 1998.

[APC$^+$96] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, 21– May 1996.

[ARZ92] F. Allen, B.K. Rosen, and K. Zadeck. *Optimization in Compilers, Chapter 6, Value Numbering* (unpublished). ACM Press/Addison Wesley, 1992.

[ASG97] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 134–145, June 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equalities of variables in programs. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

[BA98] Rastislav Bodik and Sadun Anik. Path-sensitive value-flow analysis. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998.

[BC94] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 159–170, June 1994.

[BDB99] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical Report HPL–99–78, Hewlett-Packard Laboratories, 1999.

[BG96a] Rastislav Bodik and Rajiv Gupta. Array data flow analysis for load-store optimizations in fine-grain architectures. *International Journal of Parallel Programming*, 24(6):481–512, December 1996.

[BG96b] Rastislav Bodik and Rajiv Gupta. Array data flow analysis for load-store optimizations in fine-grain architectures. *International Journal of Parallel Programming*, 24(6):481–512, December 1996.

[BG97a] Rastislav Bodik and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 159–170, June 1997.

[BG97b] Rastislav Bodík and Rajiv Gupta. Partial dead code elimination using slicing transformations. *SIGPLAN Notices*, 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*.

[BGS97a] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 146–158, June 1997.

[BGS97b] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 146–158, June 1997.

[BGS97c] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In *ESEC97*, pages 361–377. LNCS Nr. 1301, Springer–Verlag, September 1997.

[BGS98a] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 1–14, June 1998.

[BGS98b] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 1–14, June 1998.

[BGS98c] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Load-reuse analysis: Design and evaluation. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1998.

[BL94] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[BL96a] Thomas Ball and James R. Larus. Efficient path profiling. In *29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, 1996.

[BL96b] Thomas Ball and James R. Larus. Efficient path profiling. In *29th Annual IEEE/ACM International Symposium on Microarchitecture*, Paris, France, 1996.

[BMO90] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative programs. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 257–271, June 1990.

[BMS98] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: The showdown. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998.

[Car95] Paul R. Carini. Automatic inlining. Technical Report IBM Research Report RC-20286, IBM T.J. Watson Research Center, November 1995.

[CC77] P. Cousot and R. Cousot. Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 53–65, June 1990.

[CCK$^+$97] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 273–286, June 1997.

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 152–161, July 1986.

[CE98] George Chrysos and Joel Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *Computer Architecture News*, pages 142–153, New York, June 1998. ACM Press.

[CFR$^+$89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth Zadeck. An efficient method of computing static single assignment form. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)*, pages 25–35, Austin, TX, USA, January 1989.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CHK92] K. Cooper, M. Hall, and K. Kennedy. Procedure cloning. In *Intl. Conf. on Computer Languages*, Oakland, CA, 1992.

[CK88] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. *SIGPLAN Notices*, 23(7):57–66, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[CK94a] Steve Carr and Ken Kennedy. Scalar replacement in the presence of conditional control flow. *Software Practice and Experience*, 24(1):51–77, January 1994.

[CK94b] Steven Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, November 1994.

[CKL$^+$98] Fred Chow, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. Register promotion by partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 26–37, Montreal, Canada, May 1998.

[Cli95] Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 246–257, La Jolla, California, June 18–21, 1995.

[CMC$^+$91] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, volume 19, pages 266–275, New York, NY, June 1991. ACM Press.

[CMCH92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[CMT94] S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support forProgramming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 1994.

[CN96] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 145–156, New York, NY, USA, 1996. ACM Press.

[Coc70] John Cocke. Global common subexpression elimination. In Proceedings of a Symposium on Compiler Optimization, volume 5, pages 20–24, July 1970.

[CS69] John Cocke and Jacob T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Inst of Math. Sci., NYU, NY, NY, 1969.

[DGS93a] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 68–77, June 1993.

[DGS93b] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 68–77, June 1993.

[DGS95] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48, San Francisco, California, January 1995.

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

[DH88] Jack. W. Davidson and Anne. M. Holler. A study of a C function inliner. *Software, Practice and Experience*, 18(8):775–790, August 1988.

[Dha91] Dhananjay M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

[DI80] D. M. Dhamdhere and J. R. Isaac. A composite algorithm for strength reduction and code movement optimization. *International Journal of Computer and Information Sciences*, 9(3):243–273, June 1980.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 106–117, Montreal, Canada, 17–19 June 1998. *SIGPLAN Notices* 33(5), May 1998.

[DRZ92] Dhanajay M. Dhamdhere, Barry K. Rosen, and Kenneth F. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 212–223, July 1992.

[DS88] K. Drechsler and M. Stadel. A solution to a problem with Morel and Renvoise's "global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[DS93] K. Drechsler and M. Stadel. A variation of Knoop, Rüthing, and Steffen's *lazy code motion*. *ACM SIGPLAN Notices*, 28(5):635–640, May 1993.

[Dul98] Carole Dulong. The IA-64 architecture at work. *Computer*, 31(7):24–32, July 1998.

[ED95] Alexandre E. Eichenberger and Edward S. Davidson. Register allocation for predicated code. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 180–191, Ann Arbor, Michigan, November 29–December 1, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[FF92] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Boston, Massachusetts, October 12–15, 1992. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[FH95] Christopher W. Fraser and David R. Hanson. *A retargetable C compiler: design and implementation*. Benjamin/Cummings, 1995. ISBN 0-8053-1670-1.

[GBF97a] R. Gupta, D. Berson, and J.Z. Fang. Path profile guided partial dead code elimination using predication. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 102–115, November 1997.

[GBF97b] R. Gupta, D. Berson, and J.Z. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 358–368, December 1997.

[GBF97c] R. Gupta, D.A. Berson, and J.Z. Fang. Path profile guided partial dead code elimination using predication. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 102–115, San Francisco, California, November 10–14, 1997. IEEE Computer Society Press.

[GBF98] R. Gupta, D. Berson, and J.Z. Fang. Path profile guided partial redundancy elimination using speculation. In *IEEE International Conference on Computer Languages*, May 1998.

[GKKG98] Benjamin Goldberg, Hansoo Kim, Vinod Kathail, and John Gyllenhaal. The trimaran compiler infrastructure for instruction level parallelism research. Technical Report `http://www.trimaran.org`, Hewlett-Packard Laboratories, University of Illinois, NYU, 1998.

[GKT91] G. Goff, K. Kennedy, and C.-W. Tseng. Practical Dependency Testing. In *Proceedings Conference on Programming Language Design and Implementation*, pages 15–29, Ottawa, CDN, June 1991. ACM SIGPLAN. SIGPLAN Notices, 26(6).

[GMP$^+$97] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 163–178, New York, June 12–13 1997. ACM Press.

[GT93] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 90–99, June 1993.

[Gup90] Rajiv Gupta. A fresh look at optimizing array bound checking. *SIGPLAN Notices*, 25(6):272–282, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.

[HH97] R.Nigel Horspool and H.C. Ho. Partial redundancy elimination based on a cost-benefit analysis. In *Proceedings of 8th Israeli Conference on Computer Systems and Software Engineering (ICSSE'97)*, pages 111–118, Herzliya, Israel, June 1997. IEEE Computer Society.

[HHR95] Richard E. Hank, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Region-based compilation: An introduction and motivation. In *28th Annual IEEE/ACM International Symposium on Microarchitecture*, Ann Arbor, Michigan, 1995.

[HMC$^+$92] 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. In *The Journal of Supercomputing*, volume (this issue). 1992.

[HR81] L. Howard Holley and Barry K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, 7(1):60–78, January 1981.

[JC96] Johan Janssen and Henk Corporaal. Controlled node splitting. In *Compiler Construction, 6th International Conference*, volume 1060 of *Springer Lecture Notes in Computer Science*, pages 44–58, Sweden, April 1996.

[Joh91] William Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991. ISBN 0-13-875634-1.

[JS96] Richard Johnson and Michael Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual International Symposium on Microprogramming*, pages 100–113, December 1996.

[KEH91] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Enginnering, University of Washington, November 1991.

[Kno98] Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*, volume 1428 of *Lecture Notes in Computer Science Tutorial*. Springer–Verlag, Heidelberg, Germany, 1998. PhD dissertation, Department of Computer Science, University of Kiel, 1993.

[Kra94] Andreas Krall. Improving semi-static branch prediction by code replication. *SIGPLAN Notices*, 29(6):97–106, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[KRS92] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *SIGPLAN Notices*, 27(7):224–234, July 1992. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

[KRS93] Jens Knoop, O. Rüthing, and Bernhard Steffen. Lazy strength reduction. *International Journal of Programming Languages*, 1:71–91, 1993.

[KRS94a] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[KRS94b] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Trans. on Progr. Languages and Systems*, 16(4):1117–1155, 1994.

[KRS94c] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 147–158, New York, NY, USA, June 1994. ACM Press.

[KSR94] Vinod Kathail, Michael S. Schlansker, and B. Ramakrishna Rau. Hpl playdoh architecture specification: Version 1.0. Technical Report HPL–93–80, Hewlett-Packard Laboratories, 1994.

[KU77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[KW95] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. *SIGPLAN Notices*, 30(6):270–278, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.

[Lar99] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 259–269, 1999.

[LC97] John Lu and Keith Cooper. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 308–319, New York, June15–18 1997. ACM Press.

[LCK$^+$98] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng 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 (PLDI)*, pages 26–37, Montreal, Canada, 17–19 June 1998.

[LFK$^+$93] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[LH96] Daniel M. Lavery and Wen-mei W. Hwu. Modulo scheduling of loops in control-intensive non-numeric programs. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 126–137, Paris, France, December 2–4, 1996.

[LS97] M. H. Lipasti and J. P. Shen. The performance potential of value and dependence prediction. *Lecture Notes in Computer Science*, 1300:1043–??, 1997.

[MCB$^+$93] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. *ACM Transactions on Computer Systems*, 11(4):376–408, 1993.

[MCB99] Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient incremental run-time specialization for free. *ACM SIGPLAN Notices*, 34(5):281–292, May 1999.

[MH86] Scott McFarling and John Hennessy. Reducing the cost of branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, Tokyo, Japan, June 2–5, 1986. IEEE Computer Society TCCA, ACM SIGARCH, and the Information Processing Society of Japan.

[MLC$^+$92] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In Wen mei Hwu, editor, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, Portland, OR, December 1992. IEEE Computer Society Press.

[MR79] E. Morel and C. Renviose. Global optimization by supression of partial redundancies. *CACM*, 22(2):96–103, 1979.

[Muc97] Steven S. Muchnik. *Advanced Compiler Design and Implementation*. Addison Wesley, 1997.

[MW92a] Frank Mueller and David B. Whalley. Avoiding unconditional jumps by code replication. In *Programming Language Design and Implemenation Conference*, pages 322–330. ACM SIGPLAN, ACM Press, June 1992.

[MW92b] Frank Mueller and David B. Whalley. Avoiding unconditional jumps by code replication. *SIGPLAN Notices*, 27(7):322–330, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

[MW95a] Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 30 of *ACM SIGPLAN Notices*, pages 56–66. ACM SIGPLAN, ACM Press, June 1995.

[MW95b] Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. *SIGPLAN Notices*, 30(6):56–66, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.

[PC94] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *j-SIGPLAN*, 29(10):324–, 1994.

[PC95] John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Eighth Annual Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, volume 1033, pages 566–580, Columbus, Ohio, August 1995.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, September 1994.

[Ram96] G. Ramalingam. Data flow frequency analysis. In *Proceedings of the ACM SIGPLAN '96 Conf. on Progr. Language Design and Implementation*, pages 267–277, June 1996.

[Rau91] B. R. Rau. Data flow and dependence analysis for instruction level parallelism. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, LNCS, pages 236–250. Springer-Verlag, 1991.

[RCT$^+$98] Glenn Reinman, Brad Calder, Dean Tullsen, Gary Tyson, and Todd Austin. Profile guided load marking for memory renaming. Technical Report UCSD-CS98-593, University of California, San Diego, 1998.

[RG81] B. Ramakrishna Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. 14th Annual Workshop on Microprogramming*, pages 183–198, 1981.

[RG89] Stephen Richardson and Mahadevan Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3):137–142, 1989.

[RL77] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *Conference Record of the Fourth annual ACM Symposium on Principles of Programming Languages*, pages 104–118. ACM, ACM, January 1977.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, California, January 1988.

[SG95] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 62–73, January 1995.

[Sit93] Richard L. Sites. Alpha AXP architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

[SJ98] A. V. S. Sastry and Roy D. C. Ju. A new algorithm for scalar register promotion based on SSA form. *ACM SIGPLAN Notices*, 33(5):15–25, May 1998.

[SLM96] Stuart Sechrest, Chih-Chieh Lee, and Trevor Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 22–32, Philadelphia, Pennsylvania, May 22–24, 1996.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.

[SS97] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 194–205, New YOrk, June2–4 1997. ACM Press.

[SS98] Avinash Sodani and Gurindar S. Sohi. Understanding the differences between value prediction and instruction reuse. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, Dallas, TX, December 2–4, 1998.

[Ste96] Bernhard Steffen. Property oriented expansion. In *Proc. Int. Static Analysis Symposium (SAS'96)*, volume 1145 of *LNCS*, pages 22–41, Germany, September 1996. Springer.

[SVS96] Yiannakis Sazeides, Stamatis Vassiliadis, and James E. Smith. The performance potential of data dependence speculation and collapsing. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 238–247, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[TP95] P. Tu and D. Padua. Gated SSA-Based demand-driven symbolic analysis for parallelizing compilers. In *Conference proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3–7, 1995*, pages 414–423, 1995.

[TS99] Omri Traub and Michael D. Smith. Ephemeral instrumentation for lightweight program profiling. In *submitted to the 32nd Annual International Symposium on Microprogramming*, December 1999.

[Tu99] Peng Tu. Personal communication. 1999.

[Weg75a] B. Wegbreit. Property extraction in well-founded property sets. *IEEE Transactions on Software Engineering*, 1(3):270–285, September 1975.

[Weg75b] Mark Wegman. *THESIS*. PhD thesis, University of California, Berkeley, 1975.

[Wu96] Youfeng Wu. Conflict Ratio Profiling for Memory References. Technical Report MRL Compiler Technical Report 96012, Intel Corp., 1996.

[YGS95] Cliff Young, Nicolas Gloy, and Michael D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Intl. Symposium on Computer Architecture*, Italy, 1995.

[YP91] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, Albuquerque, New Mexico, November 18–20, 1991. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[YS94] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–241, San Jose, California, October 4–7, 1994. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.