12

Path Splitting: a Technique for Improving Data Flow Analysis

by

Massimiliano Antonio Poletto

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science

and

Master of Engineering

in Computer Science and Engineering

at the

Massachusetts Institute of Technology

May 1995

© Massimiliano Antonio Poletto, 1995. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 12, 1995

Certified by .... [illegible]
M. Frans Kaashoek
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .... [illegible]
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

AUG 10 1995

LIBRARIES

Barker Eng

Path Splitting: a Technique for Improving Data Flow Analysis

by

Massimiliano Antonio Poletto

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1995, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science
and
Master of Engineering
in Computer Science and Engineering

Abstract

Path splitting is a new technique for improving the amount of data flow information statically available to the compiler about a fragment of code. Path splitting replicates code in order to provide optimal reaching definitions information within given regions of the control flow graph. This improved information is used to extend the applicability of various classical code optimizations, including copy and constant propagation, common subexpression elimination, dead code elimination, and code hoisting. In addition, path splitting can contribute to decreasing register pressure, and creates long instruction sequences potentially useful for trace scheduling.

Path splitting was implemented in the SUIF compiler. Experimental results indicate that path splitting effectively restructures loops, modifying the control flow graph so as to improve data flow information and hence enable further "classical" optimizations. Path splitting can decrease the cycle count of loops by over a factor of two. Such transformations result in over 7% run time performance improvements for large C benchmarks. Although path splitting can cause exponential growth in code size, applying it only to regions where it is beneficial limits code growth to below 5% for realistic C programs. Cache simulations reveal that in most cases this code growth does not harm program performance very much.

Thesis Supervisor: M. Frans Kaashoek
Title: Assistant Professor of Computer Science and Engineering

Path Splitting: a Technique for Improving Data Flow Analysis

by

Massimiliano Antonio Poletto

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1995, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science
and
Master of Engineering
in Computer Science and Engineering

Abstract

Path splitting is a new technique for improving the amount of data flow information statically available to the compiler about a fragment of code. Path splitting replicates code in order to provide optimal reaching definitions information within given regions of the control flow graph. This improved information is used to extend the applicability of various classical code optimizations, including copy and constant propagation, common subexpression elimination, dead code elimination, and code hoisting. In addition, path splitting can contribute to decreasing register pressure, and creates long instruction sequences potentially useful for trace scheduling.

Path splitting was implemented in the SUIF compiler. Experimental results indicate that path splitting effectively restructures loops, modifying the control flow graph so as to improve data flow information and hence enable further "classical" optimizations. Path splitting can decrease the cycle count of loops by over a factor of two. Such transformations result in over 7% run time performance improvements for large C benchmarks. Although path splitting can cause exponential growth in code size, applying it only to regions where it is beneficial limits code growth to below 5% for realistic C programs. Cache simulations reveal that in most cases this code growth does not harm program performance very much.

Thesis Supervisor: M. Frans Kaashoek
Title: Assistant Professor of Computer Science and Engineering

Acknowledgments

This thesis would likely not exist without Frans Kaashoek, who provided help and enthusiasm, and argued in its favor when I was on the verge of scrapping the entire project. Many thanks to Dawson Engler, who had the original idea of replicating code to improve optimizations, and to Wilson Hsieh, who also provided feedback and ideas. Thanks also to Dr. Tom Knight, who willingly listened to my ideas on various occasions, and to André DeHon and Jeremy Brown, with whom I spent a year hacking and flaming about specialization. I am grateful to the SUIF group at Stanford for providing a good tool and supporting it well; special thanks to Chris Wilson, Todd Mowry, and Mike Smith.

I am indebted to everyone in the PDOS group at LCS for providing an enjoyable and supportive working environment: Dawson Engler, Sanjay Ghemawat, Sandeep Gupta, Wilson Hsieh, Kirk Johnson, Anthony Joseph, Professors Corbató and Kaashoek, Kevin Lew, Neena Lyall, Dom Sartorio, Josh Tauber, and Debby Wallach. Thanks to Dawson, Sandeep, Anthony, Josh, and Debby for providing helpful feedback on many sections of this document.

Sincere thanks to all those who, each in their own way, have made my undergraduate years at MIT interesting and wonderful. I name only a few: Arvind Malhan, my roommate for a long time, who patiently and unswervingly bore my music, sarcasm, and compulsive tidiness; Charlie Kemp, who provided friendship and inspiration whenever it was needed; Neeraj "Goop" Gupta, with whom I did "the drugs" so many times on Athena, and who cheerfully proofread a full draft of this thesis; Megan Jasek and Kimball Thurston, for much good company; people at Nu Delta, and especially "Fico" Bernal, Hans Liemke, Eric Martin, and Paul Hsiao, whose heroic if ultimately unsuccessful efforts to make me into a party animal did not go unnoticed; Lynn Yang, who initiated many lunch breaks and put me in charge of her incontinent pet turtle for a summer; Julia Ogrydziak, whose performances I hear much less often than I would like; Eddie Kohler, with whom I look forward to a few more years of this place; Wilson Hsieh, who was a good 6.035 TA and a great friend and officemate for the last year, even though he did occasionally leave toothmarks on my forearms.

Un grazie particolare ai miei genitori, i quali mi hanno sempre dato libertà di decisione e l'opportunità di perseguire i miei interessi e desideri. Non avrei potuto chiedere loro altro, e non posso ringraziarli abbastanza.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Path splitting is a new compiler technique for transforming the structure of a program's control flow graph in order to increase the amount of data flow information available to subsequent passes. The goal of this work is to increase the performance of compiled code by extending the range of applicability of optimizations that depend on data flow information, such as copy propagation, common subexpression elimination, and dead code elimination [1, 5]. In short, path splitting increases the accuracy of reaching definitions information by replicating code to remove joins.

Path splitting was implemented in the SUIF compiler. Experimental results indicate that this technique effectively restructures loops, modifying the control flow graph so as to improve reaching definitions information and hence enable further "classical" optimizations. Path splitting can decrease the cycle count of loops by over a factor of two. Measurements show that such transformations can result in over 7% run time performance improvements for large C benchmarks. Although path splitting can cause exponential growth in code size, applying it only to regions where it is beneficial limits code growth to below 5% for realistic C programs. Cache simulations reveal that in most cases this code growth does not harm program performance very much.

This thesis describes the path splitting algorithm, places it in the context of data flow analysis techniques, and evaluates its performance on several benchmarks. Furthermore, it outlines the implementation of path splitting in the SUIF compiler toolkit, and discusses how path splitting may be useful for extracting additional instruction-level parallelism on super-scalar machines. Lastly, based on the experience accumulated during the implementation of path splitting, the thesis makes some general comments on compiler design.

The following section provides some background necessary for understanding subsequent chapters. Section 1.2 presents motivations for path splitting. Section 1.3 outlines the rest of the thesis.

1.1 Background

The purpose of code optimizations in a compiler is to improve the performance of emitted code to a level ever nearer the best achievable, while respecting reasonable constraints on compilation time and code size. As mentioned in [1], "optimization" in this context is a misnomer: it is rarely possible to guarantee that the code produced by a compiler is the best possible. Nevertheless, transformations performed by the compiler to improve naively written or generated code can have a large impact on performance.

Code-improving transformations fall under two general categories: machine-dependent optimizations, such as register allocation and "peephole" optimizations [33, 13], and machine-independent transformations. The latter use information available statically at compile time to restructure the code in ways intended to improve performance independently of the specific characteristics of the target machine. These techniques include constant folding, copy propagation, common subexpression elimination, dead code elimination, and various loop transformations, such as unrolling, induction variable elimination, code hoisting, and unswitching [1, 5]. All such machine independent optimizations rely on some form of data flow analysis, the process of collecting information about the way in which variables are used at various points throughout a program.

Path splitting enables code improvement techniques that otherwise could not be performed due to the structure of the control flow graph. Before providing motivating examples for path splitting, we first define a few terms.

A *control flow graph* is a program representation used to specify and reason about the structure of a piece of code. It can be defined as a triple $G = (N, E, n_0)$, where N is a finite set of nodes corresponding to program statements, E (the *edges*) is a subset of $N \times N$, and $n_0 \in N$ is the initial node. Edge (x, y) leaves node x and enters node y, indicating that y may be executed immediately after x during some execution of the program. There is a path from n_0 to every node (for the purposes of the control flow graph, we can ignore code which is unreachable and hence never executed).

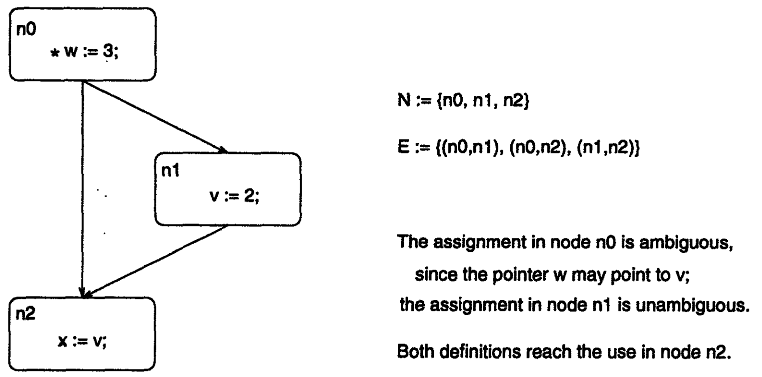

n0
*w := 3;
n1
v := 2;
n2
x := v;
N := {n0, n1, n2}
E := {(n0,n1), (n0,n2), (n1,n2)}
The assignment in node n0 is ambiguous,
since the pointer w may point to v;
the assignment in node n1 is unambiguous.
Both definitions reach the use in node n2.

Figure 1.1. An example of a control flow graph.

Figure 1.1 gives an example of a control flow graph. Node n_0 ends in a *fork*: more than one edge (namely edges (n_0, n_1) and (n_0, n_2)) leaves it. Node n_2, on the other hand, begins with a *join*: more than one edge enters it. A *basic block* is any sequence of instructions without forks or joins. Throughout this thesis, every node in the graphical representation of a flow graph is intended to represent a basic block. For simplicity, in each node we explicitly portray only the least number of instructions necessary to convey a point. However, of course, a basic block may consist of arbitrarily many instructions.

A cycle in a flow graph is called a *loop*. If one node n_1 in a loop *dominates* all the others, meaning that every path from n_0 to any node in the loop contains n_1, then the loop is *reducible*, and n_1 is called the *head* of the loop. Any node n_2 in the loop, such that $(n_2, n_1) \in E$, is referred to as a *tail* of the loop, and the edge (n_2, n_1) is a *back edge* of the loop. Figures 1.2 (a) and (b) depict a reducible and irreducible flow graph, respectively. For example, in Figure 1.2(a), node a_0 dominates all nodes in the cycle, so the cycle is a reducible loop with back edge (a_2, a_0). On the other hand, in Figure 1.2(b) there is a cycle containing nodes b_1 and b_2, but neither dominates the other, since either can be reached from b_0 without passing through the other. The loop is therefore irreducible. Henceforth, we only consider reducible loops.

A *definition* of a variable v is a statement (corresponding to some $n \in N$) that either assigns or may assign a value to v. A definition that definitely assigns a value to v, such as an assignment, is an *unambiguous* definition. *Ambiguous* definitions are those in which

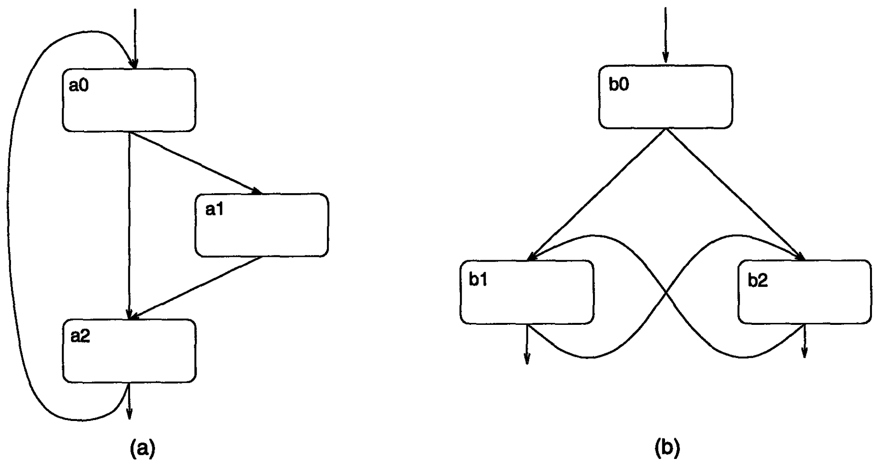


Figure 1.2. A reducible loop and an irreducible loop.

assignment may, but cannot be proven to, occur. Ambiguous definitions include (1) assignments to pointers that may refer to v , and (2) procedure calls in which v is either passed by reference or in the scope of the callee, or aliased to another variable that is itself either passed by reference or in the scope of the callee. A *use* of v is any statement that references v (i.e., explicitly uses its value).

Every unambiguous definition of v is said to *kill* any definition prior to it in the control flow graph. A definition **d** is *live* if the value it defines is used on some path following **d** before being redefined (killed). If a statement is not live, it is *dead.* A definition **d** *reaches* a use **u** of v if there is a path in the control flow graph of the procedure from the point immediately following **d** to **u** on which v is not killed. The classical reaching definitions problem consists of determining, for all variables, which uses of each variable are reached by which definition (and conversely, which definitions of each variable reach which use). A detailed description of the problem and its solution is given in [1].

An expression **e** is *available* at a point **u** if it is evaluated on every path from the source of the control flow graph to **u**, and if, on each path, after the last evaluation prior to **u**, there are no assignments (either ambiguous or unambiguous) to any of the operands in **e**.

1.2 Motivation

Path splitting is a technique intended to be composed with other optimization algorithms so as to improve their applicability. It increases the accuracy of reaching definitions information within selected regions of the control flow graph by replicating code to remove joins.

This section motivates the use of path splitting; Chapter 2 will describe the path splitting algorithm in detail.

Copy propagation is one example of an optimization technique that can be improved by path splitting. Copy propagation aims to eliminate statements **s** of the form x := y, by substituting y for x in each use of x reached by **s**. A copy may be "propagated" in this fashion to each use **u** of x if the following two conditions hold:

1. **s** is the only definition of x reaching **u**.

2. There are no assignments to y on any path from **s** to **u**, including cyclic paths that contain **u**.

Copy propagation reduces to constant propagation in the case when y is a constant, trivially satisfying condition (2) above.

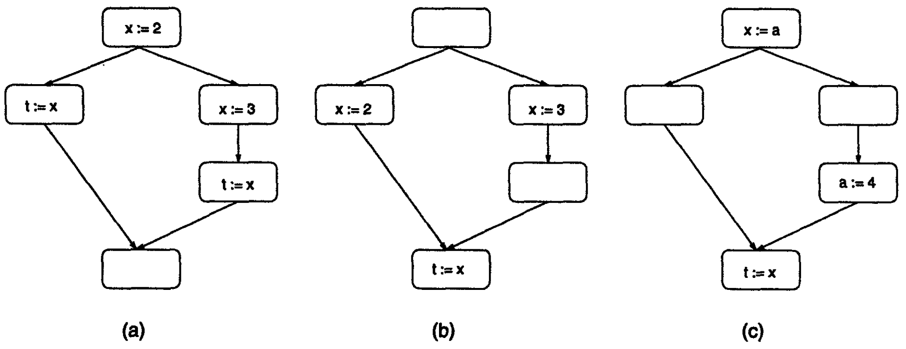


Figure 1.3. Control flow graphs that benefit from copy propagation and path splitting.

Figure 1.3 illustrates three flow graphs that could benefit from copy propagation and path splitting. In Figure 1.3(a), all definitions of x can be propagated to the appropriate use without path splitting. In (b), however, too many definitions reach the use of x, whereas in (c) copy propagation cannot occur because a is redefined along the path on the right, violating condition (2). As a result, copy propagation normally could not occur in cases (b) and (c).

Path splitting is intended to solve the problems encountered in cases (b) and (c), by increasing the amount of data flow information available to the compiler. Figure 1.4 illustrates its effects. Dashed ovals indicate instructions that can be removed after some optimizations. In part (a), no path splitting needs to be done. The constant values are propagated, and the assignments to x, now dead, can be removed. In (b), path splitting replicates instructions so

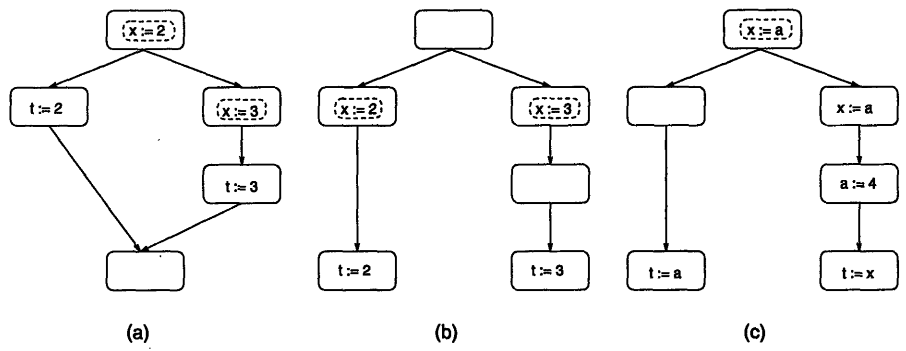


Figure 1.4. Control flow graphs after path splitting and copy propagation.

that only one definition reaches each use of x. This enables constant propagation (and dead code elimination, as in (a)) to both uses. In (c), by path splitting we can copy propagate the assignment to a along one of the two paths. Thus x := a becomes dead along that path, so partial dead code elimination [28] might be used to place it only onto the path on which it is used.

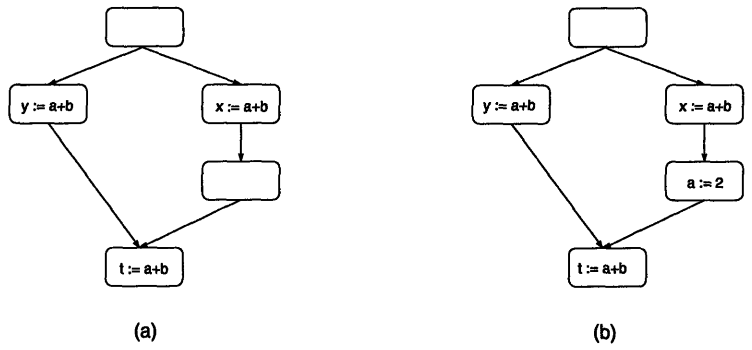


Figure 1.5. Control flow graphs that benefit from CSE and path splitting.

Common subexpression elimination (CSE) is another important code improvement technique that can be enabled by path splitting. It relies on available expressions. If an expression **e** computed in a statement **u** is available at that point in the flow graph, its value may be assigned to a temporary variable along each of the paths coming in to **u**, and this variable may be referenced rather than re-evaluating the entire expression. Figures 1.5(a) and 1.6(a) illustrate common subexpression elimination. Unfortunately, as shown in Figure 1.5(b), if even one assignment to any operand of **e** appears on any path coming into **u** after the last evaluation of **e**, then common subexpression elimination cannot be performed.

It is certainly possible for the operands of **e** to each have multiple reaching definitions at **u** and for **e** to be available: this happens, for example, when occurrences of **e** on different paths leading to **u** are preceded by different definitions of operands in **e** on two or more of these paths. However, if each operand in **e** has exactly one reaching definition at **u**, and **e** is computed on each path leading to **u**, and **e** is available when considering any one path entering **u** alone, then **e** is available at **u**, and common subexpression elimination may be performed.

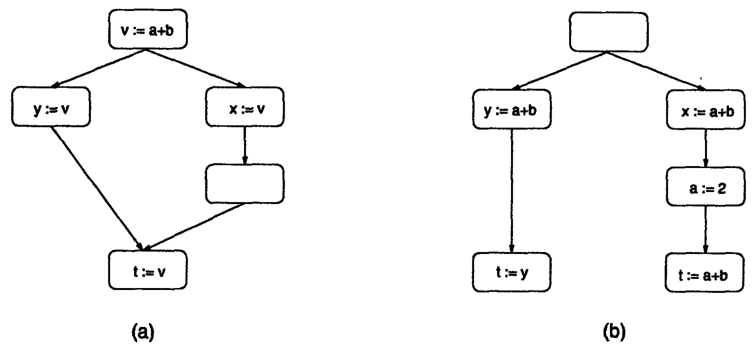


Figure 1.6. Control flow graphs after path splitting and CSE.

As a result, by performing path splitting based on the definitions reaching `a` and `b` in `t := a+b`, we enable the maximum possible amount of common subexpression elimination within this region of code. In other words, CSE will become possible on every path to the use **u** of an expression **e** along which **e** is evaluated, independently of assignments to operands of **e** along any other paths reaching **u**. Path splitting and the common subexpression elimination which it enables for the flow graph of Figure 1.5(b) are illustrated in Figure 1.6(b).

It can be useful to view path splitting in terms of static single assignment (SSA) form [12]. In this program representation, each variable is assigned only once in the program text, with the advantage that only one use-definition chain is needed for each variable use. In SSA form, whenever two paths in the control flow graph having different data flow information meet, a ϕ-function is used to model the resulting loss of data flow information. A ϕ-function is a function whose value is equal to one of its inputs. Consider Figure 1.7: we must assign $\phi(x_0,x_1)$ to x_2 because we cannot know which value of x will reach that point in the flow graph at run time. In this context, path splitting is an attempt to minimize the number of ϕ-functions required in the SSA representation of a procedure. In other words, it increases the accuracy of reaching definitions information within the flow graph.

```
if (a=2)                    if (a=2)
   x := 3;                     x0 := 3;
else                        else
   x := 4;                     x1 := 4;
...                         x2 := φ(x0,x1);
j := x;                     ...
                            j := x2;
```

All assignments to x are converted to assignments to distinct "versions" of x, such that no variable is assigned more than once.

Figure 1.7. Inserting a ϕ-function at a join in SSA form.

In summary, it is possible to improve the effectiveness of copy propagation, common subexpression elimination, dead code elimination, and other code optimization algorithms by restructuring a program's control flow graph. Path splitting is a technique for identifying where data flow information could be improved, and then replicating code to remove problematic joins and restructure the code appropriately.

1.3 Overview of Thesis

This thesis is organized as follows: Chapter 2 discusses the path splitting algorithm and how the compiler decides to apply it. Chapter 3 analyzes its effects in terms of a lattice-theoretical data flow framework, showing that path splitting results in optimal downward data flow information in selected regions of code. Chapter 4 describes the implementation of path splitting, including SUIF, the Stanford University compiler toolkit within which it was developed. Chapter 5 presents concrete examples of code improved by path splitting, and discusses results obtained by compiling several benchmarks using path splitting. Chapter 6 outlines alternative algorithms for path splitting and techniques for limiting code growth, and relates path splitting to trace scheduling and compilation on super-scalar machines. It also makes some general comments on compiler implementation and design. Chapter 7 discusses related work. Lastly, Chapter 8 concludes.

Chapter 2

The Path Splitting Algorithm

This chapter presents the details of the path splitting algorithm. Roughly speaking, path splitting can be divided into three phases:

1. Removal of joins within a loop to create a tree, where each leaf is a back edge of the loop;
2. "Path specialization," a process by which the path from the head of the loop to each of the back edges is replicated and appended to itself, "in place" of the back edge;
3. Removal of unnecessary code.

This process is outlined in Algorithm 2.1. Since loops are the regions of a program executed most often, they are the places where it is usually most profitable to invest compilation time and easiest to justify code size growth. Consequently, path splitting is performed only within loops.

The rest of this chapter is divided into three sections, corresponding to the three phases described above. Throughout the text we refer to a simple example in which path splitting provides benefits. The original flow graph, before path splitting or any other optimizations, appears in Figure 2.1.

2.1 Removing Joins

In order to minimize code size growth, code is replicated to remove joins only when doing so will improve reaching definitions information at some point in the code. The compiler

```
SPLIT-PATHS(p:procedure)
    find-reaching-definitions(p)
    foreach l:loop in reverse-post-order(loops(p)) do
        foreach s:statement in reverse-bfs-order(statements(l)) do
            o:operand := find-specializable-operand(s)
            if (o $\neq \emptyset$) then
                specialize-operand(o, s, l)
                find-reaching-definitions(p)
        specialize-loop(l)
        remove-unnecessary-code(l)
```

Code

- **find-reaching-definitions**(p) solves the reaching definitions problem for procedure p.
- **find-specializable-operand**(s) finds an operand of s, if any, which is reached by more than one *useful* definition at s. A *useful* definition is one that could be profitable in later analysis stages, such as an assignment from a scalar variable or a constant. If no such operand exists, it returns a null value.
- **loops**(p) returns a tree, the nodes of which are the reducible loops in procedure p. This tree is structured as follows:
  - If a loop A is lexically within a loop B, then the tree node corresponding to B is an ancestor of the node corresponding to A.
  - If loop A textually follows loop B at the same scoping level in the code, then the node corresponding to B is a left sibling of A in the tree.
- **remove-unnecessary-code**(l) performs on l the task described in Section 2.3.
- **reverse-bfs-order**(g) performs a breadth-first search over a flow graph g', which is derived from g by reversing all edges in g (i.e., for every directed edge (x, y) in the set of edges of g, the set of edges of g' contains a directed edge (y, x)). The search starts from the node corresponding to the "sink" of g (the point where control exits g). If g has more than one sink (a loop may have more than one exit point), then breadth-first searches are performed from one or more of the nodes of g' corresponding to these sinks, in an arbitrary order, until all nodes ("statements") in the loop body have been visited.
- **reverse-post-order**(t) returns each of the loops in the tree of loops t by traversing t in reverse post-order (right child, left child, parent).
- **specialize-loop**(l) is described in Algorithm 2.4.
- **specialize-operand**(o, s, l) is described in Algorithm 2.2.
- **statements**(l) is the flow graph of all statements in the body of loop l (this does not include loop back edges or jumps out of the loop).

Legend

Algorithm 2.1. Path Splitting: Top Level Algorithm.

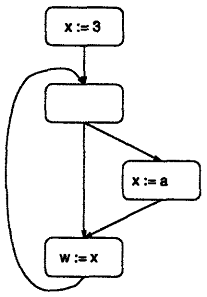


Figure 2.1. Example flow graph, before any changes.

therefore first performs reaching definitions analysis, and then runs FIND-SPECIALIZABLE-OPERAND (see description in Algorithm 2.1) to identify statements that use one or more variables reached by one or more *useful* definitions. In this chapter, an operand refers to any value or location used or defined by a statement. A *useful* definition is one that could be profitable in later analysis stages, such as an assignment from a scalar variable or a constant. Once a statement that could benefit from path splitting is found, the compiler attempts to improve data flow information by executing SPECIALIZE-OPERAND (Algorithm 2.2).

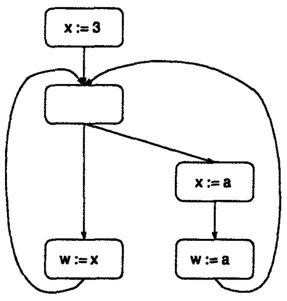


Figure 2.2. Example flow graph, after removing joins.

This process of restructuring the flow graph of the loop body is detailed in algorithms 2.2 (SPECIALIZE-OPERAND) and 2.3 (SPECIALIZE-REPLICATE-CODE). Its effect on our example flow graph (see Figure 2.1) is illustrated in Figure 2.2. At a high level, path splitting eliminates joins that cause any statement following them in the loop body to have ambiguous reaching definitions information caused by definitions within the loop. If the reaching definitions information for a variable is already ambiguous on loop entry, path splitting cannot generally improve things, and so will not be applied. If, however, as in the example, it *can* be applied, then it results in a flow graph in which each node

```
SPECIALIZE-OPERAND(o:operand, s:statement, l:loop)
    foreach st:statement in statements(l) do
        color[st] := white
    color[s] := gray
    Enqueue(Q:queue, s)
    while Q $\neq \emptyset$ do
        sc:statement := head[Q]
        foreach sp:statement in predecessors[sc] do
            if has-unique-reaching-definition(sp, o) then
                specialize-replicate-code(o, sp, sc, l)
            else if color[sp] = white then
                color[sp] := gray
                Enqueue(Q, sp)
        Dequeue(Q)
        color[sc] := black
```

Code

- This algorithm is essentially a slightly modified version of breadth-first search [11].
- **has-unique-reaching-definition** is true if only one definition of *o* reaches its use at *sp*, and if this definition is an assignment from a scalar variable or a constant.

Legend

Algorithm 2.2. Path Splitting: function SPECIALIZE-OPERAND.

in the loop body is reached by no more than one definition of each of its operands in the loop body. As a result, the loop body is mostly a tree, in which joins exist only if they do not corrupt downward data flow information, and otherwise each basic block only has one direct predecessor. In our example, for instance, the original loop has been "split" into two loops (or, more accurately, into a sequence of code with two back edges), neither of which contains a join.

2.2 Creating Specialized Loops

Converting the flow graph within a loop to tree form may often be insufficient to improve data flow information in the loop. Information that flows down through the control flow graph will reenter the loop through the back edges, so that at the head of the loop there is no more data flow information than there was before joins were removed. This is a problem,

```
SPECIALIZE-REPLICATE-CODE(o:operand, sp, sc:statement, l:loop)
    if is-unconditional-jump(sp) then
        seq:sequence := copy(code-sequence(sc, l))
        mark-end-of-path(seq)
        insert-sequence(seq, sp)
        delete-statement(sp)
    else if is-conditional-branch(sp) then
        l:statement := new-label()
        insert-after(l, sp)
        reverse-and-redirect(sp, l)
        seq:sequence := copy(code-sequence(sc, l))
        mark-end-of-path(seq)
        insert-sequence(seq, sp)
        delete-statement(sp)
    else if not is-indirect-jump(sp) then
        seq:sequence := copy(code-sequence(sc, l))
        mark-end-of-path(seq)
        insert-sequence(seq, sp)
```

Code

- **code-sequence**(s, l) returns the straight-line code sequence from s to the end of l, inclusive.
- **copy** copies a code sequence. All labels are renamed uniquely. The destination of a jump is renamed appropriately if the corresponding label is within the copied sequence.
- **is-unconditional-jump** returns true if the instruction is an unconditional jump.
- **is-conditional-branch** returns true if the instruction is a conditional branch.
- **insert-after**(s_1, s_2) inserts s_1 after s_2.
- **insert-sequence**(seq, q) inserts sequence s after statement q.
- **mark-end-of-path***(seq)* marks the last statement of *seq* as an "end of path" node, required to find points of specialization during the second phase of path splitting.
- **new-label** returns a new, unique label.
- **reverse-and-redirect***(j, l)* reverses the test condition of jump j and redirects it to label l.

Legend

Algorithm 2.3. Path Splitting: function SPECIALIZE-REPLICATE-CODE.

since it implies that maximally precise downward data flow information is only available at the bottom of the loop. For example, it can now become difficult to perform dead code elimination after copy propagation. In Figure 2.2, for instance, after removing joins we are able to propagate the copy `x := a`, converting the assignment `w := x` to `w := a`, but cannot remove the assignment `x := a`.

To counter this difficulty, path splitting unrolls once each of the join-free loops created in the previous step. Algorithm 2.4 describes this process in some detail. In more abstract terms, we find the paths from the end of each loop (i.e., from the nodes ending in loop back-edges, namely those containing `w := x` and `w := a` in Figure 2.2) to the head of the loop, and append a copy of this path to the end of each loop. This process transforms the flow graph in Figure 2.2 into that shown in Figure 2.3.

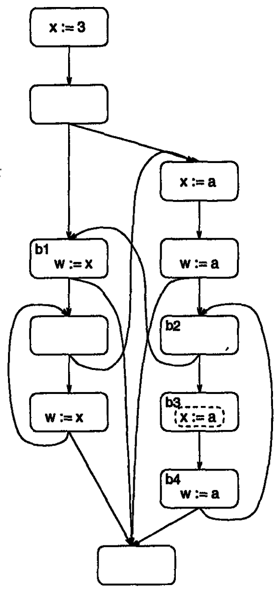


Figure 2.3. Example flow graph, after creating specialized loops.

At this point, the original loop containing joins has been transformed into a tree-like structure, in which each of the leaves (1) contains precise reaching definitions information, and (2) is followed by a "specialized" loop that is a copy of the path between itself and the head of the loop. Each such loop contains outgoing but no incoming edges, so the accurate data flow information entering it is preserved throughout its body. As a result, it is possible to make strong statements about the values of variables inside some loops, and thus perform effective optimizations tailored to that particular set of data flow information — hence the term "specialized" loop.

In Figure 2.3, for example, the unrolled loop on the right hand side can only be entered by first passing through the section of the tree on which the assignment `x := a` occurs. Since `x` is never reassigned inside the loop, the assignment `x := a` inside the unrolled loop (in block b_3) can be removed, as indicated by the dashed box surrounding it in the figure.

The key to effectively creating specialized loops is that, as mentioned earlier, no edges *enter* any of these loops (except through the loop head, of course). Edges that leave these loops are inevitable, since conditional branches inside the loops cannot in general be removed, but they are redirected into the "upper" parts of the unrolled loop structure. As a result, the "tree" created by splitting joins can be re-traversed, making the appropriate data flow information precise and available before control enters the destination specialized loop.

2.3 Removing Unnecessary Code

Path splitting improves information regarding predecessors of a given block. All data flow information that flows downward, such as reaching definitions and available expressions, is made more precise. Data flow information that propagates upward in the control flow graph, such as liveness information, is not improved by the code splitting alone, however. In fact, liveness information propagating backward along the edges of the control flow graph can be detrimental to the elimination of code made unnecessary by path splitting.

Figure 2.3 contains a concrete example of this problem. In the previous section, we mentioned that the assignment `x := a` in block b_3 can be eliminated. This is true, but the removal will not happen if dead code elimination is performed in the standard fashion [1], because the use of `x` in block b_1, together with the existence of edges (b_4, b_2) and (b_2, b_1), keeps the assignment in b_3 live. However, there is no reason why this assignment should not be eliminated, since the only definition of its destination operand that reaches it is its own original, and the source operand is not reassigned within the body of the loop. The code is therefore unnecessary, although not technically dead. To avoid wasting these opportunities for "unnecessary" code elimination, we search the control flow graph, and immediately remove any copy that meets the following criteria:

```
SPECIALIZE-LOOP(l:loop)
    foreach s:statement in statements(l)do
        if is-end-of-path-mark(s) then
            seq:sequence := copy(reverse-BFS-path(s, head(l)))
            insert-sequence(seq, s)
            adjust-control-flow(seq, s)
```

Code

- **adjust-control-flow***(seq, s)* inserts the sequence *seq* after statement *s*.
  - If *s* (the end of the path of which *seq* is a copy) is an unconditional jump to the top of the current loop (*l*), it is removed, so control flow simply falls from the predecessor of *s* into *seq*. If *s* is a conditional branch, then its test condition is reversed, and it is redirected to a new label following *l*.
  - A new label (the same used in the case when *s* is a conditional branch) is appended to *l*, and an unconditional jump to it is placed directly after *s*. The different specialized paths are laid out sequentially in the text segment, so without the branch, flow of control would not exit the loop after falling out of *seq*, but would incorrectly fall into the textually successive specialized sequence.
- **is-end-of-path-mark***(s)* returns true if *s* is marked as an "end-of-path" statement (always guaranteed to be a loop back-edge), i.e., if we are at a leaf of the tree formed by the first stage of path splitting.
- **reverse-BFS-path***(x, y)* returns the sequence of statements on the shortest path from x to y, moving along reverse control flow arcs, from a node to its predecessor. The sequence is thus the list $(y, p_1, p_2, \ldots, x)$, where y is a predecessor of p_1, p_1 is a predecessor of p_2, and so forth.

Legend

Algorithm 2.4. Path Splitting: function SPECIALIZE-LOOP.

1. It is in a basic block that was created as a copy for the purposes of path splitting.
2. The only definition of the destination operand is that corresponding to the assignment in the original block.
3. The source operand is not redefined within the body of the loop.

Chapter 3

Lattice-theoretical Properties of Path Splitting

This chapter describes a framework for data flow analysis, and discusses the advantages in data flow information resulting from path splitting. It opens with some definitions necessary for the presentation, outlines monotone and distributive data flow frameworks and the limits imposed on data flow information by the structure of the control flow graph, and describes the effects of path splitting on available data flow information. The purpose of this chapter is to explain how path splitting can provide optimal downward data flow information over regions of interest.

3.1 Preliminary Definitions

We begin by presenting and explaining some standard definitions required to provide a base for the rest of the discussion.

Definition 1 *A* semi-lattice *is a set L with a binary meet operation $\wedge$ defined as follows, for $a, b, x_i \in L$:*

$$a \wedge a = a \tag{3.1}$$

$$a \wedge b = b \wedge a \tag{3.2}$$

$$a \wedge (b \wedge c) = (a \wedge b) \wedge c \tag{3.3}$$

$$a \geq b \text{ iff } a \wedge b = b \tag{3.4}$$

$$a > b \text{ iff } a \wedge b = b \text{ and } a \neq b \tag{3.5}$$

$$\bigwedge_{1 \leq i \leq n} x_i = x_1 \wedge x_2 \wedge \dots x_n \tag{3.6}$$

Definition 2 *A semi-lattice has a* zero element 0 *if, for all $x \in L$, $0 \wedge x = 0$. A semi-lattice has a* one element 1 *if, for all $x \in L$, $1 \wedge x = x$. We henceforth assume that every semi-lattice has a zero element.*

In other words, a semi-lattice is a partially ordered set of values. The meet operator is idempotent, commutative, and associative. Definitions 3.4 and 3.5 indicate that the meet operator takes the lower bound of the operands to which it is applied — it cannot be the case that $a \wedge b = a$ and $a > b$. Given this, the one element 1 is the greatest element in the set of values, and the zero element 0 is the least element.

Definition 3 *Given a semi-lattice L, a* chain *is a sequence $x_1 \dots x_n \in L$, such that $x_i > x_{i+1}$ for $1 \leq i < n$.*

Definition 4 *L is bounded if for each $x \in L$ there exists a constant b_x, such that each chain beginning with x has length at most b_x.*

The concept of a bounded semi-lattice is important in data flow analysis. Definition 4 above does not put a bound on the size of the lattice L — it may well contain an infinite number of elements. However, it implies that every *totally ordered* subset of L is of finite size. We will show later why this is relevant.

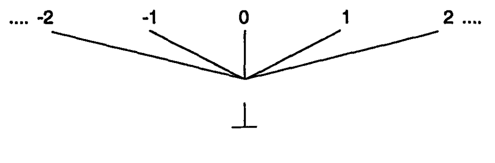


Figure 3.1. An example bounded semi-lattice.

This idea is illustrated in Figure 3.1. In this case, L consists of all the integers, plus the special symbol $\perp$, denoted "bottom." The meet operator $\wedge$ is defined so as to create the following partial order: for any integers i, j, $i \wedge j = \perp$ if $i \neq j$, and $i \wedge j = i = j$ if $i = j$. Also, for any integer i, $i \wedge \perp = \perp$. The lattice thus has an infinite number of elements, but the height of the longest chain is 2, because no ordering relationship is defined between any integers. $\perp$ is the 0 element. If the meet operator is applied to two identical integers, then the result is that same integer. If it is applied to two different integers, then the result is $\perp$. The order relationship implies that if the meet is applied to some number of integers, and any two differ, then the result is $\perp$.

3.2 Monotone and Distributive Frameworks

This section begins with five definitions necessary to describe a monotone data flow framework, and then goes on to map these definitions onto a concrete instance of a program representation.

Definition 5 *Given a bounded semi-lattice L, a set of functions F on L is a* monotone function space associated with L *if the following are true:*

- *Each $f \in F$ satisfies the monotonicity condition,*

$$\forall x, y \in L, \forall f \in F, f(x \wedge y) \leq f(x) \wedge f(y) \tag{3.7}$$

- *There exists an identity function i in F, such that*

$$\forall x \in L, i(x) = x \tag{3.8}$$

- *F is closed under composition,*

$$f, g \in F \Rightarrow fg \in F \tag{3.9}$$

$$\forall x \in L, fg(x) = f(g(x)) \tag{3.10}$$

- *For each $x \in L$, there exists $f \in F$ such that $f(x) = 0$.*

Definition 6 *A* monotone framework *is a triple $D = (L, \wedge, F)$, where*

- *L is a bounded semi-lattice with meet $\wedge$ as above.*
- *F is a monotone function space associated with L.*

Definition 7 *A* distributive framework *is a monotone framework $D = (L, \wedge, F)$ that satisfies the* distributivity *condition:*

$$\forall x, y \in L, \forall f \in F, f(x \wedge y) = f(x) \wedge f(y) \tag{3.11}$$

Definition 8 *An* instance of a monotone framework *is a pair $I = (G, M)$, where*

- $G = (N, E, n_0)$ *is a flow graph.*
- $M : N \rightarrow F$ *maps each node in N to a function in F.*

Definition 9 *Given an instance $I = (G, M)$ of $D = (L, \wedge, F)$, f_n denotes $M(n)$, the function in F associated with node n. In addition, let $P = n_1, n_2, \ldots, n_m$ be a path in G. Then $f_P(x) = f_{n_1} \circ f_{n_2} \circ \ldots \circ f_{n_{m-1}}$.*

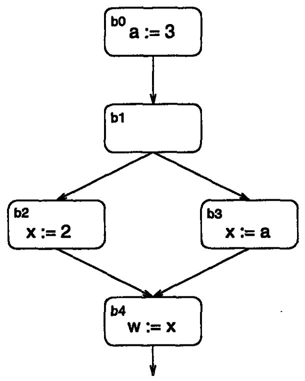


Figure 3.2. Another example control flow graph.

Having stated all these definitions, we now attempt to make them more concrete by placing them in the context of the example control flow graph in Figure 3.2 and the data flow values associated with it. In this case, the flow graph G in Definition 8 consists of nodes $N = \{b_0, b_1, b_2, b_3, b_4\}$, edges $E = \{(b_0, b_1), (b_1, b_2), (b_1, b_3), (b_2, b_4), (b_3, b_4)\}$, and initial node b_0.

Let us construct a data flow framework useful for copy propagation over this graph. Informally, L will be some set of sets of pairs (v, k), where v is any variable in the program (x, a, or w), and k is any variable or integer. Then define the meet operator $\wedge$ to be set intersection. Let F in Definition 5 consist of functions f_{b_k}, such that M in Definition 8 maps $b_k \mapsto f_{b_k}$, and each f_{b_k} models the assignments, if any, in the corresponding block b_k in the flow graph. For the sake of simplicity in this example, we let such a "block" contain exactly one instruction, and we ignore pointers. Thus, if a block b consists of an assignment to a variable v_1 from a variable v_2 or a constant c, then $f_b(x)$ (where $x \in L$) is a function that removes all ordered pairs (v_1, k) for any k from x, and then adds to x the ordered pair (v_1, v_2) or (v_1, c), respectively.

Having constructed this framework, we can now find which copies reach which nodes in the flow graph. For each node $n \in N$, we shall denote the copies entering n by $A[n]$. We must initialize each $A[n]$ to some value. Since no copies can enter the root node, and since we do not want to miss possible copy information anywhere else, we set $A[n_0] = 0$ (the 0 element, ie. the empty set of copies), and $A[n] = 1$ (the 1 element, ie. the universe of all possible pairs of copies)) for all other nodes n. Then, for each n, $A[n]$ is the meet of $f(A[p])$ over all predecessors p of n. We repeatedly apply this process until we reach a fixed point over all $A[n]$. This is guaranteed to occur eventually, since, by Definitions 3.4 and 3.5, $a \wedge b \geq a$ and $a \wedge b \geq b$ for all $a, b \in L$, and L is bounded. The fact that L is bounded, emphasized earlier, is necessary in order for this to hold true.

At the end of this iterative relaxation procedure (which in this case actually requires only one iteration, if we traverse the nodes in the direction of control flow, since there are no loops), in the case of our example we find that $A[b_0] = 0$ (it never had an opportunity to change, since b_0 has no predecessors), $A[b_1] = A[b_2] = A[b_3] = A[b_4] = \{(a, 3)\}$. Nothing about x can be said in b_4, since $\{(a, 3), (x, 2)\} \cap \{(a, 3), (x, a)\} = \{(a, 3)\}$.

3.3 Solving Data Flow Analysis Problems

This section adds some details to and formalizes the concepts introduced in the example above. It turns out that many other data flow analysis problems of interest, such as live variable analysis, can be expressed in terms of the monotone frameworks described in the previous section [25]. The relaxation method for solving such problems, presented by example above, is due to Kildall [27], and is outlined in Algorithm 3.1. Iterative algorithms (such as those used in path splitting) to solve data flow problems can all be reduced to this generic prototype. Depending on the specific problem, the nature of $\wedge$ and F, the types of values stored in elements of L (the $A[N]$s in Algorithm 3.1), and the direction of information flow (whether the meet is taken over all predecessors or all successors), need to be defined appropriately.

Executing this algorithm results in $A[n]$ for each n, which are the maximum fixed point (MFP) solution to the following set of simultaneous equations:

$$A[n_0] = 0 \tag{3.12}$$

```
foreach $n \in N$ do
        if $n = n_0$ then $A[n] = 0$ else $A[n] = 1$
while changes to any $A[n]$ occur do
        foreach $n \in N$ do
                $A[n] = \bigwedge_{p \in PRED(n)} f_p(A[p])$
```

where $PRED(n)$ refers to the predecessors of n, ie. all $m \in N$ such that $(m, n) \in E$.

Algorithm 3.1. Monotone Framework Iterative Analysis.

$$\forall n \in N - \{n_0\}, A[n] = \bigwedge_{p \in PRED(n)} f_p(A[p]) \tag{3.13}$$

The desired solution to a data flow problem is generally referred to as the *meet over all paths* (MOP) solution. If we define $PATH(n)$ to be the set of paths in G from n_0 to $n \in N$, then the MOP solution is $\bigwedge_{P \in PATH(n)} f_P(0)$ for each n. Theorem 1 formalizes the relationship between this MOP solution and the MFP solution obtainable by Kildall's iterative relaxation algorithm.

Theorem 1 *Given an instance $I = (G, M)$ of $D = (L, \wedge, F)$, executing Algorithm 3.1 we obtain, for each $n \in N$,*

$$A[n] \leq \bigwedge_{P \in PATH(n)} f_P(0) \tag{3.14}$$

Proof: *See [25].*

When a data flow problem can be expressed in terms of a distributive framework $D = (L, \wedge, F)$, then the MFP solution is always the MOP solution [27]. However, in the case of a framework which is monotone but *not* distributive, the maximum fixed point of the data flow equations is not necessarily the MOP solution — it is at most equal to the MOP solution [1].

The copy propagation framework described in the previous section, for instance, is not distributive. A small and classic example of a framework which is monotone but not distributive is constant propagation, as presented in [27]. Constant propagation can be formalized as a monotone framework CONST $= (L, \wedge, F)$, where $L \subset 2^{V \times R}$, $V = \{A_1, A_2, \ldots\}$ is an infinite set of variables, R is the set of all real numbers, and $\wedge$ is set intersection. This definition is simply a more formal way of describing a lattice similar in structure to the one

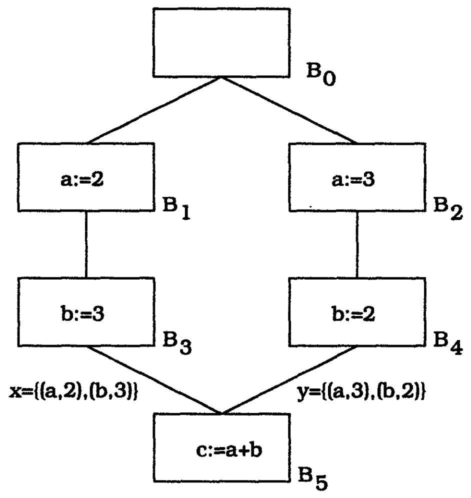


Figure 3.3. Example in which MFP < MOP because CONST is not distributive.

used as an example in Section 3.2. Intuitively, each $z \in L$ is the set of information about variables at a given point in the program. Thus, for $A \in V$ and $r \in R$, $(A, r) \in L$ indicates that the variable A has value r at some $n \in N$. The functions in F model the effect of assignments within each node in N on the sets of definitions in L.

A simple example taken from [1] illustrating CONST's lack of distributivity and the resulting disadvantages in terms of data flow analysis appears in Figure 3.3. In this case, $N = \{B_0, \dots, B_5\}$. Let $M(B_5) = f_{B_5} \in F$, and define $f_{B_5} = \langle C := A + B \rangle$. Then $x \wedge y = \emptyset$, so $f(x \wedge y) = \emptyset$, whereas $f(x) \wedge f(y) = \{(C, 5)\} \wedge \{(C, 5)\} = \{(C, 5)\}$. Essentially, in the case of the maximum fixed point solution data flow information flows down to B_5 as if there existed paths $B_0 \to B_2 \to B_3 \to B_5$ and $B_0 \to B_1 \to B_4 \to B_5$.

3.4 Improvements Resulting from Path Splitting

Path splitting transforms the control flow graph so as to increase the amount of information available during data flow analysis. It provides two main advantages, which will be discussed in more depth in the following sections. It improves the quality of the MOP solution at nodes of interest, and it removes limitations, such as that described in the last section in the case of CONST, associated with non-distributive monotone frameworks.

3.4.1 Improving the MOP Solution

Path splitting creates specialized sequences of nodes in the control flow graph corresponding to every path that may be taken through a loop. This section shows that the downward data flow information available in each of these specialized loops is at least as good as before path splitting, and in fact that it is optimal, in the sense that it cannot be improved further by rearrangements of the control flow graph.

We assert, omitting the proof of correctness of the algorithm, that each of the specialized loops created by algorithm SPECIALIZE-LOOP contains no joins, except when such joins do not corrupt downward reaching definitions information. Recall also that control may enter the sequence only through the header node, which is immediately preceded by one of the leaves of the tree created by function SPECIALIZE-REPLICATE-CODE.

Pick any one of these specialized loops, let N' be the set of all its component nodes, and let $n'_0 \in N'$ be its head. The fact that there are no joins inside this loop that corrupt data flow information means that for every n' in that specialized loop, there are no two paths P_i and P_j from n'_0 to n' such that one has different data flow information than another. In other words, if there in fact is more than one distinct downward path from n'_0 to any n', then for every pair of such paths, P_i and P_j, $f_{P_i}(A[n'_0]) \wedge f_{P_j}(A[n'_0]) = f_{P_i}(A[n'_0]) = f_{P_j}(A[n'_0])$. So we can conclude that after path splitting, within each specialized loop,

$$(\forall n' \in N')[A[n'] = f_P(A[n'_0])] \tag{3.15}$$

where P can be picked from any one of the paths from n'_0 to n'.

From the definitions of a lattice in Section 3.1, we know that, given a lattice L and $a, b \in L$,

$$a \geq a \wedge \left(\bigwedge_{1 \leq i \leq j} b_i \right) \tag{3.16}$$

As a result, if we let PATH be the set of all paths from the original loop head (n_0) to any other node n in the original loop body, and PATH contains at least one pair of paths P_i and P_j such that $f_{P_i}(A[n_0]) \wedge f_{P_j}(A[n_0]) < f_{P_i}(A[n_0])$ or $f_{P_i}(A[n_0]) \wedge f_{P_j}(A[n_0]) < f_{P_j}(A[n_0])$ (i.e., the flow graph contains joins harmful to downward data flow information), then

$$f_P(A[n'_0]) \geq \bigwedge_{P \in PATH(n_0,n)} f_P(A[n_0]) \tag{3.17}$$

meaning that downward data flow information in each specialized loop after path splitting is at least as good as before path splitting. Moreover, since we are not taking a meet over more than one path, this data flow information cannot be improved.

3.4.2 Tackling Non-distributivity

Over regions on which it is performed, path splitting also solves problems, such as those illustrated in the CONST example, due to joins in non-distributive data flow frameworks.

From Definition 7, a monotone framework $D = (L, \wedge, F)$ is not distributive, if there exist $x, y \in L$ and $f \in F$, such that $f(x \wedge y) < f(x) \wedge f(y)$. After path splitting, as claimed earlier, within each specialized loop there are no joins that corrupt data flow. Since there are no joins within these regions of code, the meet operator never needs to be applied, so the issue of distributivity inside these pieces of code disappears.

Chapter 4

Implementation

This chapter describes the implementation of path splitting used for this thesis. Section 4.1 gives an overview of the SUIF compiler toolkit from Stanford University, which was used as a base for this work. Section 4.2 describes parts of the actual implementation of path splitting.

4.1 The SUIF System

This thesis was implemented using SUIF, the Stanford University Intermediate Format [3, 4, 21], version 1.0.1. The SUIF compiler consists of a set of programs which implement different compiler passes, built on top of a library written in C++ that provides an object-oriented implementation of the intermediate format.

This format is essentially an abstract syntax tree annotated with a hierarchy of symbol tables. As illustrated in Figure 4.1, the abstract syntax tree can expose code structure at different levels of detail. At one level, referred to as "high-SUIF," it consists of language-independent structures such as "loop," "block," and "if." This level is well-suited for passes which need to look at the high-level structure of the code.

The leaves of this abstract syntax tree comprise "low-SUIF," and consist of nodes which represent individual instructions in "quad" format. This form works well for many scalar optimizations and for code generation. SUIF supports both expression trees, in which the instructions for an expression are grouped together, and flat lists of instructions, in which instructions are totally ordered, losing the structure of an expression but facilitating low-level activities such as instruction scheduling. Every SUIF node can contain "annotations," which are essentially pointers to arbitrary data.

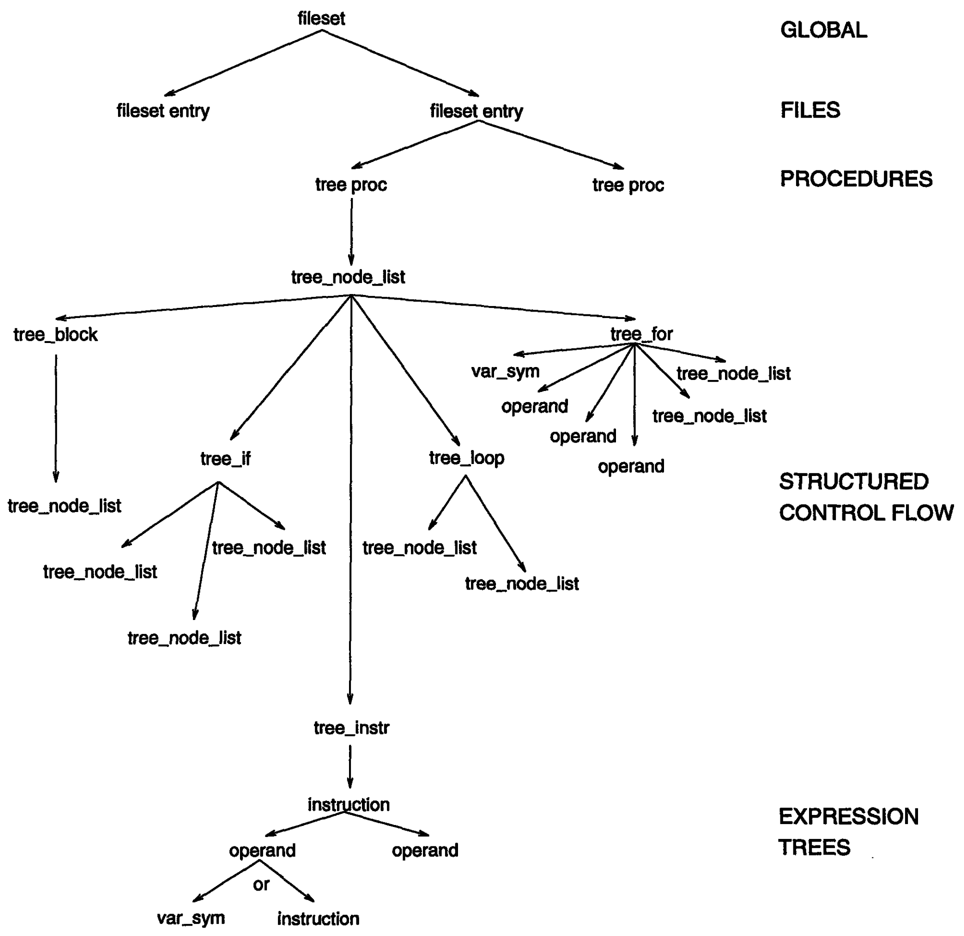


Figure 4.1. The SUIF abstract syntax tree structure (from the SUIF on-line documentation, `http://suif.stanford.edu`).

Each program (or pass) in the SUIF system performs a single analysis or transformation and writes the results out to a file. All the files share the same SUIF format, so passes can be reordered simply by running the programs in a different order, and new passes can be freely inserted at any point in the compilation. The existence of a consistent underlying format and large libraries which use it encourages code reuse, and makes experimentation with different passes or combinations of passes easy. A sample usage of the SUIF toolkit, appears in Figure 4.2. The top half of the figure consists of passes which parse the C or FORTRAN source into SUIF and then do various high-level parallelizing optimizations. The bottom part of the figure outlines the available back-end options. Those of primary

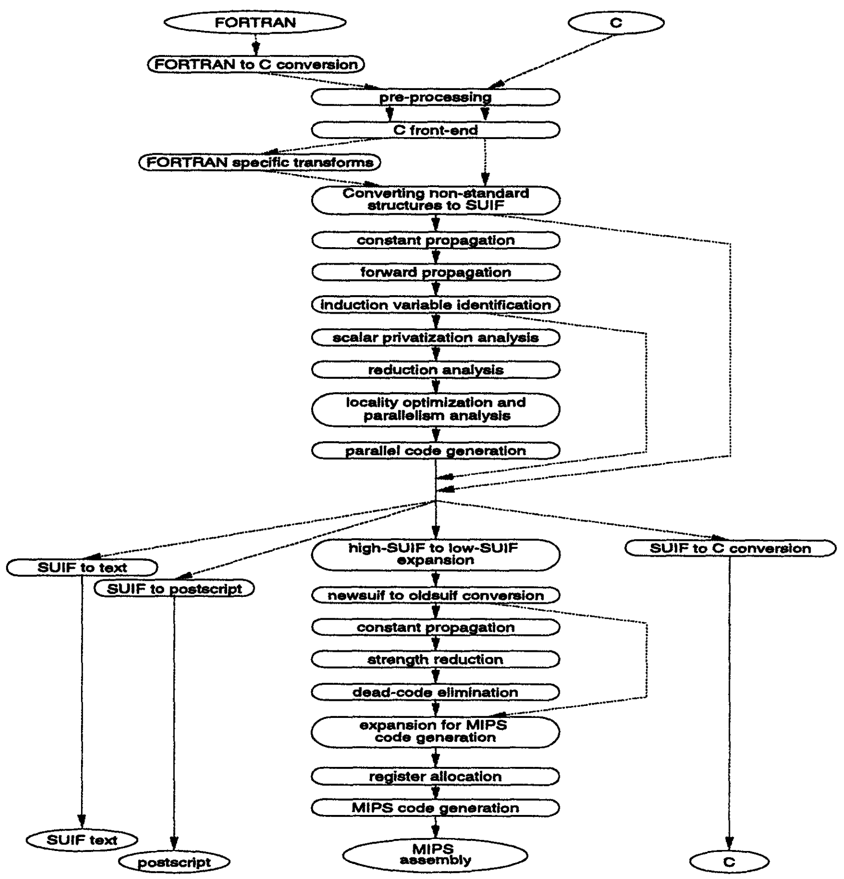


Figure 4.2. A sample ordering of SUIF compiler passes (from the SUIF on-line documentation, `http://suif.stanford.edu`).

interest are an optimizing back-end targeted to the MIPS architecture, and the SUIF-to-C converter, which provides portability to other architectures.

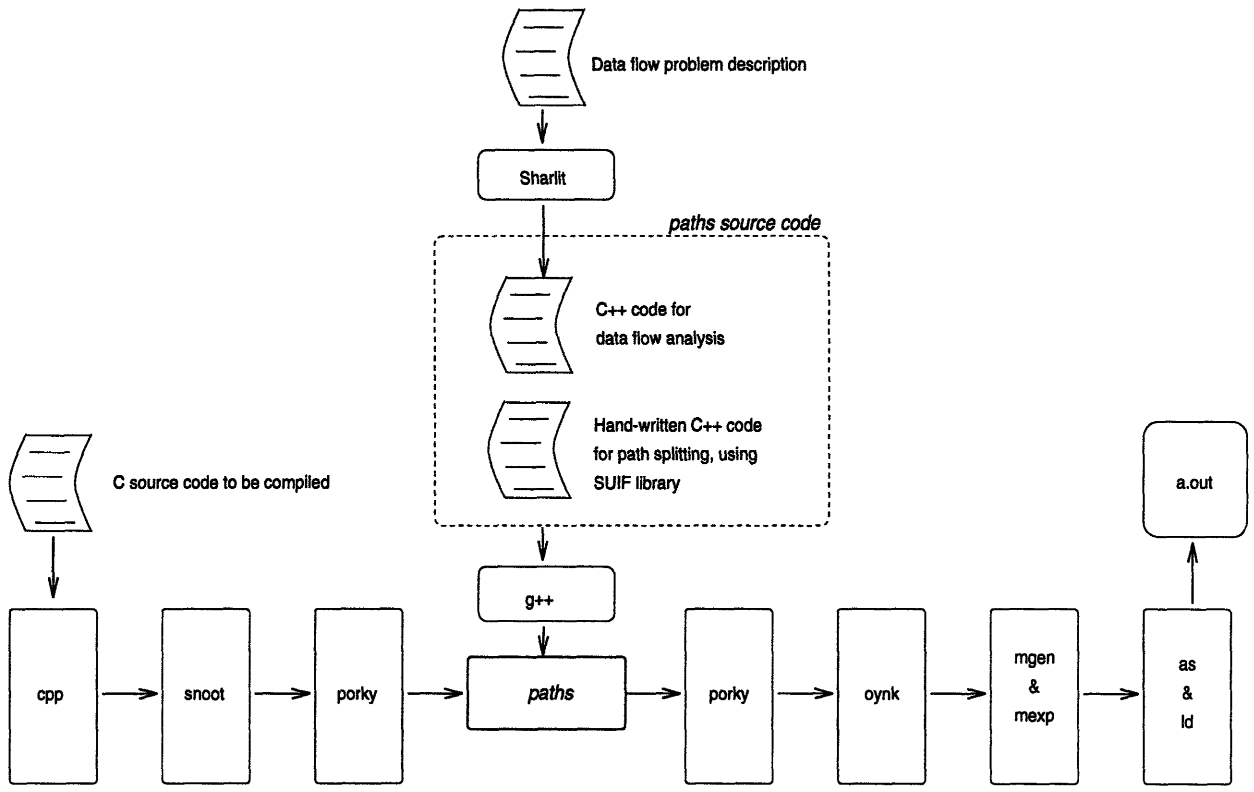
In addition to the core intermediate language kernel and the compiler system written on top of it, SUIF provides several libraries (most of them useful for compilation of parallel programs) and various utility programs. Among them is Sharlit, a data flow analyzer generator [41, 42]. Similarly to how YACC or other parser generators create source code for a parser given a grammar, Sharlit uses a specification of a data flow problem to generate a program that performs data flow analysis on a SUIF program representation. Using Sharlit, one can describe various types of data flow analyses, including iterative relaxation and interval analysis. In our experience, Sharlit has proved to be useful and versatile. It is a good tool for quickly describing and prototyping data flow analyses.

4.2 The Implementation of Path Splitting

Due to the structure of SUIF, the path splitting program, `paths`, is not excessively complicated. Reaching definitions and liveness analysis data flow problems were specified using Sharlit, and the rest of the algorithm was written using the main SUIF library. The entire path splitting algorithm, together with flow graph and data flow information abstractions, facilities for I/O and debugging, and Sharlit data flow specifications is a little over 4000 lines of commented C++ code.

The consistent interface between passes made inserting the path-splitting pass into the rest of the compiler simple. As shown in Figure 4.3, the path-splitting program, `paths`, is invoked after invoking the `porky` program. A C program to be compiled is first fed through the preprocessor, then converted to SUIF by the program `snoot`, and subsequently modified by `porky`. `porky` performs various optimizations, including constant folding and propagation, induction variable elimination, and forward propagation of loop tests, and ensures that no irreducible loops are associated with high-level "loop" constructs in the abstract syntax tree. `paths` reads the intermediate form produced by `porky`, and for each procedure performs reaching definitions analysis, finds well-structured loops (by searching for SUIF "loop" constructs), and performs path splitting. It then writes the binary intermediate form out to disk for use by the back end.

After path splitting, `porky` and `oynk` perform additional scalar optimizations, and then `mgen` and `mexp` generate MIPS assembler code, which is passed to the system assembler and linker. At various points throughout the compilation, other minor passes convert the intermediate representation to and from two slightly different flavors of the SUIF format used by different passes, but their operation is transparent and we omit them for simplicity. When emitting code to a machine other than the MIPS, a SUIF-to-C translator is invoked in place of `oynk` and the rest of the back end. Optimization and code generation are then performed by the system C compiler.



Data flow problem description
Sharlit
paths source code
C++ code for data flow analysis
Hand-written C++ code for path splitting, using SUIF library
C source code to be compiled
a.out
g++
cpp
snoot
porky
paths
porky
oynk
mgen & mexp
as & ld

Figure 4.3. Path splitting inside the SUIF framework.

Chapter 5

An Evaluation of Path Splitting

Previous chapters outlined the design and implementation of path splitting: this chapter describes and evaluates its effects. First, Section 5.1 presents simple examples of code for which path splitting improves data flow analysis, enabling other data flow optimizations, such as copy propagation and common subexpression elimination. Subsequently, Section 5.2 reports the results of performing path splitting on a variety of benchmarks.

5.1 Examples

This section presents small examples of code for which path splitting is useful, and illustrates the algorithm's effect on them. In each figure, the fragment of code on the left can be improved by applying path splitting together with traditional optimizations. The fragment on the right is the C representation of the result of applying these optimizations to the fragment on the left.

Consider Figure 5.1. The value of `x` is a function of `a`, and may be determined directly for all values of `a`, but the code is written in such a way that standard data flow analysis will not provide any useful information. After path splitting, however, each assignment to `y` is specialized to a set of values of `a`, so that the appropriate value of `x` is known in each case. As a result, `x` is no longer needed, enabling the removal of the assignment `x = 2` and potentially freeing one register or avoiding a memory reference.

Figure 5.2 is an example of how path splitting can help in common subexpression elimination. In the loop, the expression `k+z` would be available at the assignment to `x` for all values of `a` other than `a=50`. As a result of this exception, CSE cannot be performed. After

```
...
int x = 2;
int y = 0;
while (a<100) {
  if (a>50)
    x = 3;
  y = y+x;
  a++;
}
...
```

before

```
...
int y = 0;
while (a<100) {
  if (a>50) {
    y = y+3;
    a++; continue;
  } else {
    y = y+2;
    a++; continue;
  }
}
...
```

after

Figure 5.1. Code example 1: constant propagation.

```
...
int x = 2;
while (a<100) {
  w = k+z;
  ...
  if (a==50)
    k++;

  x = k+z;
  a++;
}
...
```

before

```
...
int x = 2;
while (a<100) {
  w = k+z;
  ...
  if (a==50) {
    k++; x = k+z;
    a++;
    continue;
  } else {
    x = w; a++;
    continue;
  }
}
...
```

after

Figure 5.2. Code example 2: common subexpression elimination.

path splitting on the use of `k` in `x=k+z`, however, each copy of this assignment is reached by a unique definition of its operands, and `k+z` is available at one of the assignments, allowing it to be substituted with a reference to `w`.

Figure 5.3 illustrates how path splitting can enable copy propagation (in this case, actually, constant propagation). The improved data flow information leads to folding of an

```
...
for (j=0;j<100;j++) {
  if (j%2)
    a = 3;
  else
    a = 4;
  if (j%3)
    b = 6;
  else
    b = 7;
  c += a+b;
}
...
```

before

```
...
for (j=0;j<100;j++) {
  if (j%2) {
    if (j%3)
      c += 9;
    else
      c += 10;
  } else {
    if (j%3)
      c += 10;
    else
      c += 11;
  }
  ...
}
```

after

Figure 5.3. Code example 3: copy propagation and constant folding.

arithmetic expression on every path through the loop, decreasing the number of instructions which need to be performed on each iteration.

Figure 5.4 is another example of copy propagation, constant folding, and code motion permitted by path splitting. Note that the C `continue` statement is generally implemented by a jump to the bottom of the loop, where an exit condition is evaluated and a conditional branch taken to the top of the loop if the exit condition is false. Exploiting the ideas of [34], we save the additional jump to the loop test by duplicating the loop test at all points where a loop now ends. If the test evaluates to false, a jump is taken to the end of the entire loop "tree" so that execution can continue in the right place.

There are other small advantages to be had from path splitting. On machines with direct-mapped instruction caches, for example, we expect path splitting to improve locality in those loops where a test is made but the condition changes rarely. Consider Figure 5.5. In this case, path splitting on the value of `test` after the `if` statement would create two specialized sub-loops: one containing "code block 1," and the other without it. If `test` is indeed false, executing in the specialized loop not containing code block 1 will result in better locality. In this sort of situation, path splitting is a "conservative" version of loop un-switching [5]: it does not replicate the entire loop, but only certain portions of its body.

```
...
for (x=0;x<a;x++) {
  switch(j) {
  case 1:
    c = 4;
    d = 7;
    break;
  case 2:
    d = 2;
    break;
  case 3:
    c = 4;
    break;
  default:
  }
  g(c+d);
}
```

before

```
...
for (x=0;x<a;x++) {
  switch(j) {
  case 1:
    c = 4; d = 7;
    g(11);
    continue;
  case 2:
    d = 2; g(c+2);
    continue;
  case 3:
    c = 4; g(d+4);
    continue;
  default:
    g(c+d); continue;
  }
}
```

after

Figure 5.4. Code example 4: copy propagation, constant folding, and code motion.

```
...
test = f();
while (condition) {
  if (test)
    /* code block 1 here */
  ...
  /* other code here */
}
return y;
```

Figure 5.5. Code example 5: path splitting as a variant of loop un-switching.

5.2 Benchmarks

Path splitting introduces a tradeoff between static code size and dynamic instruction count. The more aggressively code is replicated and special-cased to certain sets of data flow values, the more effectively, presumably, other data flow optimizations will be performed, lowering the program's dynamic instruction count. Excessive code growth, however, can lead to poor cache behavior and, in extreme cases, to object files which are impractically large.

This section explores these concerns by measuring several benchmarks. The first group of benchmarks is "synthetic" — programs in this group are small, and contrived so as to especially benefit from path splitting. Each consists of one or more loops, in which joins prevent copy propagation, common subexpression elimination, constant folding, and dead code elimination from occurring. They are to some degree an upper bound on the improvements available through path splitting.

The second group of benchmarks consists of real programs, among them some SPECint92 benchmarks, the Stanford benchmarks, and some in-house utility programs for extracting statistics from text files (word histogram, word diameter[1]).

In order to obtain significant data, we performed three kinds of measurements: (1) instruction counts, both static and dynamic, (2) cache simulations, and (3) actual run time measurements. We briefly describe each in turn:

1. Static and dynamic instruction count. The former is simply the number of instructions in the object file, the latter is the number of cycles executed at run time. Note that the latter number is not an especially good indicator of program run time, since it ignores all memory hierarchy effects and operating system-related events such as network interrupts. Both measurements were taken using `pixie` and the `xsim` [39] simulator.

2. Cache "work." The numbers reported, labeled "Traffic" and "Misses," are the cumulative number of reads and writes to memory and the number of demand misses in the cache respectively. The data was collected with the `dineroIII` cache simulator from the WARTS toolkit [6, 31], modeling direct-mapped separate instruction and data caches of 64KB and 128KB respectively, as in a DECstation 5000/133 workstation.

3. Actual run times. This is how long the program actually took to run, averaged over 300 trials, each timed with the `getrusage` system call. Measurements were taken in a realistic environment, on a DECstation 5000/133 with 64MB of main memory, connected to the network and running in multi-user mode. As a sanity check, measurements for the large benchmarks were also made on a DECstation 3100 running in single user mode and disconnected from the network.

[1]The *diameter* of a list of words is the longest of all shortest paths between any two words in the list, where a path is a sequence of words in the list, each of the same length, such that two adjacent words differ only by one letter.

Before presenting the actual data, we outline the major results which can be drawn from them:

- Path splitting can provide large performance improvements over regions of code where it is applied, but at the cost of significant code growth.
- Path splitting does not significantly harm the cache performance of large programs, and can in fact sometimes improve it.
- Regions where path splitting is profitable are not extremely common. As a result, real run time benefits on large programs range between 0.2% and 7.5%, and code growth is always limited to under 5%.
- It is not always necessary to perform all of path splitting. Experiments show that simply performing the first phase, splitting the loop body into tree form, obtains noticeable performance improvements with relatively little code growth. Performing phase two (specializing loops) often degrades performance and causes code size to grow.
- Path splitting should be considered a "speculative" optimization, like loop unrolling or function inlining. In certain cases, it can cause the performance of a compiled program to degrade slightly.

5.2.1 Results for Synthetic Benchmarks

We begin by analyzing the results for the synthetic benchmarks. Figure 5.6 presents the dynamic instruction count results. The numerical data for these graphs, as well as for all other graphs in this section, is available in Appendix A. As can be seen, performing path splitting can significantly decrease the dynamic instruction count of a program.

Figure 5.7 documents actual run times of these benchmarks on a DECstation 5000/133. The black horizontal line on top of each bar is the "upper-bound" error bar corresponding to the 95% confidence interval for the data displayed. Error bars of this sort are provided in all figures which depict noisy or variable data. Please refer to Appendix A for additional information on the confidence intervals. From the figure we notice that path splitting does not provide as much run time benefit as could be hoped by looking only at the dynamic instruction counts. This is probably due primarily to two factors: (1) slightly poorer cache

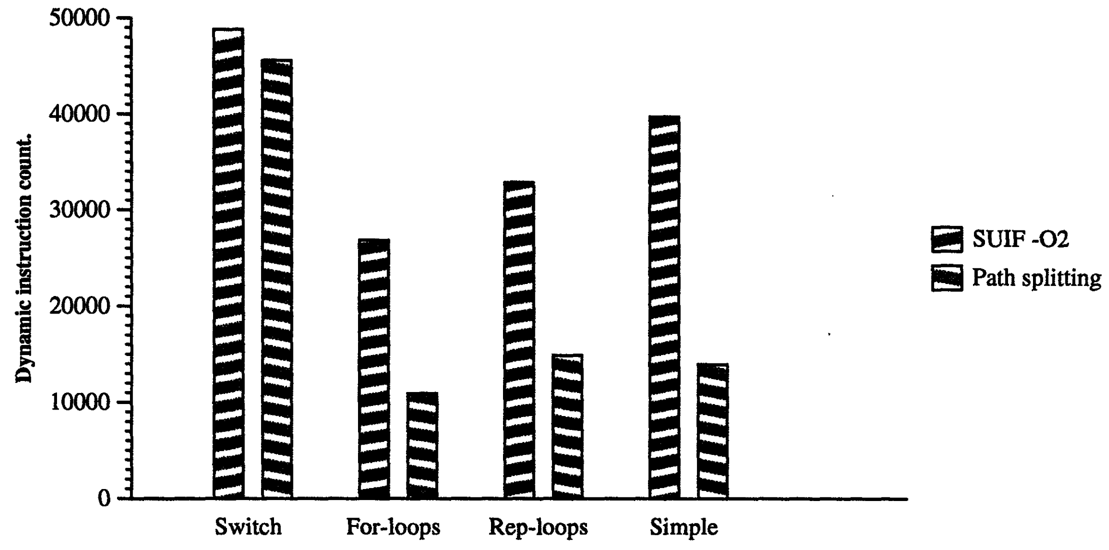


Figure 5.6. Dynamic instruction count results for synthetic benchmarks (see also Table A.1).

performance in the path splitting case (see below), and (2) noise and overhead involved with actually running the code on the machine.

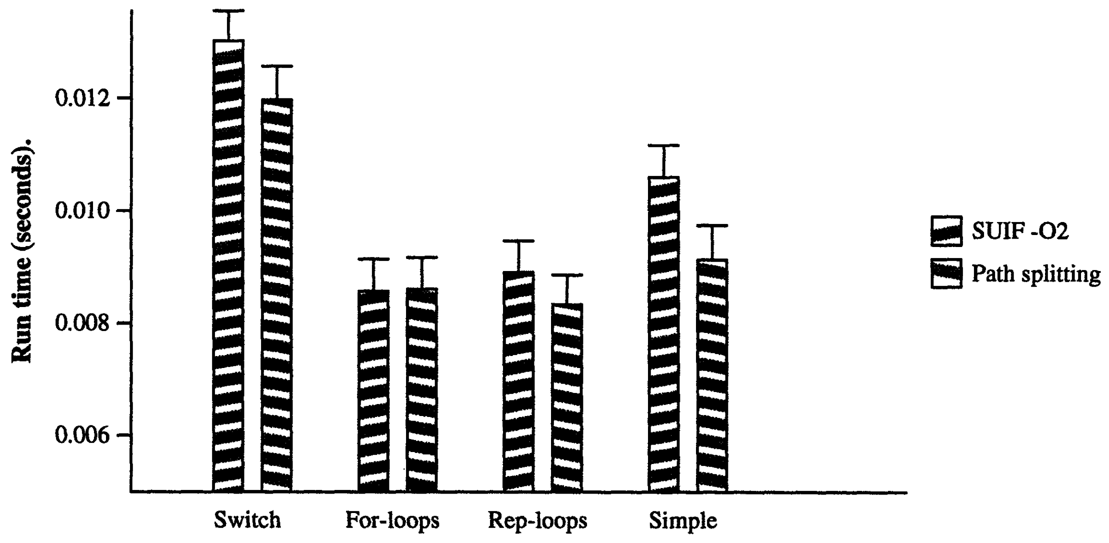


Figure 5.7. Measured run times of the synthetic benchmarks (seconds) (see also Table A.2).

Before moving on to larger benchmarks, we must consider code size growth and cache effects resulting from compiling the synthetic benchmarks with path splitting. Figure 5.8 illustrates the growth in static code size in the case of the synthetic benchmarks. We see that, especially in three of the four cases, static code size grows relatively little compared to the vast decreases in dynamic instruction count visible in Figure 5.6.

Finally, Table 5.1 shows the effect of path splitting on the synthetic benchmarks' cache

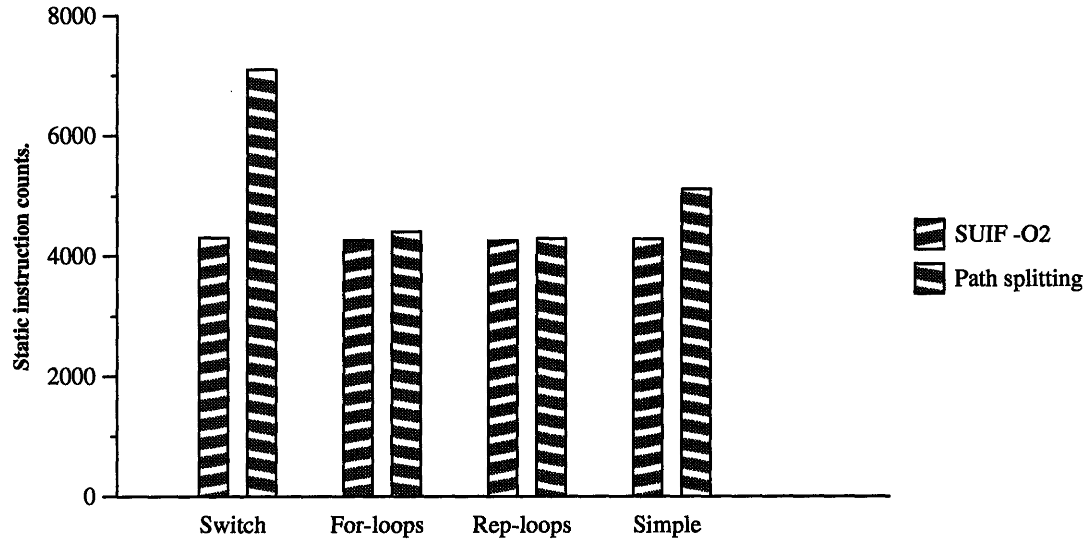


Figure 5.8. Static instruction count results for synthetic benchmarks (see also Table A.3).

performance. As mentioned previously, the "Traffic" columns refer to total traffic to memory (both reads and writes). The "Misses" column refers to the number of demand misses in the cache. The three columns, "SUIF -O2," "Paths (1)," and "Paths (2)," refer to measurements taken when the micro-benchmarks were compiled without path splitting, with only the first phase of path splitting (without unrolling specialized loops), and with complete path splitting, respectively. As expected, increasing code size has negative effects on cache performance, but this performance loss is small: memory traffic increases by no more than 5%, even in cases with dramatic code growth. This result is due in part to the small size of the benchmarks themselves. However, as discussed later, even for larger benchmarks cache performance decreases only by a small amount.

	(SUIF -O2)		Paths (1)		Paths (2)	
Benchmark	*Traffic*	*Misses*	*Traffic*	*Misses*	*Traffic*	*Misses*
Switch	2544	278	2536	278	2680	296
For-loops	2536	278	2560	281	2600	286
Rep-loops	2520	276	2536	278	2592	285
Simple	2528	277	2552	279	2592	284

Table 5.1. DineroIII results for synthetic benchmarks.

5.2.2 Results for Large Programs

Having discussed the synthetic benchmarks, we now present results on more representative, "real" code. These include 3 benchmarks from the SPECint92 suite (`008.espresso`, 023.eqntott, 026.compress), the Stanford benchmark from the MIPS architecture benchmark suite, the classic 8queens problem, Histogram, a program which measures the frequency of words in a text file, and Wordstat, which finds the diameter of a list of words.

We proceed as with the synthetic benchmarks, first presenting run time performance numbers, and then moving on to code size and cache effects. In order to obtain data about performance both in a "realistic" multi-user system and in a more controlled environment, we made run time measurements both on a fully functional DECstation 5000/133 in multi-user mode and on a DECstation 3100 in single-user mode disconnected from the network.

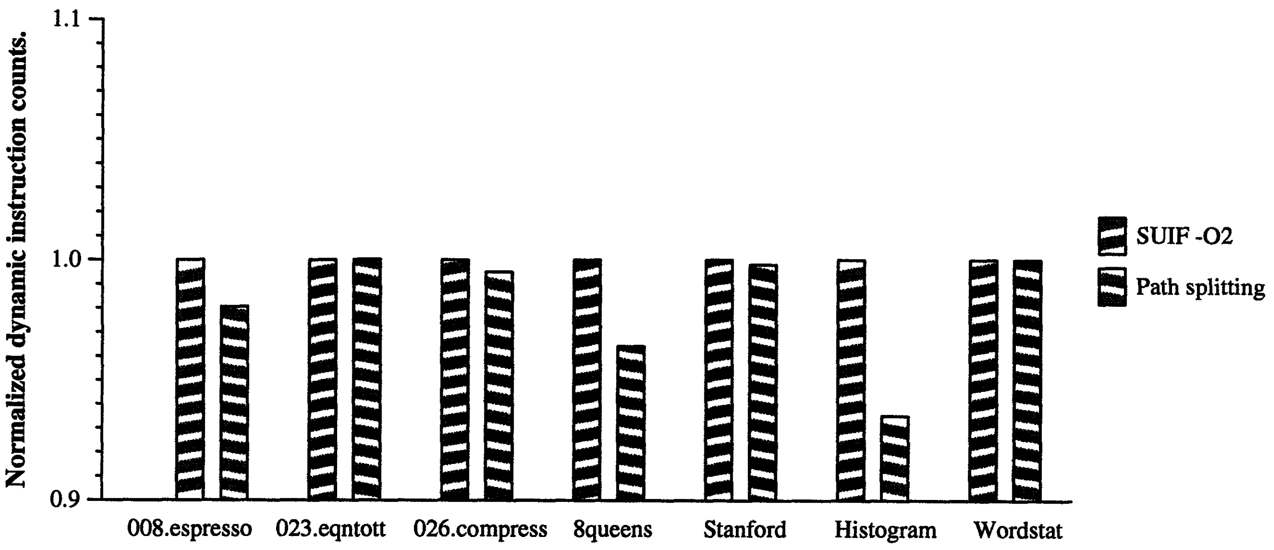


Figure 5.9. Dynamic instruction count results for the "real" benchmarks (see also Table A.4).

Figure 5.9 shows the normalized dynamic instruction count results for each one of these benchmarks, whereas Figure 5.10 is a normalized plot of measured run times on a DECstation 5000/133 in multi-user mode. The dynamic instruction count data shows that path splitting can decrease the number of instructions executed at run time by as much as 7% (Histogram). In one case, eqntott, path splitting results in an executable which actually takes more cycles at run time than the original. Although path splitting does not increase the number of *instructions* along any path, examination of assembly code has shown that in some cases (such as eqntott) restructuring loops can lead to less efficient use of delay slots, resulting in slightly negative results. This appears to be a very rare occurrence, however.

Run time performance after path splitting can improve by over 7% (Histogram). Moreover, since the 95% confidence intervals, denoted by the error bars, generally do not overlap

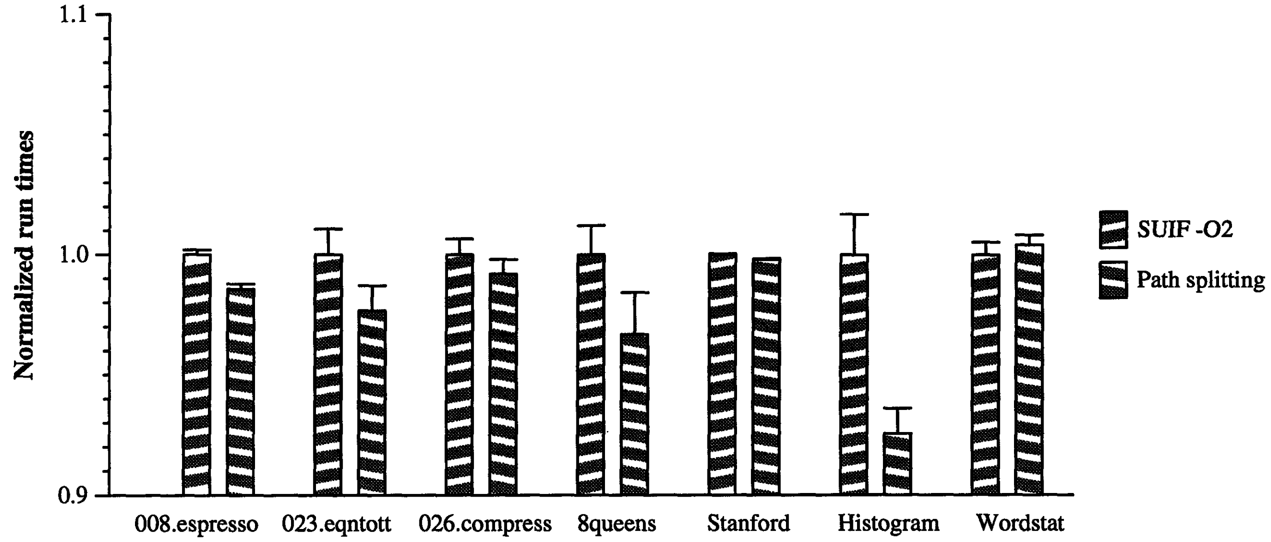


Figure 5.10. Measured run times of the "real" benchmarks (seconds) (see also Table A.5).

(in the case of `compress` there is a small overlap; only for `Wordstat` is there significant overlap), these measurements appear convincing. Only in one case (`Wordstat`) does path splitting degrade performance. Nevertheless, in that case performance only falls by 0.42%, and the confidence intervals overlap significantly, indicating that system noise influenced results to a large degree.

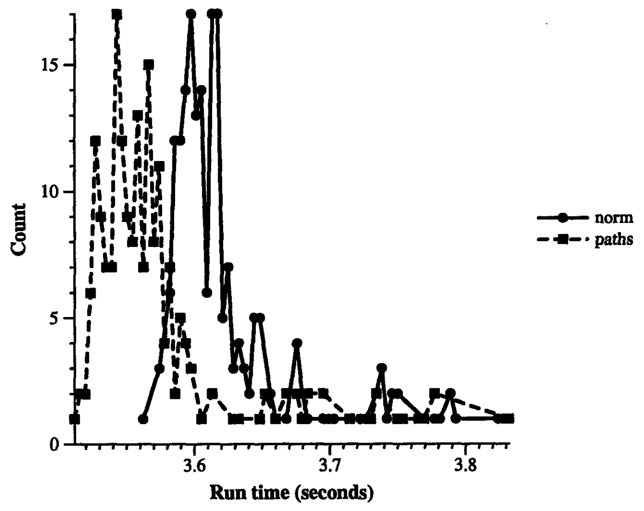


Figure 5.11. Histogram of `008.espresso` run times.

The issue of system noise when measuring actual run times on a workstation needs to be carefully considered. As an example, Figures 5.11 and 5.12 are intended to illustrate, more clearly than the error bars in Figure 5.10, the degree of noise and variability in system measurements when running the benchmarks on a fully configured workstation connected to the network. They are histograms of run times for the `008.espresso` and `Histogram` benchmarks, respectively. The x-axis measures time, in seconds, and the y-axis measures number of occurrences. In other words, a point at (x, y) indicates that the runtime of the given benchmark was x for y of the 300 measurements made. Despite the noise, path splitting still provides measurable and statistically significant performance improvements.

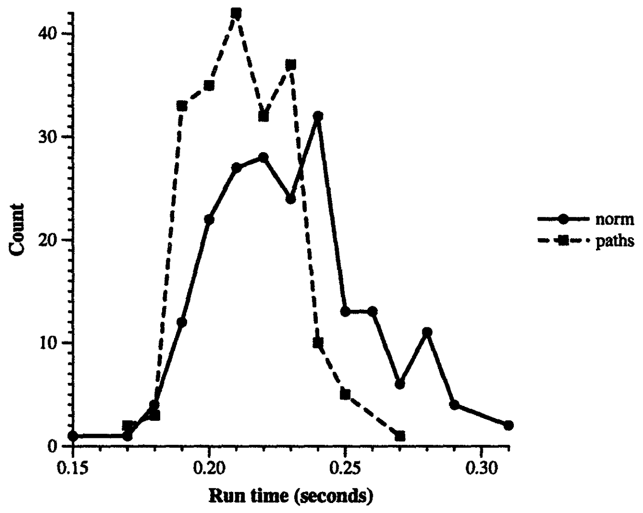


Figure 5.12. Histogram of `histogram` run times.

To provide real run time data less influenced by system noise, we also measured each benchmark's performance on a DECstation 3100 running in single-user mode and disconnected from the network. We applied the same methodology employed in the multi-user environment: 300 test runs per benchmark, with timings obtained through `getrusage`. Figure 5.13 depicts the resulting normalized data. Although performance differences change (path splitting appears to be relatively more effective on `Stanford` and `Wordstat`, and less effective on the other benchmarks), we still observe that path splitting offers significant performance improvements.

We now discuss code growth and cache performance issues. Figure 5.14 shows normalized static instruction counts for each benchmark. This data shows that, for large benchmarks, code growth caused by path splitting is generally less than 5% of the original code size.

Table 5.2 reports on cache activity for large benchmarks. In this case, unlike for the

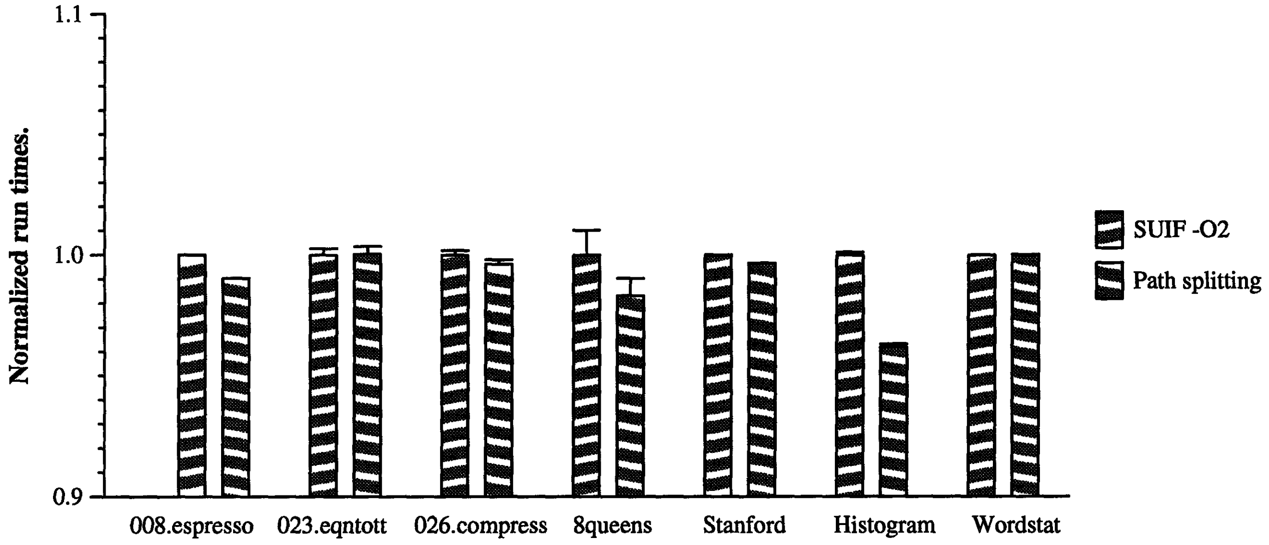


Figure 5.13. Measured run times of the "real" benchmarks in single-user-mode (see also Table A.6).

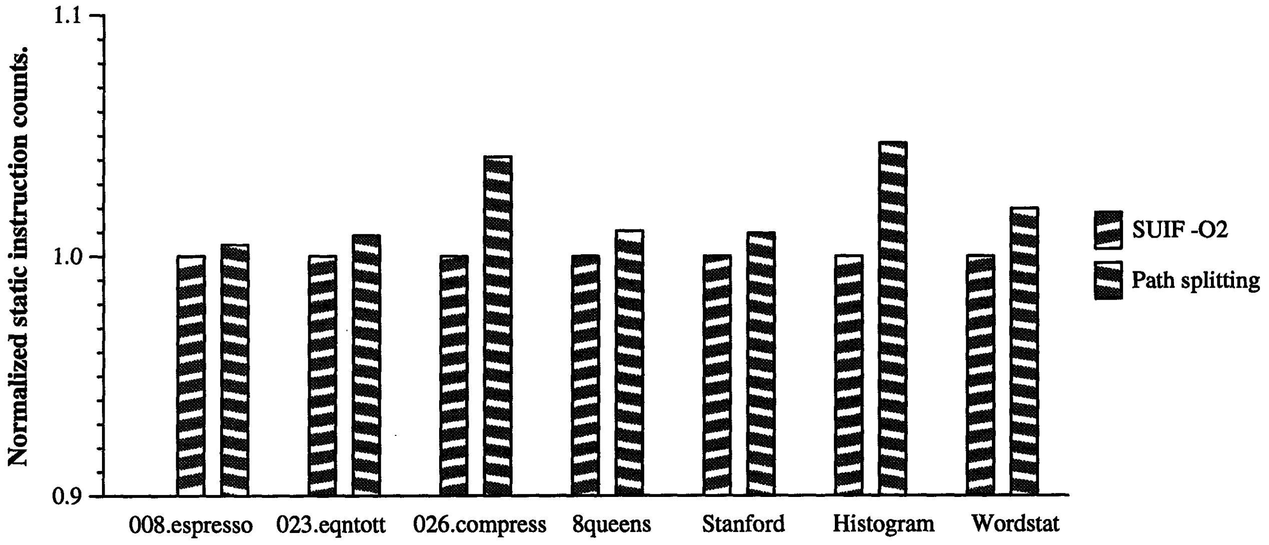


Figure 5.14. Static instruction count results for the "real" benchmarks (see also Table A.7).

synthetic tests, cache activity was only measured for the path splitting policy that produced best run time results. In general, this involved only phase one of path splitting (splitting the loop body into tree form), without specializing sub-loops, which did not improve run time performance and increased code size. As can be seen from the data, applying limited path splitting does not significantly harm cache behavior, and can sometimes even improve it (for example, `008.espresso`, `026.compress`).

	(SUIF -O2)		Path Splitting		% increase	
Benchmark	*Traffic*	*Misses*	*Traffic*	*Misses*	*Traffic*	*Misses*
008.espresso	160552	18533	146520	16779	-8.73	-9.46
023.eqntott	27376	2022	27432	2029	0.20	0.35
026.compress	806704	61431	806160	61431	-0.07	0.00
8queens	1776	179	1776	179	0.00	0.00
Stanford	536	51	536	51	0.00	0.00
Histogram	178056	13915	199536	16044	12.06	15.30
Wordstat	12936	8056	13000	1011	0.49	-87.45

Table 5.2. DineroIII results for real benchmarks.

5.3 Summary of Results

To conclude, path splitting can be useful. For most real benchmarks it provides noticeable improvements in real run time with negligible code growth. Decreases in performance, when they occur, are minimal and usually statistically insignificant, but they nonetheless indicate that path splitting is a "speculative" optimization, and should be applied with care. One slightly surprising discovery is that performing simply the first phase of path splitting, without specializing loops, is usually preferable to running both phases. Running both phases incurs large code growth penalties and related cache effects, while the additional data flow information apparently does not enable much more optimization. By running only the first phase of path splitting we reduce both code size growth and time required for compilation, while still preserving the performance benefits of path splitting.

Chapter 6

Discussion

This chapter discusses potential ramifications and extensions of path splitting, and comments on compiler design. Section 6.1 describes how path splitting can impact trace scheduling [16, 18], a technique for improving code generation on super-scalar systems. Section 6.2 describes a variant of the algorithm presented in Chapter 2 which appears useful for trace scheduling, and discusses possible methods for limiting the resultant code growth. Lastly, Section 6.3 comments on compiler construction, presenting experience gleaned through extensive work with a number of public-domain compilers.

6.1 Trace Scheduling

Trace scheduling is a technique for increasing the amount of instruction-level parallelism exploitable on a super-scalar or Very Long Instruction Word (VLIW) processor. The algorithm can be summarized as follows:

Trace selection The compiler selects a loop-free, linear section of code, possibly spanning several basic blocks, that has not yet been scheduled. This section of code is called a *trace.*

Instruction scheduling The compiler schedules instructions within the selected trace as if it were a single basic block, with no consideration for the source block of a given instruction. Two invariants are maintained: the relative order of branches is not changed, and an instruction is not moved up above a conditional branch if it kills a value live on the off-trace edge.

"Patch code" generation Where necessary, the compiler emits copies of instructions scheduled on the trace. Copies occur for two reasons:

- An instruction has been rescheduled after a conditional branch that used to follow it, so it needs to be placed before the target of the off-trace edge;
- Joins that jumped into the trace can only jump to a point after which all instructions present were originally following the join. Any instructions moved above the join must be replicated to execute before the join in the rescheduled trace.

By considering an entire trace (which may consist of any number of basic blocks) as a basic block, the compiler has a greater opportunity to schedule instructions so as to make use of the greatest number of functional units on any given cycle.

Aggressive path splitting, by replicating code to eliminate joins even when this does not directly benefit data-flow information, produces two advantages for trace scheduling:

- Like loop unrolling, inlining, and other techniques, it increases the amount of code available for selecting traces on which to reschedule instructions.
- The specialized loop bodies emitted at the end of path splitting have no joins entering them, potentially decreasing the amount of code which needs to be replicated per trace during the generation of patch code (of course, path splitting increases the size of code in any case, but this code growth can at least be used to enable other optimizations).

We are currently still determining the effects of path splitting and uncontrolled path splitting on super-scalar machines with trace-scheduling compilers. Although data is not yet available, this chapter nevertheless presents uncontrolled path splitting. Section 6.2.1 describes the algorithm. Section 6.2.2 discusses issues related to limiting code size increase.

6.2 Another Flavor of Path Splitting

The path splitting algorithm presented in Chapter 2 only performs splitting when the use of a variable is "specializable": in other words, when, according to some criterion (reaching definitions, for example), data flow information reaching a node is imprecise and could lead to additional optimizations if made more precise.

This "usefulness" check is beneficial for compiling code on normal scalar machines, since it controls code size growth. However, as described above, it may be profitable to more aggressively increase code size and restructure the bodies of loops in the interests of trace scheduling. "Uncontrolled" path splitting is a variation of the path splitting algorithm which removes joins without considering benefits to data flow information. Code size growth is limited by only performing the splitting on the most heavily executed regions of code, as determined through run-time feedback or static branch prediction.

6.2.1 Uncontrolled Path Splitting

Algorithm 6.1 outlines uncontrolled path splitting. The last two steps are identical to those of the first path splitting algorithm. The first phase of the algorithm, however, differs in that it removes *all* joins present in a given region, whether this removal improves data flow information or not. Algorithms 6.2 and 6.3 describe this phase of path splitting. Since joins are introduced either by unconditional jumps or by the meeting of the branch-taken and fall-through paths following a conditional branch, these two procedures simply traverse the given loop body, replicating code to eliminate unconditional jumps and to remove joins caused by conditional branches.

```
SPLIT-PATHS-UNCONTROLLED(p:procedure)
    foreach l:loop in reverse-post-order(loops(p)) do
        remove-unconditional-jumps(l)
        remove-joins(l)
        specialize-loop(l)
        remove-unnecessary-code(l)
```

Code

- Functions **remove-unconditional-jumps** and **remove-joins** are described in algorithms 6.2 and 6.3 respectively.
- Other notation, and functions **specialize-loop** and **remove-unnecessary-code** are as in Chapter 2.

Legend

Algorithm 6.1. Uncontrolled Path Splitting.

```
REMOVE-UNCONDITIONAL-JUMPS(l:loop)
    foreach s:statement in reverse-bfs-order(statements(l)) do
        if is-unconditional-jump(s) and not is-loop-back-edge(s) then
            j:statement := branch-taken-successor(s)
            if parent-loop(s) = parent-loop(j) then
                seq:sequence := copy(find-sequence(j))
                insert-sequence(seq,s)
                mark-end-of-path(seq)
```

Code

- **branch-taken-successor**(s) returns the successor of conditional branch s that is on the path taken when the branch condition is true.
- **find-sequence**(s) returns the sequence of instructions between s and the end of the innermost loop containing s.
- **is-loop-back-edge**(s) returns true if one of the successors of s is the head of the reducible loop that s is in.
- **parent-loop**(s) returns the innermost loop containing s.
- All other notation is as in Chapter 2.

Legend

Algorithm 6.2. Uncontrolled Path Splitting: function REMOVE-UNCONDITIONAL-JUMPS.

```
REMOVE-JOINS(l:loop)
    foreach s:statement in reverse(statements(l)) do
        if is-conditional-jump(s) and not is-loop-back-edge(s) then
            j:statement := branch-taken-successor(s)
            if parent-loop(j) = parent-loop(s)
                    and path-exists(immediate-successor(s),j) then
                seq:sequence := copy(find-sequence(j))
                insert-sequence(seq,immediate-predecessor(j))
                mark-end-of-path(seq)
```

Code

- **immediate-successor**(s) and **immediate-predecessor**(s) refer to the statement (instruction) immediately following or preceding (respectively) s in the text segment.
- **path-exists**(i, j) is true if any path not containing a loop back edge exists from i to j, false otherwise.

Legend

Algorithm 6.3. Uncontrolled Path Splitting: function REMOVE-JOINS.

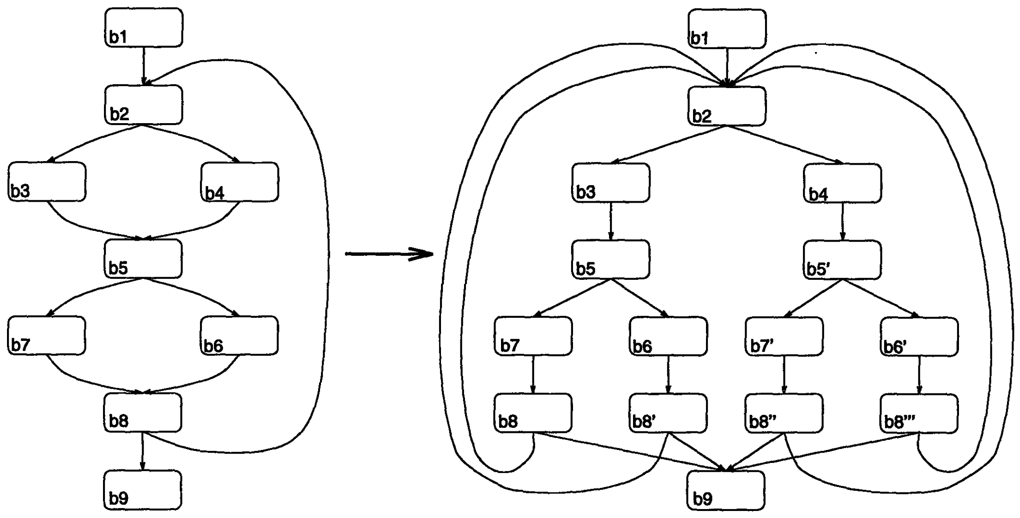


Figure 6.1. Converting the inner body of a loop from a DAG to a tree.

An example of this process appears in Figure 6.1. As can be seen, all joins in the loop body are removed, by appropriately replicating blocks of code which follow them. The same restructuring would occur after the first phase of the path splitting algorithm presented in Chapter 2, if it were the case that *each* join in the loop body polluted the reaching definition information of uses following it.

6.2.2 Code Growth Issues

"Uncontrolled" path splitting, without regard for its usefulness in improving data-flow information, can lead to code growth exponential in the number of joins in the control-flow graph. For all real applications compiled with this technique (for example, gcc and espresso), assembler file size often increased by a few orders of magnitude, rendering the technique infeasible.

One obvious method for reducing this code growth is feedback. Samples [37] presents an in-depth analysis of light-weight, efficient methods for run-time profiling. Ball and Larus [6] also discuss this problem. Another, promising technique is static branch prediction and profile estimation. Various studies [22, 44, 43] indicate that static flow analysis is almost as accurate as run-time profiling. The approximately 80% branch prediction accuracy of static methods should be sufficient to identify the critical regions of code in which it might be profitable to perform uncontrolled path splitting. Moreover, static branch prediction eliminates the possibly lengthy compilation–feedback–re-compilation loop required for runtime profiling, and is not susceptible to the quality of sample data sets used to collect run-time feedback data.

6.3 Comments on Compiler Design

We now change gears, and present some opinions and ideas about compiler construction gathered through the implementation (or in some cases, attempted implementation) of this thesis in a few widely-used publicly available compilers. From this experience, we observe a dichotomy between the power and versatility of various compilers and the elegance and modularity of their design.

On one side of this spectrum lies the GNU C compiler, popular because it emits efficient code and is easily retargetable to new architectures. Unfortunately, `gcc` suffers from a lack of internal structure, and is clearly the work of many people hacking often independently over a long period of time. The loop restructuring algorithms presented in this thesis were first implemented in `gcc`, and the resulting compiler emitted correct, appropriately restructured code. However, the other loop optimizations which should have been enabled by the restructuring, such as copy propagation, stopped working as a result of undocumented assumptions made by other passes about the structure of the intermediate representation.

At the other end of the spectrum is SUIF, which provides a clean, well-defined intermediate representation, and in which different optimization passes may be combined and reordered freely and easily. Unfortunately, as a result of its primary purpose as a compiler for parallel machines, SUIF generates scalar code slightly inferior in quality (approximately a 10% slowdown) to that of `gcc`. In addition, since compilation passes share information by reading and writing binary files to disk, and since analyses have to be performed independently by each pass, SUIF is noticeably slower than other compilers for all but the smallest programs.

Another compiler in this spectrum is `vpcc`, the Very Portable C Compiler from the University of Virginia. `vpcc` and its companion optimizer, `vpo` [7], are structured in a more rational fashion than `gcc`, but they lack SUIF's powerful hierarchical intermediate representation, and provide quite minimal facilities for code replication and for structuring data flow analyses. In addition, fields containing data flow information are embedded directly into vpo's control flow graph, making the addition of new information or analyses rather awkward.

Lastly, Fraser and Hanson's lcc [19] is a well structured, retargetable C compiler, but currently lacks any facilities for global code optimizations. Nonetheless, it efficiently emits code of reasonable quality by using aggressive peephole optimizations.

All this research has led to a few considerations about issues and desiderata in the design of a research optimizing compiler for scalar machines. These ideas are by no means new; most are "common sense" rules of software design. All are certainly issues to be carefully considered when designing compilers, not laws set in stone. Each point below is adapted from Lampson's "Hints for Computer System Design" [29], which, not too surprisingly, applies to compiler design as well.

- *Keep basic interfaces stable.* It is certainly both possible and desirable to specify an intermediate language interface and respect it throughout the compiler. SUIF does this, making experimentation fast and enjoyable. Unfortunately, gcc does not do this, so even the smallest change involves discovering obscure assumptions and "cute hacks" which cause the implementation of the intermediate language to deviate from its specification.

- *Handle normal and worst case separately.* SUIF provides an excellent facility for analyzing the intermediate representation: after every pass, a representation of the program being compiled is dumped to disk, and may be unparsed at leisure with appropriate tools. However, debugging the intermediate form is the common case for a relatively small fraction of the life-cycle of a compiler. In every-day, non-development use, SUIF can be frustratingly slow in compiling large programs. gcc, on the other hand, keeps the intermediate representation in memory throughout the compilation, and outputs a printed representation to a file after a given pass only if desired. Unfortunately, it is impossible to reconstruct the intermediate form from this printed representation, since the latter does not contain all necessary information. A combination of these two approaches, offering reasonably fast compilation but the possibility to view and edit the intermediate form independently, would be ideal. Tanenbaum *et al.* [40] take this philosophy to the extreme so as to obtain a system with both short compile times during the edit-compile-debug cycle and efficient code as a final product: they simply generated two customized versions of their base compiler, each optimized for one of these two cases.

- *Make it fast.* Peephole optimizations are very useful, especially during the initial development and debugging of a piece of software, because they provide substantial performance improvement at very little cost. Unlike most global optimizations, they can be applied in time linear with the size of input. The incremental benefit of performing global optimizations is generally not worth the compile-time cost, if compilation is occurring frequently. Thus, global optimizations usually need only be applied after the main implementation phase of a program is completed. Again, this is a result amply described in Tanenbaum *et al.* [40].

- *Do one thing well.* There appears to be a limit to the functionality that a system can support before its size and complexity make it essentially unmanageable. `gcc`'s retargetability features make it extremely useful as a tool for porting code to almost any architecture, but creeping "featurism" is contributing to making it very hard to manage. See more on this topic in the next paragraph.

- *KISS: Keep It Simple, Stupid.* `gcc` is littered with flags, special-case code, and other constructs which make development and experimentation painful. Code seems to have been added over the years without much sense for overall structure or coherence. When one adds a module to the existing code base, it becomes difficult to reason about its interactions with the rest of the system. It is likely that such special-casing generates particularly clever code in many situations, but it is not clear whether this slight performance improvement is worth the extra cost in development time and maintainability.

- *Use a good idea again.* Most classic data flow optimizations fit into a relaxation scheme wherein some operator is applied iteratively to all nodes in the flow graph until the data flow values reach a fixed point. Once this structure is identified and formalized, adding new analyses becomes simple. A data flow analyzer generator such as Sharlit makes the implementation of new analyses incalculably easier than in, for example, `gcc` or `vpcc`, because it allows the developer to focus on the properties of the specific data flow problem at a higher, more structured level, ignoring many details relating to the implementation of the relaxation algorithm and the underlying program representations.

- *Don't hide power.* SUIF provides a set of abstractions for viewing code at different levels of granularity, from the very high level ("procedure," "body," "loop") to the very low level ("load instruction," "branch instruction"). The front end presents the back end with a richly annotated, machine-independent abstract syntax tree, from which one can easily extract control- and data-flow information.

- *Divide and conquer.* vpo stores data-flow information (such as information about successors, predecessors, dominators, etc.) directly within the nodes of the control flow graph. This renders various tasks, such as code replication or insertion of additional data-flow information, more involved and less elegant than necessary. Furthermore, we have found it helpful to make a distinction between the syntactic structure of a program (including its abstract syntax tree and the layout of instructions in the text segment), and the representation of its various properties, such as control flow graphs, data flow graphs, value dependence graphs, etc.. The cost of developing and maintaining separate representations is likely to be less than that paid to augment or change a more tightly integrated, monolithic internal representation. Maintaining an abstract syntax tree as part of the intermediate representation makes analysis of the program considerably easier. For example, the lack of an abstract syntax tree representation in `lcc`'s back end is, in retrospect, one of its authors' big regrets.

- *Cache answers.* Incremental compilation and analysis are fundamental to a compiler's performance. SUIF is quite slow because the intermediate form is written to and read from disk between each phase. All auxiliary information (such as data and control flow graphs) is lost and must be recomputed for each pass. Furthermore, the code for data flow analysis generated by Sharlit can be quite slow because it does not support incremental additions to the control flow graph. Thus, whenever a substantial change is made to the program, the entire flow graph must be recomputed and the iterative data flow analysis performed once more. It would be immensely faster if both data and control flow graphs could be computed incrementally. Static single assignment form, a lightweight and incrementally modifiable implementation of reaching definition information, results in appreciable compiler performance improvements.

- *Use static analysis.* Lampson writes, "Use static analysis if you can . . . ; when a good static analysis is not possible, . . . fall back on a dynamic scheme." As described in Section 6.2.2, using runtime feedback to optimize code has various disadvantages, and techniques for static branch prediction [43, 22] and other control flow analyses are sufficiently good to make the overhead of collecting runtime feedback impractical. When necessary, dynamic code generation [15, 26] can be profitably used to exploit statically unknown runtime state.

In conclusion, the research community appears to be in need of a compiler system which generates good code and is both efficient and flexible. Such a compiler would need to be easily extensible with new optimization techniques and program representations, and quickly retargetable to new architectures. In addition, it should not meet these conditions at the expense of the quality and efficiency of emitted code. No system currently seems to exist which effectively satisfies all these requirements.

Chapter 7

Related Work

Replicating code, and hence increasing static instruction count, in order to decrease dynamic instruction count is not a new idea. Function inlining, in which the body of a function is substituted in place of a call to it in order to avoid function call overhead and trivially enable inter-procedural optimizations, has been described in various papers [38, 24, 13]. Loop unrolling [5] is a well-known technique for potentially reducing loop overhead, improving register and data cache locality, and increasing instruction-level parallelism. Code replication is fundamental in producing the long instruction sequences useful for compiling efficiently to VLIW architectures [2, 14, 16, 32].

Mueller and Whalley [34] describe the elimination of unconditional jumps by code replication. They present an algorithm for replacing unconditional jumps uniformly by replicating a sequence of instructions from the jump destination. This sequence is selected by following the shortest path from the destination to a suitable transfer of control (such as a loop end or a return from the function).

An extension of these ideas, applied to conditional branches, appears in [35]. The key idea of this work is to find paths through a loop on which the results of a conditional branch will be known, and either removing the branch (if it is never taken) or replicating code as with unconditional jumps so as to make the branch unnecessary and hence removable. Joins are introduced in the control-flow graph of a program as a result of unconditional jumps and when the fall-through and branch-taken successors of a conditional branch meet. In reading this work we realized that systematic elimination of these constructs in the appropriate places could be used to improve data-flow information following the joins.

Knoop *et al.* [28] present partial dead code elimination, a technique by which code which is dead along only some program paths is moved out of those paths and onto only those paths along which it is used. The algorithm has interesting code-restructuring effects, and decreases dynamic instruction count. Briggs and Cooper [8] illustrate partial redundancy elimination, an extension and generalization of common subexpression elimination and loop-invariant code motion. Granlund and Kenner [20] discuss eliminating conditional branches through the use of a super-optimizer, a program which finds optimal instruction sequences for small pieces of code through exhaustive search.

Warter *et al.* [45] describe enhanced modulo scheduling to improve software pipelining in loops with conditional branches. The technique uses if-conversion and code replication to transform loops with conditional branches into straight-line code before scheduling. After scheduling is performed, however, if-conversion is "undone," and the software-pipelined basic block representation is reconstructed by replacing the if-conversion predicates with conditional branches.

Chapter 8

Conclusion

This thesis has presented path splitting, a method for transforming the control flow graph of a program in order to statically expose additional data flow information to the compiler. In brief, code is replicated to remove joins which corrupt reaching definitions information in the bodies of loops, and the resulting loop bodies are unrolled to further improve data flow information in the unrolled portions. This loop restructuring increases the effectiveness of "classical" code optimizations, such as copy propagation and common subexpression elimination. In addition, it is expected to make the flow graph more amenable to trace scheduling, an important technique for extracting instruction-level parallelism on super-scalar machines.

Path splitting was implemented in the SUIF compiler, and measurements collected on scalar MIPS workstations. Measurements still need to be performed on super-scalar DEC Alpha machines employing a custom DEC compiler. For scalar machines, synthetic benchmarks indicate that the algorithm very effectively restructures loops in which additional data flow information is obtainable and useful in improving code quality. For large, real applications written in C, the improved data flow information created by path splitting can result in over 7% run time performance improvements.

In summary, path splitting should be considered a speculative optimization: it has never caused significant decreases in performance, but it may not always provide large performance benefits. However, in cases when it effectively improves the applicability of traditional optimizations, path splitting can lead to significant performance improvements.

Appendix A

Numerical Data

This appendix presents the actual numerical data from which the figures in Chapter 5 were generated.

In the tables displaying real run times, the columns labeled "95% CI" are intended to give an indication of the quality of the measurements in the presence of system noise. The two numbers in each field of these columns are the lower and upper bound of the 95% confidence interval for the mean of measured run times over the 300 times measured. For example, given the measured mean and variance for the "Switch" synthetic benchmark, there is only a 5% probability that the "actual" mean of run times (were one to continue the experiment *ad infinitum*) would be lower than 0.01251s or higher than 0.01353s.

A.1 Synthetic Benchmarks

Benchmark	Cycles (SUIF -O2)	Cycles (path splitting)	% improvement
Switch	48812	45599	6.58
For-loops	26852	10892	59.43
Rep-loops	32849	14881	54.69
Simple	39720	13903	64.99

Table A.1. Dynamic instruction count results for synthetic benchmarks (see also Figure 5.6).

	(SUIF -O2)		Path Splitting		
Benchmark	Times	95% CI	Times	95% CI	% improvement
Switch	0.01302	0.01251 - 0.01353	0.01197	0.01135 - 0.01259	8.06
For-loops	0.00857	0.00795 - 0.00919	0.00861	0.00810 - 0.00912	0.47
Rep-loops	0.00891	0.00840 - 0.00942	0.00833	0.00782 - 0.00884	6.51
Simple	0.01060	0.01009 - 0.01111	0.00913	0.00851 - 0.00975	13.87

Table A.2. Run times for synthetic benchmarks (seconds) (see also Figure 5.7).

Benchmark	Size (SUIF -O2)	Size (path splitting)	% growth
Switch	4309	7104	64.86
For-loops	4266	4410	3.37
Rep-loops	4265	4298	0.77
Simple	4296	5124	19.27

Table A.3. Static instruction count results for synthetic benchmarks (see also Figure 5.8).

A.2 Real Benchmarks

Benchmark	Cycles (SUIF -O2)	Cycles (path splitting)	% improvement
008.espresso	1694662	1661668	1.95
023.eqntott	2676450	2677321	-0.03
026.compress	9390125	9342395	0.51
8queens	664007	640173	3.59
Stanford	66870245	66732245	0.21
Histogram	9904699	9263364	6.48
Wordstat	285008	285008	0.00

Table A.4. Dynamic instruction count results for "real" benchmarks (see also Figure 5.9).

	(SUIF -O2)		Path Splitting		
Benchmark	Times	95% CI	Times	95% CI	% improvement
008.espresso	3.62721	3.62005 - 3.6343	3.57458	3.56663 - 3.58253	1.45
023.eqntott	0.10798	0.10682 - 0.10914	0.10546	0.10505 - 0.10657	2.33
026.compress	0.57727	0.57350 - 0.58104	0.57258	0.56902 - 0.57614	0.81
8queens	0.02957	0.02921 - 0.02993	0.02859	0.02808 - 0.02910	3.31
Stanford	12.49315	12.48791 - 12.49839	12.46821	12.46323 - 12.47319	0.20
Histogram	0.22930	0.22543 - 0.23317	0.21220	0.20980 - 0.21460	7.46
Wordstat	0.69185	0.68826 - 0.69544	0.69475	0.69188 - 0.69762	-0.42

Table A.5. Run times for "real" benchmarks (seconds) (see also Figure 5.10).

	(SUIF -O2)		Path Splitting		
Benchmark	Times	95% CI	Times	95% CI	% improvement
008.espresso	6.59717	6.59629 - 6.59805	6.53431	6.53343 - 6.53519	0.95
023.eqntott	0.19495	0.19444 - 0.19546	0.19503	0.19441 - 0.19565	-0.04
026.compress	0.81047	0.80895 - 0.81199	0.80749	0.80601 - 0.80897	0.37
8queens	0.05019	0.04968 - 0.05070	0.04935	0.04899 - 0.04971	1.67
Stanford	20.30852	20.30634 - 20.31070	20.24008	20.23481 - 20.24535	0.34
Histogram	0.71822	0.71734 - 0.71910	0.69102	0.69014 - 0.69190	3.79
Wordstat	2.55045	2.54994 - 2.55096	2.55071	2.55009 - 2.55133	-0.01

Table A.6. Run times for "real" benchmarks in single user mode (seconds) (see also Figure 5.13).

Benchmark	Size (SUIF -O2)	Size (path splitting)	% growth
008.espresso	49477	49712	0.47
023.eqntott	10575	10667	0.87
026.compress	5726	5962	4.21
Stanford	6529	6592	0.96
8queens	2829	2859	1.06
Histogram	5025	5260	4.68
Wordstat	5208	5311	1.98

Table A.7. Static instruction count results for "real" benchmarks (see also Figure 5.14).

Bibliography

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

[2] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 308–317, Atlanta, GA, June 1988.

[3] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[4] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. Technical report, Computer Science Division, U.C. Berkeley, Berkeley, CA, 1992.

[6] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, NM, January 1992.

[7] M. E. Benitez and Jack W. Davidson. A portable global optimizer and linker. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 329–338, Atlanta, GA, June 1988.

[8] Presont Briggs and Keith D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–169, June 1994.

[9] Craig Chambers and David Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, Portland, OR, June 1989.

[10] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice & Experience*, 21(12):1301–1321, December 1991.

[11] T. H. Cormen, C. E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

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

[13] Jack W. Davidson and D. B. Whalley. Quick compilers using peephole optimization. *Software Practice & Experience*, 19(1):79–97, January 1989.

[14] J. R. Ellis. *Bulldog: a compiler for VLIW architectures.* MIT Press, 1985.

[15] D.R. Engler and T.A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 263–272, San Jose, CA, October 1994.

[16] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30:478–490, July 1981.

[17] J.A. Fisher and B.R. Rau. Instruction-level parallel processing. *Science*, (253):1233–1241, September 1991.

[18] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Boston, MA, October 1992.

[19] Christoper W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[20] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 341–352, San Francisco, CA, June 1992.

[21] Stanford SUIF Compiler Group. SUIF: A parallelizing & optimizing research compiler. CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.

[22] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu. Superblock formation using static program analysis. Technical report, Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, 1993.

[23] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, Amsterdam, 1977.

[24] W. W. Hwu and P. P. Chang. Inline function expansion for compiling C programs. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 246–258, Portland, OR, June 1989.

[25] J. B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[26] D. Keppel, S.J. Eggers, and R.R. Henry. A case for runtime code generation. TR 91-11-04, Univ. of Washington, 1991.

[27] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, New York, NY, October 1973.

[28] J. Knoop, O. Ruething, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 147–158, Orlando, FL, June 1994.

[29] B. W. Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 33–48, Bretton Woods, NH USA, December 1983. Published as ACM Operating Systems Review, SIGOPS, volume 17, number 5.

[30] James R. Larus. Efficient program tracing. *Computer*, 26(5):52–60, May 1993.

[31] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software Practice & Experience*, 24(2):197–218, February 1994.

[32] 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. In *The Journal of Supercomputing*, volume 7, pages 51–142, 1993.

[33] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, July 1965.

[34] F. Mueller and D. B. Whalley. Avoiding unconditional jumps by code replication. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 322–330, San Francisco, CA, June 1992.

[35] F. Mueller and D. B. Whalley. Avoiding conditional branches in loops by code replication. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995. (To appear).

[36] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, NY, June 1990.

[37] Alan Dain Samples. *Profile-driven Compilation.* PhD thesis, U.C. Berkeley, April 1991. U.C. Berkeley CSD-91-627.

[38] R. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9), September 1977.

[39] Michael D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[40] Andrew S. Tanenbaum, M. F. Kaashoek, K. G. Langendoen, and C. J. H. Jacobs. The design of very fast portable compilers. *SIGPLAN Notices*, 24(11):125–131, November 1989.

[41] S. W. K. Tjiang and J. L. Hennessy. Sharlit — a tool for building optimizers. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 82–93, San Francisco, CA, June 1992.

[42] Steven W. K. Tjiang. *Automatic Generation of Data-flow Analyzers: A tool for building optimizers.* PhD thesis, Stanford University, Computer Systems Laboratory, July 1993.

[43] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, Florida, June 1994.

[44] David W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 59–70, Toronto, Ontario, Canada, June 1991.

[45] Nancy J. Warter, Grant E. Haab, and Krishna Subramanian. Enhanced modulo scheduling for loops with conditional branches. *MICRO-25 Conference Proceedings*, December 1992.

7103-9184