Interprocedural Path Profiling and the Interprocedural Express-Lane Transformation

By
David Gordon Melski

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCES)

at the
UNIVERSITY OF WISCONSIN – MADISON
2002

Dedicated to my parents, John and Linda Melski

Abstract

The contributions of this thesis can be broadly divided into two categories: we present novel path-profiling techniques, and we present techniques for performing the express-lane transformation, a program transformation that duplicates frequently executed paths in the hope that better data-flow facts result along those paths.

In path profiling, a program is instrumented with code that counts the number of times particular finite-length path fragments of the program's control flow graph are executed. This thesis presents a number of extensions to the intraprocedural path-profiling technique of Ball and Larus. Several of our techniques collect information about interprocedural paths (*i.e.*, paths that cross procedure boundaries). We show that the overhead of our techniques is not prohibitive (300–700%), and that they often capture more information than the Ball-Larus technique.

The express-lane transformation isolates and duplicates hot paths in a program, aiming for better data-flow facts along the duplicated path. We describe several variants of the interprocedural express-lane transformation, each of which duplicates hot paths from an interprocedural path profile. We show that an interprocedural express-lane transformation helps range analysis to determine the outcome of 0–7% more branches than the intraprocedural express-lane transformation and 1.5–19% more branches than performing no transformation.

Code growth is one drawback of the express-lane transformation. When a pair of duplicate control-flow vertices have the same data-flow facts, it is desirable to eliminate one of the vertices (*e.g.*, by coalescing the duplicate vertices). We present several effective techniques for eliminating duplicated code that has a redundant data-flow solution; this helps to control code growth.

We also present experimental results for program optimizations that are based on: (1) performing an express-lane transformation; (2) performing range analysis; and (3) replacing decided branches and constant expressions. We show that when used with the intraprocedural express-lane transformation, this strategy leads to larger performance benefits than previously reported (0.7–13.0%). Using the interprocedural express-lane transformation also leads to performance benefits, although usually not enough to offset the costs incurred by the transformation. It is likely that a better implementation would lower these costs, possibly leading to a net performance gain.

Acknowledgements

I love being in Madison. And I have thoroughly enjoyed being a student in Madison. Even so, graduate school is hard, and I could not have accomplished anything without help.

First and foremost, I must thank my advisor Tom Reps for his patience and his guidance. I have learned a lot from Tom, including not just specific knowledge in the field of computer science, but also about how to think about problems and how to write. (Tom is an excellent editor and I wish there were time to get more feedback on the thesis; as it is, there are many rough patches for which I must take full responsibility.) I have been glad of the opportunity to work with him.

I would also like to thank my other committee members; I have tried to make the thesis easy to read, but I know it is both long and sometimes dense. I am also thankful for all of the people in the programming languages group at Wisconsin, including Susan Horwitz, Ras Bodik, Jim Larus, Tom Ball, Charles Fischer, Mike Siff, Manuvir Das, Alexey Loginov, Glenn Ammons and many more. All of these people have offered useful feedback and support. I cannot stress this enough: without the support and feedback from these people, I could not have accomplished anything. There are also colleagues outside of Wisconsin to whom I am grateful for support and suggestions, including Mooley Sagiv, Reinhard Wilhelm, Barbara Ryder, and Laurie Hendren.

I owe thanks to Glenn Ammons for his implementation of a Ball-Larus path profiler and his implementation of the intraprocedural express-lane transformation. They were a good starting point for my own implementations. I would also like to thank Mike Siff, Glenn Ammons, and Alexey Loginov for reading my prelim and calming me down before the oral presentation of my prelim.

There are other crucial players in my support network. Chief among these are my parents, John and Linda Melski. They are *always* there for me, and they are always supportive. I think that it is impossible to underestimate the importance of their support.

I have also been blessed with many great friends during my tenure in Madison. These include Amy Millen, Berit and Mark Givens, Eric Melski (my brother), Kasey Melski (my sister), Bill Winters, Amir Roth, Chris Lukas, Alain Roy, Alexey Loginov, and Meghan Wulster. These people have lifted my spirits countless times, and they always helped to relieve the pressures of graduate school. My soccer teams, the Crystal Corner and the Madison O2, were also great for relieving stress, both on the field and off.

There are many other people who have played an important role in my life while working on my Ph.D., and I am sure that I am forgetting to mention some important people. To those people, please know that I am grateful.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In path profiling, a program is instrumented with code that counts the number of times particular finite-length path fragments of the program's control-flow graph—or *observable paths*—are executed. A path profile for a given run of a program consists of a count of how often each observable path was executed. Thus, a path profile gives information about a program's execution behavior (*e.g.*, a path profile can be used to identify frequently executed program fragments). One potential application of path profiling (which we will examine in this thesis) is to transform the profiled program by isolating and optimizing frequently executed, or *hot*, paths. We call this transformation the *express-lane transformation*, and the isolated paths *express lanes*. More specifically, an express lane p is a copy of a hot path such that p has only one entry point at its beginning; p may branch into the original code, but the original code never branches to p. After the express-lane transformation has been performed, classical data-flow analysis is likely to find sharper data-flow facts along the express lanes, which may create program optimization opportunities.

1.1 Interprocedural Path Profiling

Ball and Larus have presented an efficient path-profiling technique that records how often each intraprocedural, acyclic path is executed [12]. Path profiling is of interest because a path profile contains more information than an edge profile or a vertex profile: it is always possible to calculate an edge profile from a path profile, but not vice versa. This is shown in Figure 1. Suppose that an edge profile for the control-flow graph shown in Figure 1 shows that each edge executed 50 times. Then very little can be determined about the execution frequency of any particular path: *e.g.*, the path $[A \rightarrow B \rightarrow D \rightarrow E \rightarrow G]$ could have executed anywhere from 0 to 50 times.

Suppose we have the following path profile for the control-flow graph in Figure 1:

Path	Execution Count
$A \rightarrow B \rightarrow D \rightarrow E \rightarrow G$	50
$A \rightarrow B \rightarrow D \rightarrow F \rightarrow G$	0
$A \rightarrow C \rightarrow D \rightarrow E \rightarrow G$	0
$A \rightarrow C \rightarrow D \rightarrow F \rightarrow G$	50

Compared with the edge profile, there is a good deal more information in this path profile:

1. We can calculate the edge profile from this path profile.
2. We can see that the branch taken at vertex D was perfectly correlated with the branch taken at vertex A: the branch at D was taken only when it followed an execution of A in which the branch at A was taken.
3. We can see that every time vertex E was executed, it was preceded by the execution of B and D. Likewise, every time vertex F was executed, it was preceded by the execution of C and D.

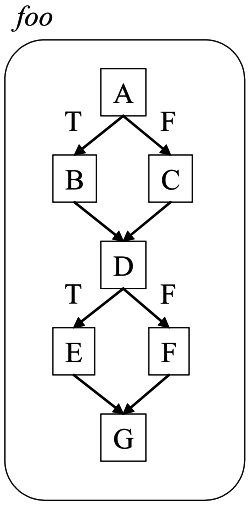


Figure 1: Example showing that a path profile contain more information than an edge profile.

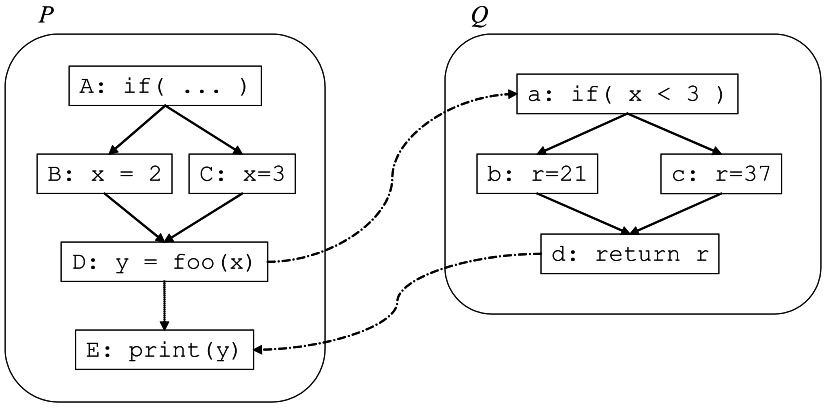


Figure 2: Example showing the use of an interprocedural path profile.

This information can be used to guide program optimization. One example of this application of path profiling is the express-lane transformation, which we will examine later [5]. (Chapter 13 discusses other work that uses path profiles to guide program optimization.)

We have extended the Ball-Larus technique in three directions:

1. **Interprocedural vs. Intraprocedural:** The Ball-Larus technique is an intraprocedural technique: it profiles each procedure separately, and the observable paths are all intraprocedural. We show how to extend the Ball-Larus technique to obtain interprocedural path-profiling techniques; that is, we present path profiling techniques in which the observable paths can cross procedure boundaries. Interprocedural path profiles are capable of capturing correlations between the execution behavior of different procedures (*e.g.*, in Figure 2, an interprocedural path profile might show that the branch at vertex a was perfectly correlated with the branch at vertex A). Furthermore, the average path in an interprocedural path profile is longer than the average path in an intraprocedural path profile, which also means that an interprocedural path profile tends to capture more of a program's run-time behavior than an intraprocedural path profile. (Longer paths may be better

for the express-lane transformation, since a long express-lane may keep a valuable data-flow fact alive longer than a short express-lane.)

2. **Context vs. Piecewise:** In piecewise path profiling, each observable path corresponds to a path that may occur as a subpath (or piece) of an execution sequence. The set of observable paths must cover every possible execution sequence. That is, every possible execution sequence can be partitioned into a sequence of observable paths. In context path profiling, each observable path corresponds to a pair $\langle C, p \rangle$, where p corresponds to a subpath of an execution sequence, and C corresponds to a context (*e.g.*, a sequence of pending calls) in which p may occur. The set of all p such that $\langle C, p \rangle$ is an observable path must cover every possible execution sequence. A context path-profiling technique generally maintains finer distinctions than a piecewise path-profiling technique.

3. **Edge Values vs. Edge Functions:** In our *functional* approach to path profiling, the numbering of observable paths is carried out via an edge-labeling scheme that is in much the same spirit as the path-numbering scheme of the Ball-Larus technique, where each edge of the control-flow graph is labeled with a number, and the "name" of a path is the sum of the numbers on the path's edges. However, in the functional approach, edges are labeled with *functions* instead of *values*. The path-profiling techniques that use edge-functions can maintain finer distinctions than those that use edge-values. For example, an interprocedural path-profiling that uses edge-values may use observable paths that begin in a procedure P and descend into a called procedure Q, or it may use observable paths that begin in a procedure Q and exit Q into a calling procedure P, but it cannot use observable paths that descend into called procedures and/or return from called procedures. Functional techniques for interprocedural path-profiling do not have this restriction.[1]

This thesis shows that for any combination of the above traits, there is at least one path-profiling technique with those traits. In effect, we give a toolkit for generating novel path-profiling techniques.

According to the above schema, the Ball-Larus technique is an example of intraprocedural piecewise path profiling; for the remainder of the thesis, we use "intraprocedural piecewise path profiling" and "Ball-Larus path profiling" interchangeably. In this thesis, we examine in detail the functional approaches to interprocedural context path profiling and interprocedural piecewise path profiling. We will show that the overhead of these techniques is reasonable (300-700%), though they are considerably more expensive than the Ball-Larus technique. Furthermore, both the interprocedural context path profile and the interprocedural piecewise path profile usually contain more information than the intraprocedural piecewise path profile. For example, the following table shows some statistics for various profiles of the SPEC95 benchmark 130.li when it is run on its reference input:

Profiling technique	Avg. num. instructions per path	Avg. num. of call-edges per path	Avg. num. of return-edges per path
Inter., context	107.7	3.0	1.0
Inter., piecewise	55.6	0.6	0.5
Intra., piecewise	36.1	-	-

A *call-edge* connects a call-site to the entry vertex of the called procedure. A *return-edge* connects an exit vertex of a procedure P to a call-site that calls P. The above table shows that the interprocedural

[1] Note that the "functional" approach to path profiling does not use a functional-programming style; it is so named because it labels edges with linear functions.

path-profiling techniques capture more information than the intraprocedural technique; furthermore, they do capture information about the "interprocedural" behavior of li (*e.g.*, correlations between the execution behavior of different procedures).

1.2 The Interprocedural, Express-Lane Transformation

The *express-lane transformation* seeks to transform a program such that subsequent data-flow analysis will find better data-flow facts along the program's frequently executed paths. Consider a hot, or frequently executed, path p. In the express-lane transformation, the program code is transformed so that there is a copy p' of p that has only one entry point at the beginning of p'; p' may branch into the original code, but the original code never branches to p'. We call p' an "express lane." The hope is that giving p' this special status may lead to sharper dataflow facts along p', thereby allowing greater optimization.

Section 2.2 summarizes the algorithm of Ammons and Larus for performing the intraprocedural express-lane transformation: the algorithm takes as input a control-flow graph, and a Ball-Larus path profile, and produces as output a hot-path graph in which the hot paths have been duplicated to form express-lanes [5]. In [5], Ammons and Larus show that the express-lane transformation does improve the results of conditional constant propagation [61]. Furthermore, they show that the express-lane transformation combined with replacing the constants found by conditional constant propagation can lead to improved program performance. However, performance is sometimes degraded due to the code growth caused by the transformation.

Ammons and Larus sought to control the problem of code growth by removing duplicated code when there was no benefit to data-flow analysis. The program optimization examined in [5] consists of the following steps: (1) perform the express-lane transformation; (2) perform conditional constant propagation; (3) reduce the hot-path graph while preserving constants found in previous step; and (4) replace constant expressions with literals. The Ammons-Larus technique for reducing the hot-path graph is based on an algorithm for minimizing deterministic, finite-state automata.

In Chapter 7, we describe how to extend the algorithm for performing the intraprocedural express-lane to an algorithm that takes as input the program supergraph (an interprocedural control-flow graph) and an interprocedural, path profile and produces as output a hot-path supergraph; the transformation of a supergraph into a hot-path supergraph is called the interprocedural express-lane transformation. We present algorithms for performing the interprocedural express-lane transformation for both interprocedural piecewise and interprocedural context path profiles.

We show that the interprocedural context express-lane transformation and the interprocedural piecewise express-lane transformation both have greater benefits for range analysis than the intraprocedural express-lane transformation does. (Some preliminary experiments suggest that the interprocedural express-lane transformations and the intraprocedural express-lane transformation have the same benefits on conditional constant propagation; therefore, we present results for range analysis, and not conditional constant propagation as [5] does.) For example, range analysis of the SPEC95 benchmark compress cannot determine if there are any conditional branches that have only one possible outcome. After the intraprocedural express-lane transformation is applied to compress, range analysis can determine that at least 2.2% of the conditional branches (weighted dynamically) have only one possible outcome. After either of the interprocedural express-lane transformations is applied to compress, range analysis can determine that at least 9.8% of the conditional branches have only one possible outcome.

1.2.1 Reducing the Hot-path Supergraph

Unfortunately, the interprocedural express-lane transformations can also cause a great deal more code growth than the intraprocedural express-lane transformation. The interprocedural, context express-lane transformation can cause 1600% code growth, although we limit the code growth to be between 20% and 400%. This amount of code growth is likely to cause performance degradation, so we examine techniques for reducing the hot-path supergraph. The technique used by Ammons and Larus cannot be applied directly to the hot-path supergraph: they use a DFA-minimization algorithm, which works on a hot-path graph (since a hot-path graph can be considered to be a deterministic, finite automaton); in contrast, a hot-path *supergraph* is a *pushdown* automaton. We show how to adapt the DFA-minimization algorithm (which is really a variant of the Coarsest Partitioning Algorithm [38, 1]) to obtain the Supergraph Partitioning Algorithm, which can be used to minimize the hot-path supergraph. The Supergraph Partitioning Algorithm is very effective: it reduces the amount of code growth (as compared to the original program) to be no more than 140% and usually less than 30%.

Even though the Supergraph Partitioning Algorithm is effective in practice, we show that there are some simple examples where it performs poorly. These examples motivated us to develop another technique for reducing the hot-path supergraph, called the Edge Redirection Algorithm. When combined with the Supergraph Partitioning Algorithm, the Edge Redirection Algorithm causes further reductions in the size of the hot-path supergraph.

1.2.2 Using the Express-Lane Transformation for Optimization

We have run experiments on using the express-lane transformation together with range analysis to optimize some of the SPEC95 benchmarks. Specifically, for five benchmarks, for three different profiling techniques, and for several different hot-path supergraph reduction strategies, we have performed the following procedure:

1. Collect a path profile.
2. Perform the express-lane transformation.
3. Perform interprocedural range analysis on the hot-path (super)graph.
4. Reduce the hot-path (super)graph.
5. Use the results of interprocedural range analysis to eliminate branches and to replace constant expressions with a literal.
6. Emit C source code for the transformed program.
7. Compile the C source code using GCC 2.95.3 -O3.
8. Compare the runtime of the new program with the runtime of the original program.

Our experiments show a greater benefit from the intraprocedural express-lane transformation than are reported in [5]: there was a 0.7–13.0% decrease in runtime for every benchmark. (Note however, that our use of the express-lane transformation is different than in [5]; for example, we perform interprocedural range analysis while intraprocedural constant propagation is performed in [5].) Interestingly, in the case of the intraprocedural express-lane transformation, the code growth caused by the express-lane transformation is not always detrimental; our experiments provide some evidence that GCC (when

optimizing at the -O3 level) is able take advantage of the express-lane transformation to perform additional optimizations. However, the greatest benefit is gained from the express-lane transformation when interprocedural range analysis is used to eliminate branches and replace constant expressions.

Our experiments also show that when the above steps are performed with an interprocedural path profile, there is a benefit to program performance, though it is usually not sufficient to overcome the costs of the interprocedural express-lane transformation. The interprocedural express-lane transformation has two associated costs: the first is due to code growth; the second is due to *entry splitting* and *exit splitting*. Entry and exit splitting are mechanisms that we use to duplicate interprocedural paths. Entry splitting allows a procedure to have more than one entry. Exit splitting allows a procedure to have more than one exit, and to return to a different location for each exit: normally, when a procedure call is made to procedure P a return address is given to P, and when P's exit is reached, control jumps to the return address; in a procedure call to a procedure P with multiple exits, a vector of return addresses is passed, one for each exit, and when one of P's exits is reached, control jumps to the appropriate return address. We used a particularly simple but inefficient implementation of entry and exit splitting. Unfortunately, this has a negative impact on our performance numbers.

Even with the high costs of the interprocedural express-lane transformation, we still occasionally get some performance benefit. Also, if we skip Step 5 of the above process (*i.e.*, eliminating branches and replacing constants), then program performance almost always drops, usually dramatically. This implies that the interprocedural express-lane transformation followed by range analysis is having a strong benefit on program performance.

The above algorithm for using the interprocedural express-lane transformation for program optimization complements the literature on profile-directed optimizations. More specifically, our approach differs from those in the literature in one or more of the following aspects:

1. We duplicate interprocedural paths before performing analysis.
2. We guide our path duplication using interprocedural path profiles.
3. We perform interprocedural range analysis on the transformed graph.
4. We attempt to eliminate duplicated code when there was no benefit to range analysis.

(Points 2 and 3 may sound redundant, but they are not. For example, in [20], edge profiles are used to duplicate intraprocedural paths.)

1.3 Organization of the Thesis

The thesis is organized as follows: Chapter 1 provides an overview of the thesis's contributions. Chapter 2 summarizes [12] and [5], which are crucial to understanding the thesis. Chapters 3 through 5 describe novel path-profiling techniques. Chapter 6 gives experimental results for our interprocedural path-profiling techniques. Chapter 7 describes how to perform the interprocedural, express-lane transformation. Chapter 8 gives experimental results for the express-lane transformation. Chapters 9 and 10 describe techniques for reducing the code growth caused by the express-lane transformation. Chapter 11 shows that this problem is NP-hard. Chapter 12 provides experimental results for the techniques presented in Chapters 9 and 10. Chapter 13 discusses related work. Chapter 14 offers some concluding remarks.

Chapter 2

Related Work

In Chapter 14, we will give an overview of related profiling and programing optimization work. In this Chapter, we give a more detailed summary of two pieces of work. Section 2.1 describes the Ball-Larus path profiling technique of [12]; our path profiling techniques are an extension of this work. Section 2.2 describes the intraprocedural express-lane transformation presented in [5]; our interprocedural express-lane transformations are an extension of this work. Familiarity with [12] and [5] will help to understand the remainder of the thesis.

2.1 Summary of the Ball-Larus Technique for Intraprocedural Path Profiling

This section summarizes the Ball-Larus path profiling technique presented in [12]. In path profiling, a program is instrumented with code that counts the number of times particular finite-length path fragments of the program's control-flow graph — or *observable paths* — are executed. A path profile for a given run of a program consists of a count of how often each observable path was executed. As mentioned above the Ball-Larus strategy can be summarized as follows:

1. Start with a graph that represents a program's control flow.
2. Apply a transformation that results in a new graph with a finite number of paths; each path through the transformed graph corresponds to a path fragment, called an *observable path*, in the original graph.
3. Number the paths in the transformed graph.
4. Instrument the program with code that counts how often each observable path is executed (by incrementing a counter associated with the path through the transformed graph).

The Ball-Larus path-numbering scheme (used in step 3 above) applies to an acyclic control-flow graph with a unique entry vertex $Entry$ and a unique exit vertex $Exit$. For purposes of numbering paths, control-flow graphs that contain cycles are modified by a preprocessing step to turn them into acyclic graphs (step 2 above):

> Every cycle must contain one backedge, which can be identified using depth-first search. For each backedge $w \rightarrow v$, add the surrogate edges $Entry \rightarrow v$ and $w \rightarrow$ *Exit* to the graph. Then remove all of the backedges from the graph.

The resulting graph is acyclic. In terms of the ultimate effect of this transformation on profiling, the result is that we go from having an infinite number of unbounded-length paths in the original control-flow graph to having a finite number of acyclic bounded-length paths in the modified graph. A path p in the original graph that proceeds several times around a loop will, in the profile, contribute "execution

counts" to several smaller "observable paths" whose concatenation makes up p. In particular, each path from $Entry$ to $Exit$ in the modified graph correspond to an observable path in the original graph (where following the edge $Entry \rightarrow v$ that was added to the modified graph corresponds to beginning a new observable path that starts with the backedge $w \rightarrow v$ of the original graph, and following the edge $w \rightarrow Exit$ that was added to the modified graph corresponds to ending an observable path in the original graph at w).

In the discussion below, when we refer to the "control-flow graph", we mean the transformed (*i.e.*, acyclic) version of the graph.

The Ball-Larus numbering scheme labels the control-flow graph with two quantities:

1. Each vertex v in the control-flow graph is labeled with a value, ***numPaths***$[v]$, which indicates the number of paths from v to the control-flow graph's $Exit$ vertex.

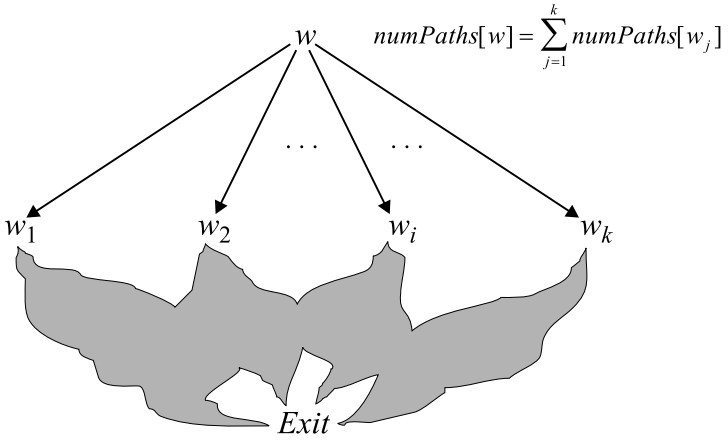
2. Each edge e in the control-flow graph is labeled with a value derived from the ***numPaths***$[]$ quantities.

For expository convenience, we will describe these two aspects of the numbering scheme as if they are generated during two separate passes over the graph. In practice, the two labeling passes can be combined into a single pass.

In the first labeling pass, vertices are considered in reverse topological order. The base case involves the $Exit$ vertex: It is labeled with 1, which accounts for the path of length 0 from $Exit$ to itself. In general, a vertex w is labeled only after all of its successors $w_1, w_2, \ldots, w_k$ are labeled. When w is considered, ***numPaths***$[w]$ is computed using the following equation:

$$\textbf{\textit{numPaths}}[w] = \sum_{i=1}^{k} \textbf{\textit{numPaths}}[w_i]. \tag{1}$$

This equation is illustrated in the following diagram:

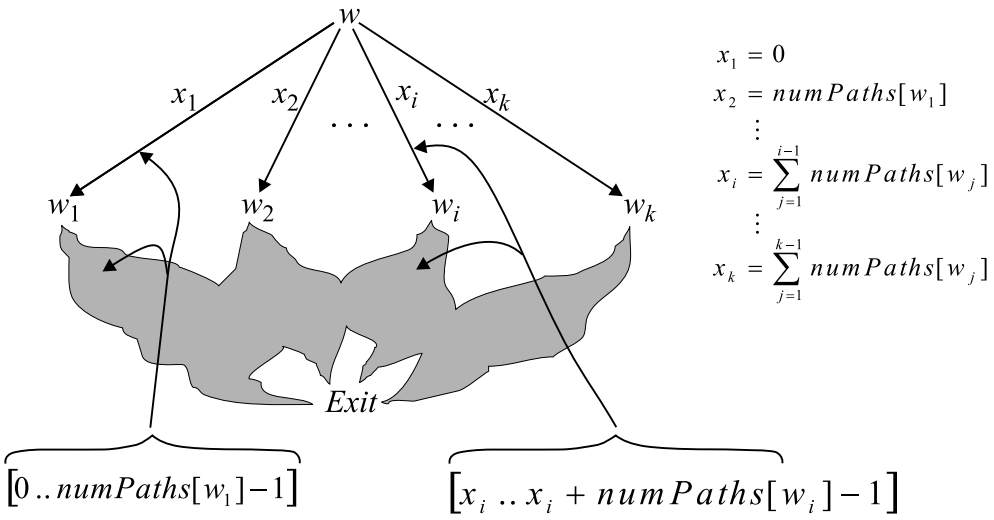


The goal of the second labeling pass is to arrive at a numbering scheme for which, for every path from $Entry$ to $Exit$, the sum of the edge labels along the path corresponds to a ***unique*** number in the range $[0..$***numPaths***$[Entry] - 1]$. That is, we want the following properties to hold:

1. Every path from $Entry$ to $Exit$ is to correspond to a number in the range $[0..$***numPaths***$[Entry] - 1]$.

2. Every number in the range $[0..\mathit{numPaths}[\mathit{Entry}]-1]$ is to correspond to some path from Entry to Exit.

Again, the graph is considered in reverse topological order. The general situation is shown below:



At this stage, we may assume that all edges along paths from each successor of w, say w_i, to Exit have been labeled with values such that the sum of the edge labels along each path corresponds to a unique number in the range $[0..\mathit{numPaths}[w_i]-1]$. Therefore, our goal is to attach a number x_i on edge $w \to w_i$ that, when added to numbers in the range $[0..\mathit{numPaths}[w_i]-1]$, distinguishes the paths of the form $w \to w_i \to \ldots \to \mathit{Exit}$ from all paths from w to Exit that begin with a different edge out of w.

This goal can be achieved by generating numbers $x_1, x_2, \ldots, x_k$ in the manner indicated in the above diagram: The number x_i is set to the sum of the number of paths to Exit from all successors of w that are to the left of w_i:

$$x_i = \sum_{j<i} \mathit{numPaths}[w_j]. \tag{2}$$

This "reserves" the range $[x_i..x_i + \mathit{numPaths}[w_i]-1]$ for the paths of the form $w \to w_i \to \ldots \to \mathit{Exit}$. The sum of the edge labels along each path from w to Exit that begins with an edge $w \to w_j$, where $j < i$, will be a number strictly less than x_i. The sum of the edge labels along each path from w to Exit that begins with an edge $w \to w_m$, where $m > i$, will be a number strictly greater than $x_i + \mathit{numPaths}[w_i] - 1$.

In some cases, the number of paths in the acyclic control flow graph is too great to fit in a single machine word (*e.g.*, $\mathit{numPaths}[\mathit{Entry}] > 2^{32}$), which can make the profiling instrumentation inefficient. In this case, an edge $u \to v$ is chosen from the graph and replaced by the edges $\mathit{Entry} \to v$ and $u \to \mathit{Exit}$ (as was done for backedges); then the labelling passes are rerun. This process is repeated until $\mathit{numPaths}[\mathit{Entry}]$ is less than or equal to the maximum value that will fit in a machine word. Each edge removed from the control-flow graph (including backedges) is called a *recording* edge.

The final step is to instrument the program, which involves introducing a counter variable and appropriate increment statements to accumulate the sum of the edge labels as the program executes along a path. Recording edges are also instrumented with code to record the current observable path and reset the counter variable to begin observing a new path.

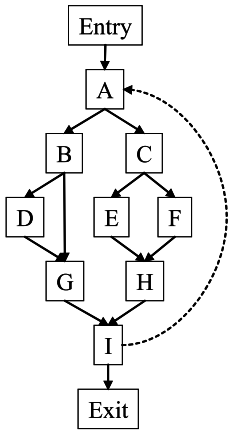


Figure 3: Example control-flow graph.

Several additional techniques are employed to reduce the runtime overheads incurred. These exploit the fact that there is actually a certain amount of flexibility in the placement of the increment statements [10, 12].

2.2 Improving Data-flow Analysis with Path Profiles

We now summarize an intraprocedural version of the *express-lane transformation* investigated by Ammons and Larus in [5]. The express-lane transformation transforms a program to isolate frequently executed paths in the hope that better data-flow facts will results along the isolated paths. Their technique consists of the following steps, performed on each procedure independently [5]:

1. Identify frequently executed, or *hot*, paths. This is done using the Ball-Larus path-profiling technique [12] summarized in Section 2.1.

2. Perform the express-lane transformation on the procedure's control-flow graph. This step creates a new control-flow graph, called the *hot-path graph* in which each hot path has been duplicated.

3. Perform data-flow analysis on the hot-path graph.

4. Reduce the hot-path graph while preserving valuable data-flow solutions. This step prevents unnecessary code growth.

5. Translate the original path profile to a path profile for the reduced hot-path graph. This step allows the profiling information to be used by subsequent compiler phases.

Section 2.2.1 elaborates on steps 2 and 5, while Section 2.2.2 discusses step 4. We will use a running example based on the control flow graph shown in Figure 3 and the profile shown in Table 1.

2.2.1 Constructing the Hot-path Graph

In [5], Ammons and Larus construct the hot-path graph by taking the cross-product of the control-flow graph G and the *hot-path automaton* A. The hot-path automaton is a deterministic finite automaton

Path	Frequency
$Entry \rightarrow A \rightarrow B \rightarrow E \rightarrow I$	15
$A \rightarrow C \rightarrow E \rightarrow H \rightarrow I$	30
$A \rightarrow C \rightarrow F \rightarrow I \rightarrow Exit$	15

Table 1: Example path profile for Figure 3.

(DFA) that recognizes hot-paths in a string of control-flow edges (from G). (For purposes of the construction, the control-flow graph G is considered to be a DFA that recognizes valid execution sequences in a string of control-flow edges: each control-flow edge $u \rightarrow v$ represents a transition from state u to state v when the control-flow edge $u \rightarrow v$ is seen in the input.) The hot-path automaton has two important properties: (1) there are only a few states that are the target of more than one transition; and (2) there is a unique state for every prefix of a hot path. These properties can be used to show that hot-path graph has the desired express-lane versions of the hot paths.

Each vertex $[v, q]$ of the hot-path graph H encodes a vertex v from the original control-flow graph G and a state q from the automaton A; the execution semantics of $[v, q]$ are taken from v. For H to contain an edge $[v, q] \rightarrow [v', q']$, it must be the case that there is an edge $v \rightarrow v'$ in G and a transition $(q, v \rightarrow v', q')$ in A; here, the notation $(q, v \rightarrow v', q')$ indicates a transition from state q to state q' labeled with the control-flow edge $v \rightarrow v'$. There is a path

$$[u, q_0] \rightarrow [v, q_1] \rightarrow [w, q_2] \ldots$$

in H iff there is a path

$$u \rightarrow v \rightarrow w \ldots$$

in G and a path

$$q_0 \rightarrow q_1 \rightarrow q_2 \ldots$$

in A where the transition from q_0 to q_1 is labeled "$u \rightarrow v$", the transition from q_1 to q_2 is labeled "$v \rightarrow w$", etc.

Ammons and Larus make use of Holley and Rosen's data-flow tracing technique to create H [37]. This technique is an efficient way of computing the cross-product of A and G that avoids creating vertices $[v, q]$ that are not reachable from $[Entry, root]$, where $Entry$ is the entry vertex of G, *root* is the start state of A, and $[Entry, root]$ is the entry vertex of the new graph, H. Figure 5 shows the data-flow tracing algorithm used in [5].

Ammons and Larus construct the hot-path automaton A using a modified version of the Aho-Corasick algorithm for matching keywords in a string [2]. In this case, the keywords are hot paths (from a Ball-Larus path profile), and, as mentioned above, the automaton works over an alphabet of edges from the control-flow graph together with the special symbol $\bullet$ that matches any recording edge. Hot-paths that do not begin at $Entry$ are prefixed with $\bullet$ to indicate that they may only be reached after traversing a recording edge.

The Aho-Corasick algorithm constructs a *trie*, or retrieval tree, from the set of key words (or hot paths in this case). Assuming that all paths in Table 1 are hot, Figure 4 shows the corresponding trie.[1]

[1]The trie (and the hot-path graph) that Ammons and Larus would construct for our running example is slightly different from those that we have shown. This is because they require that every edge from $Entry$ and every edge that targets $Exit$ be a recording edge. This makes sense in the intraprocedural case when there are no paths that may contain $Entry$ or $Exit$ in the middle of the path; we avoid this convention because it does not make sense when we begin discussing the interprocedural express-lane transformation.

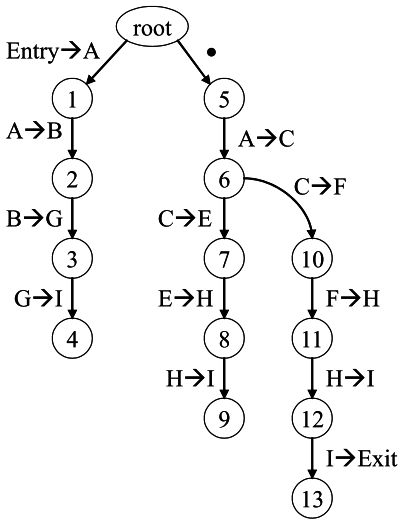


Figure 4: Hot-path trie for the path profile shown in Table 1.

Path in CFG	Path in hot-path graph
$Entry \rightarrow A \rightarrow B \rightarrow E \rightarrow E \rightarrow I$	$[Entry, root] \rightarrow [A, 1] \rightarrow [B, 2] \rightarrow [E, 3]$
$A \rightarrow C \rightarrow E \rightarrow H \rightarrow I$	$[A, 5] \rightarrow [C, 6] \rightarrow [E, 7] \rightarrow [H, 8] \rightarrow [I, 9]$
$A \rightarrow C \rightarrow F \rightarrow H \rightarrow I \rightarrow Exit$	$[A, 5] \rightarrow [C, 6] \rightarrow [F, 10] \rightarrow [H, 11] \rightarrow [I, 12] \rightarrow [Exit, 13]$

Table 2: Paths for Figure 1 translated to the hot-path graph in Figure 6.

The trie in Figure 6 shows the structure of the hot paths that must be duplicated; this gives intuition as to why the automaton is useful in performing the express-lane transformation. In Aho-Corasick, as each symbol of the input string is read, an appropriate transition is made in the trie. Specifically, the next letter of the input string is compared to the labels on outgoing edges of the current state; if a match is found, then the matching edge is followed. If no matching edge for the next letter, e, and current state, q, is found, then a *failure function*, $h(q, e)$ is consulted.

Ammons and Larus show that in the case of the hot-path trie, the failure function is trivial [5]. Regardless of the current state, if e is not a recording edge, then the failure function returns the root state, q_{root}. If e is a recording edge, then the failure function returns $q_\bullet$, where $q_\bullet$ is defined to be the target of the $\bullet$ edge from q_{root}. In Figure 4, $q_\bullet = q_5$. Together, the edges of the trie and the failure function define the transition function of the hot-path automaton: there is a state in the automaton for each state in the trie; there is a transition (q_i, e, q_j) in the automaton if there is an edge $q_i \rightarrow q_j$ in the trie labeled e or the failure function is defined so that $h(q_i, e) = q_j$. Hence forth, we will use the terms "hot-path trie" and "hot-path automaton" interchangeably.

The hot-path automaton has the important property that only two states, q_{root} and $q_\bullet$, are the target of more than one transition. (The states q_{root} and $q_\bullet$ may be the target of multiple transitions because the range of the failure function h is $\{q_{root}, q_\bullet\}$.) It follows that in the hot-path graph, hot paths are only entered from the beginning. Figure 6 shows the hot-path graph for the running example.

The algorithm in Figure 5 also identifies recording edges in the hot-path graph: an edge $[v, q] \rightarrow [v', q']$ is a recording edge in the hot-path graph iff $v \rightarrow v'$ is a recording edge in the original control flow graph. Given this set of recording edges, it is possible to translate the path profile for the original graph into a

$G \equiv (V, E)$ is the original (possibly cyclic) control flow graph.
A is the hot-path automaton.
Q is the set of states of A.
q_{root} is the start state of A.
T is the set of transitions in A.
$R \subseteq E$ is the set of recording edges.
W is a worklist of pairs $[v, q]$, where $v \in V$ and $q \in Q$.
$G_A = (V_A, E_A)$ is the hot path graph.
$R_A \subseteq E_A$ is the new set of recording edges in G.

$V = \{[Entry, q_{root}]\}$
$E_A = \emptyset$
$R_A = \emptyset$
$W = [Entry, q_{root}]$
While $W \neq \emptyset$
    $[v, q] = Take(W)$
    ForeachEdge $v \rightarrow v' \in E$
    $(q, v \rightarrow v', q') \in T$ /* The unique transition for $v \rightarrow v'$ */
    If $[v', q'] \notin V_A$
        $Put(W, [v', q'])$
        $V_A = V_A \cup [v', q']$
    $E_A = E_A \cup \{[v, q] \rightarrow [v', q']\}$
    If $v \rightarrow v' \in R$
        $R_A = R_A \cup \{[v, q] \rightarrow [v', q']\}$

Figure 5: Algorithm for creating the hot-path graph from a control-flow graph G and an deterministic, finite automaton A that recognizes hot-paths in G (see [5, 37]).

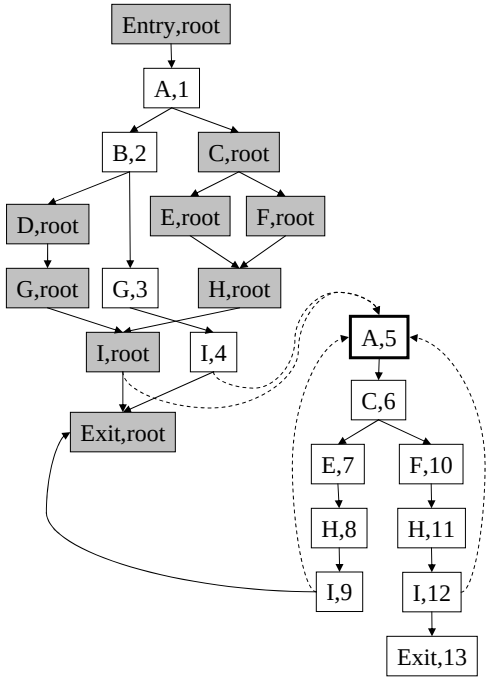


Figure 6: The hot-path graph constructed by the hot-path tracing algorithm (see Figure 5) for the control-flow graph in Figure 3 and the hot-path automaton in Figure 4. Dashed lines represent recording edges. Shaded boxes indicate vertices where the automaton is in the q_{root} state; with the exception of $[Entry, q_{root}]$, these are vertices that do not occur on an express-lane. The vertex $[A, q_5]$ is the only vertex where the automaton is in the state $q_\bullet$.

path profile for the hot-path graph. Translating an individual path is done in two steps:

1. Find the corresponding starting point in the hot-path graph. For a path starting at $Entry$, this is $[Entry, root]$. For a path starting at $v \neq Entry$, this is $[v, q_\bullet]$.

2. Inductively trace out the path: when the hot path follows an edge $v \to w$, the translated path follows an edge $[v, q] \to [w, r]$.

[5] proves that this translation produces the correct path profile for the hot-path graph. Table 2 shows the translated paths for the profile in Table 1.

2.2.2 Reducing the Hot-path Graph

A forward data-flow analysis may get better solutions on the hot-path graph than on the original control flow graph: for a vertex v of the original control flow graph and a vertex $[v, q]$ of the hot-path graph, we have $I(v) \sqsubseteq J([v, q])$, where I and J are the greatest fix-point solutions for data-flow analysis for the respective graphs [37, 5]. When $I(v) \sqsubset J([v, q])$, duplication of a hot-path has been beneficial.

However, in some cases, the code duplication may be redundant: for example, the hot-path graph may contain duplicate vertices that have the same data-flow solution. When $J([v, q]) = J([v, q'])$, it is desirable to collapse $[v, q]$ and $[v, q']$ to a single vertex and avoid unnecessary code growth. If $[v, q]$ has a high execution frequency (*i.e.,* it is *hot*) and $[v, q']$ has a low execution frequency (*i.e.,* it is *cold*), then it may be desirable to combine them even when $[v, q]$ has weaker facts than $[v, q']$ — *i.e.,* $J([v, q]) \sqsubseteq J([v, q'])$. Care must be taken that collapsing vertices $[v, q]$ and $[v, q']$ does not invalidate a desirable data-flow fact at a third vertex $[w, q'']$.

Ammons and Larus offer a technique for reducing the hot-path graph that is based on an algorithm for minimization of deterministic, finite automata (DFAs). Their algorithm consists of the following steps [5]:

1. Identify the hot vertices in the hot-path graph; we wish to preserve the data-flow facts used in these vertices. The frequency of execution for each vertex can be determined from the path profile for the hot-path graph. Ammons and Larus sort the vertices by the number of desirable data-flow facts (*e.g.,* that the variable x has constant value c) they "execute" dynamically; here, "executing" a data-flow fact means performing an operation that is described by the data-flow fact, *e.g.,* executing an operation that uses the variable x when x is known to have a constant value. Vertices are marked hot until a fixed percentage — 95% in their experiments — of the desirable data-flow facts are covered.

2. Partition the vertices of the hot-path graph into sets of *compatible* vertices; call this partition Π. Two vertices $[v, q]$ and $[v', q']$ are compatible if and only if $v = v'$ and combining the vertices does not destroy desirable data-flow fact in a hot vertex, *i.e.,* iff:

   - if $[v, q]$ is hot, then $J([v, q]) \sqsubseteq J([v', q'])$ for the data-flow facts used in v.
   - if $[v', q']$ is hot, then $J([v', q']) \sqsubseteq J([v, q])$ for the data-flow facts used in v'.

   For example, suppose the results of constant propagation for a hot vertex $[v, q]$ show that $x = 3$. If v contains a use of x, then $[v, q]$ is compatible with $[v, q']$ iff constant propagation gives $x = 3$ for $[v, q']$ or $[v, q']$ is cold and constant propagation gives $x \sqsupseteq 3$ for $[v, q']$. (Note, if $x = \top$ for $[v, q']$, then x is uninitialized, and it is safe to lower the data-flow fact $x = \top$ to the data-flow

fact $x = 3$; this is because any particular constant (*e.g.*, 3) is a possible value for an uninitialized variable (*e.g.*, x).) This definition of compatible is not transitive, and hence does not define an equivalence relation. [5] creates the partition by greedily adding each vertex $[v, q]$ to an existing set when possible.

3. Run the DFA-minimization algorithm on Π to create a new partition Π' [30]. As stated in [5], the hot-path graph can be considered to be a finite automaton with transitions labeled by the edges of the original control-flow graph. With this intuition, each set in Π is the set of final states that recognize paths ("words" in the automaton) that lead to a certain set of data-flow facts. The DFA-minimization results in a more fine-grained partition Π' such that for each pair of vertices v and v' in a block B of Π', for any "string" s of CFG-edges that drives v to a vertex w in block C, the string s drives v' to a vertex w' that is also in block C. It follows that coalescing vertices that are in the same block of Π' will not destroy any valuable data-flow facts at any vertex.

4. Replace each block B of Π' with a representative r; the data-flow solution for r is set to the meet of the data-flow solutions for the vertices in B. Let r and r' represent blocks B and B', repsectively: then there is an edge $r \to r'$ in the reduced graph iff there is an edge $[v, q] \to [v', q']$ in the hot-path graph where $[v, q] \in B$ and $[v', q'] \in B'$. The edge $r \to r'$ is a recording edge iff $v \to v'$. ([5] shows that this is well-defined.)

The path profile can be translated to the reduced hot-path graph just as it was for the hot-path graph.

Chapter 3

The Functional Approach to Interprocedural, Context Path Profiling

In path profiling, a program is instrumented with code that counts the number of times particular finite-length path fragments of the program's control-flow graph — or *observable paths* — are executed. A path profile for a given run of a program consists of a count of how often each observable path was executed. This chapter presents extensions of the *intra*procedural path-profiling technique of Ball and Larus [12]. We show that the approach used in the Ball-Larus technique can be used as a strategy for creating other path-profiling techniques. Their strategy can be summarized as follows:

1. Start with a graph that represents a program's control flow.
2. Apply a transformation that results in a new graph with a finite number of paths; each path through the transformed graph corresponds to a path fragment, called an *observable path*, in the original graph.
3. Number the paths in the transformed graph.
4. Instrument the program with code that counts how often each observable path is executed (by incrementing a counter associated with the path through the transformed graph).

All of our extensions to the Ball-Larus technique employ this strategy. In particular, we have used this approach to develop path-profiling techniques for collecting information about *inter*procedural paths (i.e., paths that may cross procedure boundaries).

Interprocedural path profiling is complicated by the need to account for a procedure's calling context. There are really two issues:

- *What is meant by a procedure's "calling context"?* Previous work by Ammons et al. [4] investigated a hybrid intra-/interprocedural scheme that collects separate *intra*procedural profiles for a procedure's different calling contexts. In their work, the "calling context" of procedure P consists of the *sequence of call sites* pending on entry to P. In general, the sequence of pending call sites is an abstraction of any of the paths ending at the call on P.

  The path-profiling techniques we have developed profile true *inter*procedural paths, which may include call and return edges between procedures, paths through pending procedures, and paths through procedures that were called in the past and completed execution. This means that, in general, our techniques maintain finer distinctions than those maintained by the profiling technique of Ammons et al.

- *How does the calling-context problem impact the profiling machinery?* In our *functional* approach, the numbering of observable paths is carried out via an edge-labeling scheme that is in much the same spirit as the path-numbering scheme of the Ball-Larus technique, where each edge

is labeled with a number, and the "name" of a path is the sum of the numbers on the path's edges. However, to handle the calling-context problem, in our methods edges are labeled with *functions* instead of *values*. In effect, the use of edge-functions allows edges to be numbered differently depending on the calling context.[1]

We have also developed non-functional (*i.e.*, value-based) techniques for interprocedural path profiling. In this approach, edges are labeled with values, as in the Ball-Larus technique. These techniques may be more efficient than the functional techniques, however, they also result in coarser profiles.

All of our techniques can be divided into two categories, which we call *context path profiling* and *piecewise path profiling*. In piecewise path profiling, each observable path corresponds to a path that may occur as a subpath (or piece) of an execution sequence. The set of observable paths is required to cover every possible execution sequence. That is, every possible execution sequence can be partitioned into a sequence of observable paths. In context path profiling, each observable path corresponds to a pair $\langle C, p \rangle$, where p corresponds to a subpath of an execution sequence, and C corresponds to a context (*e.g.*, a sequence of pending calls) in which p may occur. The set of all p such that $\langle C, p \rangle$ is an observable path must cover every possible execution sequence.

In addition to several interprocedural path-profiling mechanisms, we have also developed several extensions to the Ball-Larus technique that yield novel intraprocedural path-profiling techniques. In particular, we have developed a method for intraprocedural context path-profiling where the context of an observable path may summarize the path taken to a loop header. It is also possible to use a functional approach to intraprocedural path-profiling to unroll loops in a virtual manner for profiling purposes; one possible application of this technique might be to generate profiles that distinguish between the even and odd iterations of a loop without actually unrolling the loop.

All of the path-profiling techniques presented in this thesis can be classified according to three binary traits:

1. functional approach vs. non-functional approach
2. intraprocedural vs interprocedural
3. context vs. piecewise

In the remainder of this chapter, we present the functional approach to interprocedural, context path profiling. In Chapter 4, we present the functional approach to interprocedural, piecewise path profiling. Chapter 5 discusses other novel path profiling techniques.

In the functional approach to interprocedural context path-profiling, the "naming" of paths is carried out via an edge-labeling scheme that is in much the same spirit as the path-naming scheme of the Ball-Larus technique, where each edge is labeled with a number, and the "name" of a path is the sum of the numbers on the path's edges. However, in a functional approach to path profiling, edges are labeled with *functions* instead of *values*. In effect, the use of edge-functions allows edges to be numbered differently depending on context information. At runtime, as each edge e is traversed, the profiling machinery uses the edge function associated with e to compute a value that is added to the quantity `pathNum`. At the appropriate program points, the profile is updated with the value of `pathNum`.

Because edge functions are always of a particularly simple form (i.e., linear functions), they do not complicate the runtime-instrumentation code greatly:

[1] Note that the "functional" approach to path profiling does not use a functional programming style; it is so named because it labels edges with linear functions.

- The Ball-Larus instrumentation code performs 0 or 1 additions in each basic block; a hash-table lookup and 1 addition for each control-flow-graph backedge; 1 assignment for each procedure call; and a hash-table lookup and 1 addition for each return from a procedure.

- The technique presented here performs 0 or 2 additions in each basic block; a hash-table lookup, 1 multiplication, and 4 additions for each control-flow-graph backedge; 2 multiplications and 2 additions for each procedure call; and 1 multiplication and 1 addition for each return from a procedure.

Thus, while using functions on each edge involves more overhead than using values on each edge, we originally believed that the overhead from using edge functions would not be prohibitive. The experimental results in Chapter 6 show that the overhead for interprocedural path profiling is 300-700%; this is much more costly than the Ball-Larus technique. A significant amount of this overhead results from the fact that we must do a hash table lookup every time we record a path; the Ball-Larus technique can sometimes use an array access when it records a path.

The specific technical contributions of the work presented in this chapter include:

- In the Ball-Larus scheme, a cycle-elimination transformation of the (in general, cyclic) control-flow graph is introduced for the purpose of numbering paths. We present the interprocedural analog of this transformation for interprocedural context path profiling.

- In the case of intraprocedural path profiling, the Ball-Larus scheme produces a dense numbering of the observable paths within a given procedure: That is, in the transformed (i.e., acyclic) version of the control-flow graph for a procedure P, the sum of the edge labels along each path from P's entry vertex to P's exit vertex falls in the range $[0..\text{number of paths in } P]$, and each number in the range $[0..\text{number of paths in } P]$ corresponds to exactly one such path.

  The interprocedural techniques presented in this chapter produce a dense numbering of interprocedural observable paths. The significance of the dense-numbering property is that it ensures that the numbers manipulated by the instrumentation code have the minimal number of bits possible.

This chapter focuses on the functional approach to interprocedural context path profiling, and, except where noted, the term "interprocedural path profiling" means "the functional approach to interprocedural context path profiling". As mentioned above, the path profiling techniques explored in this thesis have four steps:

1. Start with a graph that represents a program's control flow.

2. Apply a transformation that results in a new graph with a finite number of paths; each path through the transformed graph corresponds to a path fragment, called an *observable path*, in the original graph.

3. Number the paths in the transformed graph.

4. Instrument the program with code that counts how often each observable path is executed (by incrementing a counter associated with the path through the transformed graph).

The remainder of this chapter discusses how these four steps are implemented to collect an interprocedural, context path profile: Section 3.1 presents the interprocedural control-flow graph, or *supergraph*, and defines terminology needed to describe our results. Section 3.2 describes how the supergraph is

transformed into a graph with a finite number of paths. In Sections 3.3–3.5, we describe how to number the paths in the graph defined in Section 3.2. Section 3.6 describes the instrumentation used to collect an interprocedural context path profile. Finally, Section 3.7 discusses how to profile in the presence of some language features (*e.g.*, signals) that are not discussed in earlier sections.

3.1 Background: The Program Supergraph and Call Graph

As in many interprocedural program-analysis problems, we work with an interprocedural control-flow graph called a ***supergraph***. Specifically, a program's supergraph G^* consists of a unique entry vertex $Entry_{global}$, a unique exit vertex $Exit_{global}$, and a collection of control-flow graphs (one for each procedure), one of which represents the program's main procedure. For each procedure P, the flowgraph for P has a unique entry vertex, $Entry_P$, and a unique exit vertex, $Exit_P$. The other vertices of the flowgraph represent statements and predicates of the program in the usual way,[2] except that each procedure call in the program is represented in G^* by two vertices, a *call* vertex and a *return-site* vertex. In addition to the ordinary intraprocedural edges that connect the vertices of the individual control-flow graphs, for each procedure call (represented, say, by call vertex c and return-site vertex r) to procedure P, G^* contains a ***call-edge***, $c \rightarrow Entry_P$, and a ***return-edge***, $Exit_P \rightarrow r$. The supergraph also contains the edges $Entry_{global} \rightarrow Entry_{main}$ and $Exit_{main} \rightarrow Exit_{global}$. An example of a supergraph is shown in Fig. 7(a).

For purposes of profiling, we assume that all branches are logically independent, *i.e.*, the result of one branch does not affect the ability to take any other branch. However, we do not wish to consider paths in G^* that violate the nature of procedure calls (as the path in Fig. 7(b) does). We now develop a language for describing the set of paths in G^* that we wish to consider valid. To do this, for each call site with call vertex c and return-site vertex r, let the call-edge from c be labeled with the symbol "$(_c$", and let the return-edge to r be labeled with the symbol "$)_c$". Let each edge of the form $Entry_{global} \rightarrow Entry_P$ be labeled with the symbol "$(_P$" and each edge of the form $Exit_P \rightarrow Exit_{global}$ be labeled with the symbol "$)_P$".[3] Let all other edges be labeled with the symbol e. Then a path p in G^* is a ***same-level valid path*** if and only if the string formed by concatenating the labels of p's edges is derived from the non-terminal ***SLVP*** in the following context-free grammar:

$$
\begin{array}{rcll}
SLVP & ::= & SLVP\ \ SLVP & \\
 & | & e & \\
 & | & \epsilon & \\
 & | & SLVP_1 & \\
SLVP_1 & ::= & (_c\ \ SLVP\ \ )_c & \text{foreachcallvertex } c \\
 & | & (_P\ \ SLVP\ \ )_P & \text{foreachprocedure } P
\end{array}
$$

Here, ϵ denotes the empty string. A same-level valid path p represents an execution sequence where every call-edge is properly matched with a corresponding return-edge and vice versa. If a path p spells out a word that can be derived from the non-terminal $SLVP_1$, then we call p a SLVP_1-path. A call vertex and its return-site vertex are always connected by a SLVP_1-path.

We also need to describe paths that correspond to incomplete execution sequences in which not all of the procedure calls have been completed. (For example, a path that begins in a procedure P, crosses

[2] The vertices of a flowgraph can represent individual statements and predicates; alternatively, they can represent basic blocks.

[3] For now, the only such edges are $Entry_{global} \rightarrow Entry_{main}$ and $Exit_{main} \rightarrow Exit_{global}$. In Section 3.2, we will introduce edges that connect $Entry_{global}$ and $Exit_{global}$ to other procedures.

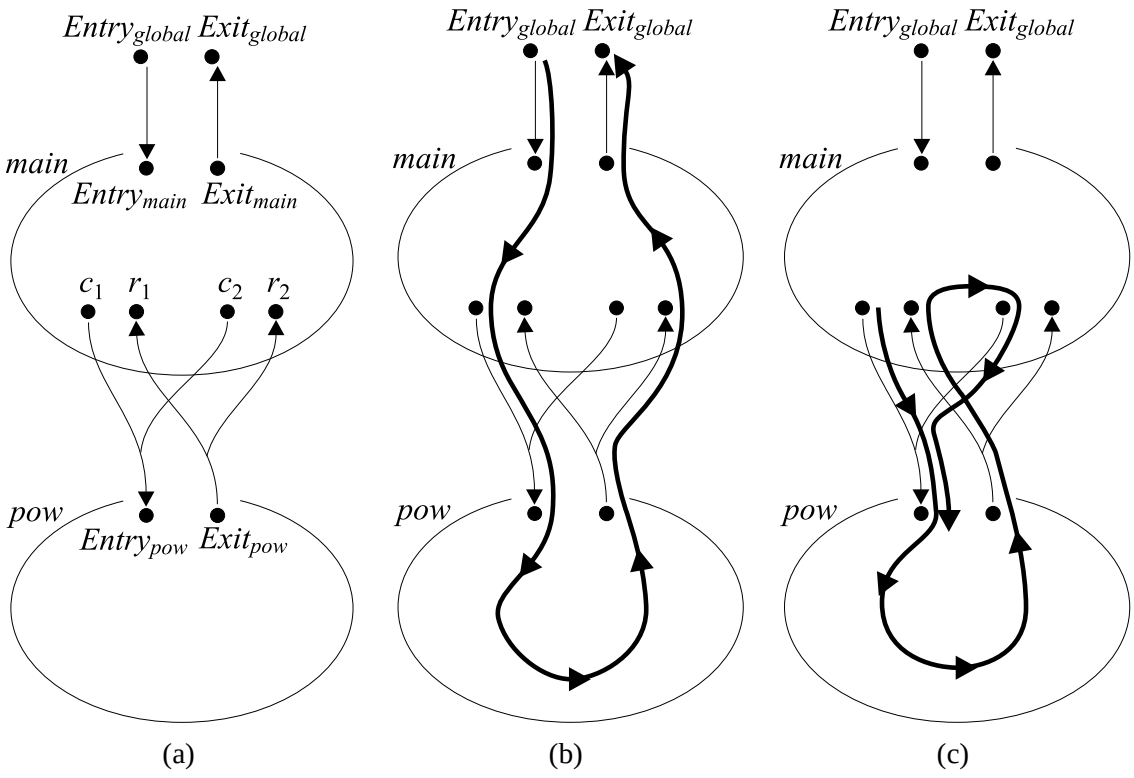


Figure 7: (a) Schematic of the supergraph of a program in which *main* has two call sites on the procedure *pow*. (b) Example of an invalid path in a supergraph. (c) Example of a cycle that may occur in a valid path.

a call-edge to a procedure Q, and ends in Q.) Such a path p is called an *unbalanced-left path*. The string formed by concatenating the labels on p's edges must be derived from the non-terminal *UnbalLeft* in the following context-free grammar:

$$
\begin{array}{lcll}
\textit{UnbalLeft} & ::= & \textit{UnbalLeft}\ (_c\ \textit{UnbalLeft} & \text{for each call vertex } c \\
 & | & \textit{UnbalLeft}\ (_P\ \textit{UnbalLeft} & \text{for each procedure } P \\
 & | & \textit{SLVP} &
\end{array}
$$

where *SLVP* is defined by the productions given above.

In the remainder of this thesis, we will also refer to a program's call graph. We define a call graph to be the graph that contains one node for each procedure P and one edge $P \rightarrow Q$ for each call site in P that calls Q. (In the literature, this is often called the call multi-graph, since there may be more than one edge from a procedure P to procedure Q.)

3.2 Modifying G^* to Eliminate Backedges and Recursive Calls

For purposes of numbering paths, the Ball-Larus technique modifies a procedure's control-flow graph to remove cycles. This section describes the analogous step for interprocedural context profiling. Specifically, this section describes modifications to G^* that remove cycles from each procedure and from the call graph associated with G^*. The resulting graph is called G^*_{fin}. Each unbalanced-left path in G^*_{fin}

from $Entry_{global}$ to $Exit_{global}$ defines an "observable path" that can be logged in an interprocedural profile. The number of unbalanced-left paths in G^*_{fin} is finite, which is the reason for the subscript "*fin*". For the remainder of the thesis, we will use the phrases, "a path in G^*_{fin} from $Entry_{global}$ to $Exit_{global}$" and "a path *through* G^*_{fin}" interchangeably.

In total, there are three transformations that are performed to create G^*_{fin}. Fig. 9 shows the transformed graph G^*_{fin} that is constructed for the example program in Fig. 8 (the labels on the vertices and edges of this graph are explained in Section 3.3).

Transformation 1: For each procedure P, add a special vertex $GExit_P$. In addition, add an edge $GExit_P \rightarrow Exit_{global}$ labeled e.

The second transformation removes cycles in each procedure's flow graph. As in the Ball-Larus technique, the procedure's control-flow graph does not need to be reducible; backedges can be determined by a depth-first search of the control-flow graph.

Transformation 2: For each procedure P, perform the following steps:

1. For each backedge target v in P, add a *surrogate* edge $Entry_P \rightarrow v$ labeled e.
2. For each backedge source w in P, add a *surrogate* edge $w \rightarrow GExit_P$ labeled e.
3. Remove all of P's backedges.

The third transformation "short-circuits" paths around recursive call sites, effectively removing cycles in the call graph. First, each call site is classified as recursive or nonrecursive. This can be done by identifying backedges in the call graph using depth-first search; the call graph need not be reducible. A call-site is considered to be recursive iff it is represented in the call graph by a backedge. Otherwise, a call-site is considered to be non-recursive.

Transformation 3: The following modifications are made:

1. For each procedure R called from a recursive call site, add the edges $Entry_{global} \rightarrow Entry_R$ and $Exit_R \rightarrow Exit_{global}$ labeled "$(_R$" and "$)_R$", respectively.
2. For each pair of vertices c and r representing a recursive call site that calls procedure R, remove the edges $c \rightarrow Entry_R$ and $Exit_R \rightarrow r$, and add the *summary* edge $c \rightarrow r$ labeled e. (Note that $c \rightarrow r$ is called a "summary" edge, but not a "surrogate" edge.)

As was mentioned above, the reason we are interested in these transformations is that each observable path—an item we log in an interprocedural path profile—corresponds to an unbalanced-left path through G^*_{fin}. Note that the observable paths do not correspond to just the same-level valid paths through G^*_{fin}: as a result of Transformation 2, an observable path p may end with $\ldots \rightarrow GExit_P \rightarrow Exit_{global}$, leaving unclosed left parentheses. Also note that a path through G^*_{fin} that is not unbalanced-left cannot represent any feasible execution path in the original graph G^*.

3.2.1 G^*_{fin} has a Finite Number of Paths

A crucial fact on which this approach to interprocedural path profiling rests is that the number of unbalanced-left paths through G^*_{fin} is finite. To prove this we start with the following observation:

Observation 3.2.1 The maximum number of times a vertex v in a procedure P can appear on an unbalanced-left path p (in G^*_{fin}) is equal to the number of times p enters the procedure P by reaching the vertex $Entry_P$. □

```
int main() {
   double t, result = 0.0;
   int i = 1;
   while( i <= 18 ) {
      if( (i%2) == 0 ) {
         t = pow( i, 2 );
         result += t;
      }
      if( (i%3) == 0 ) {
         t = pow( i, 2 );
         result += t;
      }
      i++;
   }
   return 0;
}
```

```
double pow(double base, long exp) {
   double power = 1.0;
   while( exp > 0 ) {
      power *= base;
      exp--;
   }
   return power;
}
```

Figure 8: Example program used to illustrate the path-profiling technique. (The program computes the quantity $(\sum_{j=1}^{9}(2 \cdot j)^2)) + (\sum_{k=1}^{6}(3 \cdot k)^2)$.)

This follows from the fact that G^*_{fin} contains no intraprocedural loops. It also relies on the fact that p is an unbalanced-left path; a path q that is not unbalanced-left may reach some vertices an arbitrary number of times. For example, consider Figure 10. The vertex u lies on a cyclic path that runs from u to the second call-site on procedure Q, enters Q from the second call-site, then returns to the first call-site, and then reaches u; a path may visit u are arbitrary number of times by repeatedly traversing this cycle. However, note that this cycle cannot occur in an unbalanced-left path.

Next, we wish to calculate the number *maxEnters*$[P]$, which is an upper bound on the number of times the vertex $Entry_P$ can occur on an unbalanced-left path. (As we shall see, *maxEnters*$[P]$ is well defined and finite.) We observe that for $P \neq$ *main*, the number of times $Entry_P$ occurs on an unbalanced-left path p is bounded by the maximum number of times a vertex for a call site on P can occur on the path p. Together with Observation 3.2.1 (which applies to call vertices), this implies that for all $P \neq$ *main*,

$$\textit{maxEnters}[P] = \sum_{Q} \textit{maxEnters}[Q] \cdot (\text{number of non-recursive call sites on } P \text{ in } Q).$$

We also have *maxEnters*[*main*] $= 1$. For $P \neq$ *main*, *maxEnters*$[P]$ is well defined and has a finite value because there are no recursive calls in G^*_{fin}; thus, we can solve for the *maxEnters*$[P]$ values by considering the vertices of the call graph associated with G^*_{fin} (which is acyclic) in topological order.

Observation 3.2.1, together with the fact that *maxEnters*$[P]$ is finite for all P, means that for all vertices v, there is a finite bound on the number of times v may occur in an unbalanced-left path p. This implies that there is an upper bound on the length of an unbalanced-left path, and that the number of unbalanced-left paths is finite.

3.3 Numbering Unbalanced-Left Paths: A Motivating Example

In this section, we illustrate, by means of the example shown in Fig. 8, some of the difficulties that arise in collecting an interprocedural path profile. Fig. 7(a) shows a schematic of the supergraph G^* for this program. One difficulty that arises in interprocedural path profiling comes from interprocedural cycles. Even after the transformations described in Section 3.2 are performed (which break intraprocedural cycles and cycles due to recursion), G^* will still contain cyclic paths, namely, those paths that enter a procedure from distinct call sites (see Fig. 7(c)). This complicates any interprocedural extension to the Ball-Larus technique, because the Ball-Larus numbering scheme works on acyclic graphs. There are several possible approaches to overcoming this difficulty:

- One possible approach is to create a unique copy of each procedure for each nonrecursive call site and remove all recursive call and return edges. In our example program, we would create the copies *pow1* and *pow2* of the *pow* function, as shown in Fig. 11. *pow1* can be instrumented as if it had been inlined in *main*, and likewise for *pow2*. In many cases, this approach is impractical because of the resulting code explosion.

- A second approach—which is the one developed in this chapter—is to parameterize the instrumentation in each procedure to behave differently for different calling contexts. In our example, *pow* is changed to take an extra parameter. When *pow* is called from the first call site in *main*, the value of the new parameter causes the instrumentation of *pow* to mimic the behavior of the instrumentation of *pow1* in the first approach above; when *pow* is called from the second call site in *main*, the value of the new parameter causes *pow*'s instrumentation to mimic the behavior of the instrumentation of *pow2*. Thus, by means of an appropriate parameterization, we gain the advantages of the first approach without duplicating code.

The remainder of this section gives a high-level description of our path-numbering technique. Section 3.5 gives a detailed description of the path-numbering technique.

Extending the Ball-Larus technique to number unbalanced-left paths in G^*_{fin} is complicated by the following facts:

1. As mentioned above, while the number of unbalanced-left paths is finite, an unbalanced-left path may contain cycles (such as those in Fig. 7(c)).

2. The number of paths that may be taken from a vertex v is dependent on the path taken to reach v: for a given path p to vertex v, not every path q from v forms an unbalanced-left path when concatenated with p.

These facts mean that it is not possible to assign a single integer value to each vertex and edge of G^*_{fin} as the Ball-Larus technique does. Instead, each occurrence of an edge e in a path p will contribute a value to the path number of p, but the value that an occurrence of e contributes will be dependent on the part of p that precedes that occurrence of e. In particular, e's contribution is determined by the sequence of unmatched left parentheses that precede the occurrence of e in p. (The sequence of unmatched left parentheses represents a calling context of the procedure containing e.)

Consider the example shown in Figs. 8 and 9. Notice that G^*_{fin} in Fig. 9 contains cyclic, unbalanced-left paths. For example, the following path is a cycle from u_1 to u_1 that may appear as a subpath of an unbalanced-left path:

$$u_1 \rightarrow u_3 \rightarrow u_7 \rightarrow u_8 \rightarrow v_7 \rightarrow v_8 \rightarrow v_9 \rightarrow v_{10} \rightarrow u_1.$$

Fig. 11 shows a modified version of G^*_{fin} with two copies of the procedure *pow*, one for each call site to *pow* in *main*. This modified graph is acyclic and therefore amenable to the Ball-Larus numbering scheme: Each vertex v in Fig. 11 is labeled with ***numPaths***$[v]$, the number of paths from v to $Exit_{global}$; each edge e is labeled with its Ball-Larus increment [12]. Note that there is a one-to-one and onto mapping between the paths through the graph in Fig. 11 and the unbalanced-left paths through the graph in Fig. 9. This correspondence can be used to number the unbalanced-left paths in Fig. 9: each unbalanced-left path p in Fig. 9 is assigned the path number of the corresponding path q in Fig. 11.

The following two observations capture the essence of our technique:

- Because the labeling passes of the Ball-Larus scheme work in reverse topological order, the values assigned to the vertices and edges of a procedure are dependent upon the values assigned to the exit vertices of the procedure. For instance, in Fig. 11, the values assigned to the vertices and edges of *pow1* are determined by the values assigned to $Exit_{pow1}$ and $GExit_{pow1}$ (*i.e.*, the values 5 and 1, respectively), while the values assigned to the vertices and edges of *pow2* are determined by the values assigned to $Exit_{pow2}$ and $GExit_{pow2}$ (*i.e.*, the values 1 and 1, respectively). Note that ***numPaths***$[GExit_P] = 1$ for any procedure P (since the only path from $GExit_P$ to $Exit_{global}$ is the path consisting of the edge $GExit_P \rightarrow Exit_{global}$). Thus, the values on the edges and the vertices of *pow1* differ from some of the values on the corresponding edges and vertices of *pow2* because ***numPaths***$[Exit_{pow1}] \neq$ ***numPaths***$[Exit_{pow2}]$.

- Given that a program transformation based on duplicating procedures is undesirable, a mechanism is needed that assigns vertices and edges different numbers depending on the calling context. To accomplish this, each vertex u of each procedure P is assigned a linear function ψ_u that, when given a value for ***numPaths***$[Exit_P]$, returns the value of ***numPaths***$[u]$. Similarly, each edge e of each procedure P is assigned a linear function ρ_e that, when given a value for ***numPaths***$[Exit_P]$, returns the Ball-Larus value for e.

Fig. 9 shows G^*_{fin} labeled with the appropriate ψ and ρ functions. Note that we have the desired correspondence between the linear functions in Fig. 9 and the integer values in Fig. 11. For example, in Fig. 9 vertex u_1 has the function $\psi_{u_1} = \lambda x.2 \cdot x + 2$. This function, when supplied with the value ***numPaths***$[Exit_{pow1}] = 5$ from Fig. 11 evaluates to 12, which is equal to ***numPaths***$[u'_1]$ in Fig. 11. However, when $\lambda x.2 \cdot x + 2$ is given the value ***numPaths***$[Exit_{pow2}] = 1$, it evaluates to 4, which is equal to ***numPaths***$[u''_1]$ in Fig. 11.

To collect the number associated with an unbalanced-left path p in G^*_{fin}, as p is traversed, each edge e contributes a value to p's path number. As illustrated below, the value that e contributes is dependent on the path taken to e:

Example 3.3.1 Consider the edge $u_1 \rightarrow u_3$ in G^*_{fin}, and an unbalanced-left path s that begins with the following path prefix:

$$Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow u_1 \rightarrow u_3 \tag{3}$$

In this case, the edge $u_1 \rightarrow u_3$ contributes a value of 6 to s's path number. To see that this is the correct value, consider the path prefix in Fig. 11 that corresponds to (3):

$$Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow u'_1 \rightarrow u'_3$$

In Fig. 11, the value on the edge $u'_1 \rightarrow u'_3$ is 6.

In contrast, in an unbalanced-left path t that begins with the path prefix

$$Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_9 \rightarrow v_{10} \rightarrow u_1 \rightarrow u_3 \tag{4}$$

the edge $u_1 \rightarrow u_3$ will contribute a value of 2 to t's path number. (To see that this is the correct value, consider the path prefix in Fig. 11 that corresponds to (4).)

It can even be the case that an edge e occurs more than once in a path p, with each occurrence contributing a different value to p's path number. For example, there are some unbalanced-left paths in G^*_{fin} in which the edge $u_1 \rightarrow u_3$ appears twice, contributing a value of 6 for the first occurrence and a value of 2 for the second occurrence.

To determine the value that an occurrence of the edge e should contribute to a path number, the profiling instrumentation will use the function ρ_e and the appropriate value for ***numPaths***$[Exit_P]$, where P is the procedure containing e. Thus, as noted above, an occurrence of the edge $u_1 \rightarrow u_3$ may contribute the value $(\lambda x.x + 1)(1) = 2$ or the value $(\lambda x.x + 1)(5) = 6$ to a path number, depending on the path prior to the occurrence of $u_1 \rightarrow u_3$. □

Figs. 12 and 13 show the program from Fig. 8 with additional instrumentation code — based on the linear functions in Fig. 9 — that collects an interprocedural path profile. The output from the instrumented program is as follows:

```
 0: 0    1: 0    2: 0    3: 0    4: 0    5: 0    6: 0    7: 0    8: 0
 9: 0   10: 0   11: 0   12: 0   13: 0   14: 0   15: 0   16: 1   17: 0
18: 9   19: 0   20: 0   21: 0   22: 0   23: 0   24: 9   25: 3   26: 0
27: 3   28: 3   29: 6   30: 3   31: 0   32: 3   33: 3   34: 5   35: 1
```

Section 3.5 presents an algorithm that assigns linear functions to the vertices and edges of G^*_{fin} directly, without referring to a modified version of G^*_{fin}, like the one shown in Fig. 11, in which procedures are duplicated.

3.3.1 What Do You Learn From a Profile of Unbalanced-Left Paths?

Before examining the details of interprocedural path profiling, it is useful to understand the information that is gathered in this approach:

- Each unbalanced-left path p through G^*_{fin} from $Entry_{global}$ to $Exit_{global}$ can be thought of as consisting of a *context-prefix* and an *active-suffix*. The active-suffix q'' of p is a maximal-size, surrogate-free subpath at the tail of p (though the active-suffix may contain summary edges of the form $c \rightarrow r$, where c and r represent a recursive call site). The context-prefix q' of p is the prefix of p that ends at the last surrogate edge before p's active suffix. (The context-prefix q' can be the empty path from $Entry_{global}$ to $Entry_{global}$.)

- The counter associated with the unbalanced-left path p counts the number of times during a program's execution that the active-suffix of p occurs in the context summarized by p's context-prefix.

Example 3.3.2 Consider the path in Figure 9 with path number 24 (shown in bold):

$$24 : Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow u_1 \rightarrow u_3 \rightarrow u_4 \rightarrow u_5 \rightarrow u_6 \rightarrow Exit_{global}$$

This path consists of the context-prefix $Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow u_1$ and the active-suffix $u_3 \rightarrow u_4 \rightarrow u_5$. The output obtained from running the program shown in Figs. 12 and 13 indicates that the active suffix was executed 9 times in the context summarized by the context-prefix. Note that the context-prefix not only summarizes the call site in *main* from which *pow* was called, but also the path within *main* that led to that call site. In general, a context-prefix (in an interprocedural technique) summarizes not only a sequence of procedure calls (*i.e.*, the calling context), but also the intraprocedural paths taken within each procedure in the sequence. □

3.4 Numbering L-Paths in a Finite-Path Graph

The Ball-Larus path-numbering technique applies to directed acyclic graphs (DAGs). In this section, we discuss how to generalize the Ball-Larus technique to apply to a *finite-path graphs*. Since G^*_{fin} is an example of a finite-path graph, this gives us a firm theoretical foundation to justify our technique for numbering unbalanced-left paths in G^*_{fin}. A finite-path graph is defined as follows:

Definition 3.4.1 Let *CF* be a context-free grammar over an alphabet Σ. Let G be a directed graph whose edges are labeled with members of Σ. Let G have a unique entry vertex, $Entry$, and a unique exit vertex, $Exit$. Each path in G defines a word over Σ, namely, the word obtained by concatenating, in order, the labels of the edges on the path. A path in G is an *L-path* if its word is in the language L defined by *CF*. The graph G and the context-free grammar *CF* constitute a *finite-path graph* if and only if the number of L-paths in G from $Entry$ to $Exit$ is finite.[4] □

In Section 3.2, we showed that the number of unbalanced-left paths through G^*_{fin} is finite. Thus, as mentioned above, the graph G^*_{fin}, together with the context-free grammar for unbalanced-left strings, constitutes a finite-path graph where $Entry$ is the vertex $Entry_{global}$ and $Exit$ is the vertex $Exit_{global}$. Note that a finite-path graph need not be acyclic, and hence might not be a DAG; however, just as the number of paths through a DAG is finite, the number of L-paths through a finite-path graph is finite.

We are now ready to describe a mechanism for numbering L-paths in a finite-path graph.[5] The numbering of L-paths in a finite-path graph is necessarily more complex than assigning a single integer to each vertex and edge, as is done in the Ball-Larus technique (because a finite-path graph may contain cycles). Nevertheless, a number of comparisons can be made between our technique for numbering L-paths and the Ball-Larus technique for numbering paths in an acyclic graph. In the remainder of this section, we describe the functions *numValidComps* and *edgeValueInContext* that correspond, in a sense, to the vertex and edge values of the Ball-Larus technique. (These functions are used in Section 3.5.1 to give a theoretical justification for the ψ and ρ functions that were introduced in the previous section for numbering unbalanced-left paths (*i.e.*, L-paths) in G^*_{fin} (a finite-path graph). However, in another sense, in the interprocedural path-profiling techniques it is the ψ and ρ functions that correspond to the vertex and edge values of the Ball-Larus technique, in that the ψ and ρ functions are employed at runtime to calculate the edge-increment values used by the profiling instrumentation code, whereas *numValidComps* and *edgeValueInContext* are only referred to in order to argue the correctness of the interprocedural profiling techniques.)

The following list describes aspects of the Ball-Larus technique, and the corresponding aspect of our technique for numbering L-paths:

1. In the Ball-Larus technique, each vertex v is labeled with the number *numPaths*$[v]$ of paths from v to $Exit$. In our technique, it is necessary to define a function *numValidComps* that takes an L-path prefix p from $Entry$ to a vertex v and returns the number of paths from v to $Exit$ that form an L-path when concatenated with p. Thus, *numValidComps*(p) returns the number of *valid completions* of p.

   Furthermore, *numValidComps* has two properties that are similar to corresponding properties of the *numPaths* values:

[4] Since the number of L-paths in G is finite, there must be a regular grammar R that describes the L-paths in G. We use a context-free grammar for convenience. Also, R may be exponentially larger than *CF*.

[5] Readers who are familiar with interprocedural dataflow analysis may prefer to start with the presentation given in Section 3.5.2 and Section 3.5.5 and read the present section later to understand the justification of the numbering algorithm.

- In the Ball-Larus technique, *numPaths*$[Exit]$ is 1 because the only path from $Exit$ to $Exit$ is the path of length 0. For an L-path q from $Entry$ to $Exit$, *numValidComps*(q) is defined to be 1 because the only valid completion of q is the path of length 0 from $Exit$ to itself. (In the rest of this thesis, a path of length 0 is called an *empty path*. The empty path from a vertex v to itself is denoted by "$[\epsilon : v]$.")
- In the Ball-Larus technique, the value *numPaths*$[Entry]$ is the total number of paths through the acyclic control-flow graph. In our technique, the value *numValidComps*$([\epsilon : Entry])$ is the total number of L-paths through G.

Note that the definition of the function *numValidComps* is dependent on the specific finite-path graph in question.

2. In the Ball-Larus technique, each edge e is labeled with an integer value that is used when computing path numbers. We define a function *edgeValueInContext* that takes an L-path prefix p and an edge e, and returns an integer value for the edge e in the context given by p. (As in the case for numbering unbalanced-left paths, the value that an edge e contributes to an path number may depend on the path prefix up to e.) The definition of *edgeValueInContext* is based on the concept of a *valid successor*: let p be an L-path prefix from $Entry$ to a vertex v. A vertex w is a valid successor of p if w is a successor of v, and $[p \parallel v \rightarrow w]$ is an L-path prefix.[6] We are now ready to define the function *edgeValueInContext* in terms of the function *numValidComps*: let $w_1, \ldots, w_k$ be the valid successors of the path p, where p is an L-path prefix from $Entry$ to a vertex v. Then *edgeValueInContext*$(p, v \rightarrow w_i)$ is defined as follows:

$$\textit{edgeValueInContext}(p, v \rightarrow w_i) = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{j<i} \textit{numValidComps}(p \parallel v \rightarrow w_j) & \text{otherwise} \end{cases} \quad (5)$$

Note that this equation is similar to Equation (2), which is used in the Ball-Larus technique to assign values to edges. Equation (5) is illustrated in Figure 14.

As mentioned above, the function *edgeValueInContext* is used in computing path numbers. Note that for an edge $v \rightarrow w$, the path $[p \parallel v \rightarrow w]$ must be an L-path prefix, otherwise *edgeValueInContext*$(p, v \rightarrow w)$ is not defined.

3. In the Ball-Larus technique, the path number for a path p is the sum of the values that appear on p's edges. We define the path number of an L-path p to be the following sum:

$$\sum_{[p' \| v \rightarrow w] \text{ a prefix of } p} \textit{edgeValueInContext}(p', v \rightarrow w). \quad (6)$$

(In our interprocedural path-profiling techniques, at runtime, a running total is kept of the contributions of the edges of p', and as the edge $v \rightarrow w$ is traversed, the value of *edgeValueInContext*$(p', v \rightarrow w)$ is added to this sum. The challenge is to devise a method by which the contribution of edge $v \rightarrow w$ to the running sum, which is a function of the path p' seen so far (namely, *edgeValueInContext*$(p', v \rightarrow w)$), can be determined quickly, without an expensive examination of p'. A method for doing this using ψ and ρ functions is presented in Section 3.6.)

[6] We use the notation $[p \parallel q]$ to denote the concatenation of the paths p and q; however, when $[p \parallel q]$ appears as an argument to a function, *e.g.*, *numValidComps*$(p \parallel q)$, we drop the enclosing square brackets.

4. Just as the Ball-Larus technique generates a dense numbering of the paths in an acyclic control-flow graph, we have the following theorem:

   Theorem 3.4.2 *[Dense Numbering of L-paths] Given the correct definition of the function numValidComps, Equations (5) and (6) generate a dense numbering of the L-paths through G. That is, for every L-path p through G, the path number of p is a unique value in the range $[0..(\mathit{numValidComps}([\epsilon : Entry]) - 1)]$. Furthermore, each value in this range is the path number of an L-path through G.* □

   Theorem 3.4.2 is proven in Appendix A.

Our interprocedural path-profiling techniques are based on the above technique for numbering L-paths in a finite-path graph. As mentioned above, G^*_{fin} is a finite-path graph, where unbalanced-left paths correspond to L-paths. In Section 3.3, we showed how to number unbalanced-left paths through G^*_{fin} by use of the ψ and ρ functions. As shown in Section 3.5.1, the ψ and ρ functions are actually being used to compute *numValidComps* and *edgeValueInContext*.

3.5 Numbering Unbalanced-Left Paths in G^*_{fin}

The first step in collecting an interprocedural path profile is to construct the supergraph, G^*. The second step is to construct G^*_{fin}, as described in Section 3.2, and this section assumes that G^*_{fin} has been so constructed. The remainder of this section is organized as follows: Section 3.5.1 discusses the technical connection between numbering the unbalanced-left paths in G^*_{fin} and numbering the L-paths in a finite-path graph. Section 3.5.2 describes how to assign the ψ and ρ functions that are used in numbering unbalanced-left paths in G^*_{fin}. Section 3.5.3 describes how to compute *edgeValueInContext* for interprocedural edges of G^*_{fin}. Finally, Section 3.5.5 reviews how the ψ and ρ functions are used to calculate the path number of an unbalanced-left path.

3.5.1 Connection Between Numbering Unbalanced-Left Paths in G^*_{fin} and Numbering L-Paths in a Finite-Path Graph

Motivation Behind the ψ Functions

The graph G^*_{fin}, together with the context-free grammar for unbalanced-left strings, is an example of a finite-path graph. This means that the technique presented in Section 3.4 can be used to number unbalanced-left paths in G^*_{fin}. In particular, the L-paths of Section 3.4 are the unbalanced-left paths in G^*_{fin} that start at $Entry_{global}$ and end at $Exit_{global}$. The function *numValidComps* discussed in Section 3.4 takes an unbalanced-left path p (that starts at $Entry_{global}$ in G^*_{fin}) and returns the number of valid completions of p. The function *edgeValueInContext* and the definition of a path number are exactly as described in Section 3.4.

In Section 3.5.2, we describe a technique that assigns a function ψ_v to each vertex v, and a function ρ_e to each intraprocedural edge e. These functions are used by the runtime instrumentation code to compute path numbers, which involves computing *numValidComps* and *edgeValueInContext*. This section starts by discussing properties of *numValidComps* that motivate the definition of the ψ functions. This is followed by a discussion that motivates the ρ functions.

First, observe that the following relation holds for *numValidComps*:

Let p be an unbalanced-left path from $Entry_{global}$ to v, such that $v \neq Exit_{global}$. Let $w_1 \ldots w_k$ be the set of valid successors of p. Then

$$\textit{numValidComps}(p) = \sum_{i=1}^{k} \textit{numValidComps}(p||v \rightarrow w_i). \tag{7}$$

This relation is very similar to the definition of *numPaths* (see Equation (1)). In particular, for any vertex v such that v is not an $Exit_P$ vertex, the set of valid successors for any path to v is the set of all successors of v, and so Equation (7) is identical to the definition of *numPaths*. For an $Exit_P$ vertex there is only one valid successor: for an unbalanced-left path p from $Entry_{global}$ to $Exit_P$, the label on the first edge of any valid completion of p must match the last open parenthesis that labels an edge of p. Note that this means that the number of valid completions for an unbalanced-left path p is completely determined by the last vertex of p and the sequence of unbalanced-left parentheses in p.

Now consider an unbalanced-left path p from $Entry_{global}$ to a vertex v and a same-level valid path q from v to a vertex u in the same procedure as v. Note that $[p \parallel q]$ is an unbalanced-left path and that the sequence of unbalanced-left parentheses in $[p \parallel q]$ is the same as in p alone. This implies that for an unbalanced-left path p from $Entry_{global}$ to v and a vertex u, the value of $\textit{numValidComps}(p \parallel q)$ is the same for *any* same-level valid path q from v to u.

These observations help to motivate our approach for computing *numValidComps* for unbalanced-left paths. Consider an unbalanced-left path p from $Entry_{global}$ to a vertex v in procedure P. For any same-level valid path from v to $Exit_P$, $\textit{numValidComps}(p \| q)$ is the same and is determined by the sequence of unmatched open parentheses in p. We also observe that the value $\textit{numValidComps}(p \parallel q'') = 1$, where q'' is any same-level valid path from v to $GExit_P$. (The fact that $\textit{numValidComps}(p \parallel q'')$ is always 1 follows from the fact that the only successor of $GExit_P$ is $Exit_{global}$.) Given the number of valid completions from $Exit_P$ and from $GExit_P$ (for a given p), it is possible to compute $\textit{numValidComps}(p \parallel s)$ for any same-level valid path s from v to any vertex u (that is reachable from v) in procedure P. To aid in these computations, for each vertex v of procedure P, we define a function ψ_v that, when given the number of valid completions from $Exit_P$, returns the number of valid completions from v. Note that the function ψ_v does not need to take the number of valid completions from $GExit_P$ as an explicit argument, because the number of valid completions of any path to $GExit_P$ is always 1.

The ψ functions will be used in calculating $\textit{numValidComps}(p)$ for an unbalanced-left path p that ends at a vertex $v \neq Exit_P$. They are also used to compute $\textit{numValidComps}(q)$ for an unbalanced-left path q from $Entry_{global}$ to a vertex $Exit_P$. We now consider the latter use of the ψ functions. Recall that there is only one valid successor of q—the return vertex r_1 such that the label on the edge $Exit_P \rightarrow r_1$ matches the last open parenthesis of q. Thus, we have the following:

$$\textit{numValidComps}(q) = \textit{numValidComps}(q \parallel Exit_P \rightarrow r_1).$$

Suppose r_1 occurs in procedure Q. Then the above value is equal to

$$\psi_{r_1}(\textit{numValidComps}(q \parallel Exit_P \rightarrow r_1 \parallel q')),$$

where q' is any same-level valid path from r_1 to $Exit_Q$. Recall that the function ψ_{r_1} counts the valid completions of $[q \parallel Exit_P \rightarrow r_1]$ that exit Q via $GExit_Q$, even though ψ_{r_1} only takes as an argument the number of valid completions for paths that exit Q via $Exit_Q$.

As before, there is only one valid successor of $[q \parallel Exit_P \rightarrow r_1 \parallel q']$: the return vertex r_2 such that $Exit_Q \rightarrow r_2$ is labeled with the parenthesis that closes the second-to-last open parenthesis in q.

Suppose that r_2 is in procedure R. Then the value of *numValidComps*(q) is equal to

$$\psi_{r_1}(\psi_{r_2}(\textit{numValidComps}(q \parallel \textit{Exit}_P \rightarrow r_1 \parallel q' \parallel \textit{Exit}_Q \rightarrow r_2 \parallel q'')))),$$

where q'' is any same-level valid path from r_2 to $\textit{Exit}_R$. Again, ψ_{r_2} counts valid completions that leave R via either $\textit{GExit}_R$ or $\textit{Exit}_R$.

This argument can be continued until a path s has been constructed from $\textit{Entry}_{global}$ to $\textit{Exit}_S$, where S is the first procedure that q (and s) enters. The path s has q as a prefix, and has only one unmatched parenthesis, "$(_S$", which is the same as the first unmatched parenthesis in q. The parenthesis "$(_S$" is matched by the parenthesis "$)_S$", which can only appear on the edge $\textit{Exit}_S \rightarrow \textit{Exit}_{global}$. Thus, the number of valid completions of s is 1. This implies that

$$\begin{aligned} \textit{numValidComps}(q) &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(\textit{numValidComps}(s))\ldots)) \\ &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(1)\ldots)) \end{aligned} \tag{8}$$

where $r_1 \ldots r_k$ is the sequence of return vertices determined by the unmatched parentheses in q. Figure 16 shows a schematic of the path s that is constructed to compute *numValidComps*(q).

The ψ functions are also used to calculate the total number of unbalanced-left paths through G^*_{fin}, *i.e.*, *numValidComps*$([\epsilon : \textit{Entry}_{global}])$:

$$\textit{numValidComps}([\epsilon : \textit{Entry}_{global}]) = \sum_{\textit{Entry}_P \in \textit{succ}(\textit{Entry}_{global})} \textit{numValidComps}(\textit{Entry}_{global} \rightarrow \textit{Entry}_P). \tag{9}$$

The value of *numValidComps*$(\textit{Entry}_{global} \rightarrow \textit{Entry}_P)$ can be computed using the function $\psi_{\textit{Entry}_P}$: For a same-level valid path p from $\textit{Entry}_P$ to $\textit{Exit}_P$, the value of

$$\textit{numValidComps}(\textit{Entry}_{global} \rightarrow \textit{Entry}_P \parallel p)$$

is 1, because the only valid completion of $[\textit{Entry}_{global} \rightarrow \textit{Entry}_P \parallel p]$ is the edge $\textit{Exit}_P \rightarrow \textit{Exit}_{global}$; thus, for a path consisting of the edge $\textit{Entry}_{global} \rightarrow \textit{Entry}_P$, the value of

$$\textit{numValidComps}(\textit{Entry}_{global} \rightarrow \textit{Entry}_P)$$

is given by

$$\textit{numValidComps}(\textit{Entry}_{global} \rightarrow \textit{Entry}_P) = \psi_{\textit{Entry}_P}(1). \tag{10}$$

Substituting Equation (10) into Equation (9) yields the following:

$$\textit{numValidComps}([\epsilon : \textit{Entry}_{global}]) = \sum_{\textit{Entry}_P \in \textit{succ}(\textit{Entry}_{global})} \psi_{\textit{Entry}_P}(1). \tag{11}$$

Motivation Behind the ρ Functions

In addition to the ψ functions on vertices, functions are also assigned to edges. In particular, each intraprocedural edge e is assigned a function ρ_e. While the function ψ_v is used to compute *numValidComps*(p) for a path p ending at vertex v, the function ρ_e is used to compute *edgeValueInContext*(p, e) for a path p to the source vertex of e. Specifically, for each edge e of a procedure P, the function ρ_e takes the number of valid completions from $\textit{Exit}_P$ (for an unbalanced-left path p' to $\textit{Entry}_P$ concatenated with any same-level valid path to $\textit{Exit}_P$) and returns the value of *edgeValueInContext*$(p' \parallel q, e)$, where q is any same-level valid path from $\textit{Entry}_P$ to the edge e.

In the following section, we first describe how to define the ψ functions and then show how to define the ρ functions. In Section 3.6, we show how to use these functions to instrument a program in order to collect an interprocedural profile.

3.5.2 Assigning ψ and ρ Functions

The Relationship of Sharir and Pnueli's ϕ Functions to ψ Functions

Recall that in the Ball-Larus technique, each vertex v is assigned an integer value *numPaths*$[v]$ that indicates the number of paths from v to the exit vertex. In this section, we first show that the problem of finding the value *numPaths*$[v]$ for each vertex v of a control-flow graph can be cast in a form that is similar to a backwards, intraprocedural dataflow-analysis problem. We then show that our scheme for assigning ψ functions to vertices is similar to Sharir and Pnueli's functional approach to interprocedural dataflow analysis [58].

A distributive, backwards dataflow-analysis problem includes a semi-lattice L with meet operator $\sqcap$, a set F of distributive functions from L to L that is closed under composition, a graph G, and a dataflow fact c associated with the exit vertex $Exit$ of G. Each edge $v \rightarrow w$ of the graph is labeled with a function $f_{v \rightarrow w} \in F$. Kildall showed that the meet-over-all-paths solution to a distributive, backwards dataflow-analysis problem is given by the maximal fixed point of the following equations [40]:

$$\mathit{val}[v] = \bigsqcap_{w \in \mathit{succ}(v)} f_{v \rightarrow w}(\mathit{val}[w]) \quad \text{for } v \neq \mathit{Exit} \tag{12}$$

$$\mathit{val}[Exit] = c \tag{13}$$

For a DAG, finding the quantity *numPaths*$[v]$ amounts to a technique for summing over all paths between v and $Exit$, where each path contributes a value of one to the sum. Thus, *numPaths*$[v]$ can be considered to be a "sum-over-all-paths" value. To find the *numPaths* values, the Ball-Larus technique finds the maximum fixed point of the following equations (over the integers together with ∞):

$$\mathit{numPaths}[v] = \sum_{w \in \mathit{succ}(v)} \mathit{id}(\mathit{numPaths}[w]) \qquad \text{for } v \neq \mathit{Exit} \tag{14}$$

$$\mathit{numPaths}[Exit] = 1 \tag{15}$$

The form of Equation (14) is similar to Equation (12), with the identity function *id* standing in for each edge function $f_{v \rightarrow w}$, and the addition operator $+$ playing the role of the meet operator $\sqcap$. When we simplify the right-hand side of Equation (14) to $\sum_{w \in \mathit{succ}(v)} \mathit{numPaths}[w]$, this is precisely the definition of *numPaths*$[v]$ given in Section 2.1 in Equation (1).

Thus, the problem of calculating *numPaths* values is similar to a dataflow-analysis problem (on a DAG), differing only in that addition is not an idempotent meet operator. (In fact, the problem of calculating *numPaths* values is an example of an algebraic path problem on a DAG. For a more general discussion of the relationship between algebraic path problems and dataflow-analysis problems, see [53].)

We now review the appropriate part of Sharir and Pnueli's work [58], with some rephrasing of their work to describe backwards dataflow-analysis problems instead of forwards dataflow-analysis problems. We then show how their ϕ functions are related to our ψ functions.

In Sharir and Pnueli's functional approach to interprocedural dataflow analysis, for each procedure P, and each vertex v of P, the function ϕ_v captures the transformation of dataflow facts from $Exit_P$ to v.[7] The ϕ functions are found by setting up and solving a system of equations. For an exit vertex $Exit_P$, ϕ_P is the identity function:[8]

$$\phi_{\mathit{Exit}_P} = \mathit{id}. \tag{16}$$

[7] Information flows counter to the direction of control-flow graph edges in a backwards dataflow-analysis problem.

[8] According to [58], this equation should be $\phi_{\mathit{Exit}_P} \sqsubseteq \mathit{id}$. Since we do not allow an exit vertex to be the source of an intraprocedural edge, it is safe to replace $\sqsubseteq$ with $=$.

For a call vertex c associated with return-site vertex r to the procedure Q, we have the following equation:

$$\phi_c = \phi_{Entry_Q} \circ \phi_r. \tag{17}$$

Finally, for any other vertex m in P, we have the following equation:

$$\phi_m = \bigsqcap_{n \in succ(m)} f_{m \to n} \circ \phi_n. \tag{18}$$

In a similar fashion, we wish to define, for each procedure P and each vertex v in P, a function ψ_v that calculates the number of valid completions from v to $Exit_{global}$ based on the number of valid completions from $Exit_P$ to $Exit_{global}$. The problem of finding the ψ functions differs from the one solved by Sharir and Pnueli in that the ϕ functions describe how dataflow facts are propagated in a dataflow-analysis problem and the ψ functions describe how values are propagated in the problem of assigning Ball-Larus-like values to vertices. However, as shown above, the problem of assigning Ball-Larus values to vertices is similar to a backwards dataflow-analysis problem, and as we show below, the equations that hold for the ψ functions are very similar to the equations that hold for the ϕ functions.

As in Sharir and Pnueli's functional approach to interprocedural dataflow analysis, several equations must hold. For an exit vertex $Exit_P$, ψ_{Exit_P} is the identity function:

$$\psi_{Exit_P} = id. \tag{19}$$

This is similar to Equation (16).

For a vertex of the form $GExit_P$, the following must hold:

$$\psi_{GExit_P} = \lambda x.1. \tag{20}$$

This equation reflects the fact that the number of valid completions from $GExit_P$ is always 1, regardless of the number of valid completions from $Exit_P$. Equation (20) does not have a direct analog in [58], however, $GExit_P$ could be thought of as a vertex that generates a constant dataflow fact.

For a call vertex c to a procedure Q associated with return-site vertex r, where c and r represent a non-recursive call site, we have the following:

$$\psi_c = \psi_{Entry_Q} \circ \psi_r. \tag{21}$$

This is similar to Equation (17).

For all other cases for a vertex m, the following must hold:

$$\psi_m = \sum_{n \in succ(m)} id \circ \psi_n. \tag{22}$$

where the addition $f + g$ of function values f and g is defined to be the function $\lambda x.f(x) + g(x)$. Equation (22) is similar to Equation (18) with the identity function *id* standing in for each edge function $f_{m \to n}$.

Just as Equations (16)–(18) are the interprocedural analogs of Equations (12) and (13), Equations (19)–(22) are the interprocedural analogs of Equations (14) and (15).

We now show that the solution to Equations (19)–(22) yields the desired ψ functions. Recall that, for a vertex v in procedure P, the function ψ_v takes the number of valid completions from $Exit_P$ (for an unbalanced-left path p to vertex v in procedure P concatenated with any same-level valid path from

v to $Exit_P$) and returns the number of valid completions from v (for p). Given this definition of ψ_v, it is clear that Equations (19) and (20) must hold. Equation (22) must hold because of Equation (7) and the fact that for an internal vertex v and any unbalanced-left path p to v, the valid successors of p are the same as the successors of v.

Equation (21) requires more extensive justification. Let p be an arbitrary unbalanced-left path to $Entry_P$; let p' be an arbitrary same-level valid path from $Entry_P$ to a call vertex c; let c be associated with the return vertex r; let c and r represent a nonrecursive call site on procedure Q ; and let q be an arbitrary same-level valid path from $Entry_Q$ to $Exit_Q$. (Figure 17 illustrates these paths and vertices.) The function ψ_c takes the number N_p of valid completions from $Exit_P$ (for the path p concatenated with any same-level valid path from $Entry_P$ to $Exit_P$) and returns the number of valid completions from c (for $[p \parallel p']$).

By the definition of ψ_r, we know that the number of valid completions from vertex r (for p concatenated with any same-level valid path to r) is given by $\psi_r(N_p)$. In particular, we have the following:

$$\mathit{numValidComps}(p \parallel p' \parallel c \to Entry_Q \parallel q \parallel Exit_Q \to r) = \psi_r(N_p).$$

Now consider the path $[p \parallel p' \parallel c \to Entry_Q \parallel q]$. The last unmatched parenthesis in this path is on the edge $c \to Entry_Q$. This gives us the following (see Figure 17):

$$\mathit{numValidComps}(p \parallel p' \parallel c \to Entry_Q \parallel q) = \psi_r(N_p).$$

Because q is a same-level valid path from $Entry_P$ to $Exit_P$, we have (by the definition of ψ_{Entry_Q}) the following:

$$\mathit{numValidComps}(p \parallel p' \parallel c \to Entry_Q) = \psi_{Entry_Q}(\psi_r(N_p)).$$

Because $Entry_Q$ is the only successor of c, this gives us

$$\mathit{numValidComps}(p \parallel p') = \psi_{Entry_Q}(\psi_r(N_p)).$$

However, by the definition of ψ_c, $\mathit{numValidComps}(p \parallel p') = \psi_c(N_p)$. This gives us

$$\psi_c(N_p) = \psi_{Entry_Q}(\psi_r(N_p)). \tag{23}$$

We have shown that Equation (23) holds for an arbitrary unbalanced-left path $[p \parallel p']$ to c (to which there corresponds N_P, the number of valid completions from $Exit_P$ for p concatenated with any same-level valid path to $Exit_P$) and for an arbitrary same-level valid path $[c \to Entry_Q \parallel q \parallel Exit_Q \to r]$. It follows that

$$\psi_c = \psi_{Entry_Q} \circ \psi_r.$$

Solving for ψ Functions

We now describe how to find ψ functions that satisfy Equations (19)–(22). Note that each ψ function is a linear function of one variable. This follows because $\mathit{id}(= \lambda x.x)$ and $\lambda x.1$ are both linear functions of one variable, and the space of linear functions of one variable is closed under function composition and function addition. For the composition of two linear functions, we observe that

$$(\lambda x.a \cdot x + b) \circ (\lambda y.c \cdot y + d) = \lambda z.(a \cdot c) \cdot z + (a \cdot d + b).$$

For the addition of two linear function values (as defined above), we have

$$(\lambda x.a{\cdot}x{+}b){+}(\lambda y.c{\cdot}y{+}d) = \lambda z.(\lambda x.a{\cdot}x{+}b)(z){+}(\lambda y.c{\cdot}y{+}d)(z) = \lambda z.(a{+}c){\cdot}z{+}(b{+}d).$$

The fact that each ψ function is a linear function of one variable means that they can be compactly represented as an ordered pair, with one coordinate representing the coefficient, and one coordinate representing the constant.

To find the ψ functions, each procedure P is visited in reverse topological order of the call graph, and each vertex v in P is visited in reverse topological order of P's control-flow graph. (For purposes of ordering the vertices of a procedure P, a return vertex r is considered to be a successor of its associated call vertex c.) As each vertex v is visited, the appropriate equation from Equations (19)–(22) is used to determine the function ψ_v.

The order of traversal guarantees that when vertex v is visited, all of the functions that are needed to determine ψ_v will be available. This follows from the fact that the call graph associated with G^*_{fin} is acyclic and the fact that the control-flow graph of each procedure in G^*_{fin} is acyclic. (The fact that the call graph and control-flow graphs are acyclic also explains why each vertex needs to be visited only once.) For instance, consider a call vertex c that is associated with return-site vertex r and calls procedure Q. When the vertex c is visited, the function ψ_r will be available (because vertices are visited in reverse topological order) and the function ψ_{Entry_Q} will be available (because procedures are processed in reverse topological order). Hence, Equation (21) can be used to determine ψ_c.

Solving for ρ functions

As mentioned above, a linear function is assigned to each intraprocedural edge e to aid in the calculation of *edgeValueInContext*. In particular, for an edge e in procedure P, we define the function ρ_e such that, when supplied with the number of valid completions from Exit_P (for an unbalanced-left path $[p \,\|\mathrm{e}\|\, q]$, where p is an unbalanced-left path that ends at the source vertex of edge e and q is any same-level valid path from the target vertex of e to Exit_P), it returns the value of *edgeValueInContext*(p, e).

Let v be an intraprocedural vertex that is the source of one or more intraprocedural edges. (That is, v cannot be a call vertex for a nonrecursive call–site, nor have the form Exit_P, nor have the form GExit_P.) Let $w_1 \ldots w_k$ be the successors of v. Recall that for a vertex such as v, for any unbalanced-left path p that ends at v, every successor of v is a valid successor of p. This is because no outgoing edge from v is labeled with a parenthesis. Given the definition of *edgeValueInContext* (see Equation (5)) and the definition of the ψ functions, it follows that the following equation holds:

$$\rho_{v \to w_i} = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{j<i} \psi_{w_j} & \text{otherwise} \end{cases} \tag{24}$$

Clearly, each ρ function is a linear function of one variable. Furthermore, Equation (24) can be used to find each ρ function after the appropriate ψ functions have been determined (*i.e.*, during a traversal of a procedure's vertices in reverse topological order).

3.5.3 Computing *edgeValueInContext* for interprocedural edges

The ρ functions are only assigned to intraprocedural edges and they can be used to calculate *edgeValueInContext* when the second argument to *edgeValueInContext* is an intraprocedural edge. To compute the path number for an unbalanced-left path p it is also necessary to compute *edgeValueInContext* for certain interprocedural edges. This section describes how to compute *edgeValueInContext* for interprocedural edges.

In fact, for an interprocedural edge e and an unbalanced-left path p to e, the value of *edgeValueInContext*(p, e) is almost always 0. The only situation where this is not the case is when e is of the form $Entry_{global} \rightarrow Entry_Q$ and p is the path $[\epsilon : Entry_{global}]$. (Recall that as part of creating G^*_{fin}, Transformation 3 of Section 3.2 adds edges of the form $Entry_{global} \rightarrow Entry_Q$ and $Exit_Q \rightarrow Exit_{global}$ for each recursively called procedure Q.) This follows from the fact that for an unbalanced-left path p that ends at a call vertex, an $Exit_P$ vertex, or a $GExit_P$ vertex, p has only one valid successor.

Let us consider the case of an edge of the form $Entry_{global} \rightarrow Entry_Q$. The value of *edgeValueInContext*$([\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_Q)$ is computed using Equation (5). This means that it is necessary to set a fixed (but arbitrary) ordering of the edges of the form $Entry_{global} \rightarrow Entry_P$. For convenience, we number each edge $Entry_{global} \rightarrow Entry_P$ according to this ordering, and use Q_i to refer to the procedure that is the target of the i^{th} edge. From Equation (5), the value of

$$\textit{edgeValueInContext}([\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_{Q_i})$$

is as follows:

$$\begin{cases} 0 & \text{if } i = 0 \\ \sum_{j<i} \textit{numValidComps}(Entry_{global} \rightarrow Entry_{Q_j}) & \text{otherwise} \end{cases} \tag{25}$$

As noted in Section 3.5.1, the value of *numValidComps*$(Entry_{global} \rightarrow Entry_{Q_j})$ is given by

$$\textit{numValidComps}(Entry_{global} \rightarrow Entry_{Q_j}) = \psi_{Entry_{Q_j}}(1). \tag{26}$$

Substituting Equation (26) into Equation (25) yields the following:

$$\textit{edgeValueInContext}([\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_{Q_i}) = \begin{cases} 0 & \text{if } i = 0 \\ \sum_{j<i} \psi_{Entry_{Q_j}}(1) & \text{otherwise} \end{cases} \tag{27}$$

3.5.4 Practical Considerations When Numbering Unbalanced-Left Paths

The previous sections demonstrated how to obtain a dense numbering of the paths through G^*_{fin}. This is important because it limits the number of bits needed to encode a path number. However, the number of paths through G^*_{fin} can be doubly exponential in the size of G^*_{fin}. In practice, even with a dense numbering, there are still an impractically large number of paths. For practical purposes, each path name must fit in a machine word, otherwise the profiling overhead becomes prohibitively expensive. On a machine with a word size of 64 bits, we require that there be no more than 2^{64} paths through G^*_{fin}.

Recall that in the construction of G^*_{fin} (see Section 3.2), each backedge $u \rightarrow v$ in procedure P is removed and replaced with the surrogate edges $Entry_P \rightarrow v$ and $u \rightarrow GExit_P$. We call the process of removing an edge $u \rightarrow v$ and replacing it with surrogate edges *breaking* the edge $u \rightarrow v$. We can limit the number of paths through G^*_{fin} by breaking non-backedges in a similar fashion. Consider an edge $u \rightarrow v$ in procedure P where there are c paths from $Entry_P$ to u and d paths from v to $Exit_P$ (see Figure 18. This edge participates in $c \cdot d$ paths from $Entry_P$ to $Exit_P$. If $u \rightarrow v$ is removed from the graph, $c \cdot d$ paths from $Entry_P$ to $Exit_P$ are eliminated. Adding the surrogate edge $Entry_P \rightarrow v$ creates d new paths from $Entry_P$ to $Exit_P$. Adding the surrogate edge $u \rightarrow GExit_P$ creates c new paths from $Entry_P$ to $GExit_P$. On balance, replacing the edge $u \rightarrow v$ with the surrogate edges $Entry_P \rightarrow v$ and $u \rightarrow GExit_P$ (*i.e.*, breaking the edge $u \rightarrow v$) eliminates many more paths ($c \cdot d$) than it creates ($c + d$).

It is also possible to limit the number of paths through G^*_{fin} by breaking call and return-edges at a call-site (as is done in Transformation 3 of the construction of G^*_{fin}; see Section 3.2). We also transform some call-sites by breaking the call-site's return-edge, but not the call-site's call-edge. This is shown in Figure 19. Specifically, for a pair of vertices c and r representing a call-site to a procedure Q, we can lessen the number of paths in G^*_{fin} by performing the following transformation:

> Remove the return-edge $Exit_Q \rightarrow r$. Add the surrogate edge $Exit_Q \rightarrow Exit_{global}$ and the summary-edge $c \rightarrow r$.

This transformation eliminates paths containing the edge $Exit_Q \rightarrow r$, but retains paths containing the edge $c \rightarrow Entry_Q$. The profiling machinery at the vertex c will save the path number for the current path p before calling the procedure Q. After control returns from $Exit_Q$ to the return-site vertex r, the profiling machinery will end the path from $Exit_Q$ using the surrogate-edge $Exit_Q \rightarrow Exit_{global}$. Then it will use the path number saved at c and the function $\rho_{c \rightarrow r}$ to resume recording the path $[p \| c \rightarrow r]$. After the above transformation has been performed, the appropriate ψ for c is given by

$$\psi_c = \psi_{Entry_Q} \circ \lambda x.1 + \psi_r$$

and the appropriate ρ for $c \rightarrow r$ is given by

$$\rho_{c \rightarrow r} = \psi_{Entry_Q} \circ \lambda x.1$$

As before, for any unbalanced-left path p to c, the value of $edgeValueInContext(p, c \rightarrow Entry_Q)$ is 0.

The above discussion demonstrates that the number of paths through G^*_{fin} can be lowered by replacing some edges with surrogate edges. The question becomes, which edges to break such that the number of paths through G^*_{fin} is less that 2^{WordSize}? (Here, WordSize is the number of bits in a machine word on the system where the profiling is performed.) We use a technique that breaks edges as it computes the ψ functions.

By Equation (11) in Section 3.5.1, we know that the number of paths in G^*_{fin} is equal to

$$\sum_{P \in \mathrm{Proc}} \psi_{Entry_P}(1)$$

Let NumProcs be the number of procedures in G^*_{fin}. As we compute the ψ functions for the vertices of a procedure P, we replace edges with surrogate edges until

$$\psi_{Entry_P}(1) \leq 2^{\mathrm{WordSize}}/\mathrm{NumProcs} \tag{28}$$

This guarantees that the number of paths through G^*_{fin} is less than 2^{WordSize}.

When computing the ψ functions for a procedure P, we simultaneously break edges in P to try to ensure that Equation (28) holds. When we break an edge in P is determined by a threshold value MaxPathsFromNonEntry. If after computing ψ_{Entry_P} Equation (28) does not hold, we lower the threshold for breaking edges and recompute the ψ functions in P. More precisely, we use the following algorithm to compute ψ functions in a procedure P and simultaneously break edges in P so that the number of paths in G^*_{fin} is limited:

Algorithm 3.5.1
ComputeAllPsiFnsInProc(P)
$\mathrm{MaxPathsFromEntry} = 2^{\mathrm{WordSize}}/\mathrm{NumProcs}$

MaxPathsFromNonEntry = MaxPathsFromEntry

Foreach vertex v in P in reverse topological order
If $v = Entry_P$ **Then**
$\psi_v := \sum_{w \in succ(v)} \psi_w$
Else
$ComputePsiFnAndBreakEdges(v, \text{MaxPathsFromNonEntry})$

While $\psi_{Entry_P}(1) > \text{MaxPathsFromEntry}$
Undo all of the edge breaks in P
Break all of the backedges in P
MaxPathsFromNonEntry/=2
Foreach vertex v in P in reverse topological order
If $v = Entry_P$ **Then**
$\psi_v := \sum_{w \in succ(v)} \psi_w$
Else
$ComputePsiFnAndBreakEdges(v, \text{MaxPathsFromNonEntry})$
End ComputeAllPsiFnsInProc

ComputePsiFnAndBreakEdges$(v, \text{MaxNumPaths})$
If v is not a call vertex **Then**
$\psi_v := \sum_{w \in succ(v)} \psi_w$
While $\psi_v(1) > \text{MaxNumPaths}$
break an edge $v \rightarrow w$
$\psi_v := \sum_{w \in succ(v)} \psi_w$
Return
Else
Let r denote the return-site vertex associated with the call vertex v
Let Q be the procedure called by v
$\psi_v := \psi_{Entry_Q} \circ \psi_r$
If $\psi_v(1) > \text{MaxNumPaths}$ **Then**
break the return-edge $Exit_Q \rightarrow r$
$\psi_v := \psi_{Entry_Q} \circ \lambda x.1 + \psi_r$
If $\psi_v(1) > \text{MaxNumPaths}$ **Then**
break the call-edge $v \rightarrow Entry_Q$
$\psi_v := \psi_r$
End ComputePsiFnAndBreakEdges

□

3.5.5 Calculating the Path Number of an Unbalanced-Left Path

In this section, we show how to calculate the path number of an unbalanced-left path p through G^*_{fin} from $Entry_{global}$ to $Exit_{global}$. This is be done during a single traversal of p that sums the values of *edgeValueInContext*(p', e) for each p' and e such that $[p' \parallel e]$ is a prefix of p (*cf.* Equation (6)).

For interprocedural edges, the value of *edgeValueInContext* is calculated as described in Section 3.5.3.

For an intraprocedural edge e in procedure P, the value of *edgeValueInContext*(p', e) is calculated by applying the function ρ_e to the number of valid completions from $Exit_P$. (The number of valid completions from $Exit_P$ is determined by the path taken to $Entry_P$—in this case a prefix of p'.)

We now come to the crux of the matter: how to determine the contribution of an edge e when the edge is traversed (*i.e.*, how to determine the value *edgeValueInContext*$(p', e,))$ without incurring a cost for inspecting the path p'. The trick is that, as p is traversed, we maintain a value, `numValidCompsFromExit`, to hold the number of valid completions from the exit vertex $Exit_Q$ of the procedure Q that is currently being visited (the number of valid completions from $Exit_Q$ is uniquely determined by p'—specifically, the sequence of unmatched left parentheses in p'). The value `numValidCompsFromExit` is maintained by the use of a stack, `NVCStack`, and the ψ functions for return-site vertices. The following steps describe the algorithm to compute the path number for a path p (which is accumulated in the variable `pathNum`):

- When the traversal of p is begun, `numValidCompsFromExit` is set to 1. This indicates that there is only one valid completion from $Exit_R$, where R is the first procedure that p enters: if p reaches the exit of the first procedure it enters, then it must follow the edge $Exit_P \rightarrow Exit_{global}$. The value of `pathNum` is initialized to the value *edgeValueInContext*$([\epsilon : Entry_{global}], e)$ where e is the first edge of p (see Section 3.5.3).

- As the traversal of p crosses a call-edge $c \rightarrow Entry_T$ from a procedure S to a procedure T, the value of `numValidCompsFromExit` is pushed on the stack, and is updated to ψ_r(*numValidCompsFromExit*), where r is the return-site vertex in S that corresponds to call vertex c. This reflects the fact that the number of valid completions from $Exit_T$ is equal to the number of valid completions from r.

- As the traversal of p crosses a return-edge $Exit_T \rightarrow r$ from a procedure T to a procedure S, the value of `numValidCompsFromExit` is popped from the top of the stack. This reflects the fact that the number of valid completions from the exit of the calling procedure S is unaffected by the same-level valid path that was taken through the called procedure T.

- As the traversal of p crosses an intraprocedural edge e, the value of `pathNum` is incremented by ρ_e(`numValidCompsFromExit`).

- At the end of the traversal of p, `pathNum` is output.

In essence, we have described the following algorithm:

Algorithm 3.5.2 (Calculate Path Number)
Input: An unbalanced-left path p from $Entry_{global}$ to $Exit_{global}$.
Output: p's path number.

initialize `numValidCompsFromExit` to 1
Initialize stack `NVCstack` to empty

Let e be the first edge of the path p. Calculate the value of *edgeValueInContext*$([\epsilon : Entry_{global}], e)$ as described in Section 3.5.3. Set `pathNum` to this value.

set e to the second edge of p

while e is not of the form $v \rightarrow Exit_{global}$ **do**
    if e is of the form $c \rightarrow Entry_T$ **then**
        push(`NVCstack`, `numValidCompsFromExit`)
        let r be the return vertex associated with c
        `numValidCompsFromExit` := ψ_r(`numValidCompsFromExit`)
    else if e is of the form $Exit_T \rightarrow r$ **then**
        `numValidCompsFromExit` := pop(`NVCstack`)
    else
        `pathNum` := `pathNum` + ρ_e(`numValidCompsFromExit`)
    fi
    set e to the next edge of p
od
output `pathNum`

□

3.6 Runtime Environment for Collecting a Profile

We are now ready to describe the instrumentation code that is introduced to collect an interprocedural path profile. In essence, the instrumentation code threads the algorithm described in Section 3.5.5 into the code of the instrumented program. Thus, the variables `pathNum` and `numValidCompsFromExit` become program variables. There is no explicit stack variable corresponding to `NVCstack`; instead, `numValidCompsFromExit` is passed as a value-parameter to each procedure and the program's execution stack is used in place of `NVCstack`. The instrumentation also makes use of two local variables in each procedure:

`pathNumOnEntry` stores the value of `pathNum` on entry to a procedure. When an intraprocedural backedge is traversed in a procedure P, the instrumentation code increments the count associated with the current observable path and begins recording a new observable path that has the context-prefix indicated by the value of `pathNumOnEntry`.

`pathNumBeforeCall` stores the value of `pathNum` before a recursive procedure call is made. When the recursive procedure call is made, the instrumentation begins recording a new observable path. When the recursive call returns, the instrumentation uses the value in `pathNumBeforeCall` to resume recording the observable path that was executing before the call was made.

Figs. 12 and 13 show an instrumented version of the code in Fig. 8. Appendix B gives a more detailed description of the instrumentation used to collect an interprocedural, context path profile.

3.6.1 Optimizing the Instrumentation

The code that calculates path numbers can be made more efficient than the implementation described above (and in Appendix B). As each non-backedge is traversed, this implementation requires one multiplication and two additions. However, within a given activation of a procedure P, the multiplication is always by the same value of `numValidCompsFromExit`, and the products of these multiplications are always added to the sum in `pathNum`. This means that the multiplication may be "factored out." As an example, consider a subpath in P consisting of the non-backedges e_1, e_2, and e_3 that are associated

with functions $\langle 2, 2\rangle$, $\langle 3, 4\rangle$, and $\langle 5, 3\rangle$. Let $\texttt{pathNum}_{orig}$ be the value of `pathNum` before this subpath is executed. After e_1, e_2, and e_3 are traversed, we have the following:

$$\begin{aligned}\texttt{pathNum} &= \texttt{pathNum}_{orig} \\ &+ 2 \cdot \texttt{numValidCompsFromExit} + 2 \\ &+ 3 \cdot \texttt{numValidCompsFromExit} + 4 \\ &+ 5 \cdot \texttt{numValidCompsFromExit} + 3 \\ &= \texttt{pathNum}_{orig} + 10 \cdot \texttt{numValidCompsFormExit} + 9\end{aligned}$$

Instead of incrementing `pathNum` as each edge is traversed, two temporaries t_1 and t_2 are introduced. Both are initialized to 0. The temporary t_1 is used to sum the coefficients from the edge functions. The temporary t_2 is used to sum the constant terms of the edge functions. When `pathNum` absolutely must be updated (*i.e*, before the profile is updated, before a procedure call is made, or before a procedure returns), it is incremented by $t_1 \cdot \texttt{numValidCompsFromExit} + t_2$. Note that when procedure P calls procedure Q, after control returns to P, both t_1 and t_2 should be set to 0; when control returns to P, `pathNum` will have already been updated for the current values of t_1 and t_2.

The fact that t_1 and t_2 are used to sum values as edges are traversed allows some additional optimizations to be performed. Recall that in the Ball-Larus technique, there is some flexibility in the placement of increment statements; this is used to push the increment statements to infrequently executed edges [10]. In a similar fashion, it is possible to move the statements that increment t_1 and t_2.

3.6.2 Recovering a Path From a Path Number

This section describes an algorithm that takes a path number `pathNum` as input and outputs the corresponding path p; this is the inverse operation of computing a path number. As the algorithm traverses the path p, it decrements the value in `pathNum`. After a path prefix p' has been traversed, `pathNum` holds the value that is contributed to p's original path number by the path p'', where $p = [p' \parallel p'']$. At this stage, *edgeValueInContext*$(p', v \rightarrow w_i)$ is computed for each valid successor $w_i \in \{w_1 \ldots w_k\}$ of p'. (Note that v is the last vertex of p'.) The algorithm is based on the following observations:

- Let w_j denote the valid successor of p' that gives the largest value of *edgeValueInContext*$(p', v \rightarrow w_j)$ that is less than or equal to `pathNum`. Given the definition of *edgeValueInContext*, the edge $v \rightarrow w_j$ must be the next edge of p (*i.e.*, the first edge of p'').

- For $x > j$, the value of *edgeValueInContext*$(p', v \rightarrow w_x)$ is greater than `pathNum` and $v \rightarrow w_x$ cannot be the first edge of the continuation p''.

- For $x < j$, any valid continuation of p' that starts with $v \rightarrow w_x$ will contribute a value to the path number that is less than *edgeValueInContext*$(p', v \rightarrow w_j)$ (see Figure 14), which itself is less than or equal to `pathNum`. Since the continuation p'' makes a contribution equal to `pathNum`, $v \rightarrow w_x$ cannot be the first edge of p''.

Thus, the next steps of the algorithm are: (i) set p' to $p' \parallel v \rightarrow w_j$; (ii) decrement the value of `pathNum` by *edgeValueInContext*$(p', v \rightarrow w_j)$; and (iii) start considering the valid successors of the new p'.

The algorithm uses two stacks: the first is similar to the stack used by Algorithm 3.5.2 and keeps track of the value `numValidCompsFromExit`; the second keeps track of the sequence of return vertices that correspond to the current call stack.

Algorithm 3.6.1 (Calculate Path from Path Number)
Input: The path number `pathNum` for an unbalanced-left path p in G^*_{fin} from $Entry_{global}$ to $Exit_{global}$.
Output: A listing of the edges of the path p.

initialize stack `NVCstack` to empty
initialize stack `returnStack` to empty
initialize `numValidCompsFromExit` to 1
initialize path p to $[\epsilon : Entry_{global}]$

Find the edge $Entry_{global} \rightarrow Entry_P$ that gives the largest value for

$$x = \textit{edgeValueInContext}([\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_P)$$

that is less than or equal to `pathNum` (see Section 3.5.3).
$p := [p \parallel Entry_{global} \rightarrow Entry_P]$
`pathNum` := `pathNum` $- x$
push(`returnStack`, $Exit_{global}$)
push(`NVCstack`, 1)
while p does not end at $Exit_{global}$ **do**
    let v be the last vertex of p
    if v is a call vertex c **then**
        let $c \rightarrow Entry_P$ be the call-edge from c
        $p := [p \parallel c \rightarrow Entry_P]$
        let r be the return vertex associated with c
        push(`returnStack`, r)
        push(`NVCstack`, `numValidCompsFromExit`)
        `numValidCompsFromExit` := ψ_r(`numValidCompsFromExit`)
    else if v is an exit vertex $Exit_P$ **then**
        `numValidCompsFromExit` := pop(`NVCstack`)
        r :=pop(`returnStack`)
        $p := [p \parallel Exit_P \rightarrow r]$
    else if v is an exit vertex $GExit_P$ **then**
        $p := [p \parallel GExit_P \rightarrow Exit_{global}]$
    else
        let $w_1 \ldots w_k$ denote the successors of v
        for $(i := 1; i \leq k-1; i$++$)$ **do**
            if ($\rho_{v \rightarrow w_{i+1}}$(`numValidCompsFromExit`) $>$ `numPaths`) **then**
                break
            fi
        od
        $p := [p \parallel v \rightarrow w_i]$
        `numPaths` := `numPaths` $- \rho_{v \rightarrow w_i}$(`numValidCompsFromExit`)
    fi
od
output p

□

3.7 Handling Other Language Features

In this section, we describe how to handle some additional language features that were not explicitly addressed in the previous sections. Specifically, Section 3.7.1 discusses some of the complications that arise because of signals and signal handlers. Section 3.7.2 describes how exceptions can be handled, and Section 3.7.3 describes how to take care of indirect function calls.

3.7.1 Signals

Program signals can cause a problem for path profiling because of their asynchronous nature. For example, it is possible for a signal handler to be invoked while the program is in the middle of executing an observable path p. Because it is possible that the signal handler will never return, it is possible that the program will never complete execution of the (hypothetical) path p, and hence p will not be recorded. Furthermore, it is possible that there are paths, called *pending* paths, that are "on hold" at a recursive call site that will only be completed once control returns to the call site. Thus, the current and pending path prefixes that are active at the time of the signal will not be recorded in the profile. This is a problem if the purpose of gathering a path profile is to aid in debugging. Instead, we want to record the *prefixes* of p and the pending paths that have executed at the time the signal occurs.

For either of the interprocedural techniques described in this chapter and the next chapter this can be done as follows:

- For each procedure P, for each vertex v of P that is not a call vertex and is not $Exit_P$, add a surrogate edge $v \rightarrow GExit_P$.
- For each new surrogate edge $v \rightarrow GExit_P$, the assignment of ρ functions is done such that $\rho_{v \rightarrow GExit_P} = \langle 0, 0 \rangle$. This guarantees that whenever execution reaches the vertex v, the value in `pathNum` is the path number for the current path to v concatenated with $[v \rightarrow GExit_P \rightarrow Exit_{global}]$.
- Add a global stack of `unsigned long`s called `pendingPaths`. For each recursive call site, modify the instrumentation to push the current value of `pathNum` on `pendingPaths` before the call is made and to pop `pendingPaths` after the call returns.
- For every possible signal, a signal handler is written (or modified) in which the first action of the signal handler is to update the path profile with the current value of `pathNum` and with every value that appears on the stack `pendingPaths`. Thus, if a signal s interrupts execution of a procedure P at a vertex v, the signal handler for s will record a path that ends with $v \rightarrow GExit_P \rightarrow Exit_{global}$ and, for every pending path prefix on `pendingPaths` that ends at the call vertex c, a path that ends with $c \rightarrow GExit_P \rightarrow Exit_{global}$.

These modifications guarantee that signals will not cause a loss of information in the profile, which, as mentioned above, is important for some profiling applications. Unfortunately, they also increase the number of observable paths, which creates its own problems (see Section 3.5.4). Note that a similar technique could be used for handling signals in the intraprocedural case.

It is possible to avoid the issue of pending paths by changing Transformation 3 in the construction of G^*_{fin} (see the construction for context profiling in Section 3.2). In particular, for a recursive call site represented by vertices c and r, rather than adding the summary edge $c \rightarrow r$, the surrogate edges $c \rightarrow GExit_P$ and $Entry_P \rightarrow r$ are added.[9] In this way, every observable path that contains the summary

[9] For piecewise profiling, the edge $Entry_P \rightarrow r$ becomes $GEntry_P \rightarrow r$.

edge $c \rightarrow r$ is split into two observable paths. The instrumentation code at a recursive call site records the path that ends with $c \rightarrow GExit_P \rightarrow Exit_{global}$ before making the recursive call, and begins recording a new path when the recursive call returns. When this version of Transformation 3 is used, there are never any pending paths during runtime.

3.7.2 Exceptions

Programming languages that have exceptions (*e.g.*, C++, Java) can cause complications, particularly for intraprocedural path-profiling techniques. In particular, consider a procedure P that calls procedure Q, which in turn calls procedure R. If R throws an exception that is caught in P, then there will be an incomplete path in Q that is not recorded. One way to address this is to break every observable path at each call site.

In an interprocedural path-profiling technique, exceptions can be handled by adding a surrogate edge from the $Entry_P$ vertex (or $GEntry_P$ vertex, depending on the interprocedural technique being used) to the vertex v where the exception is caught, and adding a surrogate edge from the vertex u where the exception is thrown to $GExit_R$. (Here P is the procedure containing v, and R is the procedure containing u.) When the exception is thrown, the profiling instrumentation updates the profile for the path that ends with $[u \rightarrow GExit_R \rightarrow Exit_{global}]$. When the exception is caught, the profiling instrumentation uses the edge $Entry_P \rightarrow v$ (or the edge $GEntry_P \rightarrow v$) to begin recording a new observable path.

Note that in the interprocedural path-profiling techniques we must also deal with the issue of pending path prefixes. This can be handled in the same way it is handled in Section 3.7.1: a surrogate edge $c \rightarrow GExit_P$ is added to each recursive call vertex c in each procedure P, the assignment of ρ functions is done such that the $\rho_{c \rightarrow GExit_P}$ functions are all $\langle 0, 0 \rangle$, and a stack of pending path prefixes is maintained. When an exception is thrown, the profile is updated for each value on the stack of pending path prefixes. A stack of pending path prefixes can also be used to handle exceptions for intraprocedural path profiling techniques (where pending paths may include paths that are "on hold" at call sites).

3.7.3 Indirect Procedure Calls

The easiest way to handle indirect procedure calls is to treat them as recursive procedure calls, and not allow interprocedural paths that cross through an indirect procedure call. Another possibility is to turn each indirect procedure call through a procedure variable `fp` into an if-then-else chain that has a separate (direct) procedure call for each possible value of `fp`. Well-known techniques (*e.g.*, such as flow insensitive points-to analysis [6, 59, 57]) can be used to obtain a reasonable (but still conservative) estimate of the values that `fp` may take on.

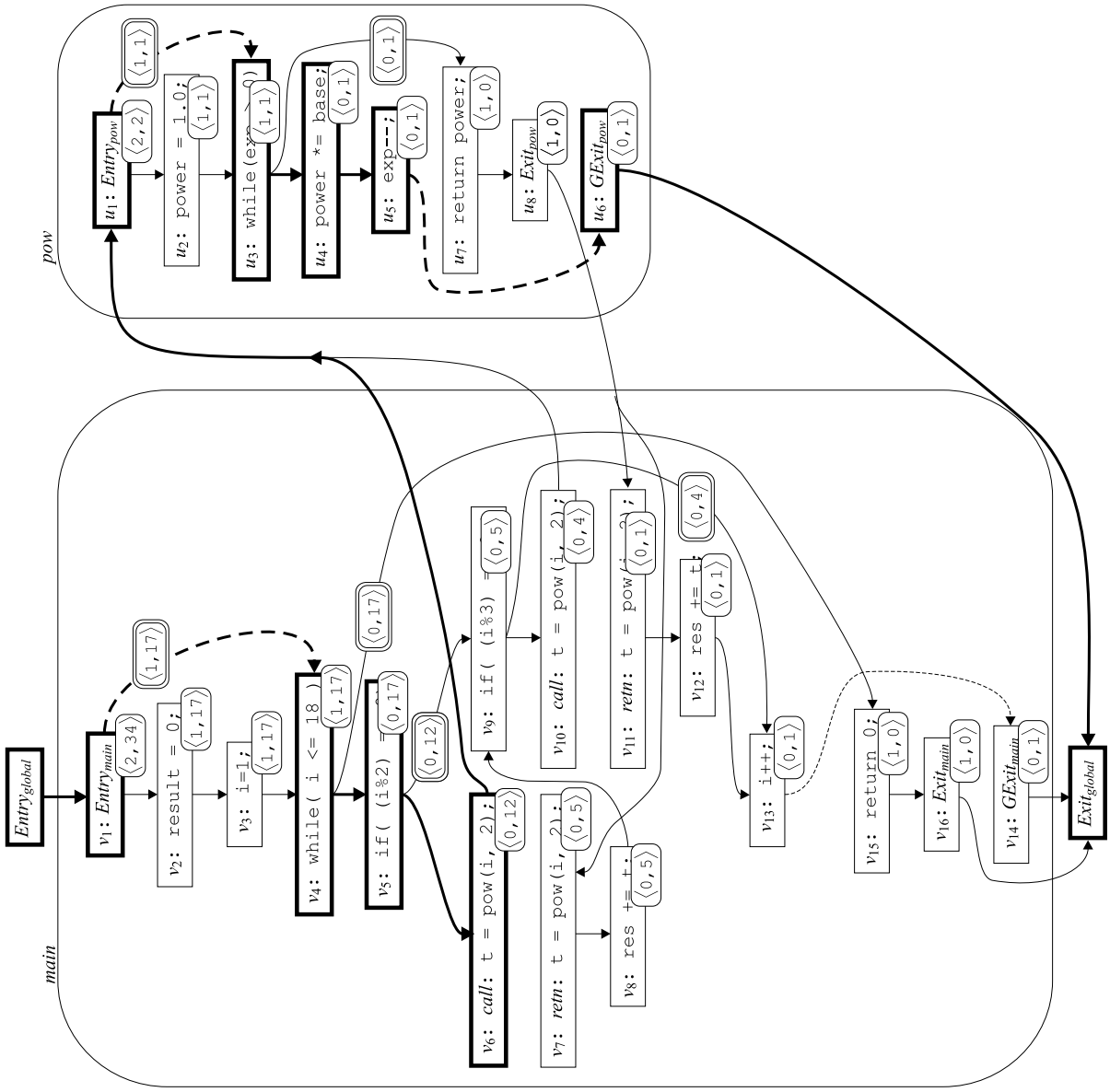


Figure 9: G^*_{fin} for the code in Fig. 8. Dashed edges represent surrogate edges; the supergraph for the program in Fig. 8 includes the backedges $v_{13} \rightarrow v_4$ and $u_5 \rightarrow u_3$, which have been removed here by Transformation 2. Here, the ordered pair $\langle a, b\rangle$ represents the linear function $\lambda x.a \cdot x + b$. Each vertex v is assigned the linear function ψ_v, which is shown in a rounded box. Each intraprocedural edge e is assigned the linear function ρ_e, which is shown in a doubled, rounded box. Unlabeled intraprocedural edges have the function $\langle 0, 0\rangle$. Interprocedural edges do not have ρ functions. Path number 24 has been shown in bold.

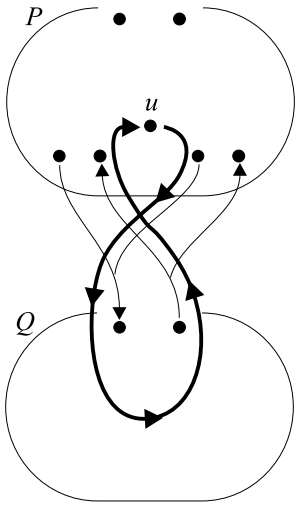


Figure 10: Example of an invalid cycle in a program supergraph.

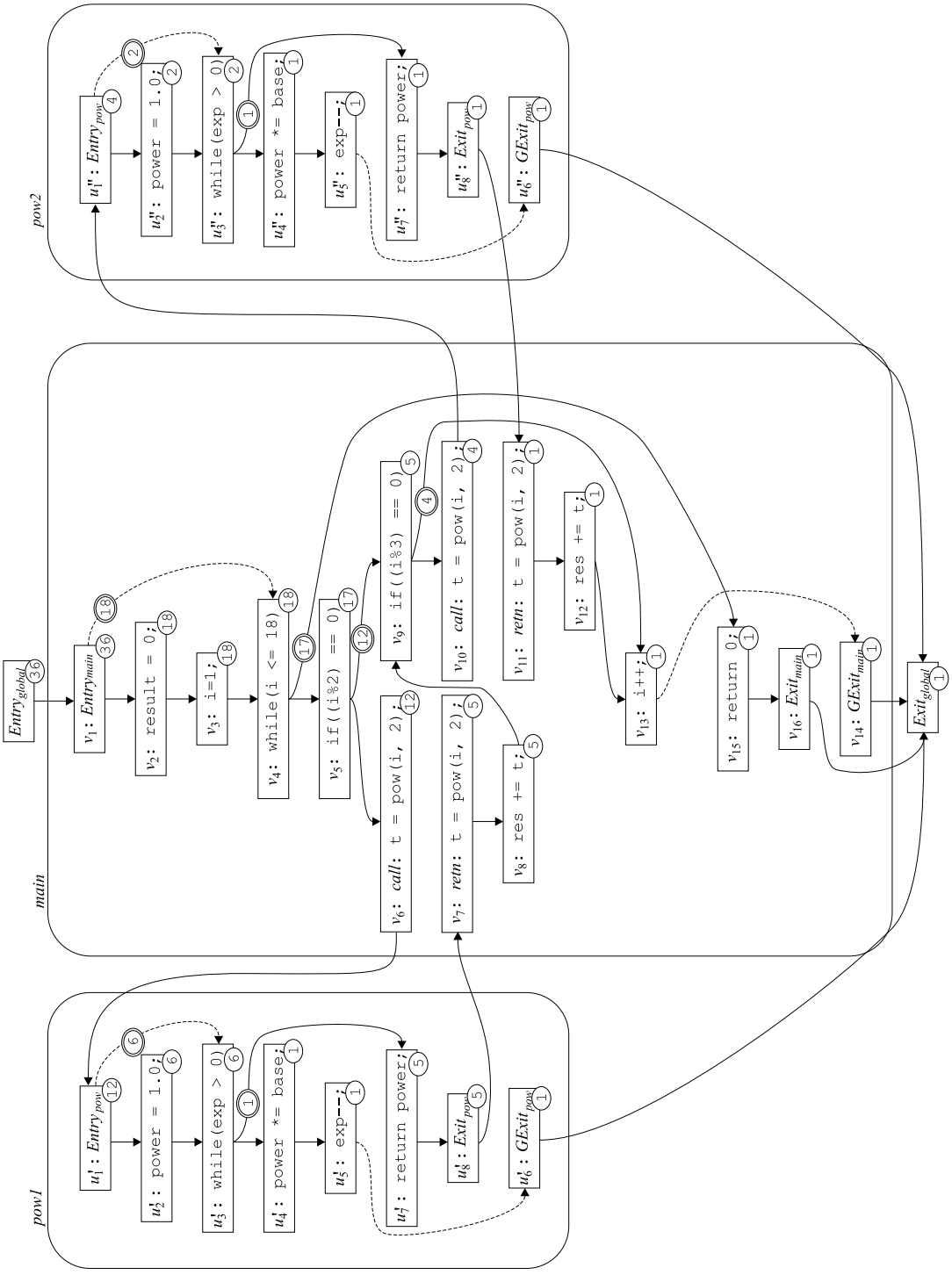


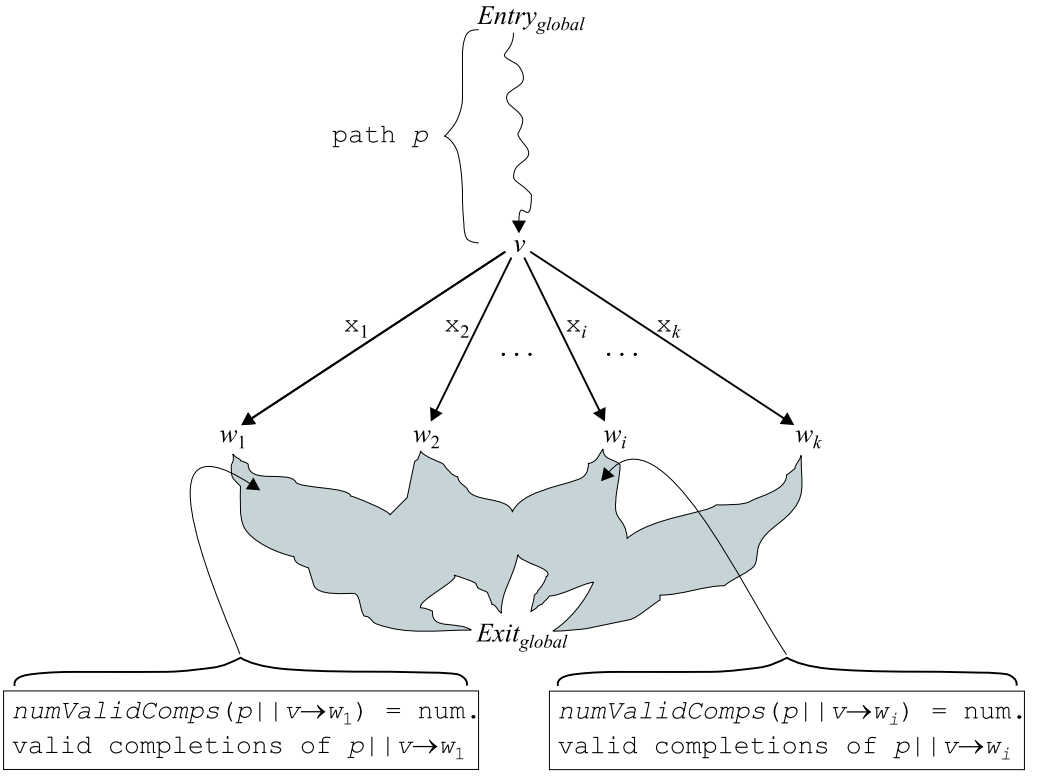
Figure 11: Modified version of G^*_{fin} from Fig. 9 with two copies of *pow*. Labels on the vertices and edges show the results of applying the Ball-Larus numbering technique to the graph. Each vertex label is shown in a circle, and each edge label is shown in a double circle. Unlabeled edges are given the value 0 by the Ball-Larus numbering scheme.

```
unsigned int profile[36];    /* 36 possible paths in total */
double pow(double base, long exp,
           unsigned int &pathNum, unsigned int numValidCompsFromExit){
   unsigned int pathNumOnEntry = pathNum; /* Save the calling context */
   double power = 1.0;
   while( exp > 0 ) {
      power *= base;
      exp--;
      profile[pathNum]++;
      /* From surrogate edge u1->u3: */
      pathNum = 1 * numValidCompsFromExit + 1 + pathNumOnEntry;
   }
   pathNum += 0 * numValidCompsFromExit + 1; /* From edge u3->u7 */
   return power;
}
```

Figure 12: Part of the instrumented version of the program from Fig. 8. Instrumentation code is shown in italics. (See also Fig. 13.)

```
int main() {
   unsigned int pathNum = 0;
   unsigned int pathNumOnEntry = 0;
   unsigned int numValidCompsFromExit = 1;
   double t, result = 0.0;
   int i = 1;
   while( i <= 18 ) {
      if( (i%2) == 0 ) {
         t = pow( i, 2, pathNum, 0 * numValidCompsFromExit + 5 );
         /* On entry to pow: pathNum is 0 or 18; fourth arg. always 5 */
         /* On exit from pow: pathNum is 1, 7, 19, or 25 */
         result += t;
      } else
         pathNum += 0 * numValidCompsFromExit + 12;
      if( (i%3) == 0 ) {
         t = pow( i, 2, pathNum, 0 * numValidCompsFromExit + 1 );
         /* On entry to pow: pathNum is 1, 7, 12, 19, 25, or 30; 4th arg. always 1 */
         /* On exit from pow: pathNum is 2, 3, 8, 9, 13, 14, 20, 21, 26, 27, 31, or 32 */
         result += t;
      } else
         pathNum += 0 * numValidCompsFromExit + 4;     /* From edge v9->v13 */
      i++;
      profile[pathNum]++;
      /* From surrogate edge v1->v4: */
      pathNum = 1 * numValidCompsFromExit + 17 + pathNumOnEntry;
   }
   pathNum += 0 * numValidCompsFromExit + 17;          /* From edge v4->v15 */
   profile[pathNum]++;
   for (i = 0; i < 36; i++) {
      cout.width(3); cout << i << ":"; cout.width(2); cout << profile[i] << "  ";
      if ((i+1) % 9 == 0) cout << endl;
   }
   return 0;
}
```

Figure 13: Part of the instrumented version of the program from Fig. 8. Instrumentation code is shown in italics. (See also Fig. 12.)



x_1 = *edgeValueInContext*$(p, v \to w_1)$ = 0

x_2 = *edgeValueInContext*$(p, v \to w_2)$ = *numValidComps*$(p||v \to w_1)$

$\vdots$

x_i = *edgeValueInContext*$(p, v \to w_i)$ = $\sum_{j<i}$ *numValidComps*$(p||v \to w_j)$

$\vdots$

x_k = *edgeValueInContext*$(p, v \to w_k)$ = $\sum_{j<k}$ *numValidComps*$(p||v \to w_j)$

Figure 14: Illustration of the definition of *edgeValueInContext* given in Equation (5).

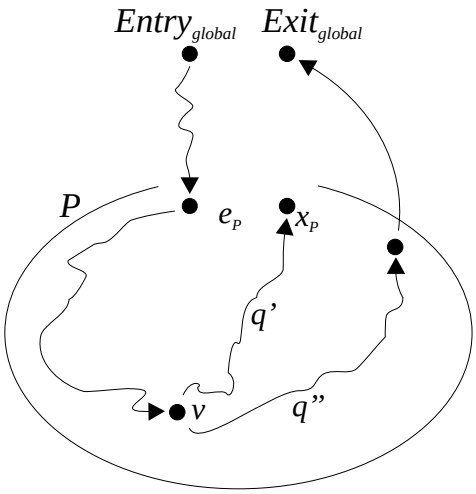


Figure 15: Schematic that illustrates the paths used to motivate the ψ functions. Vertices with labels of the form e_P, x_P, and g_P represent the vertices $Entry_P$, $Exit_P$, and $GExit_P$, respectively.

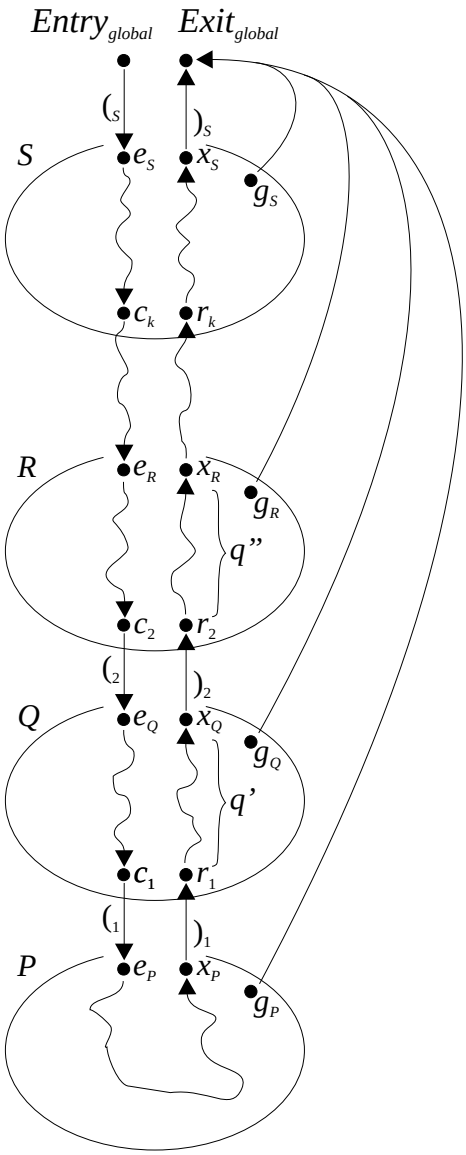


Figure 16: Schematic of the paths used to explain the use of ψ functions to compute *numValidComps*(q), where q is an unbalanced-left path from $Entry_{global}$ to $Exit_P$. The unbalanced-left path s starts at $Entry_{global}$ and ends at $Exit_S$, and has q as a prefix. Vertices with labels of the form e_P, x_P, and g_P represent the vertices $Entry_P$, $Exit_P$, and $GExit_P$, respectively.

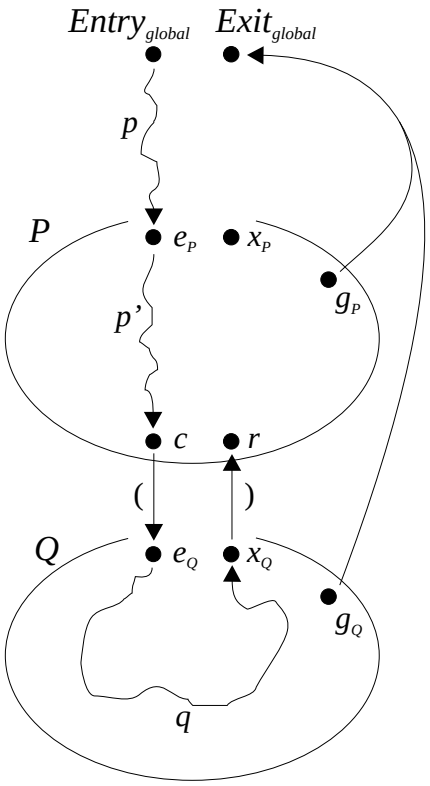


Figure 17: Paths used to illustrate the correctness of Equation (21).

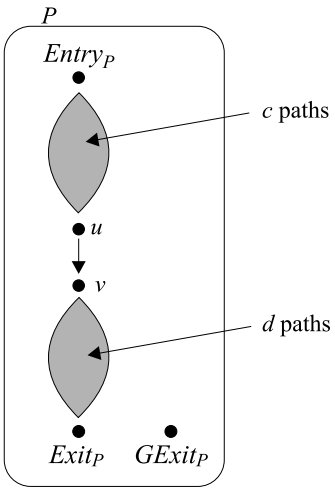


Figure 18: Example showing the effect of breaking an edge $u \rightarrow v$ on the number of paths in procedure P.

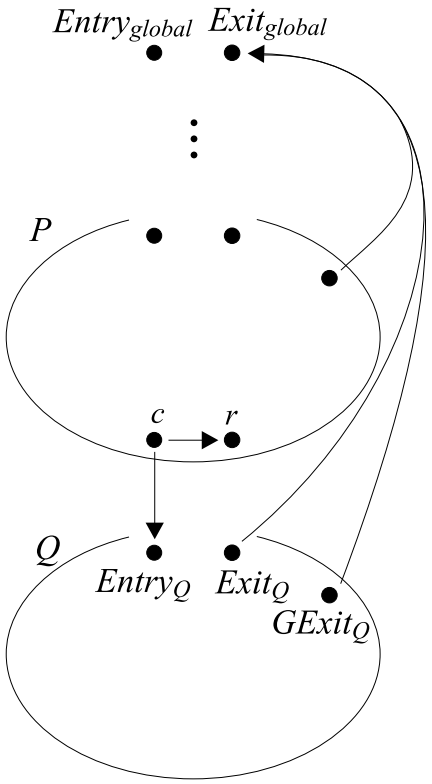


Figure 19: Schematic of G^*_{fin} with a call-site where the return-edge has been replaced by a surrogate edge, but not the call-edge.

Chapter 4

The Functional Approach to Interprocedural Piecewise Path Profiling

As mentioned in Section 3.3.1, the technique described in Chapter 3 is a functional approach to interprocedural *context* path-profiling. In particular, an unbalanced-left path p through G^*_{fin} may contain a non-empty context-prefix that summarizes the context in which the active-suffix of p occurs. In an intraprocedural context path profile, an observable path can include a context prefix that summarizes the path taken to a loop header. In *piecewise path profiling*, we define a set of observable paths such that any execution path — a same-level valid path from $Entry_{global}$ to $Exit_{global}$ in the program supergraph — is the concatenation of a sequence of observable paths. A piecewise path profile reports how many times each observable path occurs as a subpath of a given execution path.

This chapter shows how to modify the technique of Chapter 3 to give a technique that collects an interprocedural *piecewise* path profile. Since we are interested in observable paths that may begin and end in the middle of an execution path, we are interested in *unbalanced-right-left paths*. Unbalanced-right-left paths are defined in terms of *unbalanced-right paths*, which are the dual of unbalanced-left paths. A path p is called an *unbalanced-right path* if and only if the string formed by concatenating the labels of p's edges can be derived from the non-terminal *UnbalRight* in the following the context-free grammar:

UnbalRight	::=	*UnbalRight* $)_c$ *UnbalRight*	for each call vertex c
	|	*UnbalRight* $)_P$ *UnbalRight*	for each procedure P
	|	*SLVP*	

An unbalanced-right path represents part of an incomplete execution sequence. Specifically, an unbalanced-right path p may leave a procedure P (where the execution sequence that reached P is not part of p). For example, a path that begins in a procedure P, crosses a return-edge to a procedure Q, and ends in Q is an unbalanced-right path.

An unbalanced-right-left path is the concatenation of an unbalanced-right path and an unbalanced-left path. Specifically, a path is called an unbalanced-right-left path if and only if the string formed by concatenating the labels of p's edges can be derived from the non-terminal *UnbalRtLf* in the following context-free grammar:

UnbalRtLf ::= *UnbalRight* *UnbalLeft*

An unbalanced-right-left path p may leave some number of procedures (where the execution sequence that reached those procedures is not part of p) and then enter some number of procedures (where the execution that leaves those procedures is not part of p). For example a path that begins in a procedure P, crosses a return-edge to a procedure Q, then crosses a call-edge to procedure R and ends in R is an unbalanced-right-left path. Note also that every same-level-valid path, unbalanced-left path, and unbalanced-right path is also an unbalanced-right-left path.

One way to see that we are interested in unbalanced-right-left paths for interprocedural piecewise path-profiling is to compare them with the observable paths used for interprocedural context path-profiling: to modify an observable path p from the context path-profiling technique for use in a piecewise path-profiling technique, we are interested in throwing out the prefix of p that contains the context-prefix, and keeping the suffix of p that has the active-suffix; the suffix of an unbalanced-left (observable) path that is used for context profiling is an unbalanced-right-left path.

The algorithm for piecewise path profiling uses a slightly different version of G^*_{fin} from that used in the previous chapter. In particular, a new transformation must be performed before the transformations given in Section 3.2:

Transformation 0: For each procedure P add the special vertex $GEntry_P$ to the flow graph for P and an edge $Entry_{global} \rightarrow GEntry_P$.

In addition, Transformations 2 and 3 are modified so that each surrogate edge of the form $Entry_P \rightarrow v$ is replaced by a surrogate edge of the form $GEntry_P \rightarrow v$. For the purpose of classifying a path as a same-level valid path, an unbalanced-left path, an unbalanced-right path, or an unbalanced-right-left path, edges of the form $Entry_{global} \rightarrow Entry_P$ are labeled "$(_P$", edges of the form $Exit_P \rightarrow Exit_{global}$ are labeled "$)_P$", and edges of the form $Entry_{global} \rightarrow GEntry_P$ and $GExit_P \rightarrow Exit_{global}$ are labeled "e". The observable paths correspond to the unbalanced-right-left paths from $Entry_{global}$ to $Exit_{global}$ in the graph G^*_{fin} created by Transformations 0–3.

Example 4.0.1 Figure 20 shows the graph G^*_{fin} that is constructed for collecting a piecewise profile for the program in Figure 8. In Figure 20, the vertices v_{17} and u_9 are inserted by Transformation 0. The vertices v_{14} and u_6 are inserted by Transformation 1. The edges $v_{17} \rightarrow v_4$ and $v_{13} \rightarrow v_{14}$ are inserted by (the modified) Transformation 2 (and the backedge $v_{13} \rightarrow v_4$ is removed). Similarly, Transformation 2 adds the edges $u_9 \rightarrow u_3$ and $u_5 \rightarrow u_6$, and removes the backedge $u_5 \rightarrow u_3$. Transformation 3 is not illustrated in this example, because the program is non-recursive. □

As noted above, under this construction of G^*_{fin}, the observable paths correspond to unbalanced-right-left paths through G^*_{fin}. In particular, the observable paths no longer correspond just to the unbalanced-left paths in G^*_{fin}: an unbalanced-right-left path that begins $Entry_{global} \rightarrow GEntry_P \rightarrow v \rightarrow \ldots$ corresponds to an observable path that begins at vertex v (*i.e.*, in the middle of procedure P). For instance, consider the following path in Figure 20:

$$Entry_{global} \rightarrow u_9 \rightarrow u_3 \rightarrow u_7 \rightarrow u_8 \rightarrow v_7 \rightarrow v_8 \rightarrow v_9 \rightarrow v_{10} \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4 \rightarrow u_5 \rightarrow u_6 \rightarrow Exit_{global}$$

This path corresponds to an execution sequence that begins at vertex u_3 in *pow*, returns to the first call site in *main*, and then re-enters *pow* from the second call site in *main*. The sequence of parentheses generated by this path consists of an unmatched right parenthesis on the return-edge $u_4 \rightarrow v_7$ and an unmatched left parenthesis on the call-edge $v_{10} \rightarrow u_1$. Thus, the path is an unbalanced-right-left path, but is not an unbalanced-left path nor a same-level valid path.

4.1 Numbering Unbalanced-Right-Left Paths in G^*_{fin}

In this section, we describe how to obtain a dense numbering of the unbalanced-right-left paths in G^*_{fin}. The number of unbalanced-right-left paths in G^*_{fin} is finite. Thus, the graph G^*_{fin} together with the

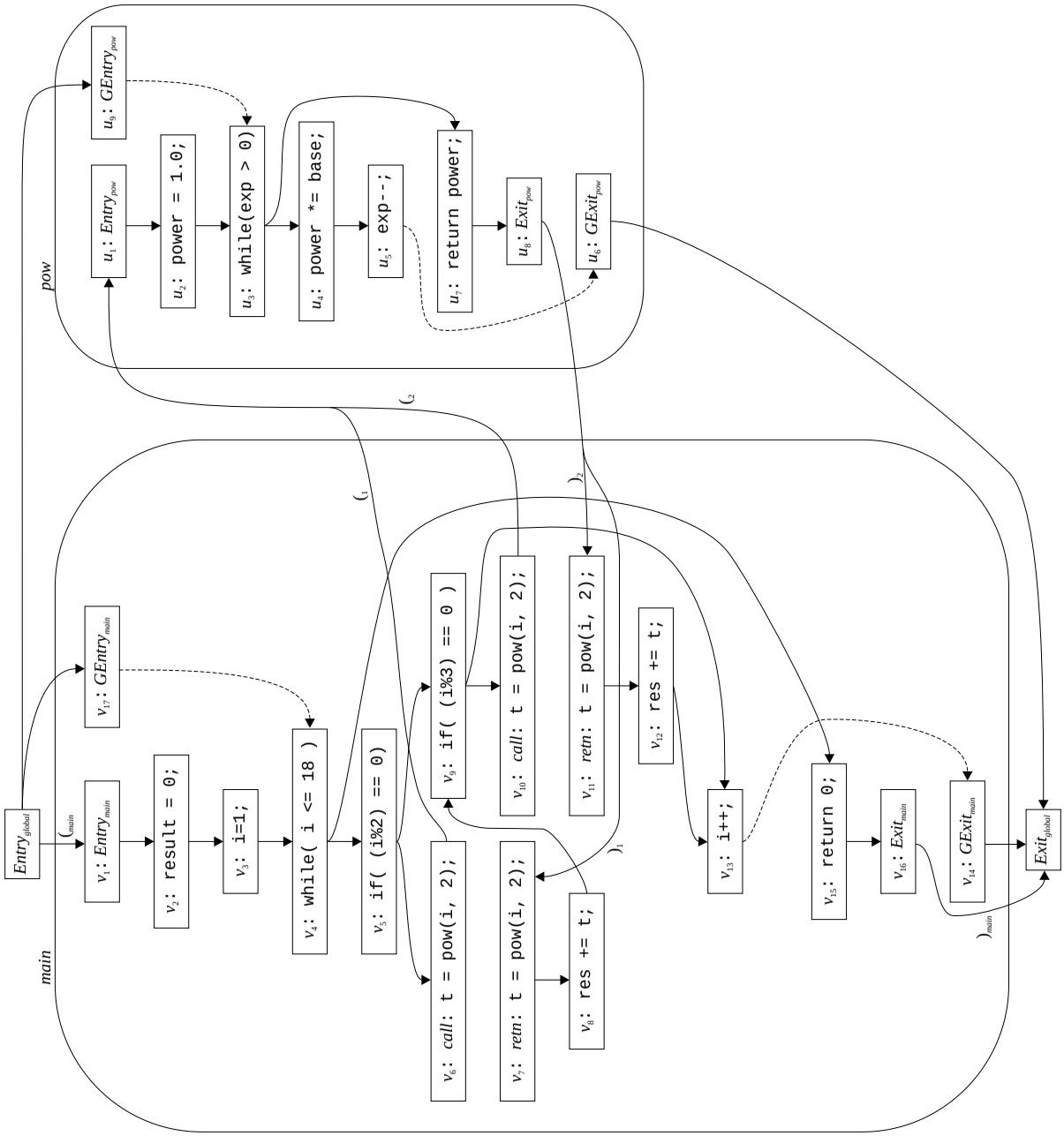


Figure 20: G^*_{fin} for piecewise-profiling instrumentation for the program given in Figure 8. Dashed edges represent surrogate edges; the supergraph for the program in Figure 8 includes the backedges $v_{13} \rightarrow v_4$ and $u_5 \rightarrow u_3$, which have been removed here by Transformation 2.

context-free grammar for unbalanced-right-left strings of parentheses constitute a finite-path graph. We will use the technique presented in Section 3.4 for numbering L-paths to number unbalanced-right-left paths in the modified G^*_{fin}. For this section, the L-paths of Section 3.4 correspond to unbalanced-right-left paths. The function *numValidComps* takes an unbalanced-right-left path p that starts at $Entry_{global}$ and returns the number of valid completions of p. The definitions of *edgeValueInContext* and path number are exactly as in Section 3.4.

The task of computing *numValidComps* and *edgeValueInContext* for an unbalanced-right-left path is similar to the task of computing these functions for an unbalanced-left path in Section 3.5. Let p be an unbalanced-right-left path from $Entry_{global}$ to a vertex v in procedure P. Our technique is based on the following observations (which are essentially the same as those made in Section 3.5):

1. The number of valid completions of p, *numValidComps*(p), is determined by the sequence of unmatched left parentheses in p and the vertex v. If $v = Exit_P$ and p contains an unmatched left parenthesis, then there is only one valid successor of p: the return vertex r such that $Exit_P \rightarrow r$ closes the last open parenthesis in p. Otherwise, any successor of v is a valid successor of p.

2. For a vertex u in P, the value of *numValidComps*$(p \parallel q)$ is the same for *any* same-level valid path q from v to u. In particular, this holds for $u = Exit_P$.

3. If $v = GExit_P$, then the number of completions of p is 1, because $GExit_P \rightarrow Exit_{global}$ is the only path out of $GExit_P$.

4. If $w_1, \ldots, w_k$ are the valid successors of p, then the value of *numValidComps*(p) is given by the following sum:

$$\textit{numValidComps}(p) = \sum_{i=1}^{k} \textit{numValidComps}(p \parallel v \rightarrow w_i).$$

These observations imply that the ψ and ρ functions described in Section 3.5 are also useful for interprocedural piecewise path profiling. Let q be any same-level valid path from v to $Exit_P$. This means that

$$\textit{numValidComps}(p) = \psi_v(\textit{numValidComps}(p \parallel q)).$$

Furthermore, for an intraprocedural edge $v \rightarrow w$, we have the following:

$$\textit{edgeValueInContext}(p, v \rightarrow w) = \rho_{v \rightarrow w}(\textit{numValidComps}(p \parallel q)).$$

As we will see in Sections 4.2 and 4.3, this allows us to use the same device we used in Sections 3.5.5 and 3.6—namely, to maintain a value `numValidCompsFromExit` so that *edgeValueInContext*$(p, v \rightarrow w)$ can be computed efficiently, as $\rho_{v \rightarrow w}$(`numValidCompsFromExit`.

Even though the same ψ and ρ functions will be used, there are two key differences in how the functions *numValidComps* and *edgeValueInContext* are computed when dealing with unbalanced-right-left paths:

1. The first difference is in how *numValidComps* is computed for an unbalanced-right-left path that ends with an $Exit_P$ vertex.

2. The second difference is in how *edgeValueInContext* is computed for an interprocedural edge.

Both of these differences stem from the fact that an unbalanced-right-left path p from $Entry_{global}$ to $Exit_P$ may have no unmatched left parentheses (*i.e.*, p may be an unbalanced-right path). In contrast, in Section 3.5, when dealing with unbalanced-left paths, we never "ran out" of unmatched left parentheses.

4.1.1 Calculating *numValidComps* from $Exit_P$

Let q be an unbalanced-right-left path from $Entry_{global}$ to $Exit_P$. In this section, we discuss how to compute *numValidComps*(q). If q contains an unmatched left parenthesis, then there is only one valid successor of q: the return vertex r_1 such that the edge $Exit_P \rightarrow r_1$ matches the last unmatched left parenthesis of q. This gives us the following:

$$\textit{numValidComps}(q) = \textit{numValidComps}(q \parallel Exit_P \rightarrow r_1).$$

Suppose r_1 occurs in procedure Q. Then the above value is equal to

$$\psi_{r_1}(\textit{numValidComps}(q \parallel Exit_P \rightarrow r_1 \parallel q')),$$

where q' is any same-level valid path from r_1 to $Exit_Q$. Recall that the function ψ_{r_1} counts the valid completions of p that exit Q via $GExit_Q$, even though it only takes as an argument the number of valid completions for paths that exit Q via $Exit_Q$. As before, if $[q \parallel Exit_P \rightarrow r_1 \parallel q']$ has an unmatched left parenthesis, then it will have only one valid successor: the return vertex r_2 such that $Exit_Q \rightarrow r_2$ is labeled with the parenthesis that closes the second-to-last unmatched left parenthesis in q. Suppose that r_2 is in procedure R. Then the above value becomes

$$\psi_{r_1}(\psi_{r_2}(\textit{numValidComps}(q \parallel Exit_P \rightarrow r_1 \parallel q' \parallel Exit_Q \rightarrow r_2 \parallel q''))),$$

where q'' is any same-level valid path from r_2 to $Exit_R$. Again, ψ_{r_2} counts valid completions that leave R via either $GExit_R$ or $Exit_R$. This argument can be continued until a path s has been constructed from $Entry_{global}$ to $Exit_S$, and one of two cases holds:

Case 1: The parenthesis "$(_S$" that appears on the edge $Entry_{global} \rightarrow Entry_S$ is the only unmatched left parenthesis in s. In this case, the only valid successor of s is $Exit_{global}$, and the number of valid completions of s is 1. This means that

$$\begin{aligned}\textit{numValidComps}(q) &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(\textit{numValidComps}(s))\ldots)) \\ &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(1)\ldots))\end{aligned}$$

where $r_1 \ldots r_k$ is the sequence of return vertices determined by the unmatched left parentheses in q. (Note that this equation is the same as Equation (8), which makes sense because q must have been an unbalanced-left path.)

Case 2: There are no unmatched left parentheses in s. (In this case, q does not contain the parenthesis "$(_S$"; note that this case cannot happen for a path q in Section 3.5, because each unbalanced-left path q starts with a parenthesis of the form "$(_S$".) In this case, s is an unbalanced-right path (which has the unbalanced-right-left path q as a prefix), and so every return vertex that is a successor of $Exit_S$ is a valid successor of s. Furthermore, any unbalanced-right-left path from $Exit_S$ to $Exit_{global}$ is a valid completion of the path s. (In contrast, consider an unbalanced-left path p that is used for interprocedural context path profiling: each valid completion of p is an unbalanced-right-left path; however, only an unbalanced-right-left path p' where the sequence of unmatched right parentheses in p' match the sequence of unmatched left parentheses in p can be a valid completion of p.)

We define the value *numUnbalRLPaths*$[v]$ to be the total number of unbalanced-right-left paths from the vertex v to $Exit_{global}$. This gives us the following:

$$\begin{aligned}\textit{numValidComps}(q) &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(\textit{numValidComps}(s))\ldots)) \\ &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(\textit{numUnbalRLPaths}[Exit_S])\ldots))\end{aligned}$$

where $r_1 \ldots r_k$ is the sequence of return vertices determined by the unmatched left parentheses in q. Note that if there are no unmatched left parentheses in q, then $Exit_S = Exit_P$ and the above equation simplifies to:

$$\textit{numValidComps}(q) = \textit{numUnbalRLPaths}[Exit_P]$$

We now show how to compute *numUnbalRLPaths*$[v]$. First, note that if *numUnbalRLPaths*$[Exit_P]$ is known, then the ψ functions can be used to compute *numUnbalRLPaths*$[v]$ for any vertex v in procedure P:

$$\textit{numUnbalRLPaths}[v] = \psi_v(\textit{numUnbalRLPaths}[Exit_P])$$

This follows from the definition of ψ_v: let p be any unbalanced-right-left path from $Entry_{global}$ to v such that p does not contain any unmatched left parentheses and let q be a same-level valid path from v to $Exit_P$—thus, p is an unbalanced-right path that starts with an edge $Entry_{global} \rightarrow GEntry_{P'}$. The number of valid completions of p is equal to the number of unbalanced-right-left paths from v to $Exit_{global}$. This implies that

$$\begin{aligned}\textit{numUnbalRLPaths}[v] &= \textit{numValidComps}(p)\\ &= \psi_v(\textit{numValidComps}(p \parallel q))\\ &= \psi_v(\textit{numUnbalRLPaths}[Exit_P])\end{aligned}$$

The value of *numUnbalRLPaths*$[Exit_P]$ is given by the following equation:

$$\textit{numUnbalRLPaths}[Exit_P] = \sum_{r \in \textit{succ}(Exit_P)} \textit{numUnbalRLPaths}[r] \tag{29}$$

where *numUnbalRLPaths*$[r]$ is given by

$$\textit{numUnbalRLPaths}[r] = \begin{cases} 1 & \text{if } r = Exit_{global} \\ \psi_r(\textit{numUnbalRLPaths}[Exit_Q]) & \text{if } r \text{ is a return-site vertex in procedure } Q \end{cases} \tag{30}$$

Equations (29) and (30) can be used to compute *numUnbalRLPaths*$[Exit_P]$ for each $Exit_P$ vertex and *numUnbalRLPaths*$[r]$ for each return-site vertex r during a traversal of the call graph associated with G^*_{fin} in topological order: Because the call graph associated with G^*_{fin} is acyclic, whenever a vertex $Exit_P$ is reached, each value *numUnbalRLPaths*$[r]$ that is needed to compute *numUnbalRLPaths*$[Exit_P]$ will be available.

Computing *edgeValueInContext* for interprocedural edges

For an unbalanced-right-left path p from $Entry_{global}$ to $Exit_P$, and an edge $Exit_P \rightarrow r$, the value of *edgeValueInContext*$(p, Exit_P \rightarrow r)$ is not always zero, as it was in Section 3.5.3. Let $r_1 \ldots r_k$ be the successors of $Exit_P$. If the path p does contain an unmatched left parenthesis, then there is only a single r_i that is a valid successor of p. This means that

$$\textit{edgeValueInContext}(p, Exit_P \rightarrow r_i) = 0.$$

Now suppose that p has no unmatched left parentheses (*i.e.*, p is an unbalanced-right path). In this case, every r_i is a valid successor of p. Then the definition of *edgeValueInContext* in Equation (5) yields the following:

$$\textit{edgeValueInContext}(p, Exit_P \rightarrow r_i) = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{j<i} \textit{numUnbalRLPaths}[r_j] & \text{otherwise} \end{cases} \tag{31}$$

Notice that the value computed by Equation (31) is the same for any unbalanced-right-left path p to $Exit_P$ that has no unmatched left parentheses (*i.e.*, for any unbalanced-right path). For an edge $Exit_P \rightarrow r_i$, we define *edgeValue*$[Exit_P \rightarrow r_i]$ to be the value computed by the right-hand side of Equation (31) (for any unbalanced-right-left path p to $Exit_P$ that has no unmatched left parentheses).

As in Section 3.5.3, we must also calculate *edgeValueInContext* for the path $[\epsilon : Entry_{global}]$ and an edge $Entry_{global} \rightarrow v$. We observe that

$$\textit{numValidComps}(Entry_{global} \rightarrow Entry_P) = \psi_{\textit{Entry}_P}(1)$$

and

$$\textit{numValidComps}(Entry_{global} \rightarrow GEntry_P) = \psi_{\textit{GEntry}_P}(\textit{numUnbalRLPaths}[Exit_P]).$$

The first of these equations is discussed in Section 3.5.3. The second holds because the path consisting of the edge $Entry_{global} \rightarrow GEntry_P$ is (trivially) an unbalanced-right path. With these observations, it is possible to apply Equation (5) to find the appropriate values of *edgeValueInContext*$([\epsilon : Entry_{global}], Entry_{global} \rightarrow v)$ for the $Entry_{global} \rightarrow Entry_P$ and $Entry_{global} \rightarrow GEntry_P$ edges.

Figure 21 shows a labeled version of G^*_{fin} from Figure 20. Figure 21 shows the values on the interprocedural edges that are calculated as discussed in this section. In addition, *numUnbalRLPaths* is shown for certain vertices.

In general, the total number of unbalanced-right-left paths through G^*_{fin} is given by

$$\textit{numValidComps}([\epsilon : Entry_{global}]) = \sum_{v \in \textit{succ}(\textit{Entry}_{global})} \textit{numValidComps}(Entry_{global} \rightarrow v)$$

where

$$\textit{numValidComps}(Entry_{global} \rightarrow Entry_P) = \psi_{\textit{Entry}_P}(1)$$

and

$$\textit{numValidComps}(Entry_{global} \rightarrow GEntry_P) = \psi_{\textit{GEntry}_P}(\textit{numUnbalRLPaths}[\textit{GExit}_P]).$$

For the graph G^*_{fin} shown in Figure 21, the total number of unbalanced-right-left paths from $Entry_{global}$ to $Exit_{global}$ is

$$\begin{aligned}
\textit{numValidComps}([\epsilon : Entry_{global}]) &= \textit{numValidComps}(Entry_{global} \rightarrow Entry_{main}) + \\
&\quad \textit{numValidComps}(Entry_{global} \rightarrow GEntry_{main}) + \\
&\quad \textit{numValidComps}(Entry_{global} \rightarrow GEntry_{pow}) \\
&= \psi_{\textit{Entry}_{main}}(1) + \psi_{\textit{GEntry}_{main}}(1) + \psi_{\textit{GEntry}_{pow}}(4) \\
&= 8 + 8 + 5 \\
&= 21
\end{aligned}$$

4.1.2 Practical Considerations When Numbering Unbalanced-Right-Left Paths

For practical purposes, it is important to limit the number of unbalanced-right-left paths through G^*_{fin} to 2^{WordSize}, where WordSize is the number of bits in a machine word, (*i.e.*, on typical commercial hardware). As in Section 3.5.4, this is done by breaking edges in G^*_{fin}. However, limiting the number of unbalanced-right-left paths in G^*_{fin} is more complicated than limiting the number of unbalanced-left paths in G^*_{fin}, as in Section 3.5.4. There are the following issues:

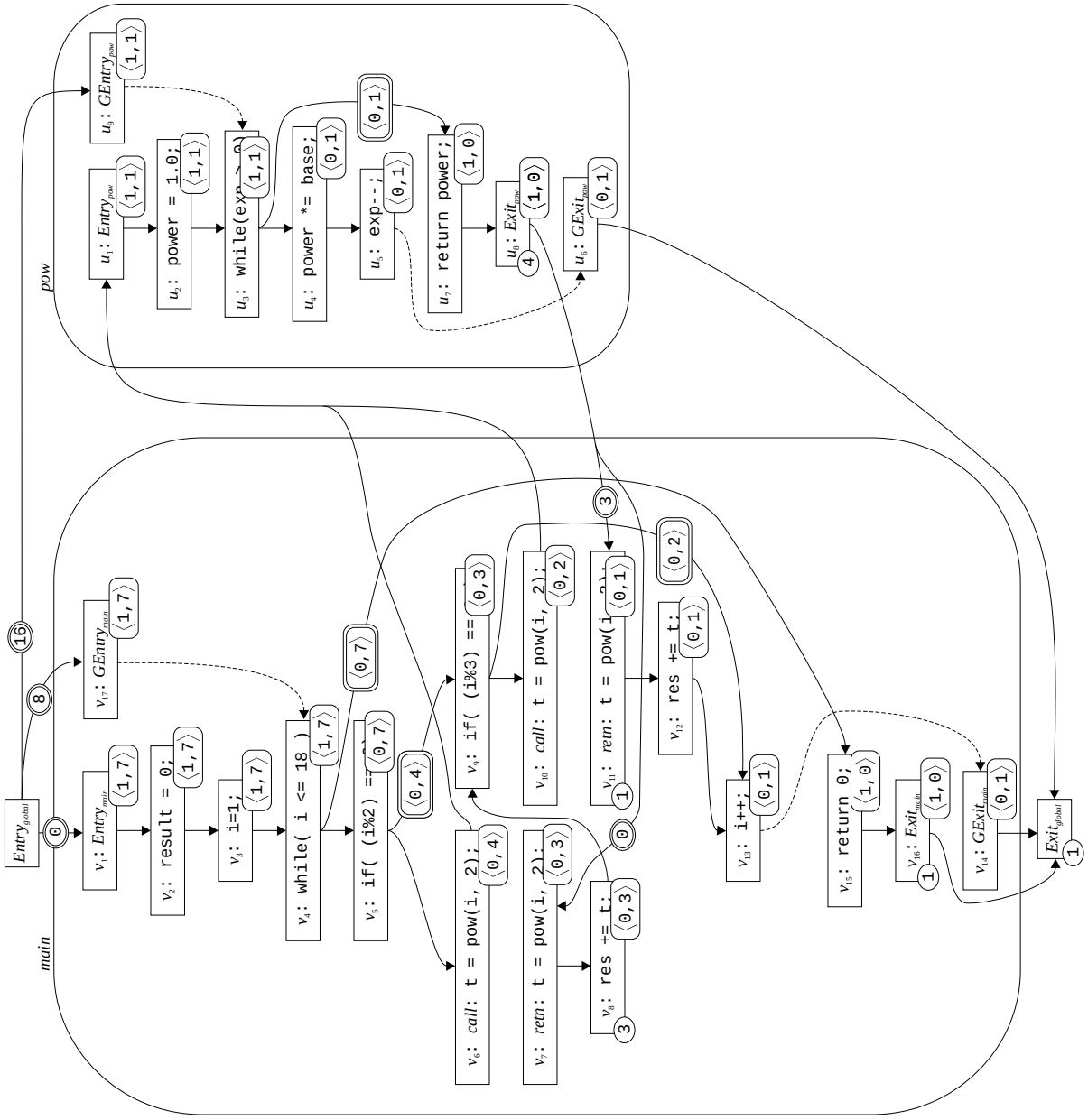


Figure 21: Labeled version of G^*_{fin} from Figure 20. For each intraprocedural vertex v, the function ψ_v is shown in a rounded box. For each intraprocedural edge e, the function ρ_e is shown in a doubled, rounded box; intraprocedural edges that are not labeled have the function $\langle 0, 0\rangle$. For the exit vertices $Exit_{global}$, $Exit_{main}$, and $Exit_{pow}$, and the return-site vertices v_7 and v_{11}, the value of *numUnbalRLPaths* is shown in a circle. For each interprocedural edge $v \to w$, the value of *edgeValueInContext*$(p, v \to w)$ for an unbalanced-right path p ending at v is shown in a doubled rounded circle. There are a total of $\psi_{Entry_{main}}(1) + \psi_{GEntry_{main}}(1) + \psi_{GEntry_{pow}}(4) = 8 + 8 + 5 = 21$ unbalanced-right-left paths from $Entry_{global}$ to $Exit_{global}$ in G^*_{fin}.

1. Calculating the number of unbalanced-right-left paths is done in two passes over G^*_{fin}, called Pass I and Pass II in the remainder of this section. (Recall that only a single pass over G^*_{fin} is required to calculate the number of unbalanced-left paths.) During Pass I, the ψ and ρ functions are calculated in almost exactly the same way as described in Section 3.5.4. Edges are broken (in order to limit the number of paths) in a manner similar to the one described in Section 3.5.4. There are two differences:

   (a) When assigning ψ and ρ functions in a procedure P, we check to make sure that

   $$\psi_{Entry_P}(1) + \psi_{G_{Entry_P}}(1) \leq 2^{\text{WordSize}}/\text{NumProcs}$$

   instead of checking that

   $$\psi_{Entry_P}(1) \leq 2^{\text{WordSize}}/\text{NumProcs}$$

   (b) For every return-site vertex r in procedure P, we add a surrogate-edge $GEntry_P \rightarrow r$. The use of these surrogate-edges is described below.[1]

2. A return-edge may be broken for some paths, but not others — this is in contrast to Section 3.5.4, where a broken edge could not be used in any path. There are two ways that a return-edge $Exit_P \rightarrow r$ may be broken:

   With respect to unbalanced-right path prefixes: In this case, the return-edge $Exit_P \rightarrow r$ cannot be used in any path $[p \| Exit_P \rightarrow r \| q]$ where p is an unbalanced-right path. In the profiling instrumentation, when execution reaches $Exit_P \rightarrow r$ along an unbalanced-right path p, the path is ended using surrogate-edge $Exit_P \rightarrow Exit_{global}$; the instrumentation then begins recording a new path using the surrogate-edge $GEntry_Q \rightarrow r$, where Q is the procedure containing Q. Even if the return-edge $Exit_P \rightarrow r$ is broken with respect to unbalanced-right path-prefixes, it may still be appear in a path $[p' \| Exit_P \rightarrow r \| q']$, where p' is not an unbalanced-right path (*i.e.*, it contains at least one unmatched, open parenthesis). Return-edges broken during Pass II are always broken with respect to unbalanced-right path prefixes.

   With respect to any path prefix: In this case, the return-edge cannot be used in any path. Return-edges broken during Pass I are always broken with respect to any path prefix.

As described above, to calculate the number of unbalanced-right-left paths in G^*_{fin}, two passes are used: Pass I calculates ψ and ρ functions (almost) as described in Section 3.5.4 (the differences are described in the preceding text); Pass II calculates *numUnbalRLPaths*$[Exit_P]$ for each procedure P (during a traversal of the call-graph associated with G^*_{fin}).

After the second pass, for every procedure P, we require that

$$\psi_{Entry_P}(1) + \psi_{G_{Entry_P}}(\textit{numUnbalRLPaths}[Exit_P]) \leq (2^{\text{WordSize}}/\text{NumProcs}) \tag{32}$$

This is sufficient to guarantee that the number of unbalanced-right-left paths in G^*_{fin} is less than 2^{WordSize}. As the second pass computes the value *numUnbalRLPaths*$[Exit_P]$, it may discover that Equation (32) is

[1] Adding these surrogate-edges to G^*_{fin} allows us to compute the number of unbalanced-right-left paths in G^*_{fin} in two passes, however, they also increase the number of paths in G^*_{fin}, and therefore may cause additional edges to be broken; this may be the reason that observable paths are shorter in an interprocedural-piecewise path profile than they are in an interprocedural-context path profile (see Chapter 6).

violated. If this is the case, the value of *numUnbalRLPaths*$[Exit_P]$ is lowered by breaking a return-edge $Exit_P \rightarrow r$ with respect to unbalanced-right path prefixes; the broken edge is replaced (for unbalanced-right path prefixes) by the surrogate-edges $Entry_Q \rightarrow r$ (which was added during the Pass I) and $Exit_P \rightarrow Exit_{global}$. We then calculate *numUnbalRLPaths*$[Exit_P]$ considering the edge $Exit_P \rightarrow r$ to have been replaced by the edge $Exit_P \rightarrow Exit_{global}$. The reason that *numUnbalRLPaths*$[Exit_P]$ is lowered by breaking a return-edge $Exit_P \rightarrow r$ only with respect to unbalanced-right path prefixes—and not with respect to all path prefixes—is that *numUnbalRLPaths*$[Exit_P]$ is used to calculate the number *numValidComps*(p) for any unbalanced-right path prefix p that ends at $Exit_P$; thus, by breaking $Exit_P \rightarrow r$ with respect to unbalanced-right path prefixes and calculating *numUnbalRLPaths*$[Exit_P]$ as if $Exit_P \rightarrow r$ is replaced by $Exit_P \rightarrow Exit_{global}$, we ensure that *numValidComps*$(p) =$ *numUnbalRLPaths*$[Exit_P]$. Furthermore, we guarantee that for any unbalanced-right path p' that ends at a vertex v in procedure P, *numValidComps*$(p') = \psi_v($*numUnbalRLPaths*$[Exit_P])$.

Pass I guarantees that, for every procedure P,

$$\psi_{Entry_P}(1) + \psi_{G_{Entry_P}}(1) \leq 2^{\mathrm{WordSize}}/\mathrm{NumProcs}$$

This implies that Pass II will be able to satisfy Equation (32) for each procedure P; to do so, it may have to break every return-edge from $Exit_P$ with respect to unbalanced-right path prefixes, in which case, *numUnbalRLPaths*$[Exit_P]$ becomes 1.

A return-edge $Exit_P \rightarrow r$ that is broken only with respect to unbalanced-right path prefixes will have an impact on the profiling machinery described below in Section 4.3. As we will see in Section 4.3, at run-time, the profiling instrumentation will keep a count of the number of open parentheses in the currently executing path. If execution reaches $Exit_P$ and the count of open parentheses is 0, then the program is currently executing an unbalanced-right prefix of an observable path, and the return-edge $Exit_P \rightarrow r$ cannot be used. However, if the count of open parentheses is greater than 0, then the return-edge $Exit_P \rightarrow r$ can be used.

4.2 Calculating the Path Number of an Unbalanced-Right-Left Path

We are now ready to give the algorithm for computing the path number of an unbalanced-right-left path p. This algorithm is very similar to the algorithm given in Section 3.5.5 for calculating the path number of an unbalanced-left path. One additional program variable, `cntOpenLfParens`, is used. This variable is used to keep track of the number of open left parentheses in the prefix p' of p that has been traversed. If `cntOpenLfParens` is zero (indicating that p' is an unbalanced-right path) and the algorithm traverses a return-edge e, then `pathNum` may be incremented by a non-zero value (see Section 4.1.1). If `cntOpenLfParens` is non-zero and the algorithm traverses a return-edge e, then `pathNum` is not incremented (which represents an increment by the value 0).

Algorithm 4.2.1 (Calculate Path Number for an Unbalanced-right-left Path)
Input: An unbalanced-right-left path p from $Entry_{global}$ to $Exit_{global}$.
Output: p's path number.

Initialize stack `NVCstack` to empty

Let e be the first edge of the path p. Calculate the value of *edgeValueInContext*$([\epsilon : Entry_{global}], e)$ as described in Section 4.1.1. Set `pathNum` to this value.

if e is of the form $Entry_{global} \rightarrow Entry_P$ **then**
    numValidCompsFromExit $:= 1$
    cntOpenLfParens $:= 1$
else /* e is of the form $Entry_{global} \rightarrow GEntry_P$ */
    numValidCompsFromExit $:=$ numUnbalRLPaths$[Exit_P]$
    cntOpenLfParens $:= 0$
fi

set e to be the second edge of p
while e is not of the form $v \rightarrow Exit_{global}$ **do**
    if e is of the form $c \rightarrow Entry_T$ **then**
        push numValidCompsFromExit on NVCstack
        let r be the return vertex associated with c
        numValidCompsFromExit $:= \psi_r$(numValidCompsFromExit)
        cntOpenLfParens++
    else if e is of the form $Exit_T \rightarrow r$ **then**
        if cntOpenLfParens $== 0$ **then**
            pathNum += edgeValue$[e]$
            let S be the procedure that contains r
            numValidCompsFromExit $:= \psi_r$(numUnbalRLPaths$[Exit_S]$)
        else
            numValidCompsFromExit $:=$ pop(NVCstack)
            cntOpenLfParens$--$
        fi
    else
        pathNum $:=$ pathNum $+ \rho_e$(numValidCompsFromExit)
    fi
    set e to the next edge of p
od
output pathNum

□

4.3 Runtime Environment for Collecting a Profile

As in Section 3.6, the instrumentation code for collecting an interprocedural piecewise path profile essentially threads Algorithm 4.2.1 into the code of the instrumented program. The instrumentation code for collecting an interprocedural piecewise path profile differs from the instrumentation code described in Section 3.6 in the following ways:

- there is no variable pathNumOnEntry;
- there is a new parameter cntOpenLfParens that is passed to every procedure except *main*, which has cntOpenLfParens as a local variable; and
- both pathNum and cntOpenLfParens are saved before a recursive call is made and restored after a recursive call returns.

```
unsigned int profile[21];       /* 21 possible paths in total */

double pow(double base, long exp,
           unsigned int &pathNum, unsigned int numValidCompsFromExit,
           unsigned int &cntOpenLfParen) {
   double power = 1.0;

   while( exp > 0 ) {
      power *= base;
      exp--;
      profile[pathNum]++;

      /* Start a new path with the edges entry_global->u9 and u9->u1 */
      cntOpenLfParen = 0;
      numValidCompsFromExit = 4; /* from numUnbalRLPaths(u8) */
      pathNum = 16;              /* from the edge entry_global->u9 */
                                 /* no additional code is needed for the */
                                 /*   function <0,0> on edge u9->u3 */
   }

   pathNum += 0 * numValidCompsFromExit + 1; /* from edge u3->u7 */

   return power;
}
```

Figure 22: Part of the instrumented version of the program that computes $(\sum_{j=1}^{9}(2 \cdot j)^2) + (\sum_{k=1}^{6}(3 \cdot k)^2)$. The original program is shown in Figure 8; the instrumentation collects an interprocedural piecewise profile. The instrumented version of *main* is shown in Figure 23. Instrumentation code is shown in italics.

Figures 22 and 23 show the program from Figure 8 with additional instrumentation code to collect an interprocedural piecewise path profile. The output from the instrumented code is as follows:

```
 0: 0    1: 0    2: 0    3: 0    4: 0    5: 0    6: 1    7: 0
 8: 9    9: 0   10: 0   11: 0   12: 3   13: 0   14: 5   15: 1
16:15   17: 3   18: 0   19: 6   20: 6
```

(The algorithm for decoding a path number to obtain the corresponding unbalanced-right-left path is left as an exercise for the reader.)

4.4 Comparing Path-Profiling Information Content

Intuitively, we expect an interprocedural path profile to have more information than an intraprocedural path profile. However, this is not always the case, as the following example shows:

Example 4.4.1 Figure 24(a) shows a schematic of an execution trace. Figure 24(b) shows the paths that the intraprocedural, piecewise path-profiling technique records given the trace shown in Figure 24(a): there are three paths, one in R and two in Q (each shown with a different style line). Notice that the

```
int main() {
   unsigned int pathNum = 0;
   unsigned int numValidCompsFromExit = 1;
   unsigned int cntOpenLfParen = 1;

   double t, result = 0.0;
   int i = 1;

   while( i <= 18 ) {
      if( (i%2) == 0 ) {
         cntOpenLfParen++;
         t = pow( i, 2, pathNum, 0 * numValidCompsFromExit + 3, cntOpenLfParen );
         /* On entry to pow: pathNum is 0 or 8; 4th arg. always 3 */
         /* On exit from pow: pathNum is 1, 9, or 17 */
         if( 0 == cntOpenLfParen ) {
            numValidCompsFromExit = 1;    /* from vertex v16 */
         } else
            cntOpenLfParen--;
         result += t;
      } else
         pathNum += 0 * numValidCompsFromExit + 4;  /* from edge v5->v9 */
      if( (i%3) == 0 ) {
         cntOpenLfParen++;
         t = pow( i, 2, pathNum, 0 * numValidCompsFromExit + 1, cntOpenLfParen );
         /* On entry to pow: pathNum is 1, 4, 9, 12 or 17; 4th arg. always 1 */
         /* On exit from pow: pathNum is 2, 5, 10, 13, 18, or 20 */
         if(0 == cntOpenLfParen) {
            pathNum += 3;              /* from edge u8->v11 */
            numValidCompsFromExit = 1; /* from vertex v16 */
         } else
            cntOpenLfParen--;
         result += t;
      } else
          pathNum += 0 * numValidCompsFromExit + 2;     /* from edge v9->v13 */
      i++;

      profile[pathNum]++;
      /* Start a new path with edges global_entry->v17 and v17->v4 */
      cntOpenLfParen = 0;
      numValidCompsFromExit = 1; /* from vertex v16 */
      pathNum = 8;               /* from edge entry_global->v17 */
   }
   pathNum += 0 * numValidCompsFromExit + 7;  /* from edge v4->v15 */
   profile[pathNum]++;
   for (i = 0; i < 21; i++) {
      cout.width(3); cout << i << ":";
      cout.width(2); cout << profile[i] << "  ";
      if ((i+1) % 9 == 0) cout << endl;
   }
   cout << endl;

   return 0;
}
```

Figure 23: Part of an instrumented version of the program in Figure 8; the instrumentation collects an interprocedural piecewise profile. The instrumented version of the function *pow* and the global declaration of *profile* is shown in Figure 22. Instrumentation code is shown in italics.

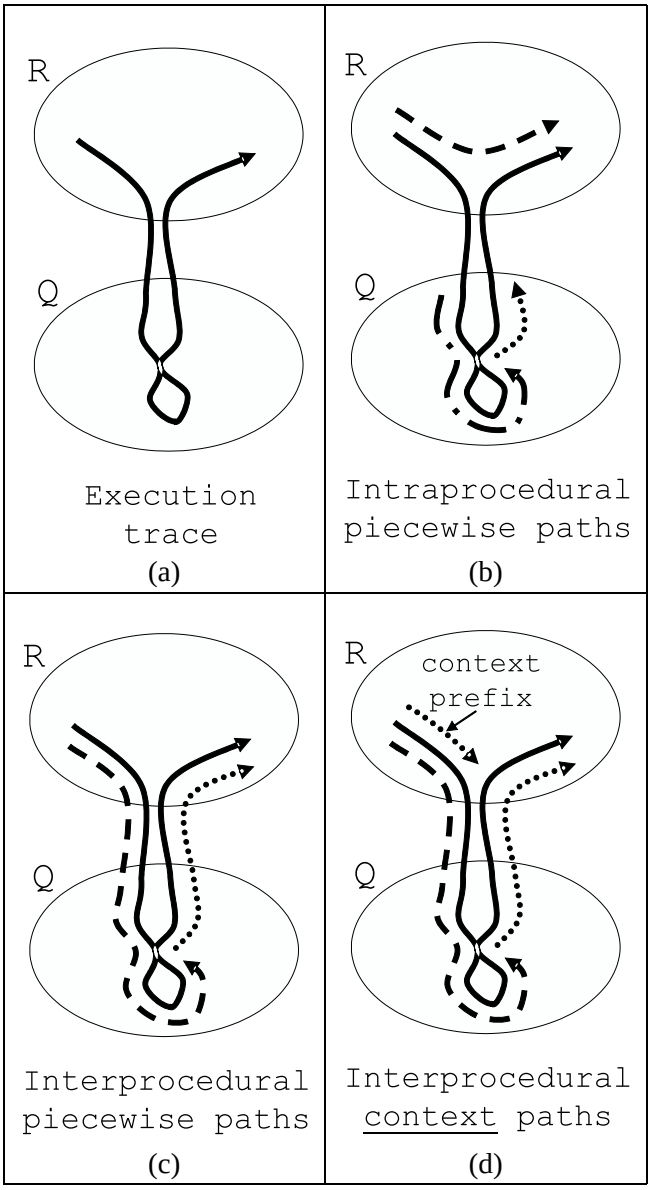


Figure 24: Comparison of the (theoretical) information content of various path profiling techniques.

path in R captures a correlation between R's behavior before the call to Q and R's behavior after the call to Q.

Figure 24(c) shows the paths that the interprocedural, piecewise path-profiling technique records, given the trace shown in Figure 24(a): there are two paths, one that starts in R and ends in Q and one that starts in Q and ends in R. Notice that the interprocedural, piecewise path profile does not capture the correlation between R's behavior before and after the call to Q. However, each path in Figure 24(c) captures a correlation between the execution behaviors of Q and R.

Figure 24(c) shows the paths that the interprocedural, context path-profiling technique records given the trace in Figure 24(a). As in the interprocedural, piecewise path profile, there are two paths, but in this case, the second path consists of a context-prefix and an active suffix. This allows the second path to capture the correlation between R's behavior before and after the call to Q. In addition, the paths in Figure 24(c) capture the same correlations between the execution behaviors of Q and R that the interprocedural piecewise paths capture. □

In practice, each of the three techniques may break edges in different places in order to guarantee that the number of paths is no greater than 2^{WordSize}, which means that the information content of the different profiles is incomprable. If this were not a factor, than an interprocedural, context path profile would always contain more information than an intraprocedural or an interprocedural piecewise path profile while the information content of intraprocedural and interprocedural piecewise path profiles would remain incomparable.

Chapter 5

Other Path-Profiling Techniques

In addition to the functional approach to interprocedural context path profiling, we have developed several other path-profiling techniques. As mentioned above techniques can be classified according to three binary traits:

1. functional approach vs. non-functional approach
2. intraprocedural vs. interprocedural
3. context vs. piecewise

In a functional approach to path profiling, edges are labeled with linear functions. In a non-functional approach to path profiling, edges are labeled with values. This means that non-functional approaches are generally more efficient than functional approaches, however, they also result in coarser profiles.

The Ball-Larus technique is an example of a non-functional approach to intraprocedural piecewise path profiling. Chapter 3 describes the functional approach to interprocedural, context path profiling. Chapter 4 describes the functional approach to interprocedural, piecewise path profiling. We have developed path-profiling techniques for every other combination of the three traits listed above. In addition, each pair of path-profiling techniques that we have developed in this paper can be hybridized to give a new technique. In the remainder of this chapter, we discuss some of these other approaches to path profiling.

5.1 Intraprocedural Context Path Profiling

This section describes how to modify the Ball-Larus path-profiling technique to collect an intraprocedural context profile. In Section 3.6, each observable path is divided into a context-prefix and an active-suffix. When these definitions are applied to the observable paths of the Ball-Larus (intraprocedural) technique, each observable path has an empty context-prefix. We now show how to modify the Ball-Larus technique so that an observable path may have a non-empty context-prefix. Under this new technique, a typical observable path will consist of a context-prefix that summarizes the path taken to a loop header and an active suffix that is a path through the loop.

The new technique gives more detailed profiling information than the Ball-Larus path-profiling technique. For example, suppose there is a correlation between the path taken to a loop header and the path(s) taken during execution of the loop body. Intraprocedural context profiling will capture the relationship between the path taken to the loop header and the paths taken on each iteration of the loop body. The Ball-Larus (piecewise) profiling technique will only capture the correspondence between the path taken to the loop header and the path taken on the first iteration of the loop; for all subsequent iterations, the Ball-Larus technique records an observable path that begins at the loop header, and ignores the context information provided by the path used to reach the loop header.

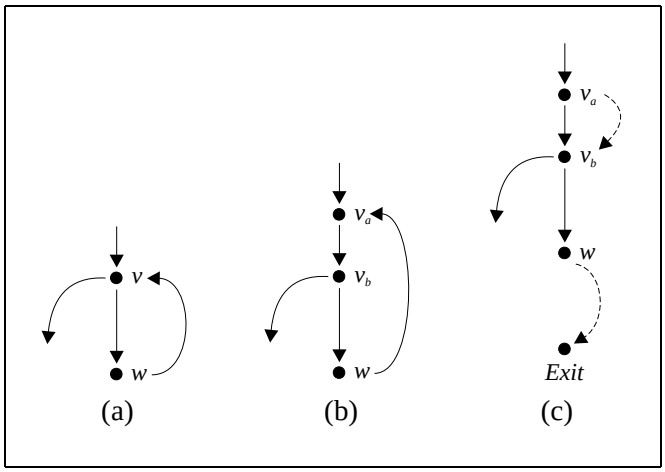


Figure 25: Illustration of Transformations 1 and 2 from Section 5.1. Figure (a) shows a piece of the control-flow graph for a loop before Transformation 1, and Figure (b) shows the same piece of the control-flow graph after the transformation. Figure (c) shows the two surrogate edges that replace the backedge $w \rightarrow v_a$. Note the difference from the Ball-Larus technique: Section 5.1 uses the surrogate edge $v_a \rightarrow v_b$ instead of the surrogate edge $Entry \rightarrow v$.

Just as in the Ball-Larus technique, modifications are made to the procedure's control-flow graph. Unlike the Ball-Larus technique, we require that the control-flow graph be reducible.[1] There are two transformations:

Transformation 1 (split backedge targets): Each vertex v that is a backedge target is split into two vertices v_a and v_b. All edges into v are changed to point to vertex v_a. All edges that have v as a source vertex are changed to have v_b as the source. An edge $v_a \rightarrow v_b$ is added to the graph; this edge is *not* considered to be a surrogate edge in the following discussion. Figures 25(a) and 25(b) illustrate this transformation.

Transformation 2 (replace backedges): For each backedge target v_a, a second edge $v_a \rightarrow v_b$ is added to the graph; this edge is considered to be a surrogate edge. (Thus, for each pair of vertices v_a and v_b introduced by Transformation 1, there are two edges of the form $v_a \rightarrow v_b$, one of which is considered to be a surrogate edge, and one that is not.) For each backedge source w, the surrogate edge $w \rightarrow Exit$ is added to the graph. Each backedge $w \rightarrow v_a$ is removed from the graph. Figure 25(c) illustrates this transformation.

The graph that results from performing these transformations is acyclic. Once the graph has been modified, the Ball-Larus edge-numbering scheme is used as before. As in the Ball-Larus technique, the path number for a path p from $Entry$ to $Exit$ is the sum of the values on p's edges.

We are now ready to describe the instrumentation that is used to collect a profile. As in the Ball-Larus technique, an integer variable `pathNum` is introduced that is used to accumulate the path number of the currently executing path. At the beginning of the procedure, `pathNum` is initialized to 0.

Let v be a backedge target in the original control-flow graph, and let v_a and v_b be the vertices that represent v in the modified control-flow graph (after Transformation 1). A new integer variable called

[1] If the control-flow graph is not reducible, then the graph can be transformed to make it reducible (see, for example, Aho et al. [3]).

`pathNumOnEntryToV` is introduced. When control reaches v, `pathNumOnEntryToV` is set to the current value in `pathNum`. `pathNum` is then incremented by the value on the non-surrogate edge $v_a \rightarrow v_b$ in the modified graph. When the backedge $w \rightarrow v$ is traversed, the following steps are taken:

1. `pathNum` is incremented by the value on the surrogate edge $w \rightarrow Exit$. The profile is updated with this value of `pathNum`.

2. `pathNum` is set to `pathNumOnEntryToV`, plus the value on the surrogate edge $v_a \rightarrow v_b$.

The second step starts recording the path number for a new path p. The path p contains the edge $v_a \rightarrow v_b$ and will have a context-prefix that ends at v_a and an active-suffix that begins at v_b. Note that this instrumentation relies on the fact that the original control-flow graph is reducible. In particular, it assumes that the backedge target v is reached—and the value of `pathNumOnEntryToV` is set—before the backedge $w \rightarrow v$ is traversed.

The remaining instrumentation is similar to the instrumentation used in the standard Ball-Larus technique. In particular, as each edge e is traversed, the value in `pathNum` is incremented by the value on e.

5.2 Interprocedural Context Path Profiling with Improved Context for Recursion

A comparison of the Ball-Larus piecewise path-profiling technique and the intraprocedural context path-profiling technique suggests a way of improving the context for recursively called procedures in interprocedural context path profiling. The way recursive call-edges are currently handled in interprocedural context path profiling is similar to the way backedges are handled in the Ball-Larus technique. In interprocedural context path profiling, each recursive call-edge is removed and an edge is added from $Entry_{global}$ to the target of the call-edge. Similarly, in the Ball-Larus technique, each backedge is removed and a surrogate edge is added from $Entry$ to the target of the backedge.

To change interprocedural context path profiling to keep more context in the presence of recursion, we change the handling of recursive call-edges to resemble the handling of backedges in intraprocedural context path-profiling. Every vertex $Entry_R$ that is the target of a recursive call-edge is split into two vertices, $Entry_R$ and $Entry'_R$ with a (non-surrogate) edge $Entry_R \rightarrow Entry'_R$ between them. The recursive call-edge is removed, and a surrogate edge is added from $Entry_R$ to $Entry'_R$. The profiling instrumentation is modified such that when the procedure R is called from a non-recursive call-site the value of `pathNum` is saved in static variable `pathNumOnNonRecrEntryToR`. When R is called from a recursive call-site, the value of `pathNum` is restored to the value in `pathNumOnNonRecrEntryToR`.

In fact, this description is an oversimplification of the technique. Special care must be taken when the call graph is not reducible. Furthermore, in the presence of mutual recursion, it is possible for more than one non-recursive call to a recursive procedure to be active at once. (Recall that a call is recursive iff it is represented by a backedge is the call-graph.) This means that `pathNumOnNonRecrEntryToR` must be a stack. Furthermore, profiling in the presence of exceptions becomes more complicated (see Section 3.7.2.

This technique is important because it may lead to a more efficient method for gathering the information present in Ammons et. al.'s *Calling Context Tree* [4] — although such a technique will be hampered by the need for a complete call graph.

5.3 Non-Functional Approaches to Interprocedural Path Profiling

In both of the functional approaches to interprocedural path profiling, the graph G^*_{fin} may contain cycles (see Figure 7(c)). If these cycles are broken, then G^*_{fin} becomes a directed acyclic graph. In this case, it is possible to number the paths in G^*_{fin} using the Ball-Larus path-numbering technique; there is no need for ψ or ρ functions — each edge is labeled with a value, and the number for a path p is the sum of the values on the path's edges.

One interesting way to remove cycles of the form found in Figure 7(c) is to remove all return edges. Specifically, we have the following graph transformation:

Remove Return-Edges: For each pair of vertices c and r representing a call site that calls procedure P, remove the return-edge $Exit_P \rightarrow Exit_{global}$, add the surrogate edge $Exit_P \rightarrow Exit_{global}$, and add the summary edge $c \rightarrow r$.

This transformation along with the appropriate modifications in the profiling machinery, can be used with either of the functional approaches to interprocedural path profiling to create a non-functional approach to interprocedural path profiling. In these non-functional approaches to interprocedural path profiling, before a procedure P calls a procedure Q, the value of `pathNum` is saved in a local variable `pathNumBeforeCall`. When control passes from the call-vertex c to the entry vertex $Entry_Q$, the value of `pathNum` is incremented by the value on $c \rightarrow Entry_Q$. When control reaches the end of Q, the profile is updated with the path ending at $Exit_Q$. When control returns to the calling procedure P, the value of `pathNum` is restored to the value in `pathNumBeforeCall` and incremented by the value on the edge $c \rightarrow r$.

There are two advantages to using a non-functional approach to interprocedural path profiling:

- There are exponentially fewer paths than in the corresponding functional approach to interprocedural path profiling.
- The instrumentation code that collects the profile is more efficient.

However, non-functional approaches to interprocedural path profiling will also generate less detailed profiles.

5.4 Hybrid Approaches to Path Profiling

Each pair of path-profiling techniques described in this thesis can be hybridized to give a new technique. For example, if one removes some but not all of the nonrecursive return-edges from G^*_{fin}, then the resulting technique could be considered to be a hybrid between a functional and a non-functional approach to interprocedural path profiling. In such an approach, some procedures may have edges labeled with linear functions while other procedures have edges labeled just with constant functions.

Other hybridizations are also possible. In a functional approach to interprocedural path profiling, the instrumentation for each procedure takes parameters at runtime; thus the instrumentation can be made to behave in different ways in different contexts. For example, with the correct parameters, the instrumentation for a leaf procedure will record an intraprocedural path profile for that procedure. With different parameters, the instrumentation for a leaf procedure will help record an interprocedural path profile for paths passing through the procedure. Thus, it is possible to use the instrumentation of a procedure to record an intraprocedural profile when called from one context, and to help record an interprocedural profile when called from another context.

Notice that if all call-edges and return-edges are removed from G^*_{fin}, then an interprocedural path-profiling technique becomes an intraprocedural path-profiling technique. In this sense, one might consider an interprocedural technique where only some of the call-edges and return-edges have been removed to be a hybridization between interprocedural path-profiling and intraprocedural path-profiling.

It is also possible to combine intraprocedural and interprocedural techniques in other ways. For example it is easy to mix intraprocedural context path profiling with interprocedural context path profiling to give an interprocedural technique where an observable path can include context information that summarizes the interprocedural path taken to a procedure entrance and the intraprocedural path taken to a loop header. One could also mix intraprocedural context path profiling with interprocedural piecewise path profiling to generate a novel technique. An observable path in this technique could include an active-suffix that crosses call and return edges and a context-prefix that summarizes the intraprocedural path taken to a loop header.

Chapter 6

Path Profiling Experimental Results

In this chapter, we present some statistics on the paths in the interprocedural path profiles of the SPEC95Int benchmarks. Our implementation of the interprocedural path profiling techniques differs from the description given in the previous chapters in one small regard: we represent a path name using a 16-bit integer for the first edge of the path (which must have the form $Entry_{global} \rightarrow Entry_P$ or $Entry_{global} \rightarrow GEntry_P$) and a 64-bit integer that holds the sum of the values contributed by each edge except the first; in the description given previously, a path name a single integer — the sum of the values contributed by each edge. The new representation simplified our implementation slightly.

We will compare the statistics for our interprocedural path profiles against statistics for the paths in the Ball-Larus (*i.e.*, intraprocedural, piecewise) path profiles of the SPECInt95 benchmarks [12]. Our results for intraprocedural, piecewise path profiling differ from those reported in [12] for several reasons:

- We test the benchmarks on different hardware.
- We implement path profiling using a tool written in SUIF 1.3.0.5 that takes C source code as input and produces an instrumented version of the source code. [12] uses EEL (the Executable Editing Library [43]) to insert profiling instrumentation directly into the executable.
- We build our control-flow structures from SUIF. [12] builds their control-flow structures from Sparc executables.
- Because we use SUIF, we do not profile library code.
- The way we define a program vertex is different. Since we construct a program supergraph, we create a call vertex c and a return-site vertex r to represent each call instruction i. The vertices c and r represent only the call instruction i. In [12], they build a collection of intraprocedural control-flow graphs where each control-flow vertex represents a basic block. In this case, a call instruction i is treated the same as any non-branching instruction: as such, i may be represented in a vertex together with other instructions.

Tables 3 through 5 and Figure 26 show statistics about path profiles of the SPEC95 integer benchmarks when run on their reference inputs. In Section 3.5.4, we described techniques for limiting the number of observable paths, such that each path name fits into a machine word. One of the techniques for limiting the number of paths that pass through a call-site is to "break" the return-edge at the call-site. This may create a small bookkeeping problem for the interprocedural, piecewise path profiling technique. The problem is that a path prefix p leading to the call vertex c of the call-site will be counted in two observable paths in the path profile: a path that continues from c along a call-edge, and a path that continues from c along a summary-edge. To avoid over-counting the execution frequency of p, we consider p to be a context-prefix in the observable path that continues from c along the call-edge. This small technical glitch means that we consider some of our piecewise observable paths to have context-prefixes, though this violates our original definition of a piecewise profile. However, the observable

Benchmark	Profiling technique	Number of distinct paths	Number of executed paths	Number of executed SUIF instructions
124.m88ksim	Inter. Context	2080	4.34E+09	1.87E+11
	Inter. Piecewise	1101	4.64E+09	〃
	Intra. Piecewise	725	4.72E+09	〃
129.compress	Inter. Context	423	1.76E+09	9.23E+10
	Inter. Piecewise	131	1.76E+09	〃
	Intra. Piecewise	124	2.91E+09	〃
130.li	Inter. Context	3943	2.24E+09	1.19E+11
	Inter. Piecewise	1343	2.22E+09	〃
	Intra. Piecewise	560	3.30E+09	〃
132.ijpeg	Inter. Context	2734	2.26E+09	1.89E+11
	Inter. Piecewise	1384	2.26E+09	〃
	Intra. Piecewise	1031	2.37E+09	〃
134.perl	Inter. Context	1656	1.15E+09	6.30E+10
	Inter. Piecewise	1497	1.20E+09	〃
	Intra. Piecewise	1096	9.77E+08	〃
147.vortex	Inter. Context	4211	1.78E+09	1.18E+11
	Inter. Piecewise	4220	1.86E+09	〃
	Intra. Piecewise	1749	2.20E+09	〃

Table 3: Path profiling statistics when the profiled SPEC benchmark is run on its reference input.

Benchmark	Profiling technique	Avg. num. edges per path	Avg num. SUIF instructions per path
124.m88ksim	Inter. Context	68.9 [31.6:37.3]	261.2 [68.7:192.5]
	Inter. Piecewise	6.8 [0.7: 6.1]	43.1 [2.7: 40.4]
	Intra. Piecewise	5.9 [: 5.9]	39.7 [: 39.7]
129.compress	Inter. Context	46.7 [20.6:26.1]	187.9 [43.1:144.8]
	Inter. Piecewise	9.0 [0.0: 9.0]	52.3 [0.0: 52.3]
	Intra. Piecewise	4.7 [: 4.7]	31.7 [: 31.7]
130.li	Inter. Context	29.7 [18.1:11.6]	107.7 [43.1: 64.6]
	Inter. Piecewise	12.0 [0.5:11.4]	55.6 [1.9: 53.7]
	Intra. Piecewise	7.2 [: 7.2]	36.1 [: 36.1]
132.ijpeg	Inter. Context	5.5 [1.9: 3.6]	102.9 [9.6: 93.3]
	Inter. Piecewise	3.6 [0.0: 3.6]	83.7 [0.0: 83.7]
	Intra. Piecewise	3.4 [: 3.4]	79.6 [: 79.6]
134.perl	Inter. Context	10.6 [1.2: 9.4]	60.8 [6.2: 54.6]
	Inter. Piecewise	9.2 [0.3: 8.9]	54.1 [1.4: 52.7]
	Intra. Piecewise	11.0 [:11.0]	64.4 [: 64.4]
147.vortex	Inter. Context	21.8 [3.2:18.6]	93.6 [10.4: 83.2]
	Inter. Piecewise	14.7 [1.7:13.0]	71.0 [7.6: 63.4]
	Intra. Piecewise	10.6 [:10.6]	53.5 [: 53.5]

Table 4: Path profiling statistics for the profiled SPEC benchmarks when run on their reference input. An entry such as "68.9 [31.6:37.3]" in the column "Average number of Edges" indicates that the average path has 68.9 edges, with 31.6 edges in the context-prefix and 37.3 edges in the active suffix. All the entries in all of the averages columns have a similar interpretation. For purposes of calculating averages, each path is weighted by its execution frequency. Blank entries in the table represent a measurement that is not applicable (*e.g.*, context-prefix length in a piecewise profile); as mentioned in the text, there is one situation where we consider an interprocedural, piecewise observable path to have a context-prefix.

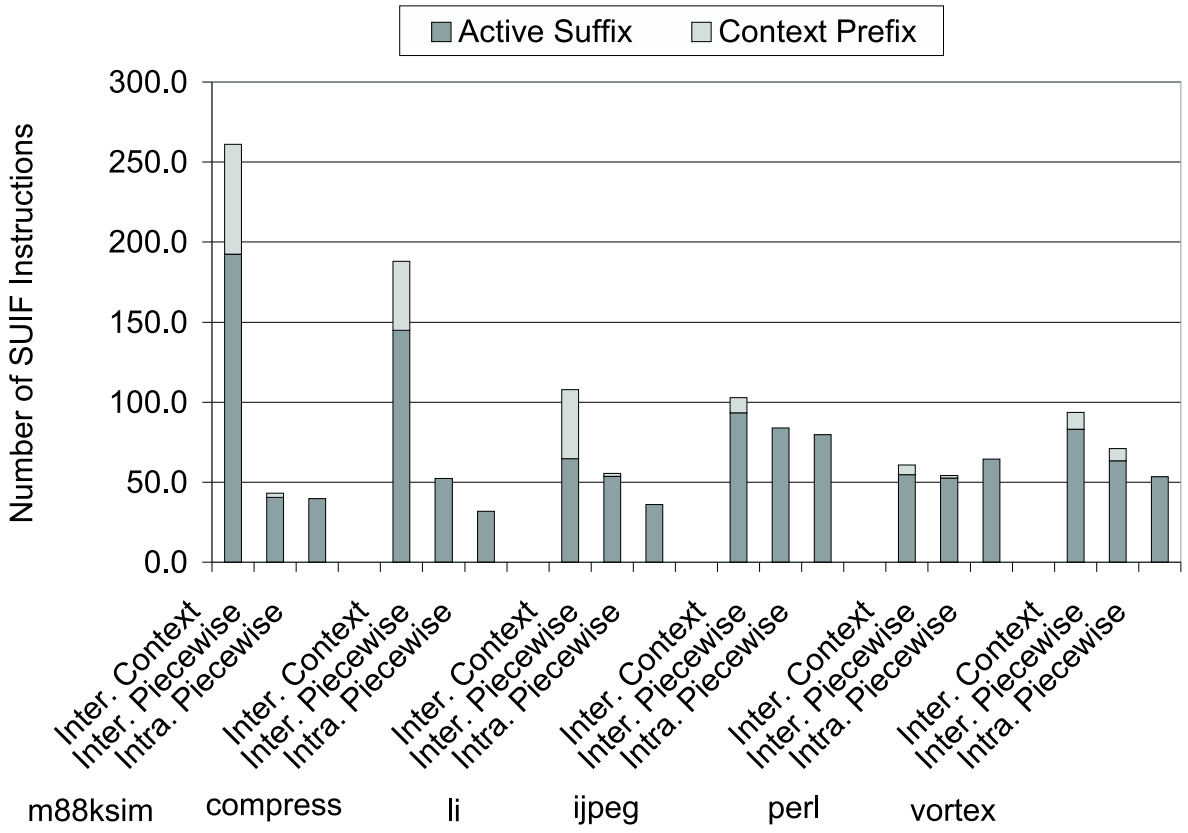


Figure 26: Graph of the average number of SUIF instructions in an observable path for interprocedural context, interprocedural piecewise, and intraprocedural piecewise path profiles of SPEC95 benchmarks when run on their reference inputs. For purposes of computing the average, each observable path is weighted by its execution frequency. (This figure is a graphical representation of the data in the last column of Table 4.)

Benchmark	Profiling technique	Avg. num. edges per path	Avg. num. call-edges per path	Avg. num. return-edges per path
124.m88ksim	Inter. Context	68.9 [31.6:37.3]	5.8 [5.6: 0.2]	3.6 [3.4: 0.2]
	Inter. Piecewise	6.8 [0.7: 6.1]	0.2 [0.0: 0.2]	0.1 [0.0: 0.1]
129.compress	Inter. Context	46.7 [20.6:26.1]	5.2 [4.6: 0.7]	2.9 [2.3: 0.7]
	Inter. Piecewise	9.0 [0.0: 9.0]	0.7 [0.0: 0.7]	0.7 [0.0: 0.7]
130.li	Inter. Context	29.7 [18.1:11.6]	3.0 [2.5: 0.6]	1.0 [0.5: 0.5]
	Inter. Piecewise	12.0 [0.5:11.4]	0.6 [0.0: 0.6]	0.5 [0.0: 0.5]
132.ijpeg	Inter. Context	5.5 [1.9: 3.6]	0.4 [0.4: 0.1]	0.1 [0.1: 0.1]
	Inter. Piecewise	3.6 [0.0: 3.6]	0.1 [0.0: 0.1]	0.1 [0.0: 0.1]
134.perl	Inter. Context	10.6 [1.2: 9.4]	0.4 [0.1: 0.3]	0.3 [0.0: 0.3]
	Inter. Piecewise	9.2 [0.3: 8.9]	0.3 [0.0: 0.3]	0.2 [0.0: 0.2]
147.vortex	Inter. Context	21.8 [3.2:18.6]	1.1 [0.4: 0.6]	0.4 [0.0: 0.4]
	Inter. Piecewise	14.7 [1.7:13.0]	0.7 [0.0: 0.6]	0.4 [0.0: 0.4]

Table 5: Path profiling statistics for the profiled SPEC benchmarks when run on their reference input. The numbers in this chart are interpreted just as the numbers in Table 4 are.

paths in a piecewise path profile still do not contain surrogate-edges, and hence do not contain "gaps" in the way that an observable path in a context-path profile does.

Table 4 and Figure 26 show that interprocedural, piecewise paths are shorter, on average, than interprocedural, context paths. This is probably due to the heuristics used to limit the number of observable paths in G^*_{fin} (see Section 4.1.2); the heuristics for interprocedural, piecewise path profiling seem to break more edges than the heuristics for interprocedural, context path profiling. Interprocedural context paths are also quite a bit longer than intraprocedural piecewise paths: the average number of SUIF instructions along an observable path is considerably higher for most of the benchmarks (see Table 4 and Figure 26). The one exception is 134.perl. Again, this is probably due to the techniques used to limit the number of observable paths in G^*_{fin} (see Section 3.5.4 and Section 4.1.2); the techniques in the interprocedural case must break an edge that is frequently executed, which has a large impact on the average path length.

Table 5 shows the degree to which the interprocedural path profiles capture interprocedural information. In some cases (*e.g.*, in the interprocedural context path profile for m88ksim), the averege observable path contains a significant number of call and return-edges, indicating that a good deal interprocedural behavior is captured in the profile. In other cases (*e.g.*, in the interprocedural piecewise path profile for ijpeg), very little interprocedural information is captured. This may mean that the program spends most of its time inside of one procedure without making function calls.

Figure 27 shows plots of the number of observable paths required to cover a given percentage of program execution. Plots are shown for interprocedural context, interprocedural piecewise, and intraprocedural piecewise path profiles of SPEC95 benchmarks when run on their reference inputs. For most of the path profiles, and for most of the benchmarks, 50 paths are sufficient to cover 80% of program execution. One exception is the interprocedural, context path profile for m88ksim; here 100 paths are required to cover 80% of program execution. Other likely exceptions include the interprocedural path profiles of go: these profiles include so many paths that we were unable to process them.

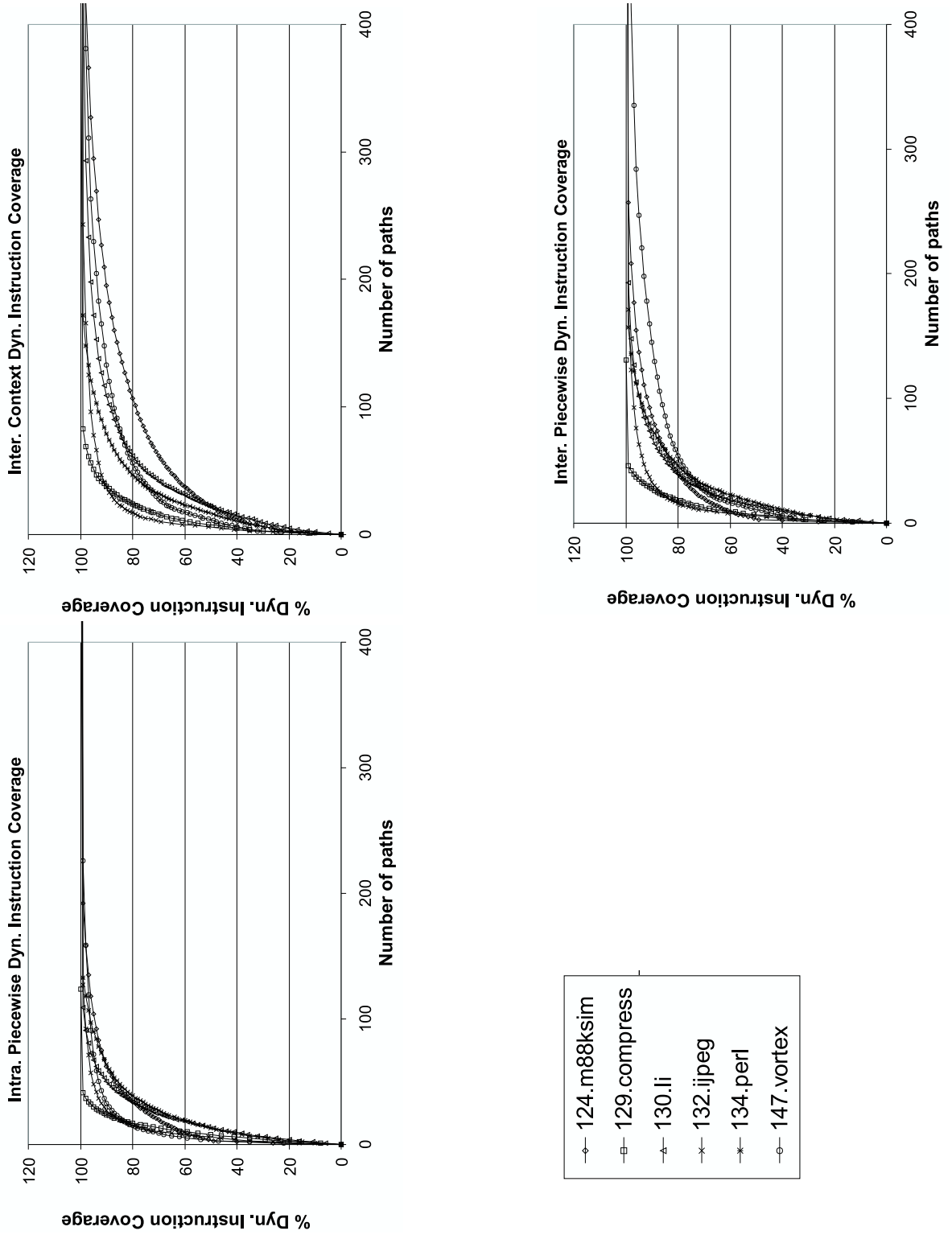


Figure 27: Number of paths versus percentage of dynamic execution covered.

Benchmark	Base Time (sec)	Time with Inter. Context. Prof. (sec)	Time with Inter. Piecewise Prof. (sec)
099.go	265	1423	1125
124.m88ksim	144	1060	1145
129.compress	135	479	478
130.li	112	641	679
132.ijpeg	152	490	479
134.perl	104	452	547
147.vortex	189	933	1028

Table 6: Runtime of the SPEC95Int benchmarks with and without interprocedural path profiling instrumentation.

Benchmark	Inter. Context % Overhead Calc. Path Name	Inter. Context % Overhead Record Path	Inter. Piecewise % Overhead Calc. Path Name	Inter. Piecewise % Overhead Record Path
099.go	138	299	111	213
124.m88ksim	252	383	199	495
129.compress	85	170	55	200
130.li	226	246	172	333
132.ijpeg	143	180	28	187
134.perl	163	170	113	311
147.vortex	200	194	148	296
average:	158	235	118	291

Table 7: Interprocedural path profiling overhead.

Table 6 shows the runtimes of the benchmarks when they are run on their reference inputs with and without profiling instrumentation. We ran the benchmarks 3 times on a 833 MhZ Pentium III with 256M RAM running Solaris 2.7 and averaged the results. The benchmarks were compiled with GCC 2.95.3 with the "-O3" option. The first column lists the benchmarks. The second column shows the runtime of the benchmarks when they are compiled "out of the box". The third column shows the runtime of the benchmark with the profiling instrumentation for collecting an interprocedural, context path profile. The fourth column shows the runtime of the benchmark with the profiling instrumentation for collecting an interprocedural, piecewise path profile.

Table 7 shows the percent overhead for the profiling machinery (as compared to the base runtime shown in Column II of Table 6). Columns II and III of Table 7 show the percent overhead of the profiling instrumentation for interprocedural, context path profiling. Columns IV and V show the percent overhead of the profiling instrumentation for interprocedural, piecewise path profiling. Columns II and IV show the percent overhead for calculating each path name (for interprocedural, context and interprocedural, piecewise path profiling, respectively); this consists of all of the increments and assignments done to `pathNum` as each supergraph edge is executed (see Sections 3.6 and 4.3). We expected the overhead in this column to be close to the overhead of the Ball-Larus path-profiling technique (30%). Instead, it is much higher. This is partly due to the fact that we did not implement the optimization used in [12] that attempts to push increment operations to infrequently executed edges (see also [10]). However, even if this optimization were performed, the profiling overhead would still be high, as is shown in Columns III and V.

Columns III and V show the percent overhead of the profiling instrumentation for recording a path execution; this involves looking up a path name in a hash table to find the path's counter and incrementing this counter. This overhead can be expected to be higher than the overhead in the Ball-Larus technique: every time one of the interprocedural techniques records a path execution, it must perform a hash-table lookup; in contrast, the Ball-Larus technique can often perform an array access. This is because the interprocedural techniques always have a large number of paths, since they consider the entire program. In contrast, the Ball-Larus intraprocedural technique can be extremely efficient (and avoid hash-table lookups) on procedures that have few paths.

Chapter 7

The Interprocedural Express-lane Transformation

The intraprocedural express-lane transformation takes a control-flow graph and an intraprocedural, piecewise path profile and creates a hot-path graph in which all of the hot paths have been duplicated. Section 2.2 summarizes the algorithm of Ammons and Larus for performing this transformation [5]. In this section, we describe how to extend their algorithm to take as input the program supergraph and an interprocedural path profile and to produce as output a hot-path supergraph; the transformation of a supergraph into a hot-path supergraph is called the interprocedural express-lane transformation. We present algorithms for performing the interprocedural express-lane transformation for both interprocedural piecewise and interprocedural context path profiles.

There are several issues that must be addressed. Some of them involve formally specifying what is meant by an interprocedural "express-lane": previously, we defined an express-lane as a copy p' of a hot-path p such that p' has only one entry point; this means that p' may have conditionals that branch out of p', but other code never branches into p' (except at the entry point). This definition works well in the intraprocedural case, but must be extended in the interprocedural case. In a context path profile, a path may consist of a non-empty context-prefix as well as an active-suffix. Also, an observable path in an interprocedural path profile may contain "gaps", *e.g.*, between the context-prefix and the active-suffix; these gaps are represented in the observable path by surrogate edges (which do not occur in the original supergraph). An express-lane version of an interprocedural observable path may have a context-prefix and an active-suffix, and may have gaps just as the hot path does.

There are also technical issues that must be resolved. The interprocedural express-lane transformation requires a mechanism for duplicating call-edges and return-edges. We will use a straightforward approach that duplicates a call edge $c \rightarrow Entry_P$ by creating copies of c and $Entry_P$ and duplicates a return edge $Exit_P \rightarrow r$ by creating copies of $Exit_P$ and r. However, this leads to a supergraph with the following non-standard properties:

1. procedures may have more than one entry;
2. procedures may have more than one exit; and
3. a call-site vertex may have multiple return-site vertices.

We will describe the executable program that corresponds to a supergraph with the above properties.

Even with a mechanism for duplicating interprocedural edges, many modifications of the intraprocedural algorithm are required to obtain an algorithm for performing the interprocedural express-lane transformation. The intraprocedural express-lane transformation uses a deterministic finite automaton (DFA) for recognizing hot-paths and combines this with the control-flow graph, which can be seen as another deterministic finite automaton. To create an automaton that recognizes a set of interprocedural hot-paths, we require a *pushdown* automaton. The supergraph can be seen as a second pushdown

automaton. Thus, if we mimic the approach used for the intraprocedural express-lane transformation, we will need to combine two pushdown automata, a problem that is uncomputable, in general. Instead, we will create a collection of deterministic finite automata, one for each procedure; the automaton for a procedure P recognizes hot-paths that start in P. The interprocedural express-lane transformation combines a family of hot-path automata with the supergraph to create the hot-path supergraph.

Finally, the construction of an automaton for recognizing paths from a context path profile is more complicated than the construction of an automaton for recognizing paths from a piecewise path profile. Again, recognizing a context-prefix is more complicated because it may contain surrogate edges.

This chapter makes the following contributions:

1. We give a formal definition of the interprocedural express-lane transformation, both for context and piecewise path profiles.

2. We give algorithms for performing the interprocedural express-lane transformation, both for context and piecewise path profiles.

Section 7.1 describes entry and exit splitting, which is the technique used to duplicate call and return-edges, respectively [15, 18]. It also discusses the non-standard supergraph properties that result. Section 7.2 defines an interprocedural express-lane. Section 7.3 gives algorithms for performing the interprocedural express-lane transformation.

7.1 Entry and Exit Splitting

The algorithm for performing the interprocedural express-lane transformation uses entry splitting to duplicate call-edges and exit splitting to duplicate return-edges. This section briefly describes entry and exit splitting. (Our definitions of entry and exit splitting are taken from [15, 18].)

Entry splitting allows a procedure P to have more than one entry. A call to P jumps (*e.g.*, via a jump-and-link instruction) to one of P's multiple entries. Figure 28 shows a schematic of a procedure with two entries.

Exit splitting allows a procedure P to have multiple exits, each of which is assigned a number. Normally, when a procedure call is made, the caller provides a return address for when the callee returns. In the case where a procedure has multiple exits, the caller provides a vector of return addresses. When the callee reaches the i^{th} exit vertex, it branches to the i^{th} return address. (In our experiments, our implementation of entry and exit splitting is much less efficient than the one described here. We implement entry-splitting by having each call vertex pass an entry number to the procedure that it calls and inserting a switch statement on this parameter at the beginning of the procedure; similarly, we implement exit-splitting by having each exit vertex return an exit number and inserting a switch statement on this value at the return-site vertex. Using this techniques we can implement the same execution semantics described above. The advantage of this implementation is that it can be done as a source-to-source transformation using the SUIF toolkit.) Figure 29 shows a schematic of a procedure with multiple exits.

As mentioned above, entry and exit splitting lead to a supergraph that may have the following nonstandard properties:

1. a procedure may have more than one entry;

2. a procedure may have more than one exit; and

3. a call vertex may be associated with more than one return-site vertex.

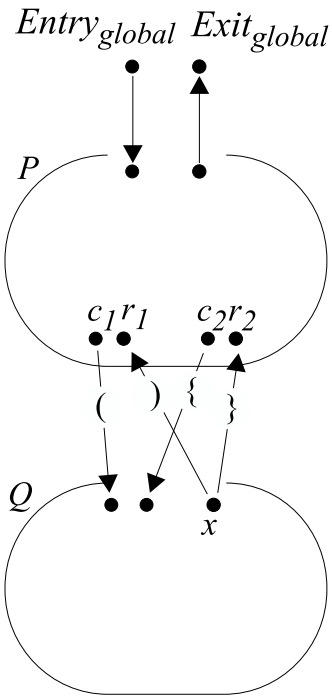


Figure 28: Schematic of a procedure Q with multiple entries; there are two call-sites that call Q, each of which calls a different entry.

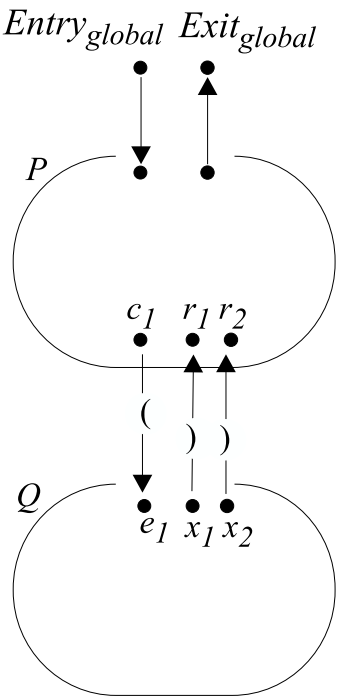


Figure 29: Schematic of a procedure Q with multiple exits; there is one call-site that calls Q, which has multiple return-site vertices.

In the interprocedural hot-path graph, we will also allow a return-site vertex to be shared by more than one call vertex.

Despite these non-standard properties, we define a same-level valid path (and unbalanced-left paths and unbalanced-right paths, etc.) in the same way as for a standard supergraph. In particular, each call site has only a single call vertex c; the call-edges associated with the call-site are labeled "$(_c$", and return-edges are labeled with "$)_c$. The context-free grammar for defining same-level valid paths (and unbalanced-left and unbalanced-right paths) is the same as before (see Section 3.1 and the beginning of Chapter 4).

In the remainder of this thesis, we will assume that the supergraph G^* has been augmented with summary-edges: that is, there is a summary-edge $c \rightarrow r$ connecting each call vertex c with its return-site vertex r. The original definition of the supergraph did not include summary-edges. Previously, we added a summary-edge to G^*_{fin} whenever we broke a return-edge. In that case, we were using a summary-edge in G^*_{fin} to help define the set of observable paths in G^*. However, for many uses of the program supergraph, *e.g.*, dataflow analysis, it is convenient for summary-edges to be present. Because they are so useful, the hot-path supergraph H^* (that we construct in this section) will also be annotated with summary-edges: for every call-edge $c \rightarrow e$ labeled "$(_c$", for every return-edge $x \rightarrow r$ labeled "$)_c$", if there is a same-level valid path in H^* from e to x, then there is a summary-edge from c to r.

7.2 Defining the Interprocedural Express-Lane

In this section, we give a definition of an interprocedural express-lane. First we consider a simple example to develop intuition about what should happen when we duplicate an observable path from an interprocedural, context path profile.

Example 7.2.1 We return to the example program in Figure 8. Figure 9 shows the graph G^*_{fin} that is constructed when collecting an interprocedural, context path profile for the program in Figure 8. Figure 9 shows the following observable path (path number 24) in bold:

$$24 : Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow u_1 \rightarrow u_3 \rightarrow u_4 \rightarrow u_5 \rightarrow u_6 \rightarrow Exit_{global}.$$

Suppose we wish to duplicate this path—that is, suppose we wish to create a hot-path version of the program in Figure 8 that has an express-lane version of path number 24. The principal difficulty in duplicating this path has to do with the the edge $u_1 \rightarrow u_3$: this surrogate-edge appears in the middle of the observable path that we wish to duplicate. However, the edge $u_1 \rightarrow u_3$ does not appear in the supergraph for the program in Figure 8. So what does it mean to create a hot-path version of the example program where this edge has been duplicated, considering that this edge does not appear in the program's supergraph?

In the interprocedural, context path profile for the example program, path number 24 has a count of 9. This means that during the program's execution, the active suffix $[u_3 \rightarrow u_4 \rightarrow u_5]$ executed 9 times in the context specified by the context-prefix $[Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow u_1]$. Another interpretation of the count of 9 for path number 24 is as follows: in the execution trace of the program, the pattern indicated by observable path number 24 occurs 9 times. Under this interpretation, the surrogate-edge $u_1 \rightarrow u_3$ matches any same-level valid path from u_1 to u_3 that ends with the backedge $u_5 \rightarrow u_3$. (The surrogate-edge $u_1 \rightarrow u_3$, along with the surrogate-edge $u_5 \rightarrow u_6$, was added to G^*_{fin} to replace the backedge $u_5 \rightarrow u_3$.) In a sense, when we duplicate the surrogate-edge $u_1 \rightarrow u_3$, we will duplicate all same-level valid paths from u_1 to u_3. It is more accurate to say that we duplicate intraprocedural paths

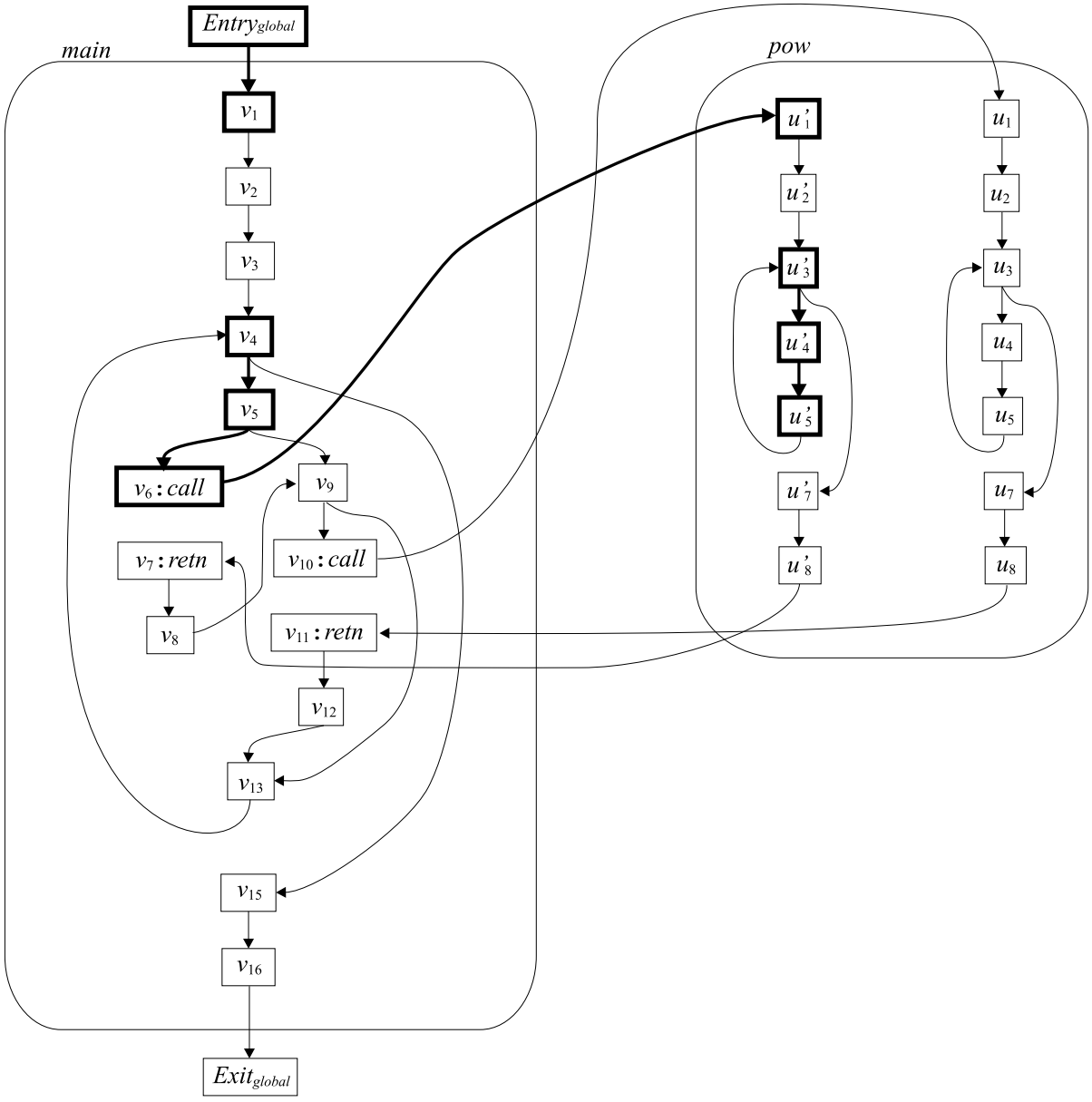


Figure 30: Example hot-path graph for the program shown in Figure 8. Observable path 24 from Figure 9 has been duplicated as an express-lane.

from u_1 to u_3 that end with the backedge $u_5 \rightarrow u_3$. (Note that given any same-level valid path p from x to y, we can create an intraprocedural path from x to y by removing any piece of p that represents the execution of a procedure call.)

When we create an express-lane version of path number 24, we will create copies of the path's context-prefix and its active-suffix. The copy of the context-prefix will end at a copy u'_1 of the vertex u_1. The copy of the active-suffix will begin at a copy u'_3 of the vertex u_3. In the hot-path graph (which is to contain the express-lane version of path number 24), we desire that any time execution reaches u'_3, it came along a path from u'_1; we want to make sure that the duplicated active-suffix executes in the context of the duplicated context-prefix. To ensure this, we may have to duplicate intraprocedural paths (*i.e.*, paths consisting of intraprocedural and summary edges) from u_1 to u_3 as paths from u'_1 to u'_3.

Figure 30 shows a hot-path graph version of the program in Figure 8 with an express-lane version of observable path number 24. The vertices of the express-lane are shown in bold. There are several notable features:

1. The active-suffix $[u'_3 \rightarrow u'_4 \rightarrow u'_5]$ can only be reached from the context-prefix $[\ldots \rightarrow v_5 \rightarrow v_6 \rightarrow u'_1]$. There is no way to reach the active-suffix from any other context-prefix (*e.g.*, there is no same-level valid path from u_1 to u'_3.

2. In order to achieve the effect noted in 1, it was necessary to duplicate vertices that are not part of path number 24. Specifically, we had to duplicate u_2, u_7, and u_8. In effect, for this example, we had to clone the procedure *pow*.

3. There are branches into the middle of the express-lane, which contradicts our previous definition of an (intraprocedural) express-lane. Specifically, there are multiple branches to the vertices u'_3 and v_4. However, note that there are no branches to the middle of the active-suffix $[u'_3 \rightarrow u'_4 \rightarrow u'_5]$.

The intraprocedural express-lane transformation has some similar properties. For example, in the hot-path graph shown in Figure 6, several vertices have been duplicated that do not belong to any hot-path; these vertices represent residual code that must be duplicated even though it is not hot. □

We are now ready to give a technical definition of an interprocedural-express lane. Let G^* be a supergraph and let p be an observable path in G^*_{fin}. The path p may be from either an interprocedural, context path profile or an interprocedural, piecewise path profile. Let H^* be a supergraph where every vertex of H^* is a copy of a vertex in G^* and the set of unbalanced-left paths in H^* is the same as in G^* (under a mapping of vertices in H^* to their originals in G^*). Then an *express-lane* version of p is a sequence of vertices $[a_1, a_2, \ldots, a_n]$ in H^* such that the following properties are satisfied:

Duplication property: a_i is a copy of the i^{th} vertex in p. If a vertex v occurs in p at positions j and k, then a_j and a_k are distinct copies of v.

Minimal predecessor property: A vertex a_i may have multiple predecessors if $a_i \equiv a_1$, or the $(i-1)^{th}$ edge of p is a surrogate edge, or a_i is a copy of a return-site vertex; otherwise a_i has exactly one predecessor, which is a_{i-1}. If a_i is a copy of return-site vertex r then there are further restrictions. Let c be the call vertex associated with r. The following must be satisfied:

- If there is a copy of c in $[a_1 \ldots a_{i-1}]$, then a_i is associated with one call vertex, the last copy of c in $[a_1 \ldots a_{i-1}]$; otherwise, a_i may be associated with many call vertices.

- If a_{i-1} is a copy of an exit vertex, then a_i is targeted by exactly one return-edge, $a_{i-1} \rightarrow a_i$; otherwise, $a_i \equiv a_1$ or a_{i-1} is a copy of a call vertex, and a_i may be targeted by multiple return-edges.

Context property: For a vertex a_i in procedure P, if there is a copy of $Entry_P$ in $[a_1 \ldots a_i]$, then a_i can reached by a same-level valid path from the last copy of $Entry_P$ in $[a_1 \ldots a_i]$ and not from any other copy of $Entry_P$.

The next two sections give more details on the second and third properties.

7.2.1 The Minimal Predecessor Property

The minimal predecessor property states the conditions under which there may be branches into the express-lane. Ideally, this would only be at the beginning of the express-lane. However, the definition allows several exceptions that are needed to define an interprocedural express-lane. One of these is for handling surrogate edges for an observable path from a context path profile. A surrogate edge $Entry_P \rightarrow v$ stands in for any same-level valid path from $Entry_P$ to v that ends with a recording edge; thus many paths may merge at the express-lane's copy of v.

The other exceptions involve observable paths that include a return vertex. A return vertex r has at least two predecessors: its intraprocedural predecessor, the call vertex c associated with r, and its interprocedural predecessor, the exit vertex $Exit_P$ for the procedure called by c. (Technically, c is not a predecessor of r in our definition of a supergraph; however, most uses of a supergraph (*e.g.*, data-flow analysis) treat c and r as intraprocedural predecessor and successor of one another.) If a hot path p is an unbalanced-right path, then p might include r but not c. In this case, p does not contain any information about the (hot) path taken to c. In the hot-path supergraph, there may be many copies of c, each encoding a different express-lane path taken to c. Some of these paths may continue to reach r along the hot path p. Thus, p's copy of r may be associated with multiple copies of c.

If a hot path p starts with r, or contains the summary-edge $c \rightarrow r$, then p has an occurrence of r but not an occurrence of r's interprocedural predecessor, $Exit_P$. This means that the hot-path supergraph may contain many copies $Exit'_P$ of $Exit_P$ such that there is a return-edge from $Exit'_P$ to p's copy of r.

7.2.2 The Context Property

The context property inductively guarantees that the context summarized by a hot path p is duplicated in the express-lane for p. Every vertex a_i must execute in the context summarized by p for the i^{th} vertex of p. If the context property holds for every vertex a_{j+1} such that the j^{th} edge of p is a surrogate edge, then the minimal predecessor property ensures that the context property holds for every vertex of the express-lane. This follows since the minimal predecessor property strongly restricts the paths that can reach most vertices of the express-lane.

The context property is trivially satisfied for paths from a piecewise path profile, providing that the other express-lane properties are satisfied. However, satisfying the context property for a path that contains a surrogate edge requires the duplication of vertices that do not occur in the path (and hence in the express-lane version of the path).

In order to create an express-lane of an observable path p containing a surrogate edge $Entry_P \rightarrow v$ it may be necessary to duplicate many vertices in the procedure P that are not on the path p (see Example 7.2.1).

7.3 Performing the Interprocedural, Express-Lane Transformation

We now present two algorithms for performing the interprocedural express-lane transformation, one for interprocedural, piecewise path profiles and one for interprocedural, context path profiles. Each algorithm takes as input a program supergraph G^* and a set of hot paths, and produces as output a hot-path supergraph H^* that contains an express-lane version of each hot-path given as input. G^* and H^* are semantically equivalent in that they have the same set of paths (and hence the same possible executions). This is stated formally in Theorem 7.4.1.

Recall that in the Ammons-Larus approach to performing the intraprocedural, express-lane transformation, a deterministic finite automaton (DFA) that recognizes hot-paths is combined with the control-flow graph to create the hot-path graph. Each vertex $[v, q]$ of the hot-path graph is a combination of vertex v in the control-flow graph and a state q of the hot-path automaton. The Ammons-Larus approach considers the control-flow graph (for the procedure P) to be a DFA that recognizes valid execution sequences (in the procedure P). (Under this interpretation, both the hot-path recognizer and the control-flow-graph-as-DFA take as input a string of control-flow-graph edges.) Thus, constructing the hot-path graph involves combining two DFA's.

For the interprocedural, express-lane transformation, we will construct a family of automata $\mathcal{A}$, one for each procedure P: the automaton for procedure P, called A_P, will recognize hot-paths that begin in procedure P. A_P is also called the hot-path automaton for P. The Interprocedural Hot-path Tracing Algorithm (or Hot-path Tracing Algorithm for short) combines the hot-path automata in $\mathcal{A}$ with the program supergraph to create the hot-path supergraph. Each vertex $[v, q]$ of the hot-path supergraph is a combination of a vertex v in the supergraph and a state q in an automaton from $\mathcal{A}$. We consider the program supergraph to specify a pushdown automaton (PDA) that recognizes valid execution sequences in the program. Under this interpretation, both the automata in $\mathcal{A}$ and the supergraph-as-PDA take as input a string of supergraph edges. The bulk of the work done by the Hot-path Tracing Algorithm consists of taking a transition $(q_i, u \rightarrow v, q_j)$ of a hot-path automaton and combining it with the supergraph edge $u \rightarrow v$ (which is considered to be a transition $(u, u \rightarrow v, v)$ in the supergraph-as-PDA) to create the hot-path supergraph edge $[u, q_i] \rightarrow [v, q_j]$.

The Hot-path Tracing Algorithm will treat the automata in $\mathcal{A}$ as DFAs, though technically they are not: an interprocedural hot path p may contain "gaps" that are represented by surrogate- or summary-edges. These gaps may be filled by same-level valid paths. This means that an automaton that recognizes the hot-path p requires the ability to skip over same-level valid paths in the input string. Since same-level valid paths can only be generated with a context-free grammar (and not with a regular grammar), a pushdown automaton is required to skip over same-level valid paths. This means that the Hot-path Tracing Algorithm must combine two PDA's; generally, this problem is undecidable. However, we will treat the hot-path automata as DFAs; the Hot-path Tracing Algorithm combines a set of DFAs and a PDA.

As stated above, we specify the automata in $\mathcal{A}$ as DFAs; we do not specify any stack or stack operations for the automata in $\mathcal{A}$. Instead, we allow the automata to have transitions that are labeled with summary-edges. A transition $(q_i, c \rightarrow r, q_j)$ that is labeled with a summary-edge $c \rightarrow r$ is considered to be an oracle transition that is capable of skipping over a $SLVP_1$-path in the input string.[1] The oracle required to skip an $SLVP_1$-path is the supergraph-as-PDA. Note that oracle transitions are sufficiently powerful to allow a hot-path automaton to handle the gaps that may occur in a hot path.

[1]Recall from Section 3.1 that an $SLVP_1$-path is a same-level valid path that consists of an open parenthesis $(_c$, followed by a same-level valid path, followed by a close parenthesis $)_c$; a call-vertex c and its matching return vertex r are always connected by an $SLVP_1$-path.

When we combine a hot-path automaton with the supergraph, an oracle transition $(q_i, c \to r, q_j)$ that is labeled with a summary-edge $c \to r$ will be combined with the summary-edge $c \to r$ of the supergraph to create the vertices $[c, q_i]$ and $[r, q_j]$ and the summary-edge $[c, q_i] \to [r, q_j]$ in the hot-path supergraph. The justification for this is that the set of $SLVP_1$-paths that an oracle transition $(q_i, c \to r, q_j)$ should skip over is precisely the set of $SLVP_1$-paths that drive the supergraph-as-PDA from c to r.

Technically, the use of oracle transitions means that the hot-path automata in $\mathcal{A}$ are not DFAs. Furthermore, a summary-edge is not actually a supergraph edge, rather it is an annotation on the supergraph; this means that a summary-edge does not represent a transition in the supergraph-as-PDA. However, our technique for using oracle transitions and combining them with summary-edges in the supergraph allows us to treat the hot-path automata as DFAs, and therefore simplifies the task of combining the hot-path automata and the supergraph.

Our approach to constructing the hot-path supergraph will consist of three phases:

1. Construct a family $\mathcal{A}$ of automata with one automaton A_p for each procedure P. The automaton A_P is specified as a DFA that recognizes (prefixes of) hot-paths that begin in P.

2. Use the Interprocedural, Hot-path Tracing Algorithm to combine $\mathcal{A}$ with the supergraph G^* to generate an initial hot-path supergraph. This step creates all of the intraprocedural path pieces that may be needed in the hot-path supergraph. An intraprocedural path may be needed in the hot-path graph if it is part of an express-lane, or a part of residual code. This stage partially connects the intraprocedural pieces by adding call-edges where appropriate. Summary are added only if they are part of an express-lane.

3. Make a pass over the generated hot-path supergraph to add return-edges and summary-edges where appropriate. This stage finishes connecting the intraprocedural paths created in the previous step.

Throughout each of the steps, it is instructive to recall the process that is used to collect a path profile: wherever the path-profiling machinery begins recording a path, the interprocedural express-lane transformation may begin a new express-lane. Wherever the path-profiling machinery may resume recording of a path (*i.e.*, at surrogate and summary edges), the interprocedural express-lane transformation may resume creation of an express-lane.

The two algorithms for performing the interprocedural express-lane transformation (one for piecewise path profiles and one for context path profiles) differ slightly in the first step. We will start by describing the automata for recognizing hot-paths from an interprocedural, piecewise path profile. Then we will describe the automata for recognizing hot-paths from interprocedural, context path profiles. Finally, we will describe phases two and three, which are the same for both algorithms. Throughout these sections, our examples use the program shown in Figure 31.

7.3.1 The Hot-Path Automata for Interprocedural, Piecewise Paths

In this section, we show how to construct the set $\mathcal{A}$ of hot-path automata for recognizing hot interprocedural, piecewise paths. We will expand our definition of $\mathcal{A}$ in that we will allow each automaton $A_P \in \mathcal{A}$ to transition to other automata in $\mathcal{A}$; thus, it is more accurate to think of $\mathcal{A}$ as one large automaton with several sub-automata.

As in [5], we will build a hot-path automaton for recognizing a set S of hot paths by building a trie A of the paths in S and defining a failure function that maps a vertex of the trie and a supergraph edge to another vertex of the trie. We then consider A to be a DFA where the edges of the trie and the failure

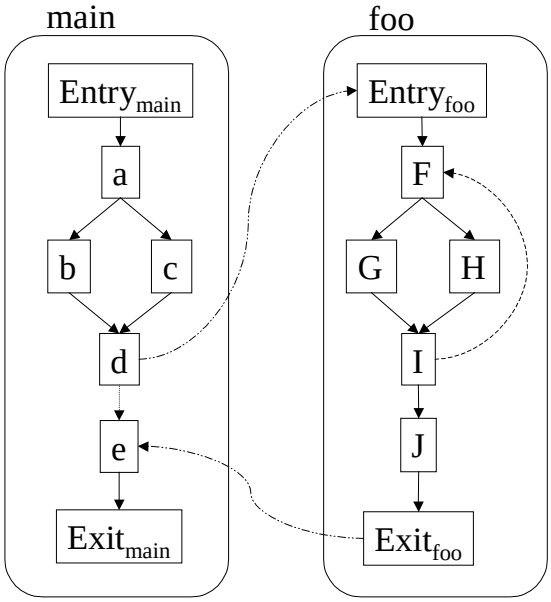


Figure 31: Supergraph used in examples of the interprocedural express-lane transformation.

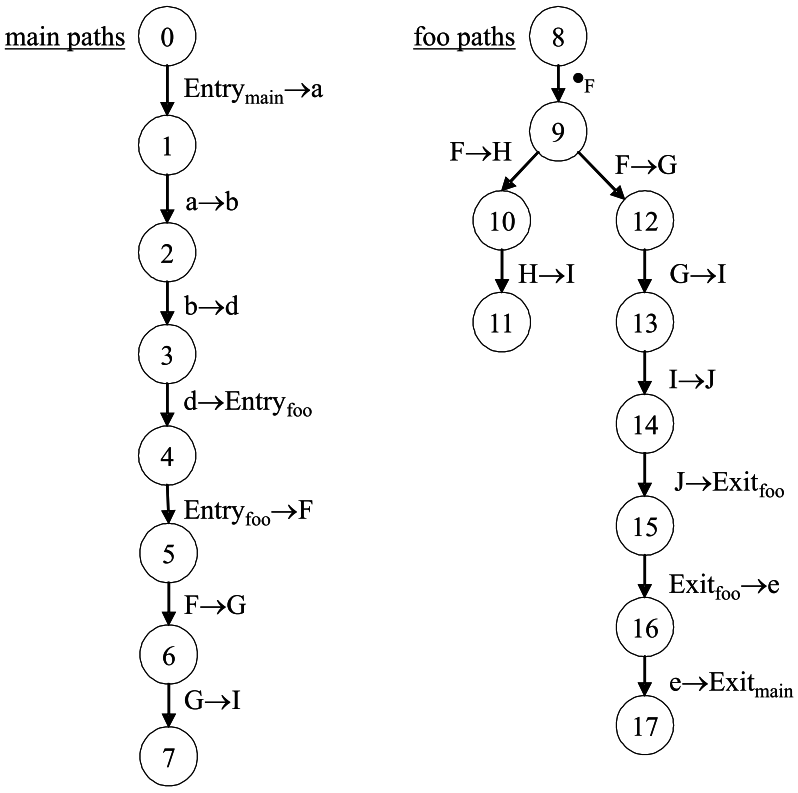


Figure 32: Path trie for an interprocedural, piecewise path profile of the supergraph in Figure 31. For $i \in [4..15]$ and a recording edge e in *foo*, $h(q_i, e) = q_9$; For $i \in [4..15]$ and a non-recording edge e in *foo*, $h(q_i, e) = q_8$. For $i \in ([0..3] \cup [16..17])$ and an edge e in *main*, $h(q_i, e) = q_0$.

function define the transition relation of the DFA. Accordingly, we refer to the trie as an automaton, and refer to the vertices of the trie as states.

For each procedure P, we create a trie of the hot paths that start in P. Similar to the intraprocedural express-lane transformation (see Section 2.2), hot paths that can only be reached by following a recording edge $u \rightarrow v$ are prefixed with the special symbol $\bullet_v$ before they are put in the trie. A transition in the trie that is labeled by $\bullet_v$ can match any recording edge that targets v. Figure 32 shows the path tries for the supergraph in Figure 31 and the following interprocedural, piecewise paths:

$$Entry_{main} \rightarrow a \rightarrow b \rightarrow d \rightarrow Entry_{foo} \rightarrow F \rightarrow G \rightarrow I$$
$$\bullet_F\ F \rightarrow H \rightarrow I$$
$$\bullet_F\ F \rightarrow G \rightarrow I \rightarrow J \rightarrow Exit_{foo} \rightarrow e \rightarrow Exit_{main}$$

Every hot-path prefix corresponds to a unique state in a path trie (which can be found by starting at the root of the trie and using the edges of the hot-path prefix to drive transitions in the trie). If a hot-path prefix ends at a vertex v and the prefix corresponds to state q other than the root state, then we say that q represents v; the root of the path trie for procedure P is said to represent $Entry_P$. The fact that q represents the vertex v is important, since for a vertex $[v, q]$ in the output hot-path graph, either $[v, q]$ is not on an express-lane and q represents an entry vertex, or q represents v. A transition $(q_0, u \rightarrow v, q_1)$ implies that q_0 represents the vertex u and q_1 represents the vertex v. In Figure 32, the state q_2 represents the vertex b. The state q_9 represents the vertex F.

As in Section 2.2, we define a failure function $h(q, u \rightarrow v)$ for a state q of any trie and an intraprocedural or summary-edge $u \rightarrow v$; the failure function is not defined for interprocedural edges. If q represents a vertex w of procedure P and $u \rightarrow v$ is not a recording edge, then $h(q, u \rightarrow v) = root\_trie_P$, where $root\_trie_P$ is the root of the trie for hot paths beginning in P. Otherwise, if $u \rightarrow v$ is a recording edge, then $h(q, u \rightarrow v) = q_{\bullet_v}$, where $q_{\bullet_v}$ is the target state in the transition $(root\_trie_P, \bullet_v, q_{\bullet_v})$; if there is no transition $(root\_trie_P, \bullet_v, q_{\bullet_v})$, then $q_{\bullet_v} = root\_trie_P$. Recall that when the piecewise path-profiling instrumentation is in a procedure P when it stops recording a path, it then begins to record a path that starts in procedure P. Similarly, when a hot-path automaton steps off of a hot-path in the procedure P, the failure function switches to the hot-path automaton that is responsible for recognizing hot-paths that begin in P. See the caption of Figure 32 for an example failure function.

The later phases of the express-lane transformation will make use of two functions, *LastActiveCaller* and *LastEntry*, which map trie states to trie states. For a state q that represents a vertex in procedure P, $LastActiveCaller(q)$ maps to the most recent ancestor of q that represents a call vertex that makes a non-recursive call to P. $LastEntry(q)$ maps to the most recent ancestor of q that represents $Entry_P$. $LastActiveCaller(q)$ and $LastEntry(q)$ are undefined for q if there is no appropriate ancestor of q in the trie. If the function $LastActiveCaller$ is undefined, then the state q must correspond to a hot-path prefix that is unbalanced-right. This is useful information for performing the interprocedural, express-lane transformation.

7.3.2 The Hot-Path Automata for Interprocedural, Context Paths

As in the previous section, a path trie is created for each procedure. Before a path is put into a trie, each surrogate edge $u \rightarrow v$ is replaced by an edge labeled with $\bullet_v$. As before, $\bullet_v$ matches any recording edge that targets v. (Recall that a surrogate edge $u \rightarrow v$ fills in for any same-level valid path that ends in a recording edge to the vertex v.) Figure 33 shows the path tries for the supergraph in Figure 31 and the following interprocedural, context paths:

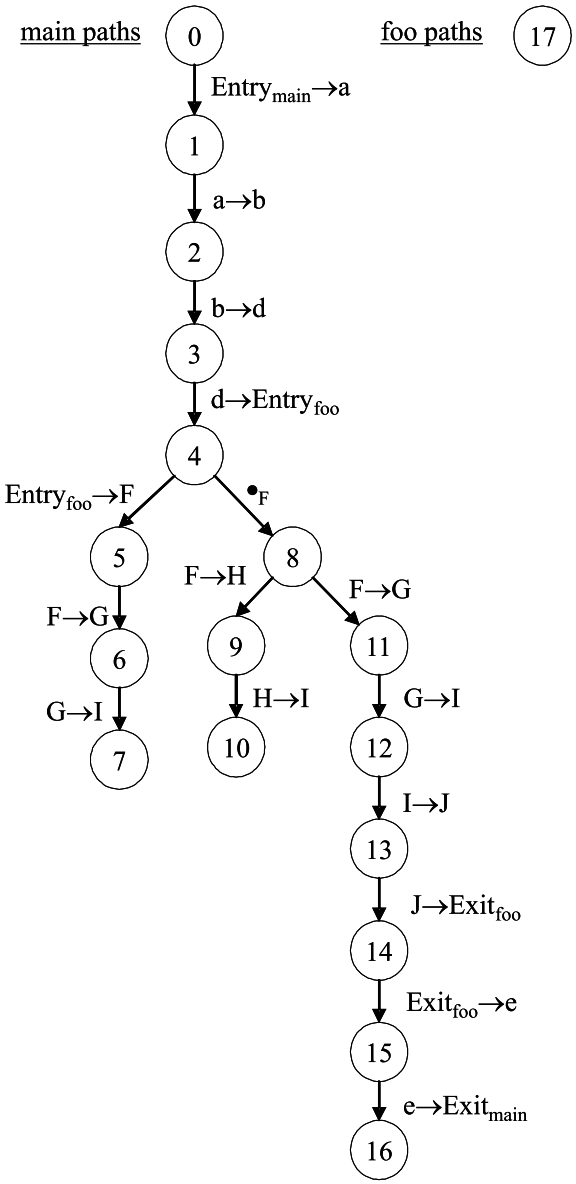


Figure 33: Path trie for an interprocedural, context path profile of the supergraph in Figure 31. For $i \in ([0..3] \cup [15..16])$ and an edge e in *main*, $h(q_i, e) = q_0$. For $i \in [4..14]$ and a recording edge e in *foo*, $h(q_i, e) = q_4$; for $i \in [4..14]$ and a non-recording edge e in *foo*, $h(q_i, e) = q_8$. For q_{17} and any edge e in *foo*, $h(q_{17}, e) = q_{17}$.

$$Entry_{main} \rightarrow a \rightarrow b \rightarrow d \rightarrow Entry_{foo} \rightarrow F \rightarrow G \rightarrow I$$
$$Entry_{main} \rightarrow a \rightarrow b \rightarrow d \rightarrow Entry_{foo} \bullet_F \ F \rightarrow H \rightarrow I$$
$$Entry_{main} \rightarrow a \rightarrow b \rightarrow d \rightarrow Entry_{foo} \bullet_F \ F \rightarrow G \rightarrow I \rightarrow J \rightarrow Exit_{foo} \rightarrow e \rightarrow Exit_{main}$$

A state q that represents an entry vertex $Entry_P$ corresponds to a hot-path prefix p that leads to the vertex $Entry_P$. The path p describes a calling context for the procedure P. This means that p could be the context-prefix for a hot-path. For this reason, states in the trie that represent entry vertices take on special importance in this section. Also, the map *LastEntry* will be important.

We define the maps *LastActiveCaller* and *LastEntry* as in the last section. A principal differnce with the last section is in the definition of the failure function. In this case, if $u \rightarrow v$ is not a recording edge, then $h(q, u \rightarrow v) = LastEntry(q)$. If $u \rightarrow v$ is a recording edge, then $h(q, u \rightarrow v) = q'$, where q' is the state reached by following the transition labeled $\bullet_v$ from $LastEntry(q)$; if there is no such state, the $q = LastEntry(q)$. Recall that when the context path-profiling instrumentation stops recording a path in procedure P, it restores the path number to the value it had when P was entered. Likewise, when the hot-path automaton steps off a hot path in procedure P, the failure function resets the recognizer to the state that corresponds to the path taken to P.

Before giving a detailed presentation of the Hot-path Tracing Algorithm, we give some intuition for how the Hot-path Tracing Algorithm will interact with an automaton for interprocedural context paths. Let us consider what happens when the Ammons-Larus technique is tracing out a cold path $[u \rightarrow v \rightarrow w \ldots]$: when on a cold path, the hot-path automaton is always at the root state, *root*. Thus, when tracing the cold path $[u \rightarrow v \rightarrow w \ldots]$, the hot-path automaton always stays in the root state, and the tracing algorithm generates the path $[[u, root] \rightarrow [v, root] \rightarrow [w, root] \ldots]$. Only when the tracing algorithm begins tracing a hot path does the hot-path automaton leave the state *root* and the tracing algorithm begin generating vertices $[x, q]$, where q is not $root$. Thus, when tracing a cold path, the tracing algorithm clones pieces of the control-flow graph, since the state of the hot-path automaton always stays stuck at $root$. See Section 2.2 and Figure 6 for an example.

For a context-prefix p that leads to a procedure P, the Interprocedural Hot-path Tracing Algorithm may have to clone parts of P. This is required to make sure that the Context Property is guaranteed for express-lanes that begin with p (see Example 7.2.1). To accomplish this, the Hot-path Tracing Algorithm may generate many vertices $[x, q]$, where q is the automaton state in hot-path automaton A that corresponds to the context-prefix p: when the hot-path automaton A is in the state q and is scanning a path $[u \rightarrow v \rightarrow w \ldots]$ in the procedure P that is cold in the context described by p, the automaton will stay in the state q. Thus, when the Interprocedural Hot-path Tracing Algorithm has traced out the context-prefix p and is in the process of tracing out the path $[u \rightarrow v \rightarrow w \ldots]$, it will generate the path $[[u, q] \rightarrow [v, q] \rightarrow [w, q] \ldots]$. Only when the tracing algorithm begins tracing a path that is hot in the context of p does the hot-path automaton move out of state q.

7.3.3 Step Two: Hot-Path Tracing of Intraprocedural Path Pieces

This section describes the hot-path tracing algorithm that can be used with a family $\mathcal{A}$ of hot-path automata from one of the previous sections. For a hot-path automaton, we define a state q to be a *reset state* if $h(q, u \rightarrow v) = q$ for some non-recording edge $u \rightarrow v$. (In the case of a hot-path automaton for interprocedural, piecewise paths, only the root state is a reset state.) Reset states are important for several reasons: (1) a context-prefix p always drives a hot-path automata to a reset-state; (2) for every vertex $[v, q]$ in the hot-path supergraph that is *not* part of an express-lane (*i.e.*, $[v, q]$ is part of residual, cold code), q is a reset state; and (3) for a reset state q and a hot-path supergraph vertex $[v, q]$, either v

is the entry vertex that q represents, or $[v, q]$ is a cold vertex. We use these facts to determine whether a hot-path supergraph vertex $[v, q]$ is part of an express-lane or not.

Figure 34 shows the Interprocedural Hot-path Tracing Algorithm and Figure 35 shows the function *ProcessCallVertex* used by the algorithm to process a call vertex $[c, q]$. The algorithm uses a worklist: when the algorithm creates a vertex $[v, q]$, it adds the vertex to a worklist W; when $[v, q]$ is removed from the worklist, the algorithm examines the supergraph vertex v and the hot-path-automaton state q to determine what edges $[v, q] \rightarrow [v', q']$ must be created.

The bulk of the work of the Interprocedural Hot-Path Tracing Algorithm is done by lines 19–21 of Figure 34. These lines process each hot-path supergraph vertex $[v, q]$ that is not a call or exit vertex. Given a hot-path supergraph vertex $[v, q]$, a supergraph edge $v \rightarrow v$, and a transition $(q, v \rightarrow v, q)$, lines 19–21 "trace out" a new edge $[v, q] \rightarrow [v', q']$ in the hot-path supergraph. if necessary, a new vertex $[v', q']$ is added to the hot-path supergraph and the worklist W. This part of the Interprocedural Hot-Path Tracing Algorithm is very similar to the Intraprocedural Hot-Path Tracing Algorithm.

The Interprocedural Hot-Path Tracing Algorithm differs from its intraprocedural counterpart in that it must take extra steps when processing call and exit vertices. Figure 35 shows the function $ProcessCallVertex$ that is used to process a call-vertex $[c, q]$. $ProcessCallVertex$ has two responsibilities: (1) it must create call-edges from $[c, q]$ (this is done in lines 25–33 of Figure 35); and (2) it must create return-site vertices that could be needed in Phase 3 of the hot-path supergraph construction (this is done in lines 34–35 of Figure 35). If there is a transition $(q, c \rightarrow Entry_P, q')$, then $[c, q]$ is part of an express-lane that continues along the call-edge $c \rightarrow Entry_P$ and lines 27–29 create an express-lane copy of the call-edge $c \rightarrow Entry_P$. Otherwise, for any call-edge $c \rightarrow Entry_P$, lines 31–33 hook up $[c, q]$ to a cold copy of $Entry_P$.

Lines 34–35 trace the summary-edge $c \rightarrow r$ from c and the transition $(q, c \rightarrow r, q')$ from q in order to create the return-site vertex $[r, q']$. The return-site vertex $[r, q']$ may be used in Phase 3, or it may not: if q' is a reset state, and $[r, q']$ is a cold vertex, then $[r, q']$ will be used if there is a cold path that contains a return-edge $x \rightarrow r$; however, if there is no cold path containing a return-edge $x \rightarrow r$, then the cold return-site vertex $[r, q']$ is unnecessary. In the later case, Phase 3 does not create any edges that target $[r, q']$, so $[r, q']$ is unreachable, and is eventually removed from the hot-path supergraph.

Figures 38 thru 40 show various stages in the construction of the hot-path supergraph for the supergraph in Figure 31 and the hot-path automata in Figure 32.

7.3.4 Step Three: Connecting Intraprocedural Path Pieces

The third phase of the interprocedural express-lane transformation is responsible for completing the hot-path supergraph H^*. It must add the appropriate summary-edges and return-edges.

Formally, this phase of the interprocedural express-lane transformation ensures that the following is true:

For each call vertex $[c, q]$
  For each call-edge $[c, q] \rightarrow [Entry_P, q']$
    For each exit vertex $[Exit_P, q'']$ reachable from $[Entry_P, q']$ by a SLVP
      There must be a return-site vertex $[r, q''']$ such that
        1. There is a summary-edge $[c, q] \rightarrow [r, q''']$
        2. There is a return-edge $[Exit_P, q''] \rightarrow [r, q''']$ labeled "$)_{[c,q]}$"

G^* is supergraph.
$\mathcal{A}$ is a family of hot-path automata, with one automaton for each procedure in G^*.
/* $A_P \in \mathcal{A}$ *denotes the automaton for procedure* P */
/* T_P *denotes the transition relation of* A_P */
$\mathcal{T}$ is the disjoint union of all T_P
$root\_trie_{main}$ is the start state of A_{main}.
W is a worklist of hot-path supergraph vertices.
$H^* \equiv (V, E)$ is the hot-path supergraph.

Main()
/* *First, create all the vertices that might begin a hot-path* */
1: $V = Entry'_{global}, Exit'_{global}$
2: **Foreach** procedure P
3: $CreateVertex([Entry_P, root\_trie_P])$ /* *See below* */
4: **If** there is a transition $(root\_trie_P, \bullet_r, q')$ where r is a return-site vertex
/* *For hot-paths that begin at return-site vertices,* */
/* *create the beginning of the express-lane.* */
5: $CreateVertex([r, q'])$
6: $E = \{Entry'_{global} \rightarrow [Entry_{main}, root\_trie_{main}]\}$

8: **While** $W \neq \emptyset$
9: $[v, q] = Take(W)$ /* *select and remove an element from W* */
10: **If** v is a call vertex
11: $ProcessCallVertex([v, q])$
12: **Else If** v is an exit vertex
13: **ForeachEdge** $v \rightarrow r$ in G^*
14: /* $v \rightarrow r$ *is a return-edge* */
15: **If** there is a transition $(q, v \rightarrow r, q) \in \mathcal{T}$.
16: $CreateVertex([r, q])$ /* *See below* */
17: **Else**
18: **ForeachEdge** $v \rightarrow v$ in G^*
19: Let q' be the unique state such that $(q, v \rightarrow v', q') \in \mathcal{T}$.
20: $CreateVertex([v', q'])$
21: $E = E \cup \{[v, q] \rightarrow [v', q']\}$
21a: **Foreach** vertex $[Exit_{main}, q] \in V$
22b: $E = E \cup \{[Exit_{main}, q] \rightarrow Exit'_{global}\}$
End Main

Figure 34: Interprocedural Hot-Path Tracing Algorithm.

```
     CreateVertex([v, q])
22:      If [v, q] ∉ V
23:          V = V ∪ {[v, q]}
24:          Put(W, [v, q])
     End CreateVertex


     ProcessCallVertex([c, q]) /* c is a call vertex */
25:      Let r be the return-site vertex associated with c
         /* Create call edges to all appropriate entry vertices */
26:      ForeachEdge c → Entry_P
             /* v may have many callees if it is an indirect call-site */
27:          If (q, c → Entry_P, q') ∈ T
                 /* There is a hot path continuing from c along the edge c → Entry_P */
28:              CreateVertex([Entry_P, q'])
29:              E = E ∪ {[c, q] → [Entry_P, q']}
29a:             Label [c, q] → [Entry_P, q'] with "(_[c,q]"
30:          Else
                 /* Hook up [c, q] to a cold copy of Entry_P */
31:              CreateVertex([Entry_P, root_trie_P])
32:              E = E ∪ {[c, q] → [Entry_P, root_trie_P]}
33:              Label the call-edge [c, q] → [Entry_P, root_trie_P] with "(_[c,q]"

         /* Create every return-site vertex [r,q'] that could be needed in phase 3 */
34:      Let q' be the unique state such that (q, v → r, q') ∈ T
35:      CreateVertex([r, q'])
         End ProcessCallVertex
```

Figure 35: The procedures *CreateVertex* and *ProcessCallVertex* used by the algorithm in Figure 34.

The algorithm for performing Phase 3 of the interprocedural express-lane transformation is shown in Figure 36. The algorithm uses a worklist of call-site vertices. For each call-edge $[c, q_0] \rightarrow [Entry_P, q_1]$, for each exit vertex $[Exit_P, q_2]$ such that there is a same-level valid path from $[Entry_P, q_1]$ to $[Exit_P, q_2]$, the algorithm finds a return-site vertex $[r, q_3]$ such that r is the return-site vertex associated with c in the supergraph and adds a summary-edge $[c, q_0] \rightarrow [Entry_P, q_1]$ and a return-edge $[Exit_P, q_2] \rightarrow [r, q_3]$ labeled "$)_{[c,q_0]}$". The return-site vertex $[r, q_3]$ is chosen as follows:

1. If there is a hot-path-automaton transition $(q_2, Exit_P \rightarrow r, q')$, then the exit vertex $[Exit_P, q_2]$ is part of an express-lane. The next vertex of the express-lane is $[r, q']$, so $[r, q_3]$ is chosen to be $[r, q']$. (See lines 48–51 of Figure 36.)

2. Otherwise, $[Exit_P, q_2]$ is not part of an express-lane, or is at the end of an express-lane. If there is a hot-path p that begins with r, $[r, q_3]$ is chosen to be the first vertex of the express-lane version of p. (See lines 53–56 and 64–70.)

3. Otherwise, $[r, q_3]$ is chosen to be $[r, q']$ where q' is the unique state determined by the summary-edge transition $(q_0, c \rightarrow r, q')$. In this case, $[r, q_3]$ may be a cold vertex, or it may be part of an express-lane for a hot-path that includes the summary-edge $c \rightarrow r$. (See lines 53–56 and 72–73.)

The algorithm uses a mapping *Exits* that maps each entry vertex $[Entry_P, q]$ to the set of exit vertices reachable from $[Entry_P, q]$ along a same-level valid path. This mapping is maintained by calculating the vertices reachable from each entry vertex $Entry_P$ along intraprocedural and summary-edges; as more summary-edges are added to the hot-path supergraph, the mapping must be updated (see lines 57–63 of Figure 36). When the mapping *Exits* changes, it means that more same-level valid paths have been discovered. Call vertices are added to the worklist (at line 63) to ensure that the new reachability information is accounted for.

Figures 41 and 42 shows the hot-path supergraph for the hot-path automata in Figures 32 and 33, respectively, and the supergraph in Figure 31.

7.4 Graph Congruence of the Supergraph and the Hot-path Supergraph

In this section, we show that a hot-path supergraph H^* created from a supergraph G^*, has the same execution behavior as G^*. We have the following definitions:

Congruent edges: Edges $u \rightarrow v$ and $u' \rightarrow v'$ are said to be *congruent* if u and u' are "duplicate vertices" (*i.e.*, duplicates of the same supergraph vertex), v and v' are duplicate vertices, and $u \rightarrow v$ and $u' \rightarrow v'$ are the same kind of edge (*e.g.*, a true branch, a false branch, a "case c:"-branch, return-edge, etc.). The labels on call- and return-edges do not affect congruence.

Congruent paths: Paths p and p' are said to be *congruent* iff they are of equal length and, for all i, the i^{th} edge of p is congruent to the i^{th} edge of p'.

Path congruent graphs: Graphs G and G' are *path congruent* iff: for every path in G there is a congruent path in G'; and for every path in G' there is a congruent path in G. *Unbalanced-left path congruent* is defined in an analogous fashion.

If two graphs are unbalanced-left congruent, then they have the same execution behavior. This follows from the fact that any prefix of a program's execution trace is an unbalanced-left path. We have the following theorem:

$H^* \equiv (V, E)$ is the partially completed hot-path supergraph.
$\mathcal{A}$ is the set of hot-path automata
$\mathcal{T}$ is the disjoint union of the transition relations for the automata in $\mathcal{A}$
Exits is a mapping from each entry vertex to the set of exit vertices
that are reachable by a same-level valid path.
W is a worklist of call vertices.

Main()
36: Add every call vertex $[c, q]$ of H^* to W
37: Perform reachability over intraprocedural and summary-edges to initialize *Exits*

38: **While** $W \neq \emptyset$
39: $[c, q] = Take(W)$
45: Let r be the return vertex associated with c in the supergraph.
46: **ForeachEdge** $[c, q] \rightarrow [Entry_P, q']$
47: **Foreach** $[Exit_P, q''] \in Exits([Entry_P, q'])$
48: **If** $(q'', Exit_P \rightarrow r, q''') \in \mathcal{T}$
49: $E = E \cup \{[Exit_P, q''] \rightarrow [r, q''']\}$
50: Label the return-edge $[Exit_P, q''] \rightarrow [r, q''']$ with "$)_{[c,q']}$"
51: $AddSummaryEdge([c, q], [r, q'''])$
52: **Else**
53: $[r, q'''] = GetDefaultRtn([c, q])$
54: $E = E \cup \{[Exit_P, q''] \rightarrow [r, q''']\}$
55: Label the return-edge $[Exit_P, q''] \rightarrow [r, q''']$ with "$)_{[c,q']}$"
56: $AddSummaryEdge([c, q], [r, q'''])$
End Main

Figure 36: Algorithm for the third phase of the interprocedural express-lane transformation. See also Figure 37.

AddSummaryEdge$([c,q],[r,q'])$
57: P is the procedure containing $[c,q]$ and $[r,q']$
58: **If** $[c,q] \rightarrow [r,q'] \notin E$
59: $E = E \cup [c,q] \rightarrow [r,q']$
60: Use reachability over summary- and intraprocedural edges to update *Exits*
61: **Foreach** $[Entry_P, t]$ where $Exits([Entry_P, t])$ changed
62: **ForeachEdge** $[c',t'] \rightarrow [Entry_P, t]$
63: $Put(W, [c',t'])$
End AddSummaryEdge

GetDefaultRtn$([c,q])$
64: Let r be the return vertex associated with c in the supergraph.
65: P is the procedure containing c and r.
66: **If** $(root\_trie_P, Entry_P \rightarrow r, q') \in \mathcal{T}$
70: **Return** $[r,q']$
71: **Else**
72: Let q' be the unique state such that $(q, c \rightarrow r, q') \in \mathcal{T}$
73: **Return** $[r,q']$
End GetDefaultRtn

Figure 37: Auxially functions for Figure 36.

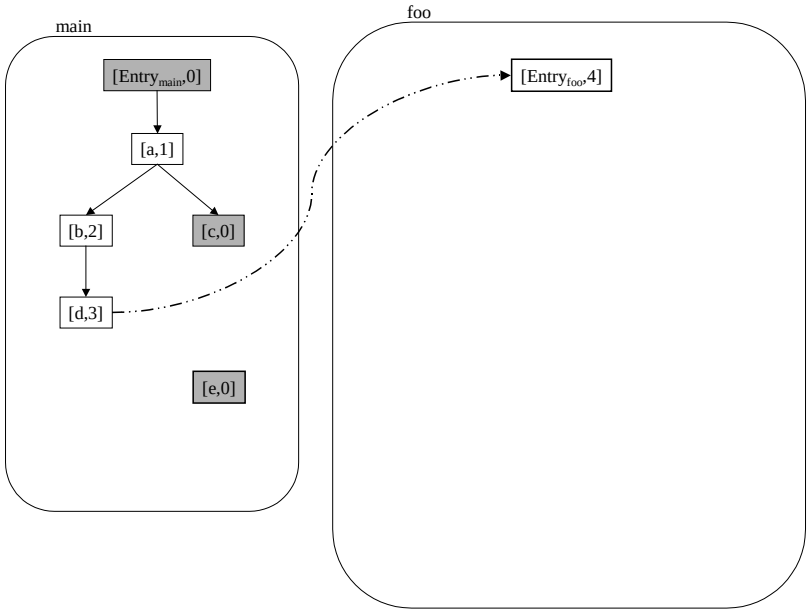


Figure 38: Stage during the hot-path tracing algorithm (see Figure 34) while operating on the supergraph in Figure 31 and the hot-path automata shown in Figure 32. The call vertex $[d,3]$ has just been processed; the vertices $[e,0]$ and $[Entry_{foo},4]$ and the call-edge $[d,3] \rightarrow [Entry_{foo},4]$ were added as a result.

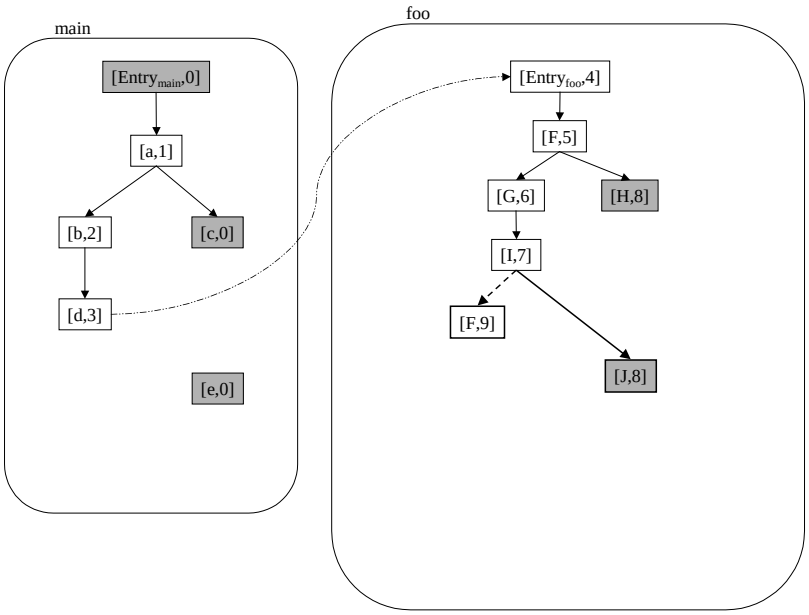


Figure 39: Stage during the hot-path tracing algorithm (see Figure 34) while operating on the supergraph in Figure 31 and the hot-path automata shown in Figure 32. The vertex $[I, 7]$ has just been processed; the vertices $[F, 9]$ and $[J, 8]$ and the edges $[I, 7] \to [F, 9]$ and $[I, 7] \to [J, 8]$ were added as a result.

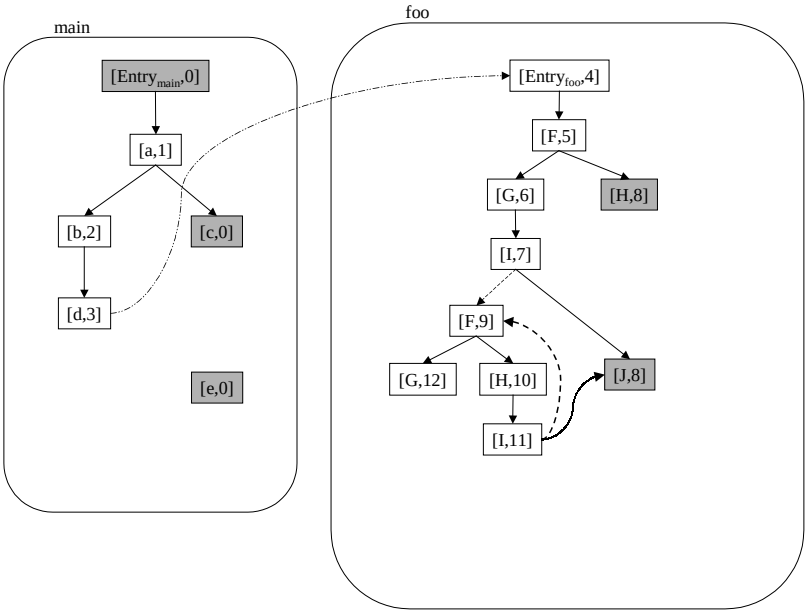


Figure 40: Stage during the hot-path tracing algorithm (see Figure 34) while operating on the supergraph in Figure 31 and the hot-path automata shown in Figure 32. The vertex $[I, 11]$ has just been processed; the edges $[I, 11] \to [F, 9]$ and $[I, 11] \to [J, 8]$ were added as a result. The vertices $[F, 9]$ and $[J, 8]$ were already present.

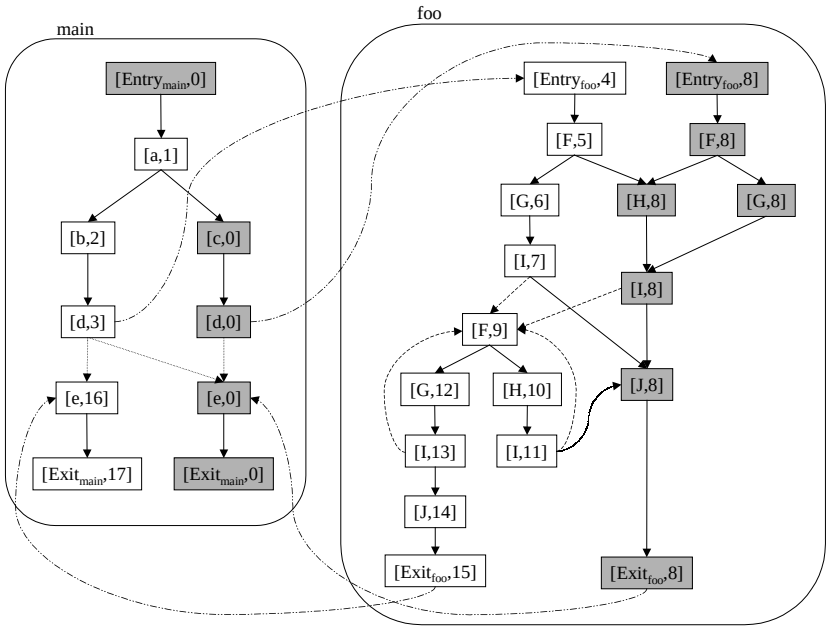


Figure 41: Hot-path supergrah for the supergraph in Figure 31 and the hot-path automaton shown in Figure 32. Most of the graph is constructed during Phase 2 of the construction. The edges $[d,3] \rightarrow [e,16]$, $[d,3] \rightarrow [e,16]$, $[d,3] \rightarrow [e,0]$, $[d,0] \rightarrow [e,0]$, and $[Exit_{foo},8] \rightarrow [e,0]$ are added during Phase 3. Each shaded vertex $[v,q]$ has a state q that is a reset state; with the exception of $[Entry_{main},0]$ and $[Entry_{foo},4]$, these are cold vertices that do not appear on any express-lane.

Theorem 7.4.1 *Let H^* be the hot-path supergraph produced when the Interprocedural Hot-path Tracing Algorithm is run on the supergraph G^* and the path profile pp. G^* and H^* are unbalanced-left congruent.*

Proof: We must show that every unbalanced-left path in G^*, there is a congruent path in H^* that is unbalanced-left, and that for every unbalanced-left path in H^*, there is a congruent path in G^* that is unbalanced-left. First, we use induction to show that for every unbalanced-left path p in G^* *that starts at* $Entry_{global}$, there is a congruent, unbalanced-left path p' in H^*:

Base case: p is the path of length 0 from $Entry_{global}$ to $Entry_{global}$. Then the path p' of length 0 from $Entry'_{global}$ to $Entry'_{global}$ is congruent to p and is an unbalanced-left path. Here, $Entry'_{global}$ denotes the root vertex of the hot-path supergraph.

Induction step: For our induction hypothesis, we assume that for all unbalanced-left paths in G^* that starts at $Entry_{global}$ and have i or fewer edges, there is a congruent, unbalanced-left path in H^* that starts at $Entry'_{global}$. Let p be an unbalanced-left path in G^* that has $i+1$ edges. We will show that there is a congruent unbalanced-left path p' in H^* that starts at $Entry'_{global}$. There are two cases that we must consider:

1. The last edge of p is a return-edge. Then there are (in G^*) an unbalanced-left path a, a call-edge $c \rightarrow e$ labeled "$(_c$", same-level valid path b, and a return-edge $x \rightarrow r$ labeled "$)_c$" such that $p = [a \| c \rightarrow e \| bx \rightarrow r]$; this follows from the fact that p is an unbalanced-left path. By our induction hypothesis, there are (in H^*) an unbalanced-left path a', a call-edge

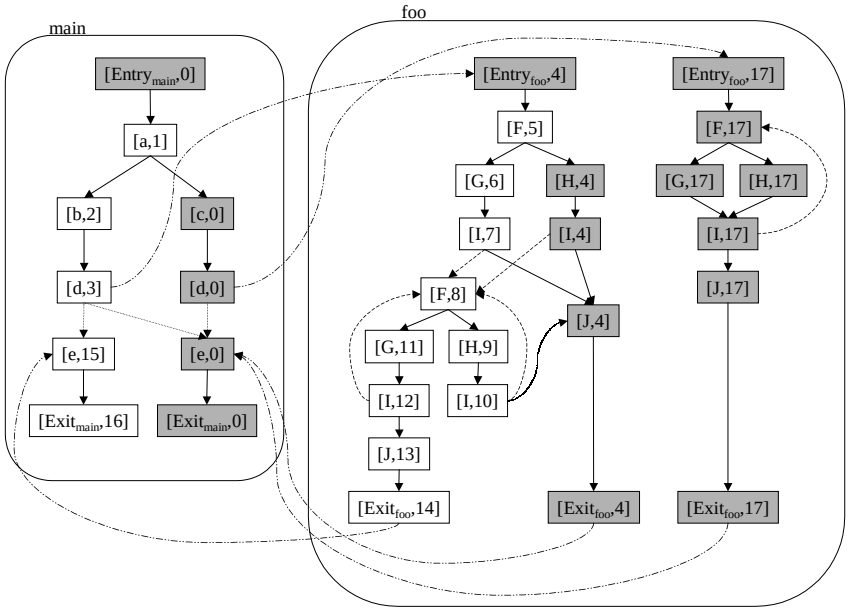


Figure 42: Hot-path supergrah for the hot-path automaton shown in Figure 33 and the supergraph in Figure 31. Most of the graph is constructed during Phase 2. The edges $[d,3] \rightarrow [e,15]$, $[d,3] \rightarrow [e,0]$, $[d,0] \rightarrow [e,0]$, and $[Exit_{foo},17] \rightarrow [e,0]$ are added during Phase 3. Each shaded vertex $[v,q]$ has a state q that is a reset state; with the exception of $[Entry_{main},0]$ and $[Entry_{foo},4]$, these are cold vertices that do not appear on any express-lane.

$c' \rightarrow e'$ labeled "$(_{c'}$", and a same-level valid path b' from e' to x' such that $[a'\|c' \rightarrow e'\|b']$ is congruent to $[a\|c \rightarrow e\|b]$. If we can find a return-edge $x' \rightarrow r'$ in H^* that is labeled "$)_{c'}$" and is congruent to $x \rightarrow r$, then $p' = [a'\|c' \rightarrow e'\|b'\|x' \rightarrow r']$ is an unbalanced-left path in H^* that is congruent to p. The call-edge $c' \rightarrow e'$ and the same-level valid path b' from e' to x' imply that the third phase of the interprocedural express-lane transformation added a return-edge $x' \rightarrow r'$ labeled "$)_{c'}$" to H^* see Figure 36. Furthermore, since c' and x' must be duplicates of c and x, respectively, the algorithm for the third phase will construct an r' that is a duplicate of r. This means that $x' \rightarrow r'$ is congruent to $x \rightarrow r$.

2. The last edge of p is not a return-edge. Then there are (in G^*) an unbalanced-left path a and an edge $u \rightarrow v$ such that $p = [a\|u \rightarrow v]$. By the induction hypothesis, there is (in H^*) and unbalanced-left path from $Entry'_{global}$ to u' that is congruent to a. Here, the vertex u' is a duplicate of u. When the Interprocedural Hot-Path Tracing Algorithm processed the vertex u', it must have created (either at line 21 of Figure 34 or line 29 or line 32 of Figure 36) an $u' \rightarrow v'$ that is congruent to $u \rightarrow v$. The path $p' = [a\|u' \rightarrow v']$ is an unbalanced-left path in H^* that is congruent to p.

There is an (almost identical) inductive proof to show that for every unbalanced-left path p' in H^* that starts at $Entry'_{global}$, there is a congruent, unbalanced-left path p in G^*.

In order to show that for any unbalanced-left path in G^* there is an unbalanced-left path in H^* we need the following additional assumption: for any vertex v in G^*, there is an unbalanced-left path p from $Entry_{global}$ to v. (If this is not true, then v is a unreachable vertex that may be removed from G^* without changing the program's execution behavior.) If this assumption is true, then for any vertex v in

G^*, there must be at least one duplicate vertex v' in H^*; this follows from the fact that there is path p' in H^* that is congruent to the unbalanced-left path p from $Entry_{global}$ to v.

Now we can use an inductive argument to show that for any unbalanced-left path p in G^*, there is a congruent, unbalanced-left path p' in H^*.

Base case: p is path of length 0 from v to v. By the argument given above, there must be at least one vertex v' in H^* that is a copy of v. The path p' of length 0 from v' to v' is congruent to p and is unbalanced-left.

Induction step: This induction step is identical to the induction step above, so we will not repeat it.

Again, a similar inductive argument shows that for any unbalanced-left path p' in H^*, there is a congruent, unbalanced-left path p in G^*. *QED* □

Chapter 8

Experimental Results for the Express-lane Transformation

We have implemented a tool called the Interprocedural Path Weasel (IPW) that performs the interprocedural express-lane transformation[1]. IPW is implemented in SUIF 1.3.0.5. The program takes as input a set of C source files for a program P and a path profile for P. The path profile can be an an interprocedural context path profile, an interprocedural piecewise path profile, or an intraprocedural piecewise path profile. Depending on the type of profile, IPW performs the appropriate express-lane transformation on P (see Chapter 3, Chapter 4, and Section 2.2). IPW then performs interprocedural range analysis on the hot-path supergraph.

This section presents results of experiments with IPW. The experiments were run on an 833 MhZ Pentium III with 256M RAM running Solaris 2.7. We compiled with GCC 2.95.3 -O3. Each test was run 3 times, and the run times averaged. For each of the different express-lane transformations, we measured the amount of code growth caused by the transformation, and the benefit of the transformation for range analysis.

As the number of paths duplicated by an express-lane transformation increases, the amount of code growth increases, and the time required to perform range analysis increases. However, the benefit to range analysis may also increase. As in [5], we evaluate this tradeoff by introducing a parameter C_A, called the *dynamic instruction coverage percentage*, to each of the express-lane transformations. Given a value for C_A, an express-lane transformation duplicates the smallest set of paths that cover C_A-percent of the program's execution [2].

Figure 43 plots the amount of code growth that results as a function of C_A for each of the express-lane transformations. Figure 44 shows the increase in analysis time for range analysis as a function of C_A for each of the express-lane transformations. For each of the express-lane transformations, there is a sharp increase in code growth and analysis time as C_A is changed from 99% to 100%. However, in the following discussion, we will see that the results of range analysis do not benefit significantly when C_A is changed from 99% to 100%. This suggests that the express-lane transformation should not be used with $C_A > 99\%$. Figure 48 compares the amount of code growth and the increase in analysis time for the three different express-lane transformations at $C_A = 99\%$. The increase in analysis time is measured against the time to perform the analysis on the original supergraph. In all cases, the interprocedural, context express-lane transformation causes the most code growth. The interprocedural, piecewise express-lane transformation causes about as much code growth as the intraprocedural, express-lane transformation, except in the case of compress. On the other hand, the range analysis has much better results on compress after the interprocedural, express-lane transformations than it does after the intraprocedural, express-lane transformation (see below). Interestingly, performing range analysis is sometimes more costly after the intraprocedural express-lane transformation than it is after the

[1] The tool is named after, and based on, Glenn Ammon's tool Path Weasel, which performs the intraprocedural express-lane transformation [5]. We give many thanks for Glenn for his implementation of Path Weasel.

[2] In [5], C_A is called the *path coverage*.

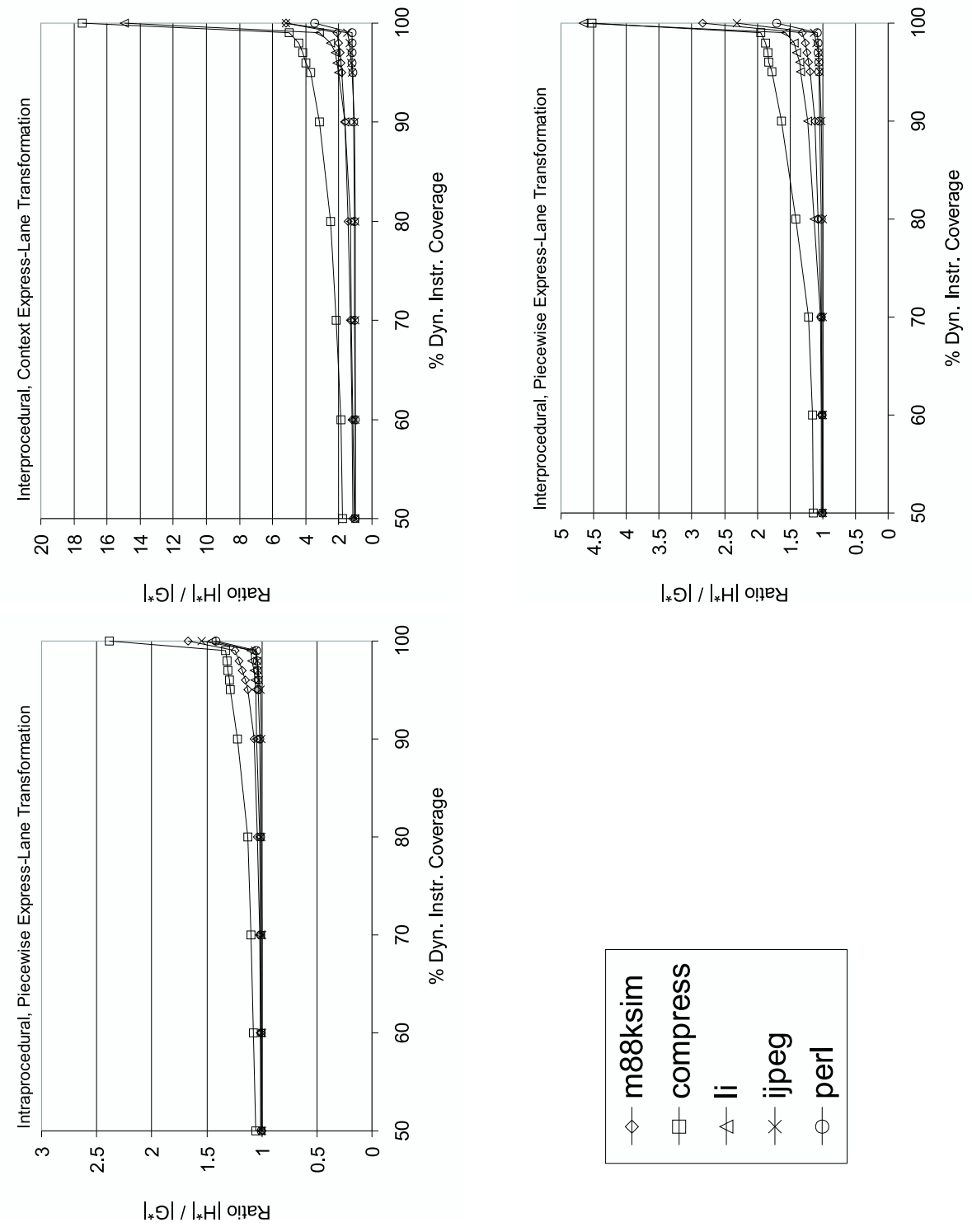


Figure 43: Code growth caused by the express-lane transformations.

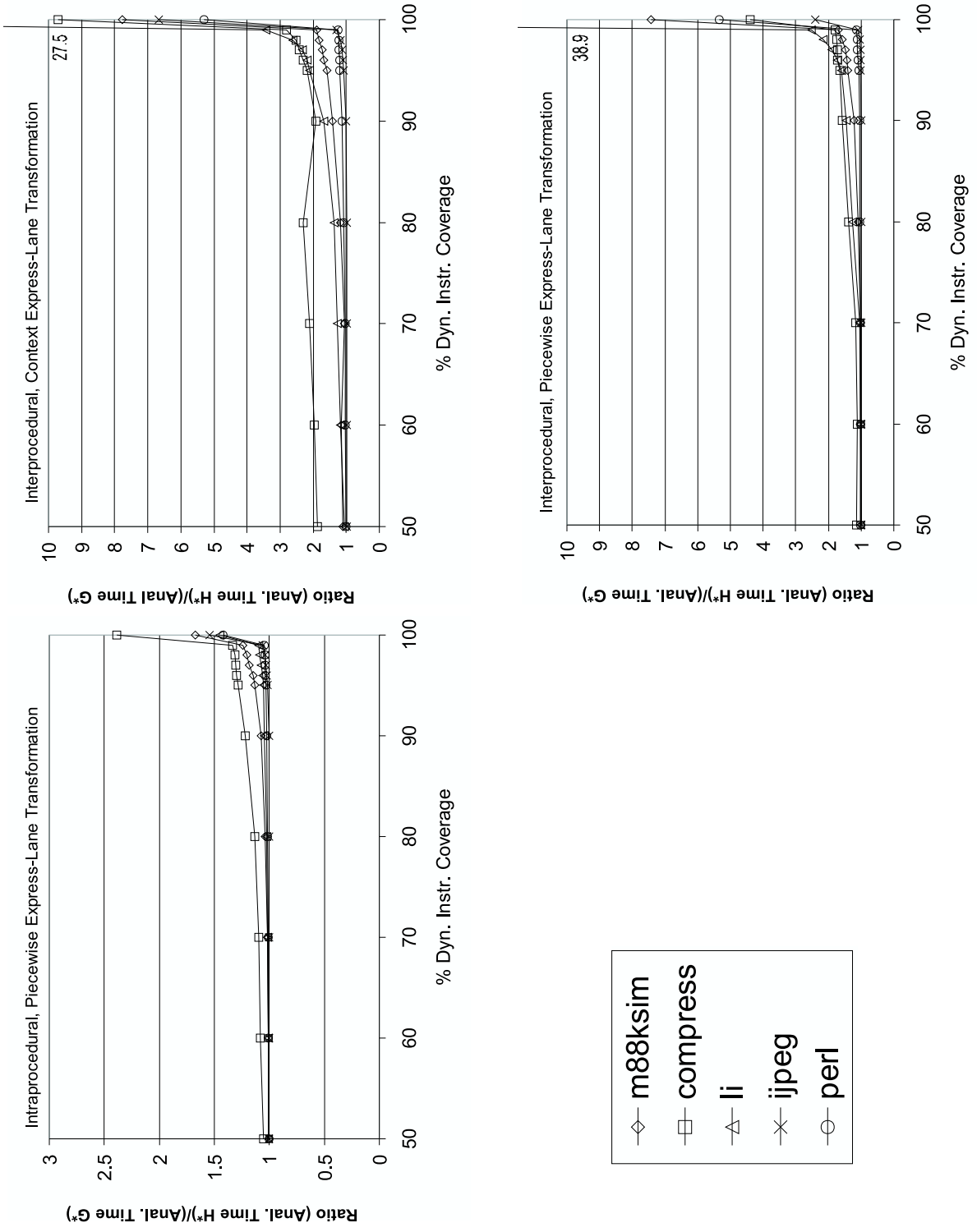


Figure 44: Increase in runtime of range analysis versus the percent code coverage.

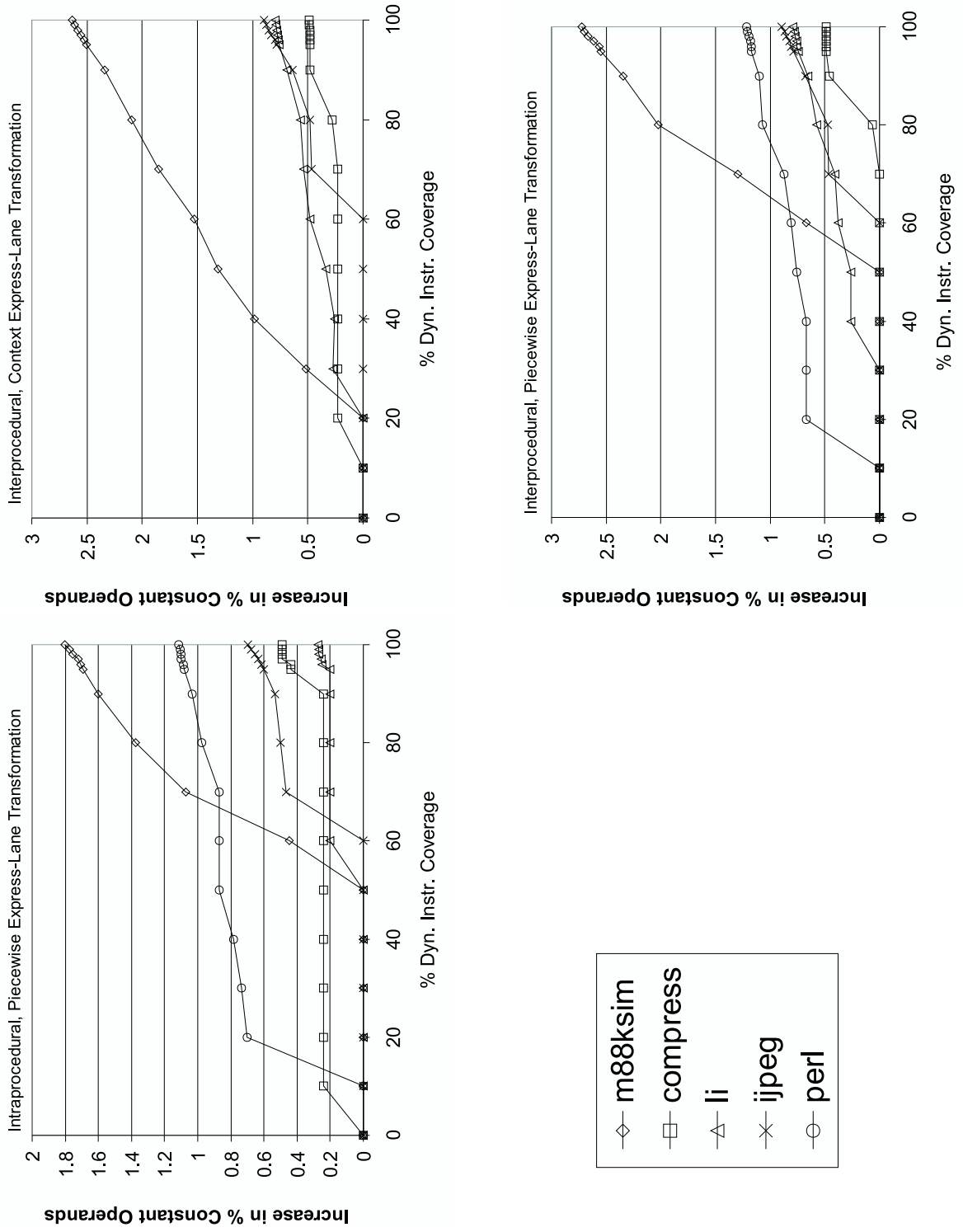


Figure 45: Increase in percentage of instruction operands that have a constant value versus the percent of code coverage.

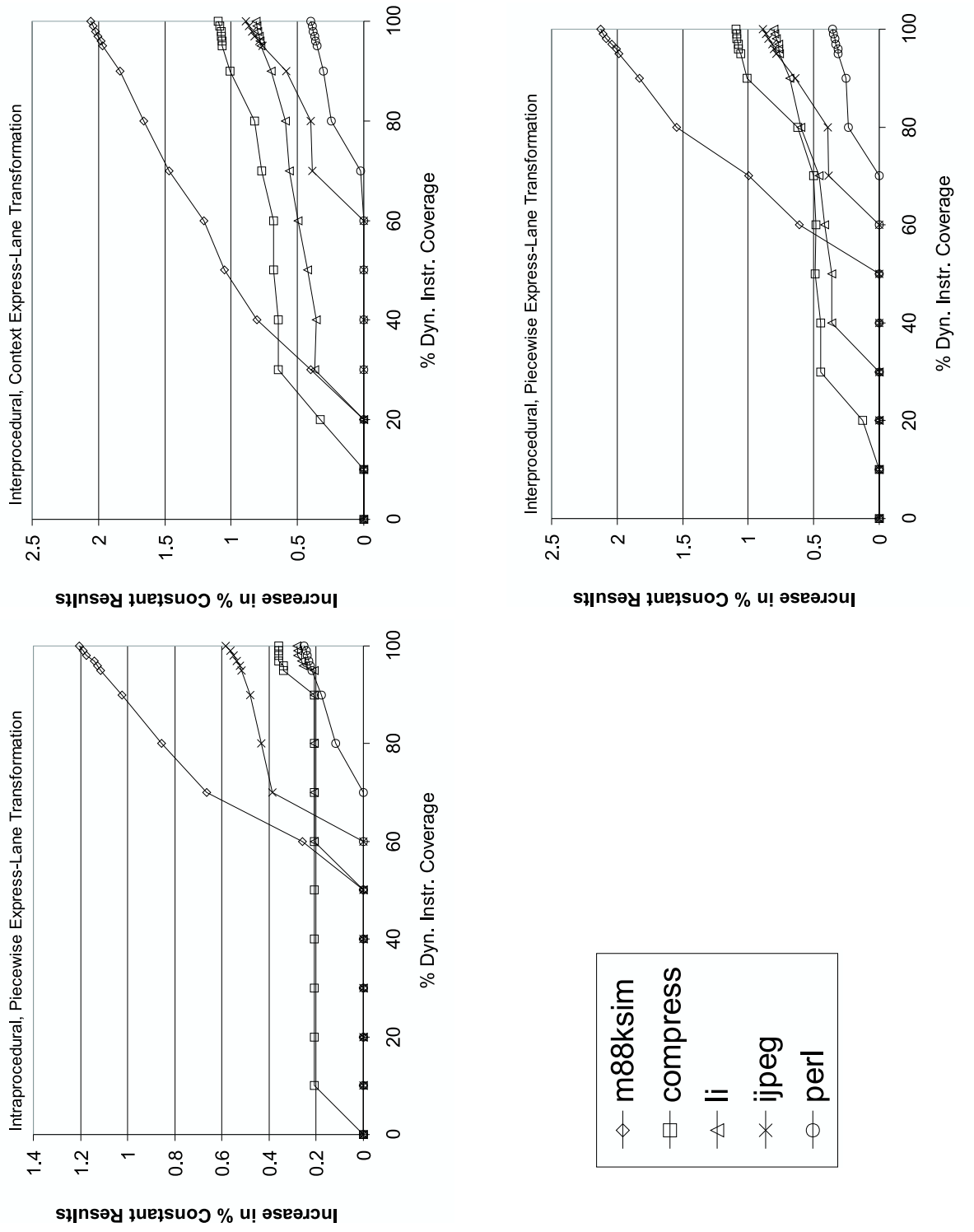


Figure 46: Increase in percentage of instructions that have a constant result versus the percent of code coverage.

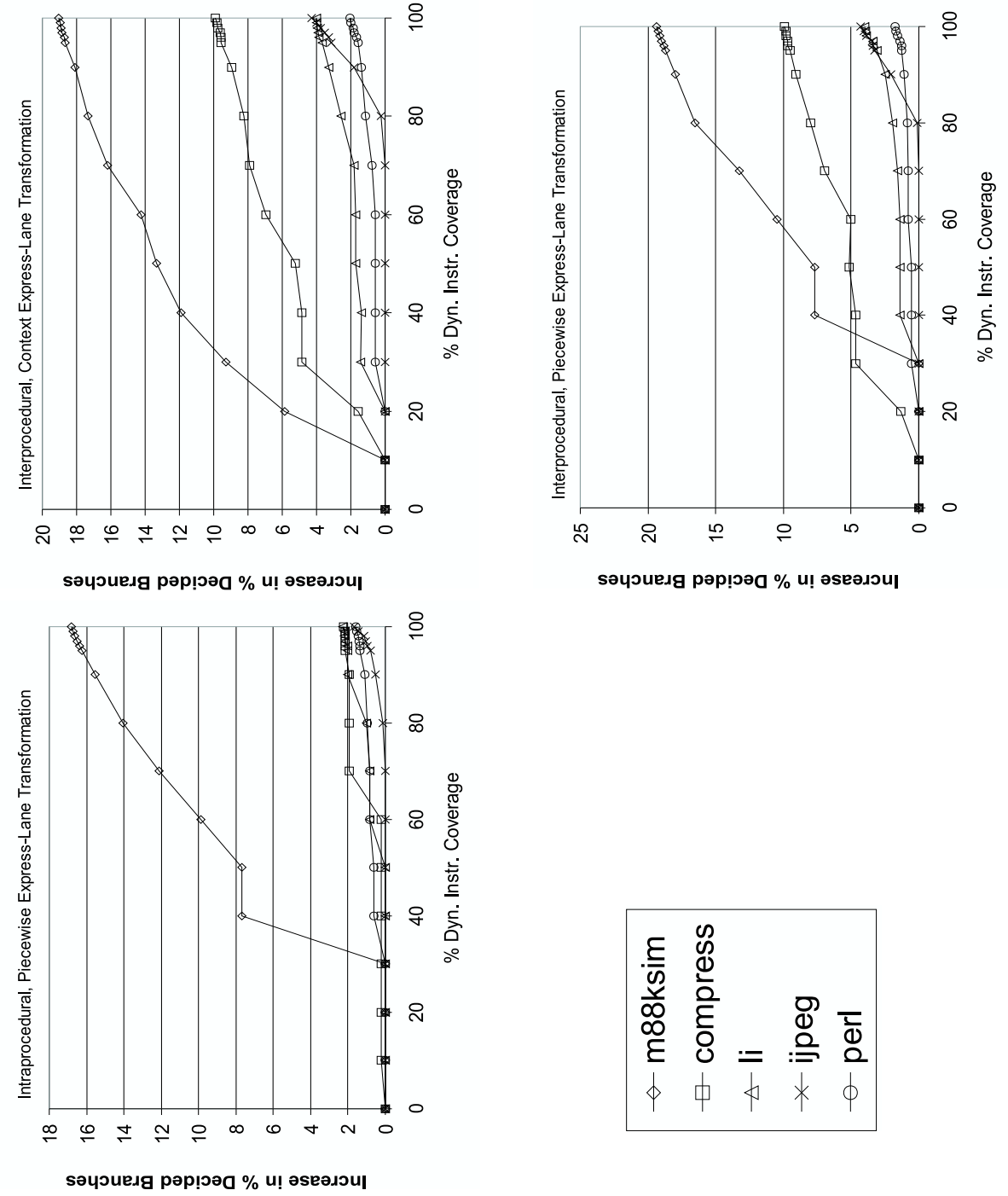


Figure 47: Increase in percentage of branch instructions that have a constant result versus the percent of code coverage.

Benchmark	E-Lane Trans.	Trans. Time (s)	Size H^*	DFA Time (s)
124.m88ksim	Inter., Context	9.8	24032	569.5
	Inter., Piecewise	4.9	15113	508.4
	Intra., Piecewise	3.0	14218	734.2
	None	-	11455	300.8
129.compress	Inter., Context	1.4	2610	14.7
	Inter., Piecewise	0.3	1014	9.4
	Intra., Piecewise	0.2	696	10.2
	None	-	522	5.2
130.li	Inter., Context	12.9	23125	99.1
	Inter., Piecewise	5.3	11319	73.2
	Intra., Piecewise	1.9	7940	35.7
	None	-	7240	29.0
132.ijpeg	Inter., Context	13.0	18087	628.8
	Inter., Piecewise	8.5	13768	526.1
	Intra., Piecewise	7.1	12955	504.3
	None	-	12192	488.2
134.perl	Inter., Context	10.3	33863	713.8
	Inter., Piecewise	9.0	30189	655.2
	Intra., Piecewise	6.7	29309	718.6
	None	-	27988	573.9

Table 8: Comparison of the cost of performing various express-lane transformations and the cost of performing interprocedural range analysis after an express-lane transformation has been performed. For these experiments, $C_A = 99\%$. Times are measured in seconds. Graph sizes are measured in number of vertices.

Benchmark	E-Lane Trans.	% const. operands	% const. results	% decided branches
124.m88ksim	Inter., Context	28.5	33.1	19.7
	Inter., Piecewise	28.6	33.2	20.0
	Intra., Piecewise	27.7	32.3	17.5
	None	25.9	31.1	0.8
129.compress	Inter., Context	21.3	26.9	9.8
	Inter., Piecewise	21.3	26.9	9.8
	Intra., Piecewise	21.3	26.2	2.2
	None	20.8	25.8	0.0
130.li	Inter., Context	24.1	27.3	4.0
	Inter., Piecewise	24.1	27.3	3.9
	Intra., Piecewise	23.6	26.8	2.2
	None	23.3	26.5	0.0
132.ijpeg	Inter., Context	16.8	23.6	4.0
	Inter., Piecewise	16.8	23.6	4.0
	Intra., Piecewise	16.6	23.3	1.4
	None	15.9	22.7	0.0
134.perl	Inter., Context	24.3	28.8	3.3
	Inter., Piecewise	24.2	28.8	3.0
	Intra., Piecewise	24.1	28.7	2.8
	None	23.0	28.5	1.3

Table 9: Comparison of the results of range analysis after various express-lane transformations have been performed.

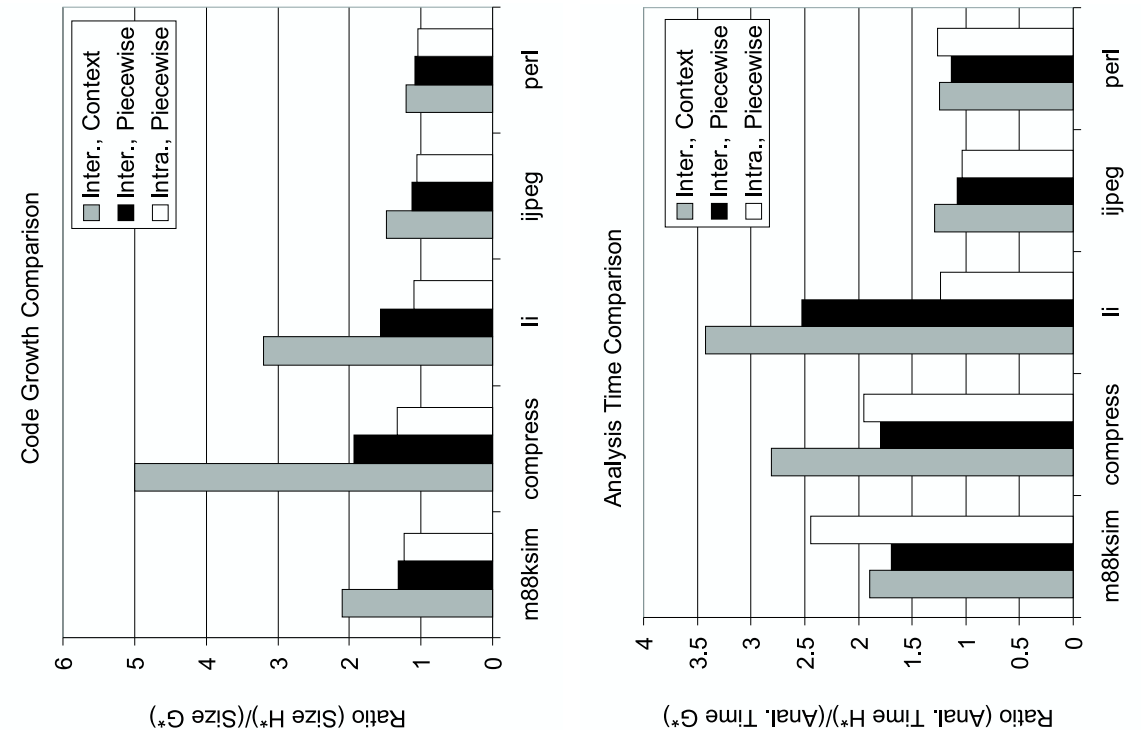


Figure 48: Graphs comparing the code growth and increase in analysis time for the three express-lane transformations for $C_A = 99\%$.

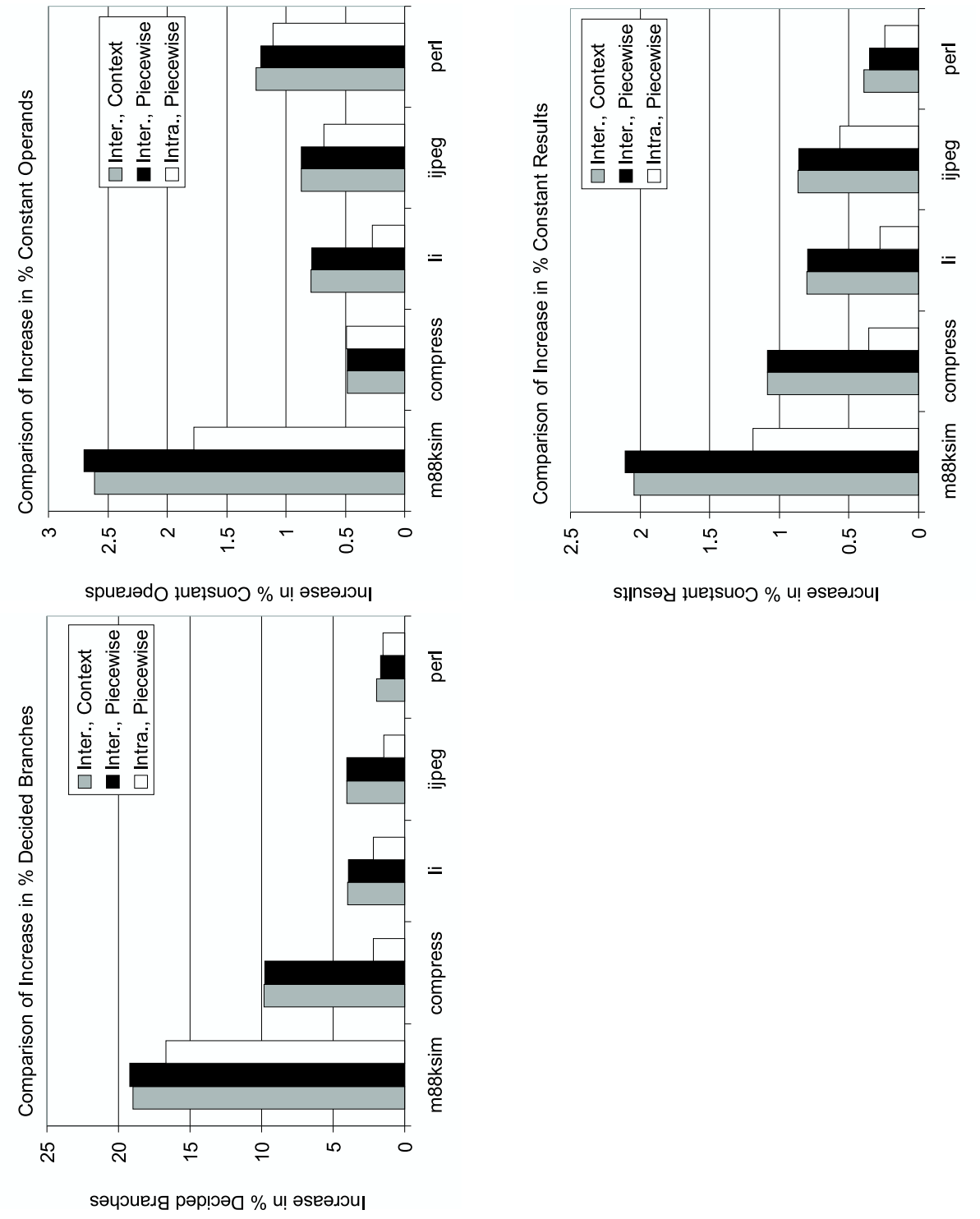


Figure 49: Graphs comparing the results of range analysis on the hot-path supergraphs created by the three express-lane transformations when $C_A = 99\%$.

interprocedural express-lane transformation.

To evaluate the results of range analysis on a program P, we weighted each data-flow fact by its execution frequency during a run of P. For example, let J be the solution for range analysis on a hot-path supergraph H^* that was constructed from compress and an interprocedural, context path profile of compress when run on the reference input. Then, for each vertex in v, we weight the data-flow facts in $J(v)$ by v's execution frequency when H^* (*i.e.*, the transformed version of compress) is run on compress's reference input. This measurement is useful when comparing range analysis on a supergraph G^* and range analysis on a hot-path supergraph H^* constructed from G^*: G^* and H^* have the same execution behavior, but statically they are very different. H^* has more vertices than G^*, and hence there are simply more data-flow facts, both good and bad, in H^* than in G^*. Hence, an unweighted comparison of the results of range analysis for G^* and H^* is not very informative. We used the same profiles to perform the express-lane transformation and to measure the effect of the transformation on range analysis. It would also be interesting if for each SPEC benchmark b we performed the interprocedural-express lane transformation based on a profile of b when run on b's *training* input, and then measured the benefit to data-flow analysis based on a profile of b when run on b's *reference* input.

Figure 45 shows the increase in the percent of instruction operands that have constant values as a function of the path coverage (C_A). Figure 46 shows the increase in the percent of instructions that have constant results as a function of the path coverage. (An instruction may have a constant result even if one of its operands is not constant: *e.g.*, the instruction "`a=b*0`" will always have the constant result zero.) Figure 47 plots the increase in the percent of conditional branch instructions that are decided to have only one possible target. In all of these graphs, we can see that most of the benefit of the express-lane transformations is realized with $C_A = 99\%$. Together with Figures 43 and 44, this suggests that 99% is the optimal value for C_A.

Figure 49 compares the results of range analysis after the express-lane transformations have been performed with $C_A = 99\%$. In all cases, the interprocedural express-lane transformations do better than the intraprocedural express-lane transformation. In the case of compress, after either of the interprocedural express-lane transformations, range analysis does much better at deciding conditional branches. This is because compress contains a function called *get_byte* that returns a character in the range $[0..255]$ when not at the end of the input buffer, and returns -1 (EOF) when the end of the input buffer is reached. There is a loop that repeatedly calls get_byte and uses a conditional branch to test if the return value is -1. After an interprocedural express-lane transformation has been performed, the path from get_byte where the return value is -1 is separated from the path where the return value is in the range $[0..255]$; this means that the condition that tests the return value of get_byte can be decided.

The range analysis we use allows the upper bound of a range to be increased once before it widens the upper bound to $(\mathrm{MaxVal}-1)$. Similarly, the lower bound of a range may be decreased once before it is widened to $(\mathrm{MinVal}+1)$. Our range analysis is similar to Wegman and Zadeck's conditional constant propagation [61] in that (1) it simultaneously performs dead code analysis and (2) it uses conditional branches to refine the data-flow facts. For example, as the analysis progresses, suppose that it determines that the vertex v is live and that `x` must be in the range $[0..10]$ in v. If there is a conditional branch on the condition "`x < 5`" at the end of v, then the analysis will determine that both the true branch and the false branch from v are executable. Furthermore, it will determine that on the true branch from v, `x` must be in the range $[0..4]$ and on the false branch, `x` must be in the range $[5..10]$. Throughout the analysis, dead code is not allowed to affect the results of the range analysis.

The results we present here for the intraprocedural, express-lane transformation are different from those in [5] for several reasons:

- The definition of a program vertex is different. IPW works on program supergraphs. For each call instruction i, IPW creates two supergraph vertices: a call vertex and a return-site vertex. These vertices represent the call instruction i and *only* the call instruction i. In contrast, in [5] they work on a collection of control-flow graphs (CFGs). Call instructions are not treated differently from other instructions: a single CFG vertex may represent a call instruction together with other instructions. The result of this difference is that our program representation has many more vertices and edges than the program representation used in [5]. Therefore, the average number of edges in our observable paths may be higher than in [5].

- When we do measurements on the intraprocedural, express-lane transformation, we perform the transformation on all of the program's procedures and then perform interprocedural range analysis. This allows us to compare the results of intraprocedural express-lane transformation with the interprocedural express-lane transformation. In [5], they perform the intraprocedural express-lane transformation on each procedure and then perform the intraprocedural version of Wegman and Zadeck's conditional constant propagation [61]. This means that they will find strictly fewer constant operands, instructions with constant values, and decided branches.

In Chapter 12, we present the results in which we use the results of range analysis to preform optimizations such as replacing a constant expression with a literal and replacing a decided branch with an unconditional jump.

Chapter 9

Reducing the Hot-path (Super)graph: Partitioning Algorithms

The express-lane transformation creates a hot-path graph in which hot paths are isolated from one another and from cold code. The hope is that a data-flow analysis may find better solutions along the duplicated paths. However, the analysis may find the same solution for some or all of the duplicated code. When the hot-path graph contains two copies of a vertex such that they share a common data-flow solution, it is desirable to collapse them into a single copy and avoid unnecessary code growth.

In this and the following two chapters, we discuss algorithms for reducing the hot-path (super)graph. Section 9.1 gives a formal definition of the problem. Section 9.2 discusses the Ammons-Larus Reduction Technique [5] for reducing the hot-path graph via the Coarsest Partitioning Algorithm. Section 9.3 describes how to adapt the key component of the Ammons-Larus Reduction Technique to work on the hot-path supergraph. In Chapter 10 we present a new reduction algorithm that is based on *redirecting* hot-path supergraph edges. Section 10.7 shows how to combine the Ammons-Larus Reduction Algorithm and the Edge-Redirection Algorithm. Chapter 11 shows that an optimal solution to the problem of reducing the hot-path graph is NP-hard. Chapter 12 presents experimental results for our algorithms on reducing the hot-path supergraph.

9.1 Definition of a Hot-path Graph Reduction Algorithm

In this section, we define what an algorithm for reducing a hot-path graph must accomplish. We will also give an analogous definition for algorithms that reduce hot-path supergraphs.

First, we give some preliminary definitions. Some of these definitions are taken from Section 7.4. Others are standard definitions from the literature.

Congruent edges: Edges $u \rightarrow v$ and $u' \rightarrow v'$ are said to be *congruent* if u and u' are "duplicate vertices" (*i.e.*, duplicates of the same supergraph vertex), v and v' are duplicate vertices, and $u \rightarrow v$ and $u' \rightarrow v'$ are the same kind of edge (*e.g.*, a true branch, a false branch, a "case c:"-branch, return-edge, etc.). The labels on call- and return-edges do not affect congruence.

Congruent paths: Paths p and p' are said to be *congruent* iff they are of equal length and, for all i, the i^{th} edge of p is congruent to the i^{th} edge of p'.

Path congruent graphs: Graphs G and G' are *path congruent* iff: for every path in G there is a congruent path in G'; and for every path in G' there is a congruent path in G. *Unbalanced-left path congruent* is defined in an analogous fashion.

Data-flow frameworks: Following the standard definitions, a data-flow framework $\mathcal{F}$ is a triple $(L, \sqcap, F)$ where L is a complete semilattice with meet operator $\sqcap$ and F is a set of functions from L to L that

is closed under composition. $\mathcal{F}$ is said to be monotonic if all the functions in F are monotonic. Likewise, $\mathcal{F}$ is said to be distributive if all the functions in F are distributive.

Data-flow problems: A data-flow problem is a tuple $(L, \sqcap, F, G, Entry, l, M)$ where $(L, \sqcap, F)$ is a data-flow framework, G is a control-flow graph, $Entry$ is the entry of G, $l \in L$ is the data-flow fact associated with $Entry$, and M is a map from the vertices of G to the functions in F. The data-flow problem $(L, \sqcap, F, G, Entry, l, M)$ is said to be an instance of the data-flow framework $(L, \sqcap, F)$.

Default data-flow instance for a graph: Given a data-flow framework $\mathcal{F} = (L, \sqcap, F)$ and a graph G, we call $\mathcal{F}_G$ the default instance of the $\mathcal{F}$ on G. We have $\mathcal{F}_G = (L, \sqcap, F, G, Entry, l, M)$ where $l = \bot$ and $M(v)$ is the appropriate transfer function as determined by the statements in v.

Valid data-flow solutions: A solution J for a data-flow problem on (super)graph G is *valid* if it approximates the meet-over-all (valid) paths solution at each vertex — *i.e.*, for all v in G, $J(v) \sqsubseteq MOP(v)$ (or $J(v) \sqsubseteq MOVP(v)$).

We are now ready to give the requirements of a hot-path graph reduction algorithm: an algorithm for reducing the hot-path graph takes three inputs: (1) a hot-path graph H; (2) a path profile pp of H; (3) and a valid solution J for a data-flow problem $\mathcal{F}_H$ where $\mathcal{F}$ is a monotonic data-flow framework. A hot-path graph reduction algorithm produces as output: (1) a reduced hot-path graph H' that is path congruent to (and preferably smaller than) H; (2) a path profile pp' translated from pp onto H'; (3) and a valid solution J' for the data-flow problem $\mathcal{F}_{H'}$. An algorithm for reducing a hot-path supergraph has similar requirements, except that the reduced hot-path supergraph must be unbalanced-left path congruent to the original supergraph.

This definition admits many trivial, uninteresting algorithms. For example, the algorithm that simply returns H, pp, and J without any modifications is a hot-path graph reduction algorithm. Other hot-path graph reduction algorithms may return any H' that is path congruent to H (*e.g.*, the original control-flow graph, or a supergraph that is larger than H) and the data-flow solution $J_\bot$ that maps everything to $\bot$.

The requirements for reducing the hot-path graph also leave room for some promising possibilities. For example, suppose J is the greatest fixed-point solution to the data-flow problem $\mathcal{F}_H$. It may be that J' is actually *better* than the greatest fixed-point solution for $\mathcal{F}_{H'}$. For example, suppose $\mathcal{F}$ is non-distributive. Then J' may contain data-flow facts that (due to the non-distributive nature of the data-flow problem) are not found in the greatest fixed-point solution.

Ideally, we want a hot-path graph reduction algorithm that minimizes the size of H' while preserving the desirable data-flow facts of J in J'. What a "desirable data-flow fact" means is dependent on the data-flow analysis in question. For classical constant propagation, a desirable data-flow fact may mean a use of a variable x that has been shown to have a constant value. A use of x in a vertex v where $J(v)(x) = \bot$ (an undesirable data-flow fact) would not have to be preserved (*i.e.*, H' need not contain a vertex v' where $J'(v')(x) = \bot$). Likewise, a data-flow fact that x is constant in a vertex v that does not use x would not have to be preserved. Furthermore, we do not care about preserving desirable data-flow facts in vertices that are cold.

Definition 9.1.1 *We say that J' preserves the valuable data-flow facts of J iff the following is true:*

> *Let p be a path in H that starts at H's entry and ends at a hot vertex v that contains a use of a "desirable" data-flow fact in $J(v)$. Let p' be the path in H' that is congruent to p. Let v' be the last vertex of p'. Then, $J'(v')$ is at least as good as $J(v)$ ($J(v) \sqsubseteq J'(v')$) for any desirable data-flow fact used in v.*

In Chapter 11, we show that finding H' and J' such that H' is minimal and J' preserves the valuable data-flow facts of J is an NP-hard problem. The algorithms we consider for reducing the hot-path (super)graph will preserve valuable data-flow facts (finding these facts was the whole purpose of creating the hot-path graph), but they will not result in a minimal graph.

9.1.1 A Paradigm Shift?

A reasonable objection to the above definition of "preserving the valuable data-flow facts" is that it seems to be a paradigm shift from the strategy of the express-lane transformation: the express-lane transformation focuses on duplicating hot *paths*, while our hot-path reduction algorithms focus on preserving data-flow facts at hot *vertices*. We feel that this shift is justified because a solution to a data-flow problem contains data-flow facts for each individual vertex, not each individual path. The meet-over-all paths data-flow solution $\mathrm{MOP}(v)$ for the vertex v is a single set of data-flow facts for v, namely, the meet of the data-flow solutions for all of the paths to v.

In contrast to our stated goal of preserving the desirable data-flow facts for hot vertices, the following Path-Preservation Algorithm reduces the hot-path graph by preserving the most valuable of the duplicated paths and discarding the others:

1. For each express-lane p in the hot-path graph H, compute the weighted benefit, b_p, of the express-lane:

   $$b_p = (\text{number of uses of desirable data-flow facts along } p) \cdot (\text{the execution frequency of } p)$$

2. Find the smallest subset $\mathcal{P}$ of the express-lanes such that

   $$\left(\sum_{p \in \mathcal{P}} b_p\right) \geq \begin{pmatrix}\text{95\% of the uses of desirable data-flow}\\ \text{facts in } H\text{, weighted by execution}\\ \text{frequency}\end{pmatrix}$$

3. For any express-lane $q \notin \mathcal{P}$, remove q from H. This can be done by coalescing the vertices of q with vertices that are in another express-lane $q' \notin \mathcal{P}$ or with vertices that are not on an express-lane. The data-flow facts on a vertex w that results from coalescing the vertices u and v are set to the meet of the data-flow facts on u and v: $J'(w) = J(u) \sqcap J(v)$. (See [5], Section 2.2.2, and Section 9.2.3 for technical details on coalescing vertices.)

4. Output the reduced graph H' with the new data-flow solution J'.

The above strategy has the advantage of being simple — all of the required machinery has been introduced in previous sections. However, it is likely to overlook many opportunities for reducing the hot-path graph. For an express-lane p, it may be that the valuable data-flow facts are concentrated in one part (*e.g.*, a prefix) of p and that it is not necessary to preserve the entire express-lane in order to preserve the valuable data-flow facts in p. Furthermore, two express-lanes p and q may have pieces that are congruent and that have the same data-flow facts; in this case, the congruent pieces may be merged. For example, both p and q may contain a copy of the path a and they may both have the same data-flow facts along their individual copy of a. In this case, it may be possible for p and q to share a single copy of a. The path-preservation algorithm does not recognize this.

It is unclear that there is any advantage in an algorithm that concentrates on preserving "valuable paths" rather than preserving valuable data-flow facts at vertices. Even if the problems mentioned above were addressed, the Path-Preservation Algorithm would still suffer from all the shortcomings of the Ball-Larus Reduction Algorithm that are described in the following section.

9.2 The Ammons/Larus Approach to Reducing the Hot-path Graph

Section 2.2.2 summarizes the approach used by Ammons and Larus to reduce the hot-path graph [5]. This section gives further details of their algorithm, and shows some of the limitations of their approach. Sections 9.2.1 thru 9.2.3 discuss the first three steps of the Ammons/Larus hot-path graph reduction algorithm. The fourth and final step is straightforward and does not require further discussion. In the remainder of this chapter, we refer to the Ammons-Larus approach to reducing the hot-path graph as the *Ammons-Larus Reduction Algorithm*.

9.2.1 Step One: Identify Hot Vertices

The first step of the Ammons-Larus approach identifies hot vertices in the hot-path graph. The execution frequency of each vertex is easily determined from the path profile (translated from the control-flow graph to the hot-path graph). The vertices are sorted based on the number of uses of desirable data-flow facts (*e.g.*, constants) they contain, weighted by execution frequency. Vertices are marked as hot until a fixed percentage — 95% in their experiments — of the desirable data-flow facts (weighted by execution frequency) have been included in hot vertices.

We desire a graph reduction algorithm that preserves hot data-flow facts. By definition, this means preserving desirable data-flow facts that are found in hot vertices. However, labeling vertices hot according to their execution frequency in the hot-path graph does not necessarily preserve the hottest data-flow facts with respect to the original control flow graph:

Example 9.2.1 Let v be a vertex in the control flow graph C that is executed 110 times. Suppose that there is a use of the program variable x in v. Let the hot-path graph H contain four copies of v: v_1, v_2, v_3 and v_4. Let J be a greatest fixed-point solution to constant propagation on H. The vertex v_1 has an execution frequency of 40 and $J(v_1)(x) = 2$. The vertices v_2 and v_3 each have an execution frequency of 30 and $J(v_2)(x) = J(v_3)(x) = 3$. The vertex v_4 has an execution frequency of 10 and $J(v_4)(x) = \perp$. From these facts, we conclude that during the execution of v, x has the value '2' 40 times, x has the value '3' 60 times, and x has an unknown value 10 times. From v's point of view, the most important data-flow value for x is 3. By looking only at the separate execution frequencies of $v_1, v_2, v_3,$ and v_4, it may seem that '2' is the most important data-flow value for x.

Suppose that the Ball-Larus technique marks the vertex v_1 as hot, since it has the highest execution frequency of v_1, v_2, v_3 and v_4. Then the Ball-Larus Reduction Algorithm will strive to create a reduced hot-path graph in which there is a unique copy v' of v such that $J(v')(x) = 2$ and v' has an execution frequency of 110. Since this is not possible, the algorithm instead tries to create a reduced hot-path graph H' with two copies v' and v'' of v such that $J(v')(x) = 2$, v' has an execution frequency of 40, $J(v'')(x) = \perp$ and v'' has an execution frequency of 70. However, the Ball-Larus Algorithm would do better to mark v_2 and v_3 as hot and strive for a reduced hot-path graph with two copies v' and v'' of v such that $J'(v') = 3$, v' has an execution frequency of 60, $J'(v'') = \perp$, and v'' has an execution frequency of 50. In this second scenario, the copy of v with a value of $\perp$ has a lower execution frequency. □

Thus, preserving the data-flow facts in vertices with high execution frequency might not preserve the most frequent data-flow facts. For Ammons and Larus (and for us), this is unlikely to be a problem: they mark enough vertices as hot to cover 95% of the desirable data-flow facts (and we mark enough vertices as hot to cover 99% of the desirable data-flow facts). If a lower percentage of data-flow facts are used to identify the hot vertices, then the fact that the reduction algorithm does not preserve the most frequent data-flow facts could be increasingly important.

9.2.2 Step Two: Partition Vertices into Compatible Blocks

The second step of Ammons-Larus Reduction Algorithm partitions the vertices of the hot-path graph into sets of compatible vertices. (Recall that a hot-path graph vertex is a pair: the hot-path graph vertex $[v, q]$ is a copy of the CFG vertex v and corresponds to a path that drives the hot-path automaton to state q.) The vertices $[v, q]$ and $[v', q']$ are compatible iff:

1. $v = v'$ (*i.e.*, $[v, q]$ and $[v', q']$ are duplicate vertices) and
2. the meet of $J([v, q])$ and $J([v', q'])$ does not destroy any desirable data-flow facts in a hot-vertex:
   (a) if $[v, q]$ is hot, then $J([v, q]) \sqsubseteq J([v', q'])$ for all desirable data-flow facts used in $[v, q]$;
   (b) if $[v', q']$ is hot, then $J([v', q']) \sqsubseteq J([v, q])$ for all desirable data-flow facts used in $[v', q']$.

The blocks of the partition are created greedily: as each vertex $[v, q]$ is considered, it is added to the first block with which it is compatible. (A vertex is compatible with a block of vertices iff it is compatible with all of the vertices in the block.) We call the partition that results from this process the *Ammons-Larus compatibility partition*, or the *compatibility partition* for short.

The advantage of this partitioning algorithm is that it is fast: it can be implemented in $O(d \cdot n \log n)$ steps, where d is the maximum number of data-flow facts used in a vertex and n is the number of vertices in the hot-path graph. However, it does not result in a partition with a minimal number of blocks. Finding a minimal partition is likely NP-hard: the fact that compatibility does not define an equivalence relationship makes the problem similar to finding a minimal graph coloring (see the proof of Theorem 11.0.1 below).

More importantly, some partitions interact with the Coarsest Partitioning Algorithm better than others. This is discussed in the next section.

9.2.3 Step Three: Apply the Coarsest Partitioning Algorithm

Step three of the Ammons-Larus reduction algorithm applies a coarsest partitioning algorithm to the partition created in step two. An $n \log n$ algorithm for the coarsest partitioning problem was given by Hopcroft in [38]. Our description is taken from [1], which is is turn based on [38]. Given a set S, an initial partition $\pi = \{B_1, B_2, \ldots B_n\}$ of S, and a function f on S, the coarsest partition algorithm produces a new partition $\pi' = \{C_1, C_2, \ldots C_k\}$ such that the following holds:

1. for every C_i, there is a B_j such that $C_i \subseteq B_j$;
2. for every $a, b \in C_i$, there is a C_j such that $f(a)$ and $f(b)$ are in C_j; and
3. there is no partition with fewer blocks that satisfies the previous conditions.

The new partition π' is said to be the coarsest partition that respects π and f. The coarsest partition algorithm is easily generalized to generate a partition that respects a set of functions. This is done when the coarsest partitioning algorithm is used for DFA minimization: there is one function f_a for each alphabet symbol a; f_a encodes the a-transitions of the DFA [38]. Figure 50 shows the coarsest partitioning algorithm presented in [1]. Figure 51 describes how the algorithm works. The running time for the generalized version of this algorithm is $m \cdot n \log n$ where m is the number of input functions and n is the number of elements in the input set S.

The Ammons-Larus technique uses the coarsest partitioning algorithm with a set of functions $\mathcal{F}$ containing one function for each edge in the control-flow graph. The function $f_{u \to v} \in \mathcal{F}$ for edge

$B_1, B_2, \ldots B_n$ are the blocks of the initial partition π
W is a worklist of blocks
q is the index of the last created partition block
Inverse is a set of hot-path graph vertices

Main()
    $W := \{B_1, B_2, \ldots B_n\}$
    $q := n$
    While $W \neq \emptyset$
        $B_i := Take(W)$ /* *select and remove an elt. from W* */
        Inverse := $f^{-1}(B_i)$
        Foreach B_j such that $B_j \cap$ Inverse $\neq \emptyset$ **and** $B_j \not\subseteq$ Inverse
            $q := q + 1$
            Create a new block B_q
            $B_q := B_j \cap$ Inverse
            $B_j := B_j - B_q$
            If $B_j \in W$ **Then**
                $Put(W, B_q)$
            Else If $\|B_j\| \leq \|B_q\|$ **Then**
                $Put(W, B_j)$
            Else
                $Put(W, B_q)$
    Output the partition $\pi' \equiv \{B_1, B_2, \ldots B_q\}$
End Main

Figure 50: The Coarsest Partitioning Algorithm [38, 1].

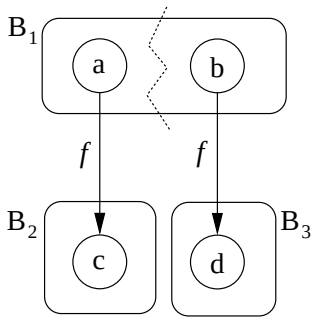


Figure 51: Example of the coarsest partition algorithm. The block B_1 must be split because f maps its members (a and b) into different blocks (B_2 and B_3, respectively). The algorithm in Figure 50 may discover this when it looks at block B_2: $f^{-1}(B_2)$ contains an element of B_1, but not both, implying that B_1 must be split.

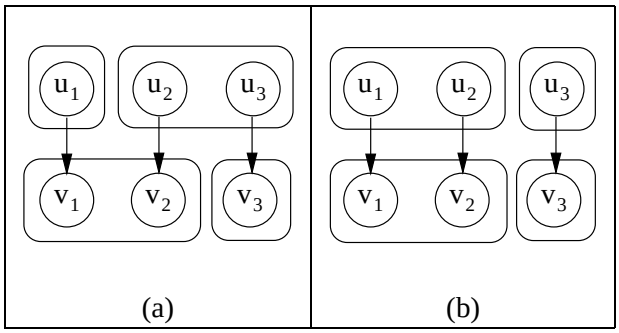


Figure 52: (a) shows a partition π_1 that the Coarsest Partitioning Algorithm splits into five blocks. (b) shows a partition π_2 of the same set that the Coarsest Partitioning Algorithm leaves as four blocks.

$u \rightarrow v$ maps any vertex $[u, q]$ (that is a copy of u) to the unique vertex $[v, q']$ (that is a copy of v) if there is an edge $[u, q] \rightarrow [v, q']$ in the hot-path graph that is a copy of the edge $u \rightarrow v$. These functions encode the control relationships of the edges of the hot-path graph: the edges representing true branches are in separate functions from the edges representing false branches; the edges representing "case i" branches are in separate functions from the edges representing "case j" branches, etc. (In fact, it is sufficient to define $\mathcal{F}$ to contain one function for encoding true branches, one for false branches, one for each possible case of a switch, and one for all other edges[1].)

Let π' be the coarsest partition that respects the compatibility partition π and the functions $f u \rightarrow v$ described above. π' can be used to reduce the hot-path graph while preserving hot data-flow facts: all the vertices in a block B_i of π' are collapsed to a single representative vertex (this is Step 4 of the Ammons-Larus reduction algorithm). The data-flow facts for a representative vertex are set to the meet of the data-flow facts on each member of the representative's block. The fact that this reduction preserves the hot data-flow facts in the collapsed vertices follows from the fact that π' respects the compatibility partition π and that the output graph is path congruent to the input graph. (See Theorem 9.3.1 for a proof.)

Over-respecting the initial partition

However, the Coarsest Partitioning Algorithm gives too much respect to π and $\mathcal{F}$ and many opportunities to reduce the hot-path graph are lost. As mentioned in the last section, some partitions of the hot-path graph's vertices interact with the coarsest partitioning algorithm in a poor fashion. Figure 52 shows two possible partitions, π_1 and π_2, of vertices in a hot-path graph. The coarsest partition that respects π_1 and the edge relationships has five blocks, while the coarsest partition that respects π_2 and the edge relationships has only four blocks. The coarsest partitioning algorithm may not find a minimal reduction of the hot-path graph if it is given the wrong initial partition.

We employ a simple heuristic to help choose a compatibility partition that interacts in a better way with the Coarsest Partitioning Algorithm. First, the duplicate vertices of v are partitioned before the duplicate vertices of u if u comes before v in a depth-first traversal of the vertices of the original control-flow graph. In Figure 52, the partition of v_1, v_2 and v_3 is chosen before the partition of u_1, u_2 and u_3.

[1]If the control-flow graph does not allow multiple edges between vertices (*e.g.*, a true branch from A to B and a false branch from A to B), then the Ammons-Larus technique can use the coarsest partitioning algorithm with one function – the edge set of the hot-path graph: true edges are never confused with false branches because they always connect incompatible vertices, and hence connect different blocks of the initial partition.

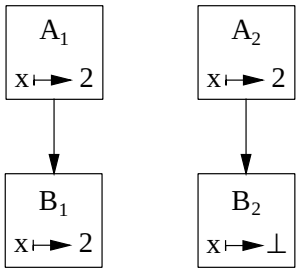


Figure 53: Example showing how edge redirection may help reduce the hot-path graph. Vertices A_1 and A_2 are compatible (they have identical data-flow facts), however vertices B_1 and B_2 are not, preventing A_1 and A_2 from being collapsed. If we redirect the edge from A_2 to target B_1 instead of B_2, then A_1 and A_2 may be collapsed.

Second, we add a requirement that if v_1 and v_2 are in separate blocks, and there are congruent-edges $u_1 \to v_1$ and $u_2 \to v_2$, then u_1 and u_2 should be put in separate blocks. If v_1 or v_2 has not yet been assigned to a block, then there is no additional restriction on the placement of u_1 and u_2. In Figure 52, the heuristic encourages us to pick the partition in Figure 52(b) over the partition in Figure 52(a). In general, forming the compatibility partition with this heuristic and then running the Coarsest Partitioning Algorithm seems to result in output partitions that are 10% smaller than when the heuristic is not used.

Over-respecting the edge set of the hot-path graph

Further opportunities for reducing the hot-path graph may be found if we change the edge set of the hot-path graph — in other words if we change the functions in the set $\mathcal{F}$. Figure 53 gives an example. The vertices A_1 and A_2 have identical data-flow facts. However, they cannot be collapsed because B_1 and B_2 cannot be collapsed — they have different data-flow facts. If the edge $A_2 \to B_2$ is redirected to B_1 — *i.e.*, the edge $A_2 \to B_2$ is replaced by the edge $A_2 \to B_1$ — then A_1 and A_2 can be coalesced. If other edges (not shown in Figure 53) that target B_2 are redirected, then B_2 may become unreachable and then may be dropped from the graph. The following example demonstrates the power of edge redirection:

Example 9.2.2 Consider the graph in Figure 54. If this graph has the hot-paths $[A \to B \to D \to E \to G]$ and $[A \to C \to D \to F \to G]$, then the intraprocedural express-lane transformation results in the hot-path graph H shown Figure 55(a). After constant propagation is performed on H, the Ammons-Larus technique for reducing the hot-path graph will group the compatible nodes of H as shown in Figure 55(b) (see Section 2.2 and [5]). The vertices D and D', E and E', and F and F' are pairs of compatible nodes that we would like to collapse. G, G', and G'' may not be collapsed because they have different data-flow facts for x. The fact that G and G'' cannot be collapsed causes the Coarsest Partitioning Algorithm to decide that E and E' cannot be collapsed, and hence that D and D' must be separated. The fact that G' and G'' cannot be combined causes the DFA-minimization to separate F and F'. Thus, the coarsest partitioning algorithm does not find any opportunities to reduce the hot-path graph in Figure 55(a).

However, the hot-path graph in Figure 55 can be reduced: the edge $E' \to G''$ could be replaced with $E' \to G$; the edge $F' \to G''$ could be replaced with $F' \to G'$; and the vertex G'' could be removed from the hot-path graph. After these transformations, each pair of vertices D and D', E and E', and F and

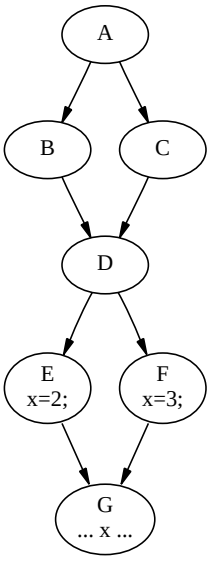


Figure 54: Sample control-flow graph. The hot-paths in this graph are $[A \to B \to D \to E \to G]$ and $[A \to C \to D \to F \to G]$.

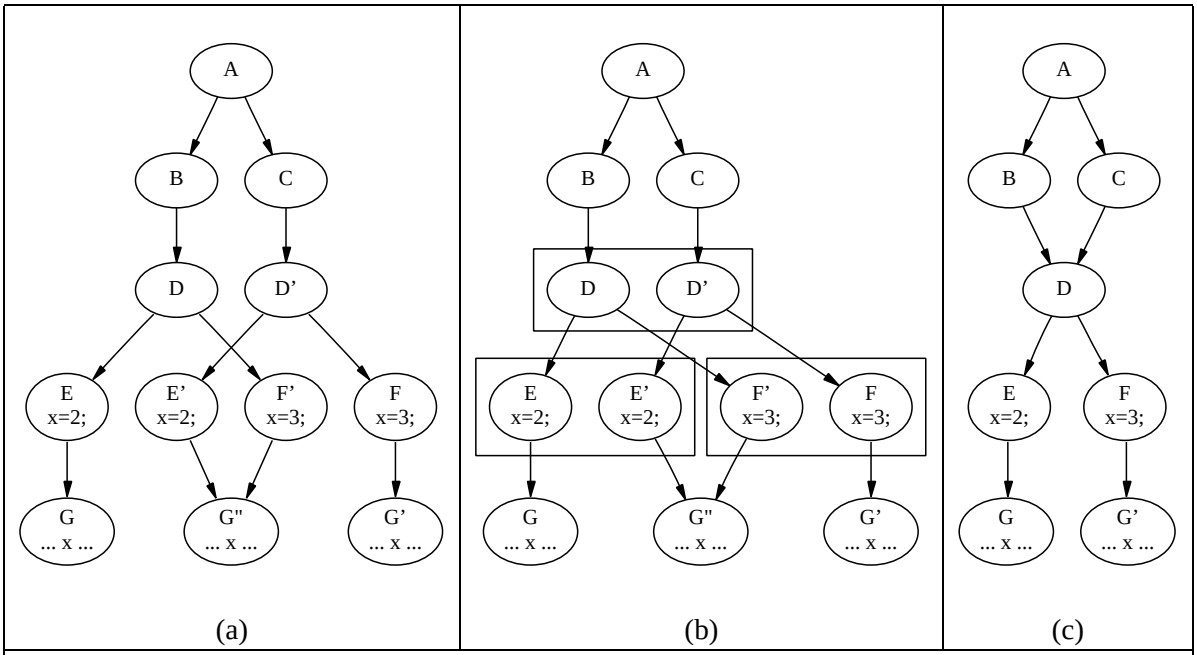


Figure 55: (a) shows the hot-path graph for Figure 54. All nodes have the same data flow fact for x (namely bottom), except G (where x is 2) and G' (where x is 3). (b) shows blocks of compatible vertices inside of boxes. (c) shows the minimal reduced hot-path graph.

F' can be combined. These steps result in the minimal reduced hot-path graph shown in Figure 55(c). □

Chapter 10 gives a new algorithm for reducing the hot-path graph that is based on the use of edge redirection.

Problems applying the Coarsest Partitioning Algorithm to the hot-path supergraph

In addition to the problems that arise when using the Coarsest Partitioning Algorithm to reduce the hot-path graph, there are difficulties in adapting the Coarsest Partitioning Algorithm to work on a supergraph. Consider the following example:

Example 9.2.3 Figure 56(a) depicts a program supergraph with two procedures, P and Q. The vertices Q are shown in an enclosed box; The vertex layout suggests that Q has been inlined in P, but that is not the case. This program is similar to the one discussed in Example 9.2.2, except that the second if statement has been put in a called procedure, and the definitions of x have been moved to the branches of the first if statement. The variable x is local to the procedure P, and not visible to Q.

Given the hot paths $[a \to b \to d \to A \to B \to D \to e \to f]$ and $[a \to c \to d \to A \to C \to D \to e \to f]$, the interprocedural express-lane transformation constructs the hot-path supergraph shown in Figure 56(b). Every duplicated vertex in Q is compatible: their data-flow facts agree on all visible variables (which is the empty set in this example). Note that if Q were inlined, none of the inlined vertices would be compatible: they would all have a different value for x.

If the Coarsest Partition Algorithm is run on this supergraph (and treated the supergraph as a normal graph), it would not collapse any of the vertices in Q. Since the vertices e and e'' cannot be collapsed, the vertices D, D'', and D' would be separated. Since the vertices e'' and e' are incompatible, the vertices D'' and D' would be separated. The separation of the set of compatible vertices $\{D, D'', D'\}$ into the singleton sets $\{D\}$, $\{D''\}$ and $\{D'\}$ causes the rest of the compatible sets in Q to be broken up. Thus, the hot-path supergraph is not reduced at all.

Figure 56(c) shows the minimum reduced hot-path supergraph for the supergraph in Figure 56(b). All of the compatible vertices in Q have been collapsed, and the vertices e'' and f'' have been removed. (It may be interesting to note that this is the graph that results from performing the intraprocedural express-lane transformation on P and Q, given the hot paths $[a \to b \to d \to e \to f]$, $[a \to c \to d \to e \to f]$, $[A \to B \to D]$, and $[A \to C \to D]$.)□

In summary, the coarsest partitioning algorithm must respect the input partition and input functions: this means that it is stuck with unwise decisions in the initial partition and that it cannot make use of the powerful edge redirection technique. Consequently, the Ammons-Larus reduction algorithm may do poorly on very simple examples like the one given in Example 9.2.2. Furthermore, there are some difficulties in adapting the Coarsest Partitioning Algorithm to work on a hot-path supergraph, although these are addressed in the next section.

9.3 Adapting the Coarsest Partitioning Algorithm for the Hot-Path Supergraph

In this section, we present a modified version of the Coarsest Partitioning Algorithm which we call the Supergraph Partitioning Algorithm. In Section 9.3.1, we state some properties of the new algorithm.

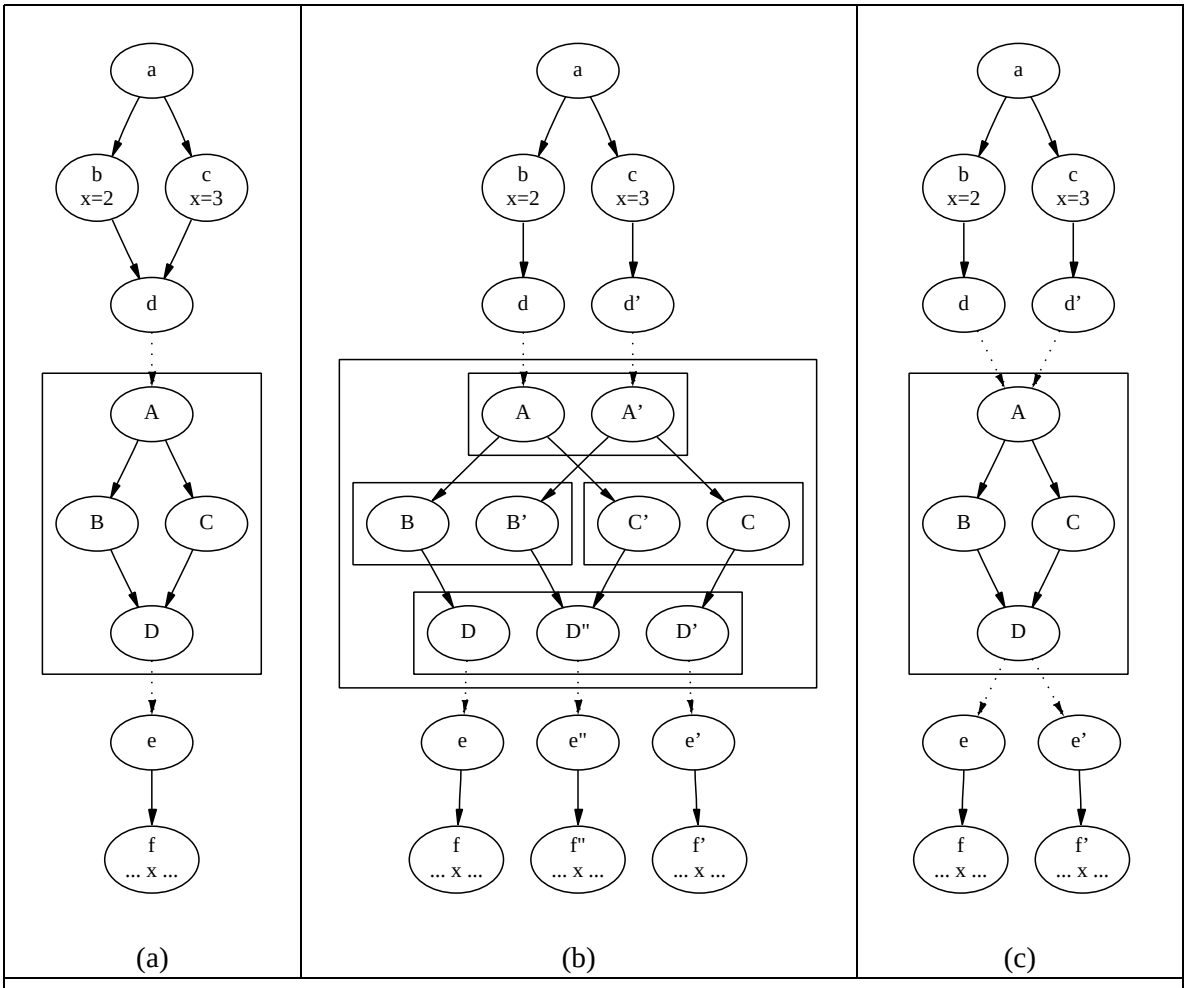


Figure 56: (a) Sample supergraph. The hot paths in this supergraph are $a \to b \to d \to A \to B \to D \to e \to f$ and $a \to c \to d \to A \to C \to D \to e \to f$. (b) Hot-path supergraph for the supergraph in (a). The large box indicates the procedure Q. The smaller boxes group compatible nodes. (c) Minimal reduced hot-path supergraph for the hot-path supergraph in (b).

Section 9.3.2 describes how the Supergraph Partitioning Algorithm can be used in the Ammons-Larus Reduction Algorithm (in place of the Coarsest Partitioning Algorithm) to obtain an algorithm for reducing the hot-path supergraph. A proof is given that the modified Ammons-Larus Reduction Algorithm is a valid hot-path supergraph reduction algorithm. In Section 9.3.3, we compare and contrast the properties of the Coarsest Partitioning Algorithm and the Supergraph Partitioning Algorithm. Finally, in Section 9.3.4, we present the Supergraph Partitioning Algorithm, prove its correctness and analyze its running time.

9.3.1 Properties of the Supergraph Partitioning Algorithm

The Supergraph Partitioning Algorithm takes as input a hot-path supergraph H^* and a partition $\pi = \{B_1, B_2, \ldots B_n\}$ of the vertices of H^*. The new algorithm produces as output a partition $\pi' = \{C_1, C_2, \ldots C_{n'}\}$ such that the following properties hold:

1. For every C_i, there is a B_j such that $C_i \subseteq B_j$.
2. For every $u_1, u_2 \in C_i$ where u_1 and u_2 are not exit vertices, for every pair of congruent edges $u_1 \rightarrow v_1$ and $u_2 \rightarrow v_2$ that are not return-edges and not summary-edges, there is a C_j such that $v_1, v_2 \in C_j$.
3. For every pair of exit vertices $x_1, x_2 \in C_i$, for every pair of call vertices $c_1, c_2 \in C_j$, for every pair of congruent return-edges $x_1 \rightarrow r_1$ and $x_2 \rightarrow r_2$ where $x_1 \rightarrow r_1$ is labeled "$)_{c_1}$" and $x_2 \rightarrow r_2$ is labeled "$)_{c_2}$", there is a C_k such that $r_1, r_2 \in C_k$.

The first property states that π' respects the partition π. The other properties guarantee that π' respects the control-flow of H^*. Specifically, the second and third properties imply that for any pair of congruent, unbalanced-left paths p_1 and p_2 in H^*, if p_1 and p_2 start in the same block, then they must end in the same block. The second and third properties also guarantee that when the Supergraph Partitioning Algorithm is used in the Ammons-Larus Reduction Algorithm, the resulting graph H'^* is unbalanced-left path congruent to H^*. These properties do not address summary-edges. Recall that a summary-edge $c \rightarrow r$ is only a notation on the supergraph that indicates that there is a same-level valid path from the call vertex c to the return-site vertex r; a summary-edge does not affect the control-flow of the supergraph.

The Supergraph Partitioning Algorithm must treat return-edges differently from other edges because it seeks to preserve unbalanced-left paths. Given an unbalanced-left path p to a vertex v that is not an exit vertex, $[p \| v \rightarrow w]$ is an unbalanced-left path for any edge $v \rightarrow w$. Given an unbalanced-left path p to an exit vertex x, there is only one return-edge $x \rightarrow r$ (uniquely determined by the last open parenthesis in p) such that $[p \| x \rightarrow r]$ is an unbalanced-left path; for any other return-edge $x \rightarrow r'$, $[p \| x \rightarrow r']$ is not an unbalanced-left path.

9.3.2 Using the Supergraph Partitioning Algorithm in the Ammons-Larus Reduction Algorithm

The Supergraph Partitioning Algorithm can be used by the Ammons-Larus Reduction Algorithm in place of the Coarsest Partitioning Algorithm. This results in a reduction algorithm for hot-path supergraphs. The steps of this new algorithm are (where H^*, *pp*, and J are the input hot-path supergraph, path profile, and data-flow solution, respectively):

1. Determine which vertices of the hot-path supergraph H^* are hot.

2. Create a compatibility partition π of the vertices of H^*.

3. Run the Supergraph Partitioning Algorithm on H^* and π to produce the partition π'.

4. Output a new graph H'^*, path profile pp', and data-flow solution J':

   - H'^* contains one vertex s_i for each block C_i in π'. H'^* contains an edge $s_i \rightarrow s_j$ if and only if H^* has an edge $u \rightarrow v$ such that $u \in C_i$ and $v \in C_j$. H'^* has a call-edge $s_i \rightarrow s_j$ labeled "$(_{s_i}$" if and only if H^* has a call-edge $c \rightarrow e$ labeled "$(_c$" such that $c \in C_i$ and $e \in C_j$. H'^* has a return-edge $s_i \rightarrow s_j$ labeled "$)_{s_k}$" if and only if H^* has a return edge $x \rightarrow r$ labeled "$)_c$" such that $x \in C_i$, $r \in C_j$, and $c \in C_k$.
   - pp' is *pp* translated onto $H'*$ by replacing each vertex v that appears in a path of *pp* with v's representative in H'^*; thus, if $v \in C_i$, it is translated to s_i.
   - J' for vertex s_i is defined by $J'(s_i) = \sqcap_{v \in C_i} J(v)$.

The output from this algorithm (H'^*, pp', and J') is valid output for a hot-path reduction algorithm and J' preserves the valuable data-flow facts of J. These statements are restated as Theorems 9.3.1 and 9.3.2 below. In the following theorems, we will use H^*, pp, J, $\pi' = \{C_1, C_2, \ldots C_{n'}\}$, H'^*, pp' and J' as they are defined in the above algorithm.

Theorem 9.3.1 *H^* and H'^* are unbalanced-left path congruent.*

Proof: See Appendix C. □

Theorem 9.3.2 *J' is a valid data-flow solution (i.e., it approximates the meet-over-all valid paths solution) for the graph H'^*.*

Proof: See Appendix C. □

These theorems are enough to show that the Ball-Larus Reduction Algorithm with the Supergraph Partitioning Algorithm is a hot-path supergraph reduction algorithm.

9.3.3 Comparing and Contrasting the Partitioning Algorithms

The Supergraph Partitioning Algorithm is very similar to the Coarsest Partitioning Algorithm except that it periodically subdivides, or repartitions, certain blocks to eliminate conflicts between return-edges. A return-edge $x_1 \rightarrow r_1$ labeled "$)_{c_1}$" conflicts with return-edge $x_2 \rightarrow r_2$ labeled "$)_{c_2}$" if x_1 and x_2 are in the same block, c_1 and c_2 are in the same block, and r_1 and r_2 are in different blocks. Conflicting return-edges indicate a violation of the third property of the Supergraph Partitioning Algorithm. The above return-edge conflict can be resolved by separating x_1 and x_2 or by separating c_1 and c_2. It usually is not feasible to put r_1 and r_2 back in the same block without creating a violation of the requirements of the Supergraph Partitioning Algorithm; we will not try to resolve return-edge conflicts in this manner.

To better understand how the Coarsest Partitioning Algorithm is modified to produce the Supergraph Partitioning Algorithm, it is instructive to compare and contrast the properties of the Coarsest Partitioning Algorithm (stated in Section 9.2.3) with the properties of the Supergraph Partitioning Algorithm (stated in Section 9.3.1). The remainder of this section discusses the similarities and differences between the properties of the two algorithms.

The first property of both algorithms states that the output partition respects the input partition. The second property of the Coarsest Partitioning Algorithm states that the output partition respects the input

functions. The second property of the Supergraph Partitioning Algorithm states that the output partition respects call-edges and the intraprocedural edges in the supergraph, with the exception of summary-edges. These two properties are very similar in that a set of congruent edges (that is not a set of summary-edges nor a set of return-edges) can be encoded as a function. This is what is done when the Coarsest Partitioning Algorithm is used in the Ammons-Larus Reduction Algorithm (see Section 9.2.3). Thus, the Supergraph Partitioning Algorithm can use the same technique as the Coarsest Partitioning Algorithm to ensure that the output partition respects the non-return, non-summary edges of the supergraph. The Supergraph Partitioning Algorithm uses the following optimization: rather than using a collection $\mathcal{F}$ of functions where each $f \in \mathcal{F}$ encodes a set of congruent edges, we use a smaller collection $\mathcal{F}'$ of functions where $\mathcal{F}'$ contains a function for encoding true-edges, a function for encoding false-edges, a function for encoding call-edges, a function for encoding "case 0:"-edges, a function for encoding "case 1:"-edges, etc. Since each $f \in \mathcal{F}$ is a subset of some function $g \in \mathcal{F}'$, if the output partition π' respects the functions in $\mathcal{F}'$, then π' respects the functions in $\mathcal{F}$ and hence the second property of the Supergraph Partitioning Algorithm.

The third property of the Supergraph Partitioning Algorithm states that the output partition respects the return-edges of the input hot-path supergraph. This requirement is more complicated than the requirement to respect non-return-edges in three ways:

1. A pair of duplicate exit vertices x_1 and x_2 might not have congruent, outgoing return-edges. This happens if x_1 and x_2 were duplicated for different calling contexts. In this case, there may be a return-edge $x_1 \rightarrow r$ labeled "$)_c$" and an return-edge $x_2 \rightarrow s$ labeled "$)_d$" where the return-site vertices r and s are not duplicates, and the call vertices c and d are not duplicate vertices. These return-edges are not congruent, and they are not in conflict. The vertices x_1 and x_2 do not need to be split. If the Supergraph Partitioning Algorithm were to treat return-edges as it treats others, then when it examines the block B containing r, it would determine that there is a return-edge from x_1 to r but no congruent return-edge from x_2 to a vertex in B. This would cause the algorithm to incorrectly conclude that x_1 and x_2 must be split.

2. Let $u_1 \rightarrow v_1$ and $u_2 \rightarrow v_2$ be congruent edges such that u_1 and u_2 are in the same block B. In most cases, if v_1 and v_2 are in different blocks, then u_1 and u_2 must be separated into different blocks. However, if $u_1 \rightarrow v_1$ is a return-edge with label "$)_{c_1}$" and $u_1 \rightarrow v_1$ is a return-edge with label "$)_{c_2}$" such that c_1 and c_2 are in different blocks, then u_1 and u_2 do not need to be split: the return-edges are not in conflict.

3. If there are conflicting return-edges $x_1 \rightarrow r_1$ labeled "$)_{c_1}$" and $x_2 \rightarrow r_2$ labeled "$)_{c_2}$", then the conflict can be resolved either by splitting the exit vertices x_1 and x_2 or by splitting the call vertices c_1 and c_2. This extra degree of freedom makes the problem harder. Also, splitting a block of call vertices may obviate the need to split a block of exit vertices, and vice-versa. In the Coarsest Partitioning Algorithm, splitting one block always increases the probability that other blocks need to be split.

The Supergraph Partitioning Algorithm includes a mechanism for eliminating return-edge conflicts by subdividing blocks of call vertices and blocks of exit vertices. This is an important difference between the Supergraph Partitioning Algorithm and the Coarsest Partitioning Algorithm.

The third property of the Coarsest Partitioning Algorithm states that the output partition has the minimum possible size while still satisfying the first two properties. There is no similar property for the Supergraph Partitioning Algorithm. The Supergraph Partitioning Algorithm uses heuristics to avoid

splitting blocks, and hence to minimize the size of the output partition. However, it is difficult to guarantee that a minimal output partition is produced. This is because of the need to respect return-edges: given two conflicting return-edges $x_1 \to r_1$ labeled "$)_{c_1}$" and $x_2 \to r_2$ labeled "$)_{c_2}$", there are two ways to fix the problem: split x_1 and x_2 or split c_1 and c_2. It is non-trivial to determine which is an optimal choice. Furthermore, suppose that the choice to split x_1 and x_2 is made. There may be a third exit vertex x_3 that does not have any conflicting edges with x_1 or x_2. When x_1 and x_2 are split, it is hard to make an optimal decision about whether x_3 should go in the sub-block with x_1 or in the sub-block with x_2. A similar problem arises if c_1 and c_2 are split.

9.3.4 The Supergraph Partitioning Algorithm

We are now ready to present our modification of the Coarsest Partitioning Algorithm, which we call the Supergraph Partitioning Algorithm (see Figure 57). The algorithm is very similar to the Coarsest Partitioning Algorithm, except that it periodically repartitions a block of call vertices or a block of exit vertices in order to eliminate conflicts between return-edges. There are two worklists, $W_{SplitPreds}$ and $W_{Repartition}$. The worklist $W_{SplitPreds}$ is processed by the auxiliary function $SplitPreds$. The core of this function is almost identical to the Coarsest Partitioning Algorithm: a block B_i is removed from the worklist $W_{SplitPreds}$; then, the incoming edges of B_i (*i.e.*, those edges $u \to v$ that target some $v \in B_i$) are examined, and predecessor blocks of B_i (*i.e.*, those blocks B_j such that there is an edge $u \to v$ where $u \in B_j$ and $v \in B_i$) are potentially split. The principal difference between the function $SplitPreds$ and the Coarsest Partitioning Algorithm is that the function $SplitPreds$ never splits a block of exit vertices (since it does not try to respect return-edges) and it may add blocks to the worklist $W_{Repartition}$. The conditions under which a block is added to $W_{Repartition}$ are discussed below.

Every block on the worklist $W_{Repartition}$ is either a call-block or an exit-block; an *call-block* is a block of call vertices. A *exit-block* is a block of exit vertices. When the Supergraph Partitioning Algorithm processes an exit-block $B \in W_{Repartition}$, it refines the current partition π' until π' respects the set of return-edges $\{x \to r : x \in B\}$ that have a member of B as their source. The block B is *repartitioned* into sub-blocks such that for any x_1 and $x_2 \neq x_1$ in a given sub-block, there are no conflicting return-edges $x_1 \to r_1$ labeled "$)_{c_1}$" and $x_2 \to r_2$ labeled "$)_{c_2}$". The sub-blocks of B are created greedily: each vertex x of B is added to the first sub-block C such that there are no conflicting return-edges $x \to r$ and $x' \to r'$ where $x' \in C$. All but the largest of the new sub-blocks are added to the worklist $W_{SplitPreds}$. The Figure 60 shows the auxiliary function $RepartitionExitBlock$ that repartitions an exit block.

The Supergraph Partitioning Algorithm uses the function $RepartitionCallBlock$ to repartition a block of call vertices (see Figure 59). This function is almost identical to $RepartitionExitBlock$ except that it creates sub-blocks such that no sub-block contains call vertices c_1 and $c_2 \neq c_1$ where there are conflicting return edges $x_1 \to r_1$ labeled "$)_{c_1}$" and $x_2 \to r_2$ labeled "$)_{c_2}$".

We are now ready to discuss when the function $SplitPreds$ puts a block onto the worklist $W_{Repartition}$. When $SplitPreds$ examines a block R of return-site vertices, it examines incoming edges and adds any of R's predecessor blocks to the worklist $W_{Repartition}$. The justification for the policy is as follows: let $x_1 \to r_1$ labeled "$)_{c_1}$" and $x_2 \to r_2$ labeled "$)_{c_2}$" be return-edges where $c_1, c_2 \in C$ and $x_1, x_2 \in X$. If $r_1, r_2 \in R$ are in the same block, then there is no conflict with these return-edges. If r_1 and r_2 are split into separate blocks, then the return-edges come into conflict. The conflict could be resolved by splitting either the c_1 and c_2 or by splitting x_1 and x_2. However, if $c_1 = c_2$, then the conflict must be resolved by splitting x_1 and x_2. If $x_1 = x_2$, then c_1 and c_2 must be split. (Note that it is not possible for

$B_1, B_2, \ldots B_n$ are the blocks of the initial partition π
$W_{SplitPreds}$ is a worklist of blocks
$W_{Repartition}$ is a worklist of blocks
q is the index of the last created partition block

Main()
1: $W_{SplitPreds} := \{B_1, B_2, \ldots B_n\}$
2: $q := n$

3: **While** $W_{SplitPreds} \neq \emptyset$ **or** $W_{Repartition} \neq \emptyset$
4: $SplitPreds()$

5: **If** $W_{Repartition} = \emptyset$ **Then**
6: **Break**

7: $B := TakeFront(W_{Repartition})$ /* *Take a block from the front of* $W_{Repartition}$ */
8: pick any $v \in B$
9: **If** v is a call vertex **Then**
10: NewSubBlocks := $RepartitionCallBlock(B)$
11: **Else** /* v *is an exit vertex* */
12: NewSubBlocks := $RepartitionExitBlock(B)$

/* *Add all but the largest block in* NewSubBlocks *to* $W_{SplitPreds}$ */
13: pick any $B \in$ NewSubBlocks
14: NewSubBlocks := NewSubBlocks $- \{B\}$
15: LargestBlock := B
16: **While** NewSubBlocks $\neq \emptyset$
17: pick any $B \in$ NewSubBlocks
18: NewSubBlocks := NewSubBlocks $- \{B\}$
19: **If** $\|$LargestBlock$\| < \|B\|$ **Then**
20: $Put(W_{SplitPreds}, \text{LargestBlock})$
21: LargestBlock := B
22: **Else**
23: $Put(W_{SplitPreds}, B)$

24: Output the partition $\{B_1, B_2, \ldots B_q\}$
End Main

Figure 57: The Supergraph Partitioning Algorithm.

```
    /* These global variables are declared in Figure 57 */
    W_SplitPreds is a worklist of blocks
    W_Repartition is a worklist of blocks
    q is the index of the last created partition block

    SplitPreds()
25:   While W_SplitPreds ≠ ∅
26:     B_i = Take(W_SplitPreds) /* select and remove an elt. from W_SplitPreds */
27:     IncomingEdges = {u → v : v ∈ B_i}
28:     While IncomingEdges ≠ ∅
29:       pick any edge u → v from IncomingEdges
30:       let t be the type of u → v /* t may be true, false, "case c:", call, ... */
31:       Inverse := R_t^{-1}(B_i)
32:       remove all the edges of type t from IncomingEdges
33:       If u → v is a return-edge Then
            /* Every vertex in Inverse is an exit; put pred. exit-blocks on W_Repartition */
34:         Foreach B_j such that B_j ∩ Inverse ≠ ∅
35:           PutFront(W_Repartition, B_j) /* if B_j ∉ W_Repartition, put it at the front */
36:         Continue /* Do not split any exit blocks */
37:       If u → v is a summary-edge Then
            /* Every v ∈ Inverse is a call; put pred. call-blocks onto W_Repartition */
38:         Foreach B_j such that B_j ⊆ Inverse
39:           PutBack(W_Repartition, B_j) /* if B_j ∉ W_Repartition, put it at the back */
            /* Fall through to the next case */
          /* Split predecessor blocks */
40:       Foreach B_j such that B_j ∩ Inverse ≠ ∅ and B_j ⊈ Inverse
41:         q := q + 1
42:         Create a new block B_q
43:         B_q := B_j ∩ Inverse
44:         B_j := B_j − B_q
45:         If B_j ∈ W_SplitPreds or ||B_q|| ≤ ||B_j|| Then
46:           Put(W_SplitPreds, B_q)
47:         Else
48:           Put(W_SplitPreds, B_j)
49:         If u → v is a summary-edge Then
              /* B_q is a new sub-block of call vertices that must be examined later */
50:           PutBack(W_Repartition, B_q) /* if B_j ∉ W_Repartition, put it at the back */
    End SplitPreds
```

Figure 58: The function *SplitPreds* used by the Supergraph Partitioning Algorithm (see Figure 57).

```
    /* These global variables are declared in Figure 57 */
    W_SplitPreds is a worklist of blocks
    W_Repartition is a worklist of blocks
    q is the index of the last created partition block

    RepartitionCallBlock(block B)
        /* B is reused as one of the new sub-blocks of the original block */
51:     OrigBlock := B
52:     B := ∅
53:     NewSubBlocks := {B}

        /* Create the sub-blocks of OrigBlock */
54:     While OrigBlock ≠ ∅
55:         pick any c ∈ OrigBlock
56:         OrigBlock := OrigBlock − {c}
57:         Foreach C ∈ NewSubBlocks
58:             If there are no conflicting return-edges x → r labeled ")_c" and
                x' → r' labeled ")_c'" s.t. c' ∈ C Then
59:                 C := C ∪ {c}
60:                 Break
61:         If c was not added to any block in NewSubBlocks Then
62:             q := q + 1
63:             create a new block B_q
64:             B_q := {c}
65:             NewSubBlocks := NewSubBlocks ∪ {B_q}

66:     Return NewSubBlocks
    End RepartitionCallBlock
```

Figure 59: The function *RepartitionCallBlock* used by the Supergraph Partitioning Algorithm (see Figure 57).

```
    /* These global variables are declared in Figure 57 */
    W_SplitPreds is a worklist of blocks
    W_Repartition is a worklist of blocks
    q is the index of the last created partition block

    RepartitionExitBlock(block B)
        /* B is reused as one of the new sub-blocks of the original block */
67:     OrigBlock := B
68:     B := ∅
69:     NewSubBlocks := {B}

        /* Create the sub-blocks of OrigBlock */
70:     While OrigBlock ≠ ∅
71:         pick any x ∈ OrigBlock
72:         OrigBlock := OrigBlock − {x}
73:         Foreach C ∈ NewSubBlocks
74:             If there are no conflicting return-edge x → r labeled ")_c" and
                x' → r' labeled ")_c'" s.t. x' ∈ C Then
75:                 C := C ∪ {x}
76:                 Break
77:         If x was not added to any block in NewSubBlocks Then
78:             q := q + 1
79:             create a new block B_q
80:             B_q := {x}
81:             NewSubBlocks := NewSubBlocks ∪ {B_q}

82:     Return NewSubBlocks
    End RepartitionExitBlock
```

Figure 60: The function *RepartitionExitBlock* used by the Supergraph Partitioning Algorithm (see Figure 57).

$x_1 = x_2 \wedge c_1 = c_2 \wedge r_1 \neq r_2$, for then the supergraph would be malformed.)

Anytime a block of return-site vertices is split, it may create conflicts between pairs of return-edges. Both the call-blocks and the exit-blocks that may be involved in these conflicts are put onto the worklist $W_{Repartition}$. It is not enough to put just the call-blocks or just the exit-blocks on the worklist since some conflicts cannot be resolved by splitting a call-block and some cannot be resolved by splitting an exit-block. The function $SplitPreds$ detects call-blocks and exit-blocks that may be involved in a return-edge conflict when it processes a block R of return-site vertices. R may have been split off from a larger block of return-site vertices, which means that it may be involved in unresolved return-edge conflicts. An exit-block X could be involved in these conflicts if there is a return edge $x \rightarrow r$ such that $x \in X$ and $r \in R$. A call-block C could be involved in the return-edge conflicts if there is a return-edge $x \rightarrow r$ labeled "$)_c$" where $c \in C$ and $r \in R$. Recall that there exists a return-edge $x \rightarrow r$ labeled "$)_c$" if and only if there is a summary-edge $c \rightarrow r$. This implies that a call-block C could be involved in the new return-edge conflicts if there is a summary-edge $c \rightarrow r$ such that $c \in C$ and $r \in R$. Thus, when $SplitPreds$ examines a block R of return-site vertices, it examines incoming summary-edges and return-edges and adds any predecessor call-blocks and exit-blocks to the worklist $W_{Repartition}$. One optimization is that $SplitPreds$ may first use the summary-edges to justify splitting a call-block (see the discussion of heuristics below). When this is done, only one of the new sub-blocks will still have summary-edges to vertices of R; only that sub-block needs to be added to $W_{Repartition}$.

Heuristics Employed

We have explained most of the Supergraph Partitioning Algorithm. All that remains is to describe the several heuristics employed by the algorithm. These heuristics aim to minimize the running time of the algorithm and to minimize the size of the output partition.

Repartitioning blocks (in the functions $RepartitionCallBlock$ and $RepartitionExitBlock$) is the most expensive step of the Supergraph Partitioning Algorithm. The first heuristic seeks to maximize the number of return-edge conflicts that are resolved each time a block is repartitioned; this minimizes the number of times the functions $RepartitionCallBlock$ and $RepartitionExitBlock$ need to be called. The heuristic is to call the function $SplitPreds$ anytime there may be blocks on the worklist $W_{SplitPreds}$. Every call to $SplitPreds$ may cause a block of return-site vertices to be split, which may in turn, create return-edge conflicts. The sooner the algorithm learns about a potential return-edge conflict, the better. Suppose the algorithm repartitions a block B into sub-blocks $\{b_1, b_2, \ldots, b_k\}$, and then later a block R of return-site vertices is split, potentially creating a new return-edge conflict. This new conflict may necessitate repartitioning of one of the sub-blocks b_i. If, on the other hand, the block R could be split before B was repartitioned, then the return-edge conflicts created by splitting R could be resolved when B is first repartitioned, thereby avoiding the second repartitioning. To maximize the number of return-edge conflicts resolved by each repartitioning, the algorithm alternates processing all of the blocks on the worklist $W_{SplitPreds}$ (which may create new return-edge conflicts) with processing a single block on the worklist $W_{Repartition}$ (which may add blocks to $W_{SplitPreds}$).

The amount of the work done by the repartitioning functions is proportional to the size of the block that is repartitioned. The second heuristic splits call-blocks into smaller sub-blocks before the function $RepartitionCallBlock$ is called. Repartitioning several smaller sub-blocks is less expensive than repartitioning one large block. The second heuristic is to have $SplitPreds$ treat summary-edges the same way it treats other intraprocedural edges and call-edges. It is a corollary of Lemma C.0.3 that for every pair of call vertices $c_1, c_2 \in B_i$, for every summary-edge $c_1 \rightarrow r_1$, there must be a a summary-edge $c_1 \rightarrow r_2$ such that r_1 and r_2 are in the same block. If there is no such summary-edge $c_1 \rightarrow r_2$, then the

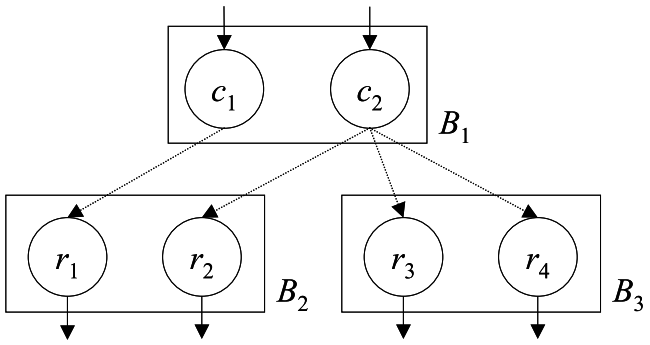


Figure 61: Schematic of a hot-path supergraph that shows a call vertex with multiple outgoing summary-edges. Blocks represent a partitioning of the vertices of the hot-path supergraph: $B_1 = \{c_1, c_2\}$, $B_2 = \{r_1, r_2\}$, and $B_3 = \{r_3, r_4\}$. The vertices c_1 and c_2 are duplicate call vertices. The vertices r_1, r_2, r_3 and r_4 are duplicate return-site vertices.

Supergraph Partitioning Algorithm will ultimately split c_1 and c_2 into different partitions. By having $SplitPreds$ examine summary-edges, c_1 and c_2 may be split earlier rather than later, which may save work in the repartitioning functions.

A set of congruent summary-edges defines a relation between call vertices and return-site vertices, but not a function. This may decrease the effectiveness of the second heuristic:

Example 9.3.3 Figure 61 shows a schematic of a hot-path supergraph with several duplicate call and return-site vertices. The figure also shows a partition of the vertices of the supergraph. The block B_1 should be split because there is a summary-edge $c_2 \rightarrow r_3$ from c_2 to a vertex (namely r_3) in B_3, and there is no summary-edge from c_1 to a vertex in B_3. The block-splitting technique in $SplitPreds$ may or may not discover that B_1 should be split: if the function examines only the block B_2, it will determine that the set of predecessors of vertices in B_2 includes all of the vertices in B_1 and conclude (incorrectly) that B_1 does not need to be split; on the other hand if B_3 is examined, then the set of predecessors of vertices in B_3 includes only some of the vertices in B_1, implying that B_1 must be split. □

Note that it is not guaranteed that both block B_2 and block B_3 are eventually examined: if blocks B_2 and B_3 were created by splitting some larger block, then only one of them would go on the worklist $W_{SplitPreds}$ to be examined later. Regardless of whether B_2 or B_3 is examined, B_1 will eventually be split. This is a corollary of Lemma C.0.3: suppose on the contrary that both c_1 and c_2 are left in the block B_1. The summary-edge $c_2 \rightarrow r_3$ indicates that there is a same-level valid path p' from c_2 to r_3. By Lemma C.0.3, there must be a congruent path p from c_1 to some vertex in B_3. The path p implies that there should be a summary-edge from c_1 to some vertex in B_3. Since there is no such summary-edge, it must be the case that the Supergraph Partitioning Algorithm eventually splits B_1.

Recall that the properties of the Supergraph Partitioning Algorithm do not mention summary-edges. Examining summary-edges is a heuristic, it is not required for correctness. If $SplitPreds$ decides to split a block of call vertices by examining summary-edges, the split is guaranteed to be correct. If $SplitPreds$ misses an opportunity to perform a split that is indicated by summary-edges, then the split will still happen, but at some different stage of the algorithm.

The third heuristic of the Supergraph Partitioning Algorithm puts exit-blocks at the beginning of $W_{Repartition}$ and call-blocks at the end of $W_{Repartition}$. This is based on experimental evidence that repartitioning exit-blocks before call-blocks leads to an output partition with fewer blocks. This may

mean that repartitioning exit-blocks resolves more return-edge conflicts than repartitioning call-blocks. If call-blocks are repartitioned first, then many return-edge conflicts may be left unresolved, such that exit-blocks must still be subdivided to fix the return-edge conflicts. On the other hand, when exit-blocks are repartitioned first, all return-edge conflicts may be resolved so that when $RepartitionCallBlock$ is subsequently called on a call-block B, it can leave B intact.

Proof of Correctness

We now turn to proving the correctness of the Supergraph Partitioning Algorithm. We have the following theorem:

Theorem 9.3.4 *When the Supergraph Partitioning Algorithm is run on a hot-path supergraph H^* and a partition $\pi = B_1, B_2, \ldots, B_n$ of the vertices of H^*, the output partition $\pi' = \{C_1, C_2, \ldots, C_{n'}\}$ satisfies the properties of the Supergraph Partitioning Algorithm listed in Section 9.3.1.*

Proof: See Appendix C. □

Analysis of Running Time

In this section, we show that the running time of the Supergraph Partitioning Algorithm is $O(\|E\| \log m + \|\mathrm{MaxRtnEdgesForCallBlk}\|^2 \|R\| \|C\| + \|\mathrm{MaxRtnEdgesForExitBlk}\|^2 \|R\| \|X\|)$, where $\|E\|$ is the number of edges of the input supergraph, m is the size of the largest block in the input partition (which is usually much smaller that the number of vertices in the supergraph), $\|\mathrm{MaxRtnEdgesForCallBlk}\|$ is the maximum number of return-edges from labeled call vertices in the same call-block, $\|\mathrm{MaxRtnEdgesforExitBlk}\|$ is the maximum number of return-edges from an exit-block, $\|R\|$ is the number of return-site vertices, $\|C\|$ is the number of call vertices, and $\|X\|$ is the number of exit vertices. The Supergraph Partitioning Algorithm performs a total of $O(\|E\| \log m)$ steps in the function $SplitPreds$ and a total of $O(\|\mathrm{MaxRtnEdgesForCallBlk}\|^2 \|R\| \|C\|)$ in the function $RepartitionCallBlock$ and $O(\|MaxRtnEdgesForExitBlk\|^2 \|R\| \|X\|)$ steps in the function $RepartitionExitBlock$

We start by discussing the total number of steps the Supergraph Partitioning Algorithm performs in the function $SplitPreds$. Throughout the following discussion, we will refer to line numbers in the function $SplitPreds$ shown in Figure 58. [1] shows the running time of the Coarsest Partitioning Algorithm to be $O(n \log n)$, where n is the number of elements in the input set S. A careful reading of their argument shows that the algorithm runs in $O(\|f\| \log m)$, where $\|f\|$ is the size of the input function f and m is the size of the largest block in the input partition π. (Since $\|f\| = n$ and in the worse case $O(m) = O(n)$, we have that $O(\|f\| \log m) = O(n \log n)$.) Using an argument closely based on the one in [1], we will show that the Supergraph Partitioning Algorithm executes a total of $O(\|E\| \log m)$ steps while in the function $SplitPreds$, where $\|E\|$ is the number of edges in the input supergraph and m is the size of the largest block in the input partition. We expect m to be much smaller than $\|V\|$, the number of vertices in the input supergraph, though in the worse case $O(m) = O(\|V\|)$.

The proper data-structures are required for an implementation of $SplitPreds$ that performs no more than $O(\|E\| \log m)$ steps. In an efficient implementation, each vertex v has a pointer to the block B that v belongs to. Each block B is represented as an object with the following data members:

- a doubly-linked list called `members` that holds the vertices in the block.
- a doubly-linked list called `intersectionInverse` that is used to temporarily hold vertices of the block that are also in the set Inverse. This list is used to efficiently compute the set $(B \cap \text{Inverse})$ that is needed in the function $SplitPreds$ at lines 34, 40, and 43.

- an integer `size` that records the number of vertices in the block.
- a boolean flag `inWSplitPreds` that tells if the block is currently on the worklist $W_{SplitPreds}$. This flag is used for maintenance of the worklist $W_{SplitPreds}$.
- a boolean flag `inWRepartition` that tells if the block is currently on the worklist $W_{Repartition}$. This flag is used for maintenance of the worklist $W_{Repartition}$.

The function $SplitPreds$ uses the following data structures:

- an array of lists of edges called `EdgeBins`. This array is indexed by edge type (*i.e.*, true, false, call, summary, etc.). Given an edge type, the list of edges of that type can be retrieved in constant time. The array is static, so that it only needs to be initialized once. The array is initialized to be all empty lists at the beginning of the program; since this takes at most $O(\|E\|)$ steps, it does not add to the overall complexity of the algorithm.
- a list of lists of edges called `IncomingEdges`. This data-structure stores the set $\mathrm{IncomingEdges}$ used by $SplitPreds$. The edges of the set $\mathrm{IncomingEdges}$ are segregated by edge type into the different lists of `IncomingEdges`. That is, there is one list in `IncomingEdges` for all the true-edges of IncomingEdges, there is one list in `IncomingEdges` for all the false-edges of IncomingEdges, etc. The union of all the edges on all the lists of `IncomingEdges` is equal to the set $\mathrm{IncomingEdges}$.
- a list of blocks called `PredBlocks`. This is used to hold the list of blocks B such that $B \cap \mathrm{Inverse} \neq \emptyset$.

Given the above data-structures, an implementation of $SplitPreds$ can perform one iteration of the outer loop (lines 25–50) in $O(\|\mathrm{IncomingEdges}\|)$ steps. We first show that the data-structure `IncomingEdges` can be initialized in $O(\|\mathrm{IncomingEdges}\|)$ steps by using the array `EdgeBins`. A linear scan of the incoming edges for each vertex of B is performed. When an edge $u \to v$ of type t is examined, it is added to the list `EdgeBins[t]`. As the edges are examined, a list of the non-empty entries of `EdgeBins` is kept. After all the incoming edges are examined, all of the non-empty lists in `EdgeBins` are moved to `IncomingEdges`.

Each iteration of the second-to-outermost loop (lines 28–50) of $SplitPreds$ processes one list L_t of the list of lists `IncomingEdges`. (Thus, one iteration processes all of the true edges, one iteration processes all of the false edges, etc.) The list L_t is the set of congruent edges encoded by the relation R_t at line 31. An efficient implementation can use L_t directly instead of creating the relation R_t. The number of steps preformed during one iteration of the loop in lines 28–50 is proportional to the size $\|L_t\|$ of the list processed. The number of steps over all iterations of the loop in lines 28–50 is $O(\|\mathrm{IncomingEdges}\|)$.

An efficient implementation of the loop in lines 28–50 does not reify the set Inverse. However, it does use efficient techniques for finding each block B such that $B \cap \mathrm{Inverse} \neq \emptyset$ and for finding the sets $(B \cap \mathrm{Inverse})$ and $(B - \mathrm{Inverse})$. This is done as follows: let L_t be the list being processed during an iteration of the loop in lines 28–50. A linear scan of the edges in L_t is performed. As each edge $u \to v$ is considered, the vertex u (which must be an element of the set Inverse) is moved from `B.members` to `B.intersectionInverse`, where B is the block containing u. If u is the first vertex to be added to `B.intersectionInverse`, then the block B is added to the list `PredBlocks`. When the scan of L_t is finished, `PredBlocks` contains a list of all the blocks B such that $B \cap \mathrm{Inverse} \neq \emptyset$. The set

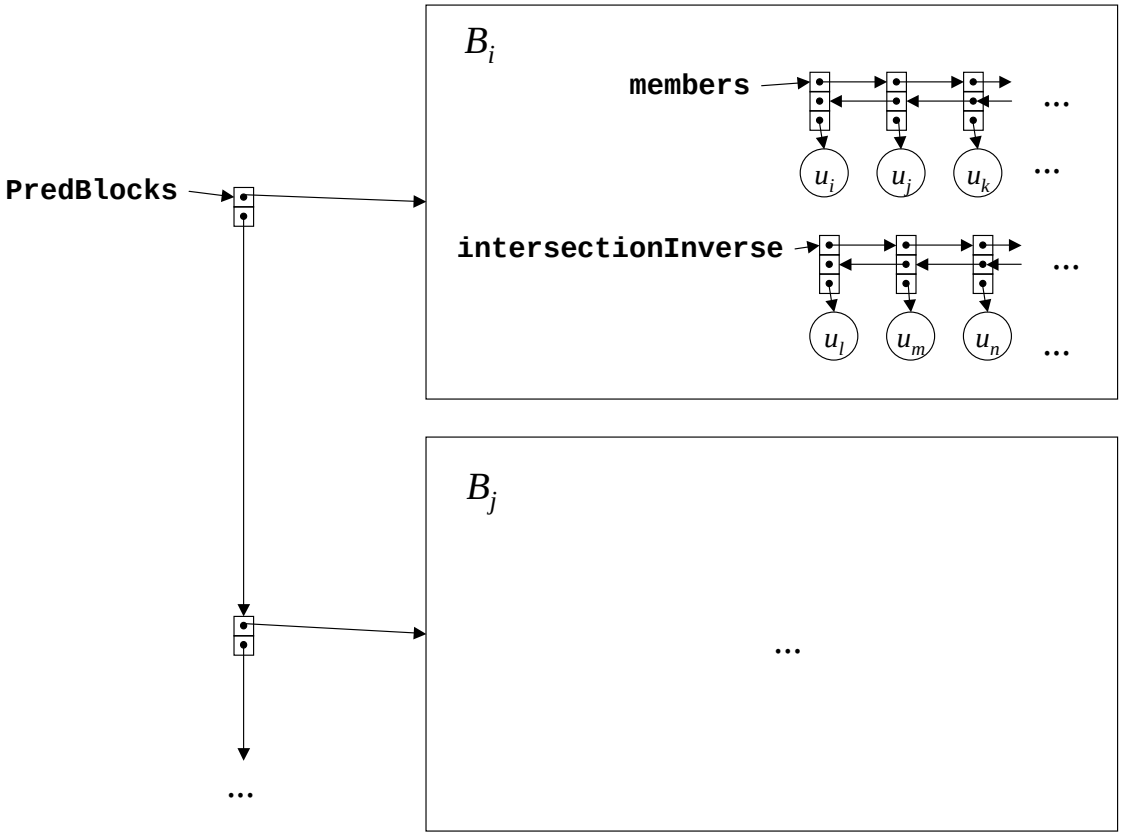


Figure 62: The list `PredBlocks` constructed in an efficient implementation of *SplitPreds*. `PredBlocks` is a list of blocks. The members of a block B are found in two lists, `B.members` and `B.intersectionInverse`. The function *SplitPreds* may split the block B by moving the vertices on the list `B.intersectionInverse` to a new block (lines 43–44 of Figure 58).

$(B \cap \text{Inverse})$ can be found in `B.intersectionInverse`. The set $(B - \text{Inverse})$ can be found in `B.members`. Figure 62 shows a schematic of the list `PredBlocks` after L_t has been examined.

At this point, we will describe how to implement lines 40–44 and trust the reader to see how lines 33–36, 37–39, 45–48 and 49–50 can be implemented. Line 40 requires that we find each block B_j such that $(B_j \cap \text{Inverse}) \neq \emptyset$ and $B_j \not\subseteq \text{Inverse}$. The list `PredBlocks` contains a list of blocks such that $(B_j \cap \text{Inverse}) \neq \emptyset$. By the above construction B_j`.intersectionInverse` contains the set $(B_j \cap \text{Inverse})$ and B_j`.members` contains the set $(B_j - \text{Inverse})$. This means that $B_j \not\subseteq \text{Inverse}$ iff B_j is on the list `PredBlocks` and B_j`.members` is not empty.

To implement lines 40–44, a scan of the list `PredBlocks` is performed. For each block B_j of `PredBlocks`, if B_j`.members` is empty, then the list B_j`.intersectionInverse` is spliced back onto the list B_j`.members`. This resets the bookkeeping for the block B_j, since no further processing of B_j is required as far as lines 40–48 are concerned — $B_j \subseteq \text{Inverse}$. If B_j`.members` is not empty, new block B_q is created (this implements lines 41–42). Each vertex on the list B_j`.intersectionInverse` is moved to B_q`.members` (this implements lines 43–44). As each vertex v is moved, the data-structure for v is updated to indicate that it now belongs to B_q, the integer B_j`.size` is decremented, and the integer B_q`.size` is incremented. The number of steps required to perform these operations is proportional to the length of B_j`.intersectionInverse`.

There are $O(\|L_t\|)$ vertices total on all of the lists `B.intersectionInverse` of blocks B on `PredBlocks` — at most one element is moved onto a list `B.intersectionInverse` for each edge in $O(\|L_t\|)$. The number of blocks on the list `PredBlocks` is also bounded by $O(\|L_t\|)$. This means that the total number of operations required to implement lines 40–44 is bounded by $(O\|L_t\|)$. In fact lines 33–50, require no more than $O(\|L_t\|)$ steps. (Given our description of how to implement lines 40–44, implementing lines 33–39 and lines 45–50 is straightforward.)

Thus, for each iteration of the loop in lines 28–50, an efficient implementation removes one list L_t from the list `IncomingEdges`, and performs $O(\|L_t\|)$ operations. The total number of operations required for all iterations of the loop is $O(\|\text{IncomingEdges}\|)$. This means that processing one block B_i on the worklist $W_{SplitPreds}$ takes at most $O(\|\text{IncomingEdges}\|)$ steps, where IncomingEdges is the set $\{u \to v : v \in B_i\}$. Suppose that each time *SplitPreds* processes a block B_i (lines 26–50), the cost of processing B_i is charged to each $v \in B_i$ in proportion to the number of edges coming into v. There is some constant c such that v is charged no more than $c \cdot \|\text{IncomingEdges}(v)\|$ for an execution of lines 26–50 of *SplitPreds*. (Here, $\text{IncomingEdges}(v) = \{a \to b : b = v\}$.)

Let us consider the number of times a vertex v can be in a block B that $W_{SplitPreds}$ takes from $W_{SplitPreds}$. This is equivalent to the number of times v has its block B moved onto $W_{SplitPreds}$. This happens once at line 1 of Figure 57. Otherwise, it may happen at lines 20 or 23 of Figure 57 or at lines 46 or 48 of Figure 58. In all these other cases, v's block is a least half the size of the block containing v when v last had its block moved onto $W_{SplitPreds}$. This means that v cannot be in a block that is moved onto $W_{SplitPreds}$ more than $1 + \log m$ times, where m is the size of the largest block in the input partition π. This means that the total amount a vertex v is charged for execution of *SplitPreds* is at most $O(\|\text{IncomingEdges}(v)\| \cdot \log m)$. Let V is the set of vertices in the input supergraph. Since $E = \bigcup_{v \in V} \text{IncomingEdges}(v)$, the total number of steps executed in *SplitPreds* is $O(\|E\| \cdot \log m)$.

We are now ready to determine the number of steps performed by the Supergraph Partitioning Algorithm while in the functions *RepartitionCallBlock* and *RepartitionExitBlock*. First, we consider the amount of time spent in a single call of *RepartitionCallBlock*. The execution time during an invocation of *RepartitionCallBlock* on block B is dominated by the innermost loop, lines 58–60 of Figure 59. For every $c_1, c_2 \in B$, the lines 58–60 may compare every return-edge $x_1 \to r_1$ labeled "$)_{c_1}$" with every other return-edge $x_2 \to r_2$ labeled $)_{c_2}$. In the worst case, this means that each execution of lines 58–60

may perform $O(\|\mathrm{MaxRtnEdgesForCallBlk}\|^2)$ comparisons, where $\|\mathrm{MaxRtnEdgesForCallBlk}\|$ is the maximum number of return-edges associated with a single call block.

The number of calls to $RepartitionCallBlock$ is equal to the number of times a call-block is put on $W_{Repartition}$. Every time a block of return-site vertices is processed in $SplitPreds$, all of the call-blocks currently in the partition π' may be added to $W_{Repartition}$. For any block B, it can be shown by induction on the size n of B that a sub-block of B is added to the worklist $W_{SplitPreds}$ at most n times. This implies that the maximum number of times a block of return-site vertices can be taken from $W_{SplitPreds}$ is $\|R\|$, where R is the set of all return-site vertices. The maximum number of call-blocks in π' is $\|C\|$, where C is the set of all call vertices. This implies that $RepartitionCallBlock$ can be invoked at most $O(\|C\| \cdot \|R\|)$ times. Thus, the Supergraph Partitioning Algorithm performs at most $O(\|\mathrm{MaxRtnEdges}\|^2 \cdot \|C\| \cdot \|R\|)$ steps in the function $RepartitionCallBlock$.

A similar argument shows that the Supergraph Partitioning Algorithm performs at most $O(\|\mathrm{MaxRtnEdges}\|^2 \cdot \|X\| \cdot \|R\|)$ steps in the function $RepartitionExitBlock$, where X is the set of all exit vertices. The lines 13–23 of Figure 57 add all but the largest of the blocks in NewSubBlocks to the worklist $W_{SplitPreds}$. The work done every time these lines execute their work can be billed to the previous execution of $RepartitionCallBlock$ or $RepartitionExitBlock$ where the list NewSubBlocks was created. This does not affect the asymptotic running time spent in these functions.

The total number of steps executed by Supergraph Partitioning Algorithm is:

$$O((\|E\| \log m) + (\|\mathrm{MaxRtnEdges}\|^2 \|R\| (\|C\| + \|X\|)))$$

In practice, the running time is much better than this. This is borne out in the experimental results presented in the next chapter.

Chapter 10

Reducing the Hot-path Supergraph Using Edge Redirection

In this section, we present a new strategy for reducing the hot-path supergraph based on the concept of *edge redirection*, or replacing an edge $u \to v$ with an $u \to v'$. This concept was introduced in Example 53. The rest of this section is organized as follows: in Section 10.1 we discuss the potential problems involved in redirecting an edge in a hot-path supergraph. This motivates Section 10.2, where we give an algorithm for determining when it is safe to perform edge redirection. Section 10.3 presents the Edge Redirection Algorithm, which decides when it is profitable to perform edge redirection. Section 10.6 shows how to translate a path profile from a hot-path supergraph onto the reduced hot-path supergraph that results from edge redirection.

10.1 Problems Created by Performing an Edge Redirection

Given two duplicate vertices v and v', suppose that the vertices reachable from v' have better data-flow facts (*i.e.*, $\sqsupseteq$) than the vertices reachable from v. Then we would prefer that the program execute v' rather than v: given an edge $u \to v$, we would like to replace it with the edge $u \to v'$. There are three general problems that must be addressed: (1) does replacing the edge $u \to v$ with the edge $u \to v'$ invalidate any of the data-flow facts in the hot-path supergraph?; (2) does replacing the edge $u \to v$ with the edge $u \to v'$ corrupt the structure of the hot-path supergraph?; (3) does replacing the edge $u \to v$ with the edge $u \to v'$ "drop" a valuable data-flow fact that is used on a path from v but not on the congruent path from v'? The following example demonstrates the first and second problems:

Example 10.1.1 Figure 63 shows a supergraph with two paths. Along one path, the local variable x takes the value 2. On the other path, x has the value $\perp$. Figure 64 shows the hot-path supergraph that results from duplicating these two paths. Notice that the data-flow facts on the path from $[h' \to E' \to F']$ are better than the data-flow facts on the path $[h'' \to E'' \to F'']$. This suggests that we may want to replace the edge $g'' \to h''$ with the edge $g'' \to h'$ so that any path to g'' will continue along the path $[g'' \to h' \to E' \to F']$ instead of the path $[g'' \to h'' \to E'' \to F'']$.

However, replacing the edge $g'' \to h''$ with $g'' \to h'$ is not enough to realize this change in control flow. With the edge $g'' \to h''$, the hot-path supergraph contains the same-level valid path $p = [A' \to C' \to D'' \to g'' \to h'' \to E'' \to F'']$. After replacing the edge $g'' \to h''$ with the edge $g'' \to h'$, we expect the path p to be replaced by the path $q = [A' \to C' \to D'' \to g'' \to h' \to E' \to F']$. Unfortunately, q is not a same-level valid path: the call edge $D'' \to g''$ is labeled "$(_{D''}$" while the return edge $h' \to E'$ is labelled "$)_{D'}$". To solve this problem, we could create a second return edge $h' \to E'$ between h' and E' that is labeled "$)_{D''}$". However, this creates other problems.

One additional problem is that the summary-edges in *main* are invalidated: the summary-edge $D'' \to E''$ must be removed and the summary-edge $D'' \to E'$ must be added.

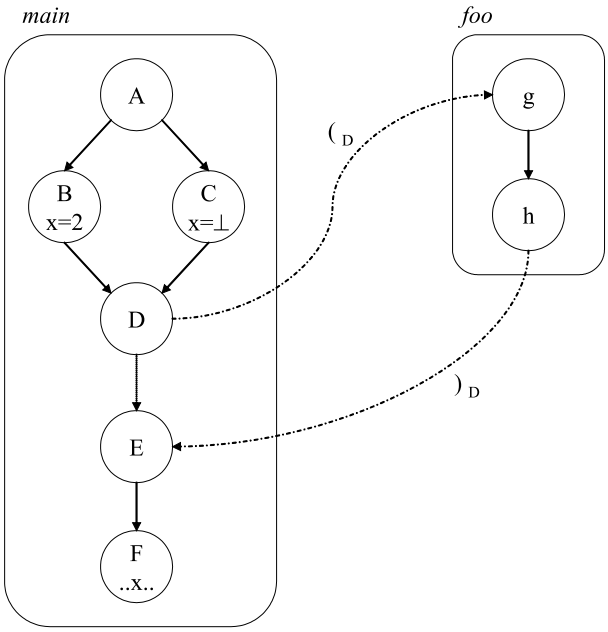


Figure 63: Simple supergraph with two paths.

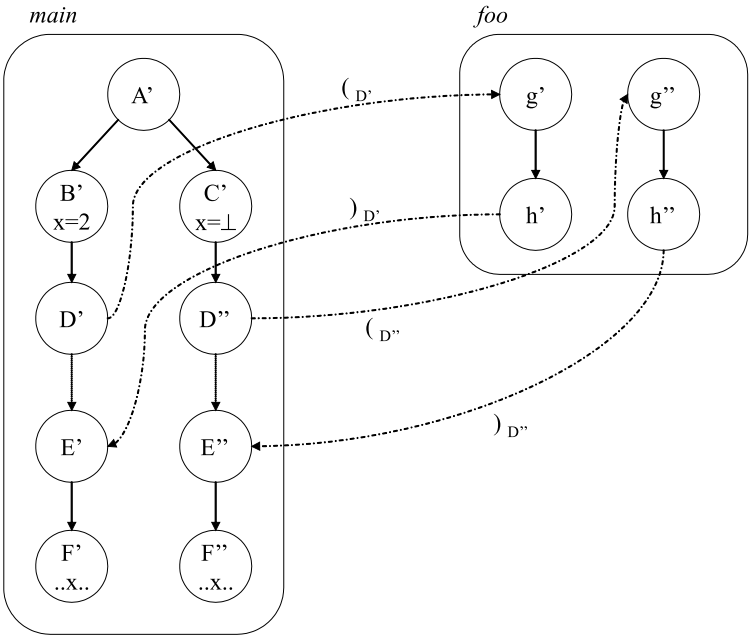


Figure 64: Hot-path supergraph for Figure 63

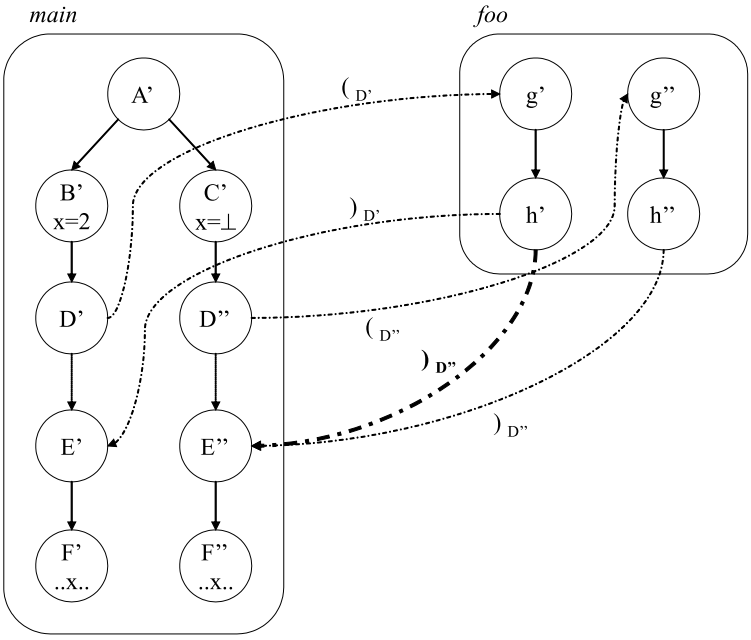


Figure 65: Hot-path supergraph from Figure 63 with an additional return-edge $h' \to E''$ labeled "$)_{D''}$".

A final problem with the redirection of the edge $g'' \to h''$ and the addition of the return-edge $h' \to E'$ labeled "$)_{E'}$" is that it invalidates the data-flow facts at E' and F': before, x had the constant value 2 in these vertices; afterwards, x has the data-flow value $\perp$ in the same vertices. This change in the data-flow solution follows from the fact that we have created a new same-level valid path from C' to E'. Notice that replacing $g'' \to h''$ with $g'' \to h'$ does not invalidate any local data-flow facts in the procedure *foo*. In general, redirecting an edge may invalidate data-flow facts that are not visible (*e.g.*, data-flow values for variables that are out of scope) at the site of the edge replacement.

We shall see later that if we wish to replace $g'' \to h''$ with $g'' \to h'$, the best way to fix the ensuing control-flow problems is to add the return-edge $h' \to E''$ labeled "$)_{D''}$". In this case the path $[g'' \to h'' \to E'' \to F'']$ is replaced by the path $[g'' \to h' \to E'' \to F'']$. □

10.2 Determining When Edge Redirection is Possible

In this section, we give an algorithm for determining when it is *safe* to redirect an edge. Informally, it is safe to redirect an edge $u \to v$ to point at a vertex v' if the following safety properties are satisfied:

1. Replacing the edge $u \to v$ with the edge $u \to v'$ does not destroy any valuable data-flow facts that are *visible* at the vertex v'. A data-flow fact $J(w)(x)$ is visible at v if the variable x is in scope at v. Note that it may be that $w \neq v$.

2. Replacing the edge $u \to v$ with the edge $u \to v'$ does not destroy any valuable data-flow facts that are not visible at the vertex v' (*i.e.*, data-flow facts for variables that are not in scope at v').

3. Replacing the edge $u \to v$ with the edge $u \to v'$ does not corrupt the hot-path supergraph in a way that is "irreparable". (Of course, the hot-path supergraph can always be repaired by taking

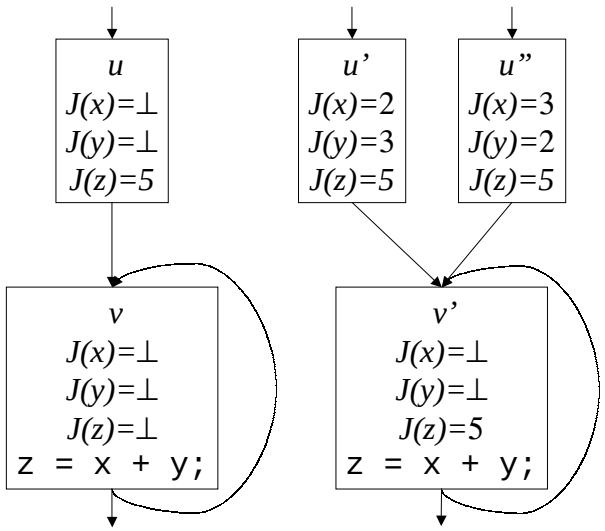


Figure 66: Example supergraph showing that if J is a meet-over-all paths solution the condition $T_u(J(u)) \sqsubseteq J(v')$ is insufficient for replacing the edge $u \to v$ with the edge $u \to v'$. The figure shows five vertices, each marked with the meet-over-all paths solution J for constant propagation. The duplicate vertices v and v' contain the statement `z = x + y`. Along every path to v', the variable z has the data-flow value 5. If the edge $u \to v$ is replaced by the edge $u \to v'$ a new path $[u \to v' \to v']$ is created where z has the data-flow value $\bot$. This happens even though $T_u(J(u)) \sqsupseteq J(v')$. In contrast, in the greatest-fixed point solution, $J(v')(z) = \bot$, and replacing the edge $u \to v$ with $u \to v'$ is safe.

out the edge $u \to v'$ and putting the edge $u \to v$ back in; we will define an irreparable corruption of the supergraph below.)

4. Replacing the edge $u \to v$ with the edge $u \to v'$ does not drop a valuable data-flow fact that is used on a path from v but not on any congruent path from v'; *i.e.*, the data-flow facts on paths from v' are at least as good as the data-flow facts on paths from v.

Checking that these requirements are satisfied is non-trivial. In several cases, we will use conditions that are sufficient, but not necessary to determine if the above requirements are met.

Of the four requirements, the first is the easiest to handle. A sufficient condition for satisfying the first requirement is that J is a greatest-fixed-point solution and $T_u(J(u)) \sqsupseteq J(v')$ where T_u is the transfer function for u. If this condition is satisfied, then adding the edge $u \to v'$ does not destroy any of the data-flow facts that are visible at v'. Figure 66 shows that if J is the meet-over-all paths solution, then the fact that $T_u(J(u)) \sqsupseteq J(v')$ is not sufficient to guarantee the first requirement. For the rest of this section, we shall assume that J is a greatest-fixed point solution.

The second, third, and fourth requirements listed are more difficult to check because they can not be verified "locally": these requirements place restrictions on the set of paths from v and v'. We will discuss how to check these requirements below, but first we elaborate on the third requirement. The only corruption of the hot-path supergraph that we consider to be "irreparable" is if we are required to add a return-edge to maintain the original control-flow behavior, but there is no way to add a return-edge that maintains control-flow and maintains the data-flow solution. For example, suppose that an edge redirection creates a new same-level valid path from entry vertex e to exit vertex x. If there is a call-edge $c \to e$ labeled "$(_c$" but no return-edge $x \to r$ labeled "$)_c$" then the modified hot-path supergraph

is malformed. To fix this problem, we must find a return-site vertex r such that adding the return-edge $x \rightarrow r$ labeled "$)_c$" restores the execution behavior of the supergraph. However, if for every such return-site vertex r, adding the return-edge $x \rightarrow r$ labeled "$)_c$" destroys the data-flow solution at r, then we cannot repair the supergraph by adding a return-edge and the supergraph has been irreparably damaged. To make sure that redirecting an edge from v to v' does not cause irreparable corruption of the supergraph, we will require (roughly) that for every path p from v that reaches a return-edge labeled "$)_c$", the congruent path p' from v' also reaches a return-edge labeled "$)_c$". As we will see in the proof of Theorem 10.4.8, this condition is also sufficient for satisfying the second requirement listed above.

To help check that safety properties 2, 3, and 4 are satisfied, we compute a *vertex subsumption relation*, or $\succ$-relation. The vertex subsumption relation is defined in terms of the *path subsumption relation*. Informally, a path p' subsumes a path p (written $p' \succ p$) if

1. p' exactly mimics the control-flow of p. In particular, p' must be congruent to p. Furthermore, if p has an unmatched return-edge $x \rightarrow r$ labeled "$)_c$", then p''s corresponding return-edge $x' \rightarrow r'$ is also labeled "$)_c$".

2. the data-flow facts along p' are at least as good as the data-flow facts along p.

A vertex v' subsumes a duplicate vertex v (written $v' \succ v$) if for every unbalanced-right-left path p from v, there is a unique unbalanced-right-left path p' from v' such that $p' \succ p$. For the time being, we use the path subsumption relation as a relation between the paths of a single hot-path graph; later, we will use the path subsumption relation as a relation between the paths of multiple path congruent graphs. This is also true of the vertex subsumption relation.

Note that in previous chapters, we were generally concerned with preserving unbalanced-left paths (and the data-flow facts along unbalanced-left paths). The path subsumption relation is defined in terms of unbalanced-right-left paths. Ultimately, this chapter is also concerned with preserving unbalanced-left paths. The suffix of an unbalanced-left path is an unbalanced-right-left path. Thus, by preserving all unbalanced-right-left paths (and the data-flow facts along unbalanced-right-left paths) we will preserve all suffixes of unbalanced-left paths (and their data-flow facts); we prove this in Theorems 10.4.8 and 10.4.6.

The condition $v' \succ v$ is sufficient to guarantee that replacing any edge $u \rightarrow v$ with the edge $u \rightarrow v'$ is safe with regard to the 2^{nd}, 3^{rd}, and 4^{th} safety properties. To check if it is safe to redirect a particular edge $u \rightarrow v$ to point to v', it is sufficient to check if $v' \succ v \wedge T_u(J(u)) \sqsupseteq J(v)$.

Why does the subsumption relation not cover the first safety property? That way, $v' \succ v$ would be a sufficient condition to guarantee that any edge $u \rightarrow v$ can be redirected to v'. The answer is, the subsumption relation "distributes" over congruent edges. That is to say, for any subsumption fact $a' \succ a$, for any pair of congruent edges $a \rightarrow b$ and $a' \rightarrow b'$ (that are not call or return-edges), it must be the case that $b' \succ b$. This observation is the key to computing the subsumption relation. If we change the subsumption relation to cover the first safety property, it will no longer distribute over congruent edges, and it will require more work to calculate. Furthermore, we already have a simple way to check the first safety property.

Let J be a greatest fixed-point solution for the data-flow problem $\mathcal{F}_{H^*}$. For unbalanced-right-left paths p and p', we say that p' *subsumes* p, (written $p' \succ p$) iff the following hold:

1. p and p' are congruent.

2. The close parentheses that are unmatched in p' are same as the close parentheses that are unmatched in p: suppose that p contains a return edge $x \rightarrow r$ labeled "$)_c$" and p does not contain a

matching call-edge for $x \rightarrow r$. Let $x' \rightarrow r'$ be the edge in p' that corresponds to p's edge $x \rightarrow r$. The edge $x' \rightarrow r'$ must also be labeled $)_c$.

3. For each vertex w of p and the corresponding vertex w' of p', $J(w) \sqsubseteq J(w')$.

For duplicate vertices v and v', we say that v' subsumes v (written $v' \succ v$) iff for every unbalanced-right-left path p from v, there is an unbalanced-right-left path p' from v' such that $p' \succ p$. Note that for any path p from v, if there is a path p' from v' such that $p' \succ p$, it is unique: since p' must mimic p so closely, there is no choice about which edges belong to p'.

In Figure 64 the only (vertex) subsumption facts are $E' \succ E''$ and $F' \succ F''$. Even though $E' \succ E''$, it is not possible to replace the edge $D'' \rightarrow E''$ with the edge $D'' \rightarrow E'$ because x has the data-flow value $\perp$ in the vertex D'', while E' requires that x has the data-flow value 2. (Furthermore, $D'' \rightarrow E''$ is a summary edge and should not be redirected without simultaneously redirecting a return edge.) For the same reason, it is not possible to replace $E'' \rightarrow F''$ with $E'' \rightarrow F'$.

Before we present the algorithm for computing the $\succ$-relation, we discuss a graph transformation that (heuristically) increases the probability that $v' \succ v$. This in turn makes it more likely that the Edge Redirection Algorithm will be able to replace the edge $u \rightarrow v$ with the edge $u \rightarrow v'$. The transformation adds additional return-edges to the hot-path graph. This increases the number of unbalanced-right-left paths from v', making it more likely that for each unbalanced-right-left path p from v, there's an unbalanced-right-left path p' from v' that mimics p; this increases the likelihood that $v' \succ v$. (Unfortunately, adding return-edges may also increase the number of unbalanced-right-left paths from v, making it less likely that $v' \succ v$; the following chapter contains experimental results that show that our technique for adding return-edges is beneficial for the Edge Redirection Algorithm.) Consider the following example:

Example 10.2.1 Figure 65 shows the hot-path supergraph in Figure 64 with the additional return edge $h' \rightarrow E''$. The new edge is labeled "$)_{D''}$". Notice that the new return edge does not change the set of unbalanced-left paths in the graph. This follows because the only edge labeled "$(_{D''}$" targets the vertex g'' and there is no same-level valid path from g'' to h'.

Without the new edge $h' \rightarrow E''$, it is not the case that $h' \succ h''$: for the path $p \equiv h'' \rightarrow E'' \rightarrow F''$ there is no congruent path from h' that has the same open parenthesis as p. After the edge $h' \rightarrow E''$ with label "$)_{D''}$" is added to Figure 64 (to create Figure 65), there is a path from h' that has the same open parenthesis as p: $h' \rightarrow E'' \rightarrow F''$. Thus, with the new return-edge, we have that $h' \succ h''$. This subsumption fact enables the replacement of the edge $g'' \rightarrow h''$ with the $g'' \rightarrow h'$. Note that this edge replacement causes the same-level valid path $[A' \rightarrow C' \rightarrow D'' \rightarrow g'' \rightarrow h'' \rightarrow E'' \rightarrow F'']$ to be replaced with the (almost identical) same-level valid path $[A' \rightarrow C' \rightarrow D'' \rightarrow g'' \rightarrow h' \rightarrow E'' \rightarrow F'']$ □

Our algorithm adds "unexecutable" return-edges to the hot-path supergraph. These return-edges are unexecutable in the sense that they never occur in an unbalanced-left path. These unexecutable return-edges may increase the number of subsumption facts $v' \succ v$ in the hot-path supergraph, which may, in turn, increase the number of opportunities for performing edge redirection. The unexecutable return-edges are added in such a way that if they become executable, they will not cause a change in execution behavior, nor will they violate any data-flow facts.

After creating a new return-edge $x \rightarrow r$ with label "$)_c$" and performing edge redirection, the transformed graph may require a new summary-edge $c \rightarrow r$. There are four requirements for adding a return-edge $x \rightarrow r$ with label "$)_c$" to the hot-path supergraph:

1. $T_x(J(x)) \sqsupseteq J(r)$ for any global data-flow facts (*e.g.*, data-flow facts associated with global variables), where T_x is the data-flow transfer function for x. Also $rtn_x \sqsupseteq rtn_r$, where rtn_x is the data-flow fact associated with the return value in x, and rtn_r is the data-flow fact associated with received return values in r. (For example, if x contains the statement "`return v1;`", then rtn_x might be the data-flow value $T_x(J(x))(v1)$ associated with `v1`; if c contains the call statement "`v2 = f(...);`", then rtn_r might be the data-flow value $J(r)(v2)$ associated with `v2`.)

2. $T_c(J(c)) \sqsupseteq J(r)$ for the local (*i.e.*, non-global) data-flow facts at the call vertex c, where T_c is the transfer function for c.

3. There must be no other return-edge $x \rightarrow r'$ from x that is labeled "$)_c$".

4. If c is a call vertex for an indirect call, then r must be the single return-site vertex associated with c. (Recall that indirect calls must have only one return-site vertex.)

The Vertex Subsumption Algorithm uses a function called $OkayToAddRtnEdge$ that takes as input a return-edge $x \rightarrow r$ and a label "$)_c$" and returns true iff the above requirements are satisfied for the return edge $x \rightarrow r$ labeled "$)_c$".

Figures 67 and 69 show the Vertex Subsumption Algorithm. This algorithm finds a conservative approximation of the $\succ$-relation: it may conclude that $v' \not\succ v$ when in fact $v' \succ v$.

1. Examine all pairs of duplicate vertices v_1 and v_2. If $J(v_1) \not\sqsupseteq J(v_2)$, then record $v_1 \not\succ v_2$ and put the pair $\langle v_1, v_2 \rangle$ on the worklist, W_1. Otherwise, optimistically assume that $v_1 \succ v_2$ and record this fact.

2. Back propagate each non-subsumption fact $v_1 \not\succ v_2$ across all edges (including summary and call-edges) except for return-edges.

3. Using the subsumption facts computed so far, add return-edges to the hot-path supergraphs. (Heuristically, these new edges increase the number of subsumption facts in the hot-path graph. This means subsequent stages are less likely to find non-subsumption facts.)

4. Examine all pairs of duplicate exit vertices x_1 and x_2 to determine if $x_1 \not\succ x_2$ because of the return edges from x_1 and x_2. If $x_1 \not\succ x_2$ and this is not already known, record the fact, and put the pair $\langle x_1, x_2 \rangle$ on the worklist.

5. Back propagate each non-subsumption fact $v_1 \not\succ v_2$ across all edges (including summary and return-edges) except for call-edges.

If $v_1 \not\succ v_2$, then for any congruent edges $u_1 \rightarrow v_1$ and $u_2 \rightarrow v_2$ that are not call nor return-edges, it follows immediately that $u_1 \not\succ u_2$ (see Figure 70); thus, we propagate the fact that $v_1 \not\succ v_2$ backwards by recording the fact that $u_1 \not\succ u_2$. This is also the procedure if $u_1 \rightarrow v_1$ and $u_2 \rightarrow v_2$ are return-edges that have the same label. If $u_1 \rightarrow v_1$ and $u_2 \rightarrow v_2$ are call-edges, then it does not necessarily follow that $u_1 \not\succ u_2$. This is shown in Figure 71. The fact that $v_1 \not\succ v_2$ implies that there is some unbalanced-right-left path p_2 from v_2 such that there is no unbalanced-right-left path from p_1 that does a good enough job at "mimicking" p_2 — every p_1 either does not have the same open parentheses as p_2 or has a worse data-flow facts than p_2. However, it may be the case that $[u_2 \rightarrow v_2 \| p_2]$ is not an unbalanced-right-left path. In this case, u_1 does not have to provide a path to mimic $[u_2 \rightarrow v_2 \| p_2]$ and it might be the case that $u_1 \succ u_2$. If we were to propagate the non-subsumption $v_1 \not\succ v_2$ to the non-subsumption fact $u_1 \not\succ u_2$,

map_VxV_to_SubsumesFlag is a map from vertex pairs to a boolean
W is a worklist of vertex pairs $\langle v_1, v_2 \rangle$, where $v1 \not\succ v2$

```
    Main()
        /* Find initial non-subsumption facts */
1:      Foreach vertex v1
2:          Foreach vertex v2 congruent to v1
3:              If J(v1) ⊉ J(v2)
4:                  map_VxV_to_SubsumesFlag(⟨v1, v2⟩) := false
5:                  Put(W, ⟨v1, v2⟩)
6:              Else /* Optimistically assume v1 ≻ v2: */
7:                  map_VxV_to_SubsumesFlag(⟨v1, v2⟩) := true
        /* Propagate non-subsumption facts, but not across return-edges */
        /* i.e., propagate non-subsumption facts along unbalanced-left paths */
8:      While W ≠ ∅
9:          ⟨v1, v2⟩ := Take(W)
10:         Foreach edge u1 → v1 that is not a return-edge
11:             Foreach edge u2 → v2 congruent to u1 → v1
12:                 If u1 ≠ u2 and map_VxV_to_SubsumesFlag(⟨u1, u2⟩) = true
13:                     map_VxV_to_SubsumesFlag(⟨u1, u2⟩) := false
14:                     Put(W1, ⟨u1, u2⟩)
        /* Add new return edges (see Figure 69) */
15:     AddRtnEdges()
        /* Find new non-subsumption facts at exits */
16:     FindNonSubsumptionAtExits()
        /* Propagate non-subsumption facts, but not across call-edges */
        /* i.e., propagate non-subsumption facts along unbalanced-right paths */
17:     While W ≠ ∅
18:         ⟨v1, v2⟩ := Take(W)
19:         Foreach edge u1 → v1 that is not a call-edge
20:             Foreach edge u2 → v2 congruent to u1 → v1
21:                 If u1 → v1 and u2 → v2 are return-edges with different labels
22:                     Continue
23:                 Else If u1 ≠ u2 and map_VxV_to_SubsumesFlag(⟨u1, u2⟩) = true
24:                     map_VxV_to_SubsumesFlag(⟨u1, u2⟩) := false
25:                     Put(W, ⟨u1, u2⟩)
26:     Output map_VxV_to_SubsumesFlag
End Main
```

Figure 67: Vertex Subsumption Algorithm for finding pairs $\langle v_1, v_2 \rangle$ such that $v_1 \succ v_2$.

```
    FindNonSubsumptionAtExits()
27:     Foreach exit vertex x_1
28:         Foreach exit vertex x_2 congruent to x_1
29:             Foreach return-edge x_2 → r_2
30:                 Let ")_c" be the label on x_2 → r_2
31:                 If there is no return edge from x_1 labeled ")_c" Then
32:                     map_VxV_to_SubsumesFlag(⟨x_1, x_2⟩) := false
33:                     Put(W, ⟨x_1, x_2⟩)
34:                 Else
35:                     Let x_1 → r_1 be the return-edge labeled ")_c"
36:                     If map_VxV_to_SubsumesFlag(⟨r_1, r_2⟩) = false Then
37:                         map_VxV_to_SubsumesFlag(⟨x_1, x_2⟩) := false
38:                         Put(W, ⟨x_1, x_2⟩)
End FindNonSubsumptionAtExits
```

Figure 68: The function *FindNonSubsumptionAtExits* used by the Vertex Subsumption Algorithm

```
/* The following global variable is defined in Figure 67 */
map_VxV_to_SubsumesFlag is a map from vertex pairs to a boolean

    AddRtnEdges()
39:     Foreach procedure P
40:         Foreach call vertex c that calls P
41:             Foreach exit vertex x of P
42:                 If there exits a return edge x → r labeled ")_c" Then
43:                     Continue
44:                 BestRtnVtx := 0
45:                 Foreach return-site vertex r congruent to a return-site vertex for c
46:                     If OkayToAddRtnEdge(x, r, ")_c") /* see page 150 */ Then
47:                         If (BestRtnVtx = 0 or
                               map_VxV_to_SubsumesFlag(⟨r, BestRtnVtx⟩) = true) Then
48:                             BestRtnVtx := r
49:                 add the edge x → BestRtnVtx to the graph; label it ")_c"
End AddRtnEdges
```

Figure 69: The function *AddRtnEdges* used by the Vertex Subsumption Algorithm in Figure 67

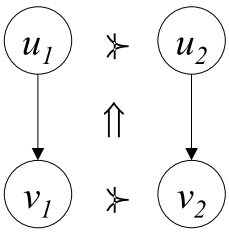


Figure 70: Example of "distribution" of non-subsumption facts across congruent edges. The fact that $v_1 \not\succ v_2$ implies that $u_1 \not\succ u_2$.

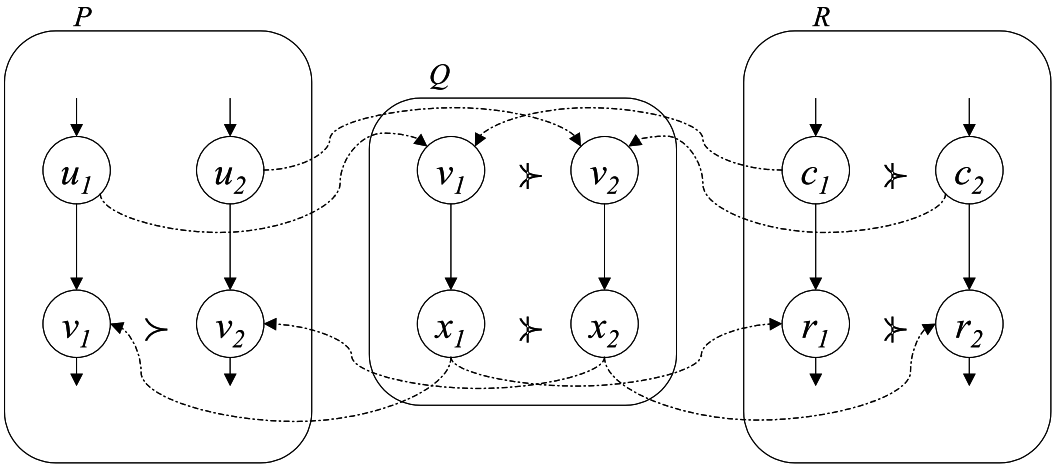


Figure 71: Example showing why non-subsumption facts might not distribute over call-edges. The non-distribution fact $v_1 \not\succ v_2$ is due to the fact that $r_1 \not\succ r_2$. However, the non-subsumption fact $r_1 \not\succ r_2$ in procedure R should not be propagated to the call vertices u_1 and u_2 in procedure P.

$\mathrm{map\_VxV\_to\_SubsumesFlag}$ is a map from vertex pairs to a boolean

Main()
/ The following also adds return-edges to the graph. */*
1: Run the Vertex Subsumption Algorithm to compute $\mathrm{map\_VxV\_to\_SubsumesFlag}$.
2: **Foreach** edge $u \rightarrow v$
3: $\mathrm{BestNewTgt} := v$
4: **Foreach** v' congruent to v
5: **If** $T_u(J(u)) \sqsupseteq J(v')$ **and** $\mathrm{map\_VxV\_to\_SubsumesFlag}(\langle v', \mathrm{BestNewTgt}\rangle) = \mathit{true}$
6: $\mathrm{BestNewTgt} := v'$
7: replace the edge $u \rightarrow v$ with $u \rightarrow \mathrm{BestNewTgt}$
End Main

Figure 72: Edge Redirection Algorithm.

we could be propagating a non-subsumption fact over a path p_2 that is not unbalanced-right-left (*i.e.*, meaning that p_2 is not a feasible execution path).

To avoid this problem, we use the same approach used in interprocedural slicing: propagation of information is performed in two phases. The first phase (stage 2 above) propagates non-subsumption facts (*e.g.*, $v_1 \not\succ v_2$) only when it knows there is an unbalanced-left path to justify the propagation. It does not propagate non-subsumption facts across return edges, and thus avoids mistakenly propagating non-subsumption facts to pairs of call vertices from inappropriate pairs of return-site vertices.

The second phase (stage 5 above) back propagates non-subsumption facts only when it knows there is an unbalanced-right path to justify the propagation. It again avoids propagating non-subsumption facts from pairs of return-site vertices to the wrong pairs of call vertices, but this time it does so by not propagating facts over call-edges. At the end of the algorithm, non-subsumption facts have been propagated across all congruent, unbalanced-right-left paths.

10.3 Determining When Edge Redirection is Profitable

Determining when edge redirection is profitable is a much harder question than determining when it is possible. We would like to use edge redirection to help reduce the size of the hot-path graph. However, just because an edge replacement is possible does not mean that it will make the hot-path graph easier to reduce. In this section, we introduce a heuristic approach that uses edge redirection in an attempt to minimize the number of vertices with incoming edges. This may increase the number of vertices with no incoming edges, which can be removed from the graph, and it may transform the hot-path supergraph in a way that aids the performance of the Ammons-Larus Reduction Algorithm.

Figure 72 shows the Edge Redirection Algorithm. For each edge $u \rightarrow v$, the algorithm performs a greedy search for the vertex v' that is congruent to v, is compatible with u's data-flow facts, and subsumes the greatest number of vertices (that are congruent to v). The greedy search assumes that it has already found the vertex *BestNewTgt* that satisfies these criteria and only updates its guess when it encounters a vertex v' that subsumes *BestNewTgt*. Hence, it may find a vertex that is a local maximum

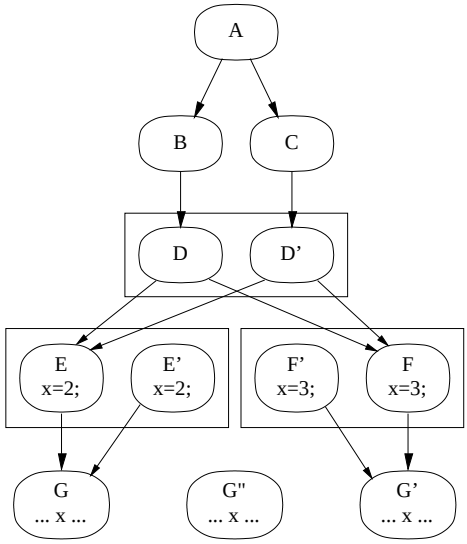


Figure 73: The hot-path graph of Figure 55 after the Edge Redirection Algorithm is run (see Figure 72).

in the subsumption relationship, but not a global maximum — there may be a vertex that satisfies the other criteria and subsumes more vertices than *BestNewTgt*.

Example 10.3.1 Let us consider the effects of this algorithm on the example hot-path graph in Figure 55(b). When the algorithm considers the edge $D' \rightarrow E'$, it will decide that it is safe to redirect the edge to E, and profitable to do so, since $E \succ E'$; the algorithm will replace the edge $D' \rightarrow E'$ with $D' \rightarrow E$. The edges $D \rightarrow F'$, $E' \rightarrow G''$, and $F' \rightarrow G''$ are replaced with the edges $D \rightarrow F$, $E' \rightarrow G$, and $F' \rightarrow G'$, respectively. Notice, when the algorithm considers the edge $E' \rightarrow G''$, it will not replace this edge with the edge $E' \rightarrow G'$, since this would violate the data-flow facts at G': x has the value 2 at E' and the value 3 at G'.

Figure 73 shows the hot-path graph from Figure 55(b) after edge redirection is performed. This graph is more amenable to the Coarsest Partitioning Algorithm than the original graph: the blocks $\{D, D'\}$, $\{E, E'\}$, and $\{F, F'\}$ are no longer split by the Coarsest Partitioning Algorithm. Also, the vertices E', F', and G'' have become unreachable and may be dropped from the graph. (In practice, one should drop unreachable vertices before performing the Coarsest Partitioning Algorithm.) □

After edge redirection is performed, the hot-path supergraph may be corrupted in three ways: (1) the summary-edges may be incorrect; (2) there may be unnecessary return-edges in the graph; and (3) there may be unreachable vertices in the graph. To fix these problems, we perform the clean-up pass shown in Figure 74. This algorithm starts by removing all of the summary-edges. Then the algorithm uses reachability over intraprocedural and summary edges to find same-level valid paths. As it finds new same-level valid paths, it adds summary-edges back to the graph. These new summary-edges are then used to find more same-level valid paths. This process is repeated until all of the summary-edges have been added to the graph. Then the clean-up algorithm uses the same-level valid paths computed in the previous stage to decide which return-edges and vertices it needs to keep. The algorithm can be made to run in $O(\mathrm{MaxEntries} \cdot \|E\| + \|\mathrm{RtnEdges}\| \cdot \mathrm{MaxEntries} \cdot \log(\mathrm{MaxCallEdges}))$, where $\mathrm{MaxEntries}$ is the maximum number of entries for a procedure, $\|E\|$ is the number of intraprocedural edges and summary-edges in the output graph, RtnEdges is the set of all return-edges, and MaxCallEdges is the

W is a worklist of vertices.
$ReachingEntries$ is a map from each vertex to a set of entry vertices

Main()
    Remove all summary-edges.
    / For each vertex v in procedure P, compute the set of P's entries that reach v */*
    / Initialize worklist */*
    Foreach vertex v
        $Put(W, v)$
    While $W \neq \emptyset$
        $v := Take(W)$
        OldReachingEntries := $ReachingEntries(v)$
        If v is an entry vertex **Then**
            $ReachingEntries(v) = \{v\}$
        Else
            Foreach intraprocedural or summary-edge $u \rightarrow v$
                $\mathrm{ReachingEntries}(v) := \mathrm{ReachingEntries}(v) \cup ReachingEntries(u)$

        If $\mathrm{ReachingEntries}(v) \neq$ OldReachingEntries **Then**
            / Add successors to worklist */*
            Foreach intraprocedural or summary-edge $v \rightarrow w$
                $Put(W, w)$
            / Update summary-edges, if needed */*
                If v is an exit vertex
                    Foreach $e \in \mathrm{ReachingEntries}(v)$
                        Foreach call-edge $c \rightarrow e$ labeled "$(_c$"
                            Foreach return-edge $v \rightarrow r$ labeled "$)c$"
                                add the summary-edge $e \rightarrow r$
                                $Put(W, r)$

    / Remove dead return-edges */*
    Foreach return-edge $x \rightarrow r$
        let "$)_c$" be the label on $x \rightarrow r$
        If there is no call-edge $c \rightarrow e$ labeled "$(_c$" s.t. $e \in \mathrm{ReachingEntries}(x)$ **Then**
            remove the return-edge $x \rightarrow r$
    / Remove unreachable vertices from the graph */*
    Foreach vertex v
        If $\mathrm{ReachingEntries}(v) = \emptyset$
            remove the vertex v
End Main

Figure 74: Clean-up algorithm that repairs the hot-path supergraph after the Edge Redirection Algorithm has been run.

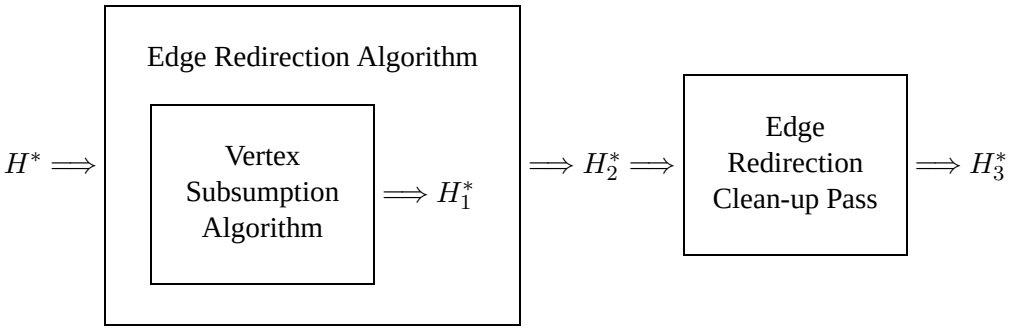


Figure 75: Stages used for minimizing a graph using edge redirection. The Edge Redirection Algorithm calls the Vertex Subsumption Algorithm as a subroutine, so the graph H_1^* is a temporary data structure used by the Edge Redirection Algorithm.

maximum number of call-edges into an entry vertex. We expect this to be cheaper than the Vertex Subsumption Algorithm.

10.4 Proof of Correctness

In this section, we present proofs that the Vertex Subsumption Algorithm and the Edge Redirection Algorithm work correctly. Throughout this section, we will use the following nomenclature:

- H^* refers to the hot-path graph that is input to the Vertex Subsumption Algorithm or the Edge Redirection Algorithm.
- J is the data-flow solution for the problem $\mathcal{F}_{H^*}$.
- H_1^* refers to the graph output by the Vertex Subsumption Algorithm. Every vertex v from H^* is renamed v_1 in H_1^*. Thus, v_1 is the same vertex as v, but renamed for the graph H^*.
- H_2^* refers to the graph output by the Edge Redirection Algorithm. Every vertex v from H^* is renamed v_2 in H_2^*.
- H_3^* refers to the graph output by the Edge Redirection Clean-up Pass. Every vertex v from H^* is renamed v_3 in H_3^*.

Figure 75 shows the relationships between the various graphs.

Before we move on to presenting lemmas and theorems, we have some observations about the subsumption relation.

Observation 10.4.1 If $v' \not\succ v$, then there must be some shortest unbalanced-right-left path p from v such that there is no unbalanced-right-left path p' from v' that subsumes p. If p is a 0-length path (*i.e.*, it is the empty-path from v to v'), then there is a congruent path p' that mimics p's control flow, namely the empty-path from v' to v'. So it must be that $J(v') \not\supseteq J(v)$. If p is of length 1 or greater, then let $p = [q \| w \rightarrow y]$. There are three possibilities:

1. There is a path q' from v' to w' such that $q' \succ q$, but there is no edge from w' that is congruent to $w \rightarrow y$.

2. There is a path q' from v' to w' such that $q' \succ q$, but $w \rightarrow y$ is a return-edge that is unmatched in p and there is no return-edge from w' with the same label as $w \rightarrow y$.

3. There is a path p' from v' to y' such that p' mimics p's control flow, but the data-flow facts at y' are not as good as the data-flow facts at y ($J(y') \not\sqsupseteq J(y)$).

□

Observation 10.4.2 The $\succ$-relation is transitive: for any paths p'', p', and p, if $p'' \succ p'$ and $p' \succ p$ then $p'' \succ p$. Similarly, for any vertices v'', v', and v, if $v'' \succ v'$ and $v' \succ v$ then $v'' \succ v$. □

Observation 10.4.3 The $\succ$-relation is reflexive: for any path p, $p \succ p$. Similarly, for any vertex v, $v \succ v$. □

Observation 10.4.4 Let $[p\|q]$ be an unbalanced-right-left path. Let p' and q' be paths such that $p' \succ p$ and $q' \succ q$. If $[p'\|q']$ is an unbalanced-right-left path, then $[p'\|q'] \succ [p\|q]$. □

All of these observations follow directly from the definition of subsumption.

We now turn to proving the Vertex Subsumption Algorithm correct. The proof is complicated by the fact that the algorithm does not compute the $\succ$-relation for the input graph: it transforms the graph (by adding return-edges) and it computes the $\succ$-relation for the transformed graph.

Theorem 10.4.5 *Let H^*be the hot-path graph input to the Vertex Subsumption Algorithm and let H_1^* be the transformed graph output by the Vertex Subsumption Algorithm. That is, H_1^* is $H^*$with the additional return-edges added by lines 39–49 of Figure 69. Every vertex subsumption assertion $v_1' \succ v_1$ output by the Vertex Subsumption Algorithm is correct for the graph H_1^*. However, the Vertex Subsumption Algorithm may conclude that $v_1' \not\succ v_1$ when in fact $v_1' \succ v_1$.*

Proof: See Appendix D. □

It is possible to compute the $\succ$-relation accurately, but the process is much more expensive than the Vertex Subsumption Algorithm both in space and time. In fact, it requires $O(\|\mathrm{MaxDuplicates}\|^4\|V_0\|^2)$ space and $O(\|\mathrm{MaxDuplicates}\|^4\|V_0\|^2\|E\|)$ time, where MaxDuplicates is the maximum number of duplicate vertices in the hot-path graph, V_0 is the set of vertices in the original supergraph, and E is the set of edges in the hot-path graph. This algorithm is described in Appendix E. Right now, we turn to the proof to the Vertex Subsumption Algorithm is correct.

Theorem 10.4.6 *If J approximates ($\sqsubseteq$) the greatest fixed-point solution for $\mathcal{F}_H^*$, then J also approximates the greatest fixed-point solution for $\mathcal{F}_{H_3}$.*

Proof: See Appendix D. □

To prove the correctness of the Edge Redirection Algorithm, we must first prove that the Edge Redirection Algorithm preserves the $\succ$-relation in the following sense: let v' and v be vertices in the input graph and let v_o' and v_o be the same vertices in the output graph (recall that the Edge Redirection Algorithm does not add vertices to the input graph). If $v' \succ v$, then $v_o' \succ v$. In other words, if $v' \succ v$, then for every path p_i from v in the *input* graph, there is a path p_o' from v_o' in the *output* graph such that $p_o' \succ p_i$; the paths from v_o' in the output graph are "better" than the paths from v in the input graph. Note that $v' \succ v$ does not imply that $v_o' \succ v_o$. We have the following lemma:

Lemma 10.4.7 *Let H^* be the hot-path graph input to Edge Redirection Algorithm. Let H_1^* be the graph that results from running the Vertex Subsumption Algorithm on H^*. (H_1^* is H^* with extra return-edges.) Let H_2^* be the graph output by the Edge Redirection Algorithm. Rename each vertex v in H^* as v_1 in H_1^* and as v_2 in H_2^*. Let v_1' and v_1 be vertices in H_1^* and let v_2' be the same vertex as v_1' but in graph H_2^*. If $v_1' \succ v_1$, then $v_2' \succ v_1$. In other words, if $v_1' \succ v_1$, then for every unbalanced-right-left path p_1 from v_1 in graph H_1^*, there must be a unbalanced-right-left path p_2' from v_2' in graph H_2^* such that $p_2' \succ p_1$.*

Proof: See Appendix D. □

Theorem 10.4.8 *Let H_3^* be the graph the results from running the Edge Redirection Algorithm (see Figure 72) and the clean-up pass in Figure 74 on the graph H^*. Then, H^* and H_3^* are unbalanced-left path congruent.*

Proof: See Appendix D. □

Theorem 10.4.9 *Let J_3 be the same data-flow solution as J, but renamed for the graph H_3^*. (By Theorem 10.4.6, J_3 is a valid data-flow solution for H_3^*.) The data-flow solution J_3 preserves the valuable data-flow facts of J.*

Proof: This follows from Lemma 10.4.7. For any unbalanced-left path p from $Entry_{global}$ in H^*, there must be a path p_3 in H_3^* such that $p_3 \succ p$. This means that p_3 has equal or better ($\sqsubseteq$) data-flow facts than p. *QED* □

10.5 Analysis of Runtime

We now turn to the problem of computing the complexity of the Vertex Subsumption and Edge Redirection Algorithms. We start with the Vertex Subsumption Algorithm. Lines 1–7 of Figure 67 find initial non-subsumption facts. To do this, every pair of duplicate vertices must be examined. This takes $O(\text{MaxDuplicates}^2 \| V_0 \| d)$ time, where MaxDuplicates in the maximum number of duplicate vertices in the hot-path graph, V_0 is the set of vertices in the original supergraph (that was traced to generate the hot-path graph), and d is the cost of comparing the data-flow facts for two vertices. Lines 8–14 and lines 17–25 propagate non-subsumption facts. This takes $O(\text{MaxDuplicates}^2 * \| E^+ \|)$ time, where E^+ is the set of edges of the input hot-path graph augmented with the return-edges added by the function call to $AddRtnEdges$. In the worse case, the call to $AddRtnEdges$ (line 15) takes $O(\text{CallSites} \cdot \text{MaxDuplicates}^2 \cdot d)$ time, where CallSites is the number of call-sites in the hot-path graph. The call to $FindNonSubsumptionAtExits$ requires $O(MaxRtnEdges^2)$ time, where MaxRtnEdges is the size of the largest set $S_x \equiv_{def} \{x' \rightarrow r' : x' \text{ is a duplicate of } x \wedge x' \rightarrow r' \text{ is a return} - \text{edge}\}$. All told, the Vertex Subsumption Algorithm requires

$$O(\text{MaxDuplicates}^2 \| V_0 \| d + \text{MaxDuplicates}^2 * \| E^+ \| + \text{CallSites} \cdot \text{MaxDuplicates}^2 \cdot d + MaxRtnEdges^2)$$

time.

The Edge Redirection Algorithm (not counting the call to the Vertex Subsumption Algorithm) requires $\| E^+ \| \cdot \text{MaxDuplicates})$ time: for each edge $u \rightarrow v$, for each duplicate v' of v, the algorithm

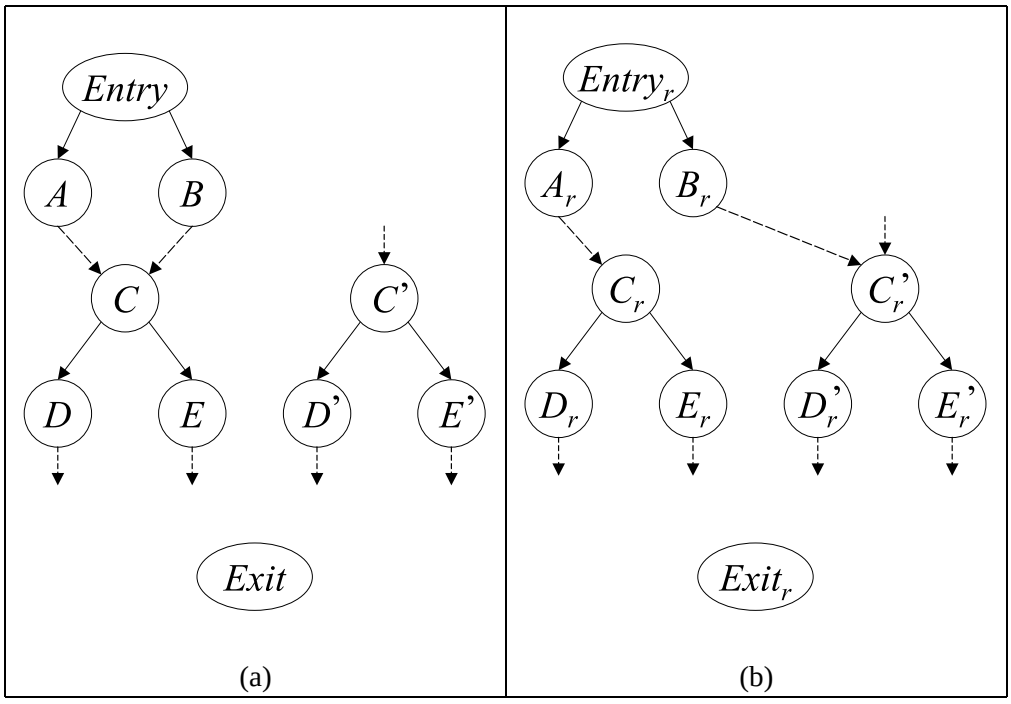


Figure 76: Example showing that translating a path profile after edge redirection is impossible. (Dotted edges indicate recording edges.)

considers redirecting the edge $u \rightarrow v$ to point at v'. This is absorbed by the time of the Vertex Subsumption Algorithm. We previously stated that the run-time of the Edge Redirection Clean-up Pass is

$$O(\mathrm{MaxEntries} \cdot \|E\| + \|\mathrm{RtnEdges}\| \cdot \mathrm{MaxEntries})$$

where $\mathrm{MaxEntries}$ is the maximum number of entries for a procedure, $\|E\|$ is the number of intraprocedural edges and summary-edges in the output graph, and RtnEdges is the set of all return-edges.

10.6 Updating a Path Profile After Edge Redirection

The previous section described how to use the Edge Redirection Algorithm to reduce a hot-path graph H^*. Let H'^* be the reduced hot-path graph that results from the process described in the last section. In this section, we discuss the problem of translating a path profile *pp* for the graph H^* to a path profile pp' for the graph H'^*. This is useful if one wishes to follow the reduction of the hot-path supergraph with profile-based optimizations (other than the express-lane transformation). Unfortunately, this cannot be done, which restricts one to doing all profile-based optimizations before reducing the hot-path supergraph. The difficulty arises when attempting to translate a path $p \in pp$ that contains a surrogate-edge $u \rightarrow v$, as the following example demonstrates.

Example 10.6.1 Figure 76(a) shows part of a simple hot-path graph. There is a surrogate-edge $Entry \rightarrow C$ that is not shown in the graph. The edges $A \rightarrow C$ and $B \rightarrow C$ are recording edges. (Recall that the path-profiling instrumentation on a recording edge $u \rightarrow v$ increments the count of the path

currently executing and begins recording a new path with the surrogate-edge $Entry \rightarrow C$.) Suppose we have the following path profile *pp* for the hot-path graph in Figure 76(a):

Path	Execution Count
$Entry \rightarrow A \rightarrow C$	2
$Entry \rightarrow B \rightarrow C$	2
$Entry \bullet C \rightarrow D \ldots$	2
$Entry \bullet C \rightarrow E \ldots$	2

Figure 76(b) shows the graph in Figure 76(a) after the Edge Redirection Algorithm has been run; the edge $B \rightarrow C$ has been replaced with the edge $B_r \rightarrow C'_r$ and each vertex v has been renamed as v_r. We would like to translate the path profile *pp* onto the graph in Figure 76(b). Unfortunately, there are many possible translations. The path $[Entry \bullet C \rightarrow D \ldots]$ could be translated as $[Entry_r \bullet C_r \rightarrow D_r \ldots]$ or as $[Entry_r \bullet C'_r \rightarrow D'r \ldots]$. Similarly, the path $[Entry \bullet C \rightarrow E \ldots]$ could be translated as $[Entry_r \bullet C_r \rightarrow E_r \ldots]$ or as $[Entry_r \bullet C'_r \rightarrow E'_r \ldots]$. The difficulty in translating the surrogate-edge $Entry \rightarrow C$ is that the profiling machinery may start a path with the surrogate-edge $Entry \rightarrow C$ after the path $Entry \rightarrow A \rightarrow C$ or after the path $Entry \rightarrow B \rightarrow C$ has executed.

Thus, the surrogate-edge $Entry \rightarrow C$ could be standing in for any same-level valid path from $Entry$ to C. One of the same-level valid paths from $Entry$ to C is translated to Figure 76(b) as a same-level valid path from $Entry_r$ to C_r. The other same-level valid path from $Entry$ to C is translated to Figure 76(b) as a same-level valid path from $Entry_r$ to C'_r. This makes it impossible to translate the surrogate-edge $entry \rightarrow C$ to Figure 76(b). □

The problem in the above example can be avoided if we prevent the Edge Redirection Algorithm from replacing the edge $B \rightarrow C$ with the edge $B \rightarrow C'$. If this is done, all of the same-level valid paths from $Entry$ to C are translated to same-level valid paths from $Entry_r$ to C_r in Figure 76(b). In this case, the surrogate-edge $Entry \rightarrow C$ is always translated to the surrogate-edge $Entry_r \rightarrow C_r$.

The following condition is sufficient for guaranteeing that a path profile can be translated to a profile for the graph created by the Edge Redirection Algorithm:

> Let H^* be a hot-path supergraph, let *pp* be a path profile on H^*, and let H^*_r be the reduced hot-path graph output by the Edge Redirection Graph. For every vertex v that is the target of a surrogate-edge in some path of *pp*, for any unbalanced-right-left path p in H^* that ends at v, the congruent path p_r in H^*_r ends at v_r. (Here, v_r is the new name of the vertex v in the graph H^*_r.)

If this condition is true, then it is straightforward to translate each path q in *pp*: we simply traverse the edges of q and simultaneously trace out the translated path q_r in H^*_r. When a surrogate-edge $u \rightarrow v$ is traversed, we know that the next vertex of q_r must be v_r.

A minor change is needed to make sure that the above condition holds after the Edge Redirection Algorithm is run. A new requirement is added to the definition of path subsumption: a path p' subsumes a path p iff

1. p' exactly mimics p's control flow;
2. p' data-flow facts are equal or better than p's data-flow facts; and
3. for every vertex v of p, v is not the target of a surrogate edge in any path of the path profile *pp*.

To implement this change, line 3 of the Vertex Subsumption Algorithm is changed to:

If $J(v_1) \not\sqsupseteq J(v_2)$ **or** v_2 is a target of a surrogate-edge that occurs in path profile *pp* **Then**

This change will mean that the Edge Redirection Algorithm will be able to make fewer edge replacements. The next chapter contains experimental measurements of the effect of this change. The other alternative is to not translate the path profile into a profile for the graph created by the Edge Redirection Algorithm.

10.7 Alternating Between Graph Reduction Strategies

The Edge Redirection Algorithm and the Supergraph Partitioning Algorithm complement one another. It is possible to use these algorithms as alternating passes in a reduction strategy for the hot-path supergraph. (In fact, the Edge Redirection Algorithm is too expensive if the hot-path supergraph has not already been partially reduced by the Supergraph Partitioning Algorithm.) The Edge Redirection Algorithm can make a hot-path supergraph more amenable to reduction by the Supergraph Partitioning Algorithm.

However, care must be taken when alternating between the algorithms. As shown in Figure 66, the Edge Redirection Algorithm requires that the input data-flow solution J approximates ($\sqsubseteq$) the greatest-fixed point solution (for $\mathcal{F}^*_H$) on the input hot-path supergraph H^*. In general, the Supergraph Partitioning Algorithm does not guarantee that the output data-flow solution J' approximates the greatest-fixed point solution (for $\mathcal{F}_{H^*_r}$) on the output supergraph H^*_r. If we wish to feed the output of the Supergraph Partitioning Algorithm as input to the Edge Redirection Algorithm, we must take care that the Supergraph Partitioning Algorithm outputs a data-flow solution that approximates the greatest-fixed point solution for $\mathcal{F}_{H^*_r}$. To do this, we require that the Supergraph Partitioning Algorithm treat every data-flow fact as desirable; this way the Supergraph Partitioning Algorithm will preserve every data-flow fact. (The Supergraph Partitioning Algorithm may still destroy data-flow facts that are "dead", *e.g.*, the fact that $x = 2$ in a vertex where the variable x is dead; since these data-flow facts are dead, destroying them does not matter.)

Chapter 11

Reducing the Hot-path Graph is NP-hard

In this chapter, we give some complexity results for the problem of reducing the hot-path graph. The principal result states that given a hot-path graph H, a distributive data-flow framework $\mathcal{F}$, and a data-flow solution J for $\mathcal{F}_H$, it is NP-hard to find a minimal reduced hot-path graph while simultaneously preserving the valuable data-flow facts of J. It is unknown if this problem in NP-complete (*i.e.*, it is unknown if a reduced graph that preserves the valuable data-flow facts of J can be verified to be the minimal such reduced graph in polynomial time). We show that when $\mathcal{F}$ is a distributive framework, it is computable (via a Turing Machine) to find a minimal reduced hot-path graph while simultaneously preserving the valuable data-flow facts of J. (In contrast, we conjecture that if $\mathcal{F}$ is not distributive, the general problem of finding a minimal reduced hot-path graph while preserving the data-flow facts of J is undecidable.)

The first theorem states our NP-hardness result:

Theorem 11.0.1 *Given a hot-path graph H, a distributive data-flow framework $\mathcal{F}$, and the meet-over-all-paths solution J to $\mathcal{F}_H$, it is NP-hard to generate a reduced hot-path graph H' such that H' has a minimal size and the meet-over-all-paths solution J' to $\mathcal{F}_{H'}$ preserves the valuable data-flow facts of J.*

The proof gives a reduction from the problem of finding a minimal graph coloring to the problem of finding a minimal H' such that J' preserves the valuable data-flow facts of J. A *coloring* of an undirected graph is a partitioning of the graph's vertices such that no two adjacent vertices appear in the same partition (*i.e.*, have the same color). A *minimal coloring* is a coloring with a minimal number of partitions. The proof uses copy constant propagation (see [3]) as an example of a distributive data-flow problem $\mathcal{P}$, but generalizes easily for any distributive data-flow problem. Also, the measure of the size of the reduced hot-path graph H' is taken to be the number of vertices in H'. Again, it is easy to modify the proof to work with any reasonable measure of size, *e.g.*, number of edges or number of instructions.
Proof: Let G be an undirected graph for a minimal graph coloring problem. We construct a program P and a path profile that result in a hot-path graph H. We give the data-flow solution J for constant propagation on H. We show how to recover a minimal coloring of G from a reduced hot-path graph H' and a constant-propagation solution J' where:

1. J' is the meet-over-all-paths solution for copy constant propagation on H';
2. J' preserves the valuable data-flow facts of J;
3. and there is no reduced hot-path graph H' with fewer vertices that satisfies the previous properties.

It follows that finding H' is NP-hard.

Let $v_1, v_2, \ldots v_n$ denote the vertices of the undirected graph G. We create a program P with the structure shown in Figure 77. P has n variables `x1, x2, ... xn`, one for each vertex v_i of G. The code contains two switch statements. In the first switch statement, the i^{th} case of the switch encodes the adjacency information for v_i in the following manner: the case statement contains the assignment

```
void main() {
    int i, x1, x2, . . . xn
    i = 0;
    while( i < n ) {
        switch( i ) {
            case 1:
                x1 = 1;
                x2 = ⊥;
                . . .
                xn = 1;
                break;
            . . .
            case n:
                x1 = 1;
                x2 = ⊥;
                . . .
                xn = 1;
                break;
            default:
                x1 = x2 = . . . = xn = ⊥;
        }
        switch( i ) {
            case 1:
                print(x1);
                break;
            case 2:
                print(x2);
                break;
            . . .
            case n:
                print(xn);
                break;
            default:
        }
    ++i;
    }
}
```

Figure 77: Example program that that results in a hot-path graph that encodes a graph coloring problem.

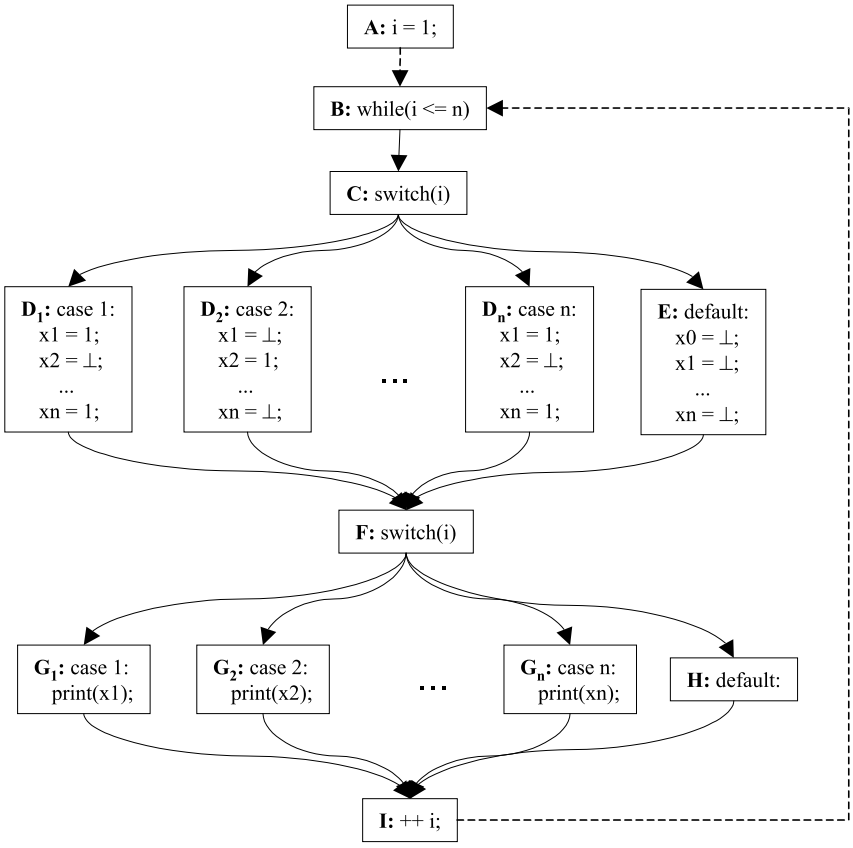


Figure 78: Schematic control-flow graph C for the program in Figure 77. The recording edges $A \to B$ and $I \to B$ are shown with dashed lines.

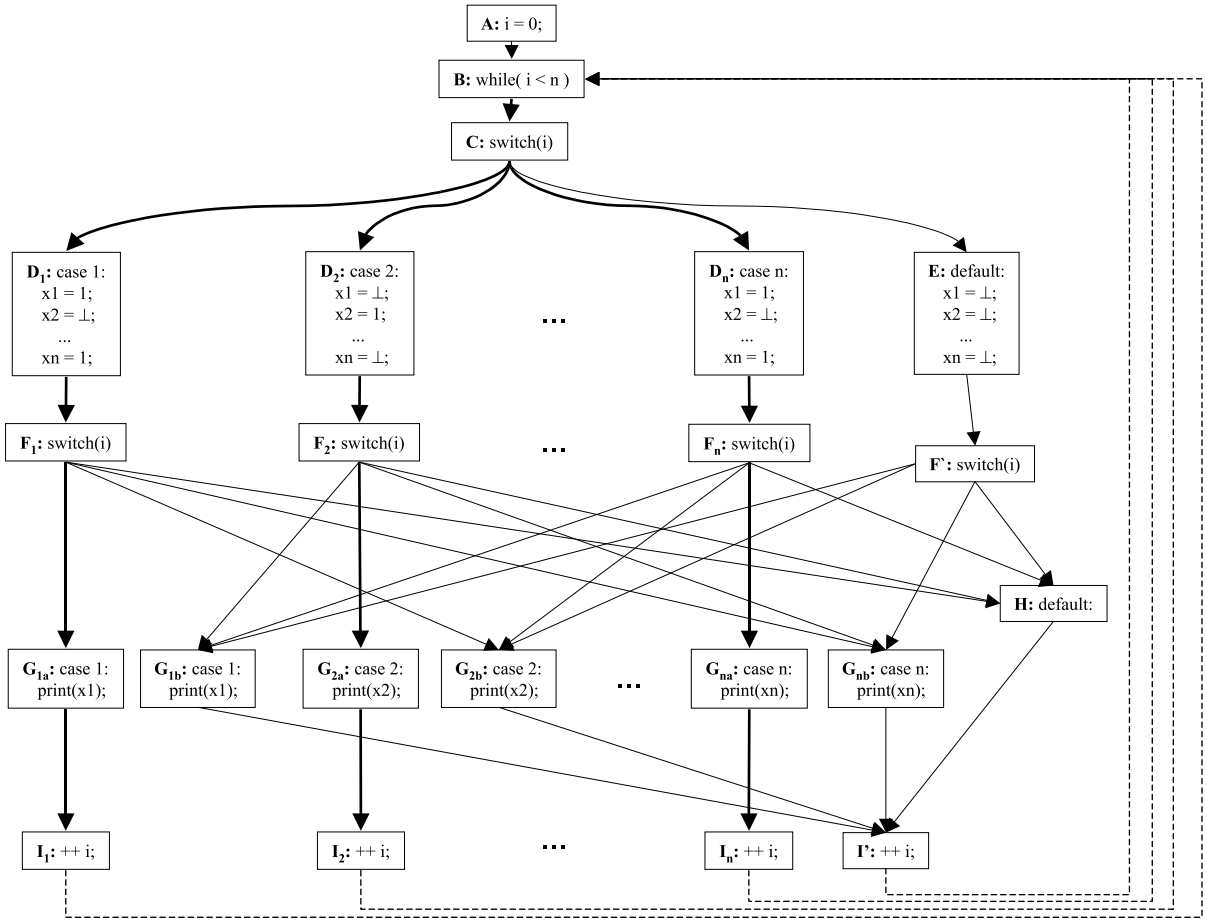


Figure 79: The hot-path graph H for the control-flow graph C shown in Figure 78. The vertices have been renamed (from the results of the hot-path tracing algorithm in Section 2.2.2) to save space. Edges that are on a hot path appear in bold.

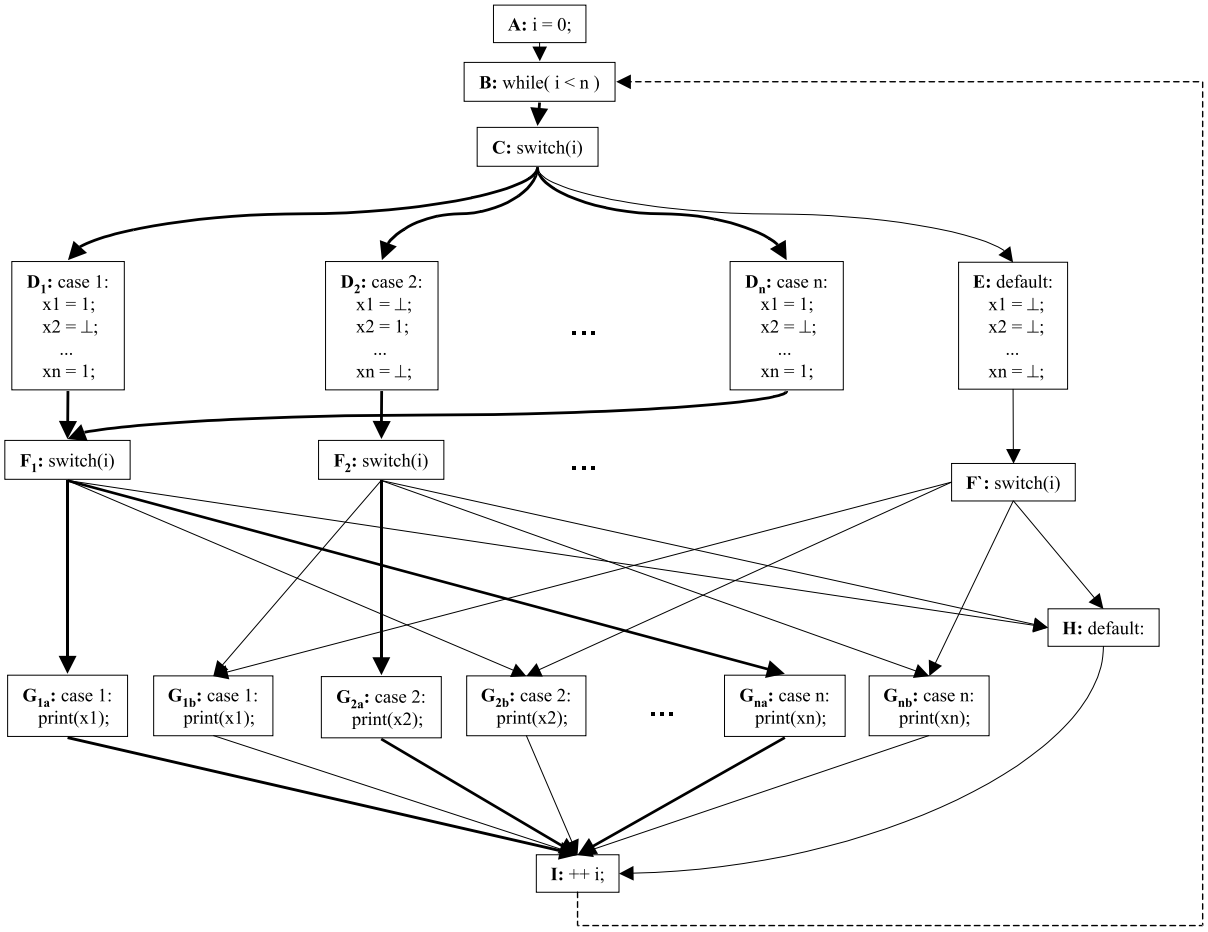


Figure 80: Reduced hot-path graph H' for the hot-path graph H in Figure 79.

`xi=1`; for $j \neq i$, if v_j is not adjacent to v_i, then the case statement contains an assignment `xj=1`; otherwise, for $j \neq i$ and v_j adjacent to v_i, the case statement contains a statement that causes constant propagation to assign the value $\perp$ to `xj`. The default case statement causes all of the variables `x1, x2, ..., xn` to go to bottom. In the second switch statement, the i^{th} case statement contains a use of `xi`.

Figure 78 shows a control-flow graph C for the program in Figure 77. If the edges $A \rightarrow B$ and $I \rightarrow B$ are chosen as recording edges, then the path profile for P has the following hot paths:

- $B \rightarrow C \rightarrow D_1 \rightarrow F \rightarrow G_1 \rightarrow I$
- $B \rightarrow C \rightarrow D_2 \rightarrow F \rightarrow G_2 \rightarrow I$

$$\vdots$$

- $B \rightarrow C \rightarrow D_n \rightarrow F \rightarrow G_n \rightarrow I$

This path profile and the control-flow graph C in Figure 78 give the hot-path graph H shown in Figure 79 (see Section 2.2.1).

The meet-over-all-paths solution J for copy constant propagation on H shows that in the vertex G_{1a}, the variable x_1 has the constant value 1; in the vertex G_{2a}, the variable x_2 has the constant value 1; and so on. These are the hot data-flow facts that must be preserved when reducing the hot-path graph. In contrast to these "good" data-flow facts, copy constant propagation shows the following: in the vertex G_{1b}, the variable x_1 has the value $\perp$ (non-constant); in the vertex G_{2b}, the variable x_2 has the value $\perp$, and so on.

Let H' be a reduced hot-path graph for H such that H' has a minimal number of vertices and the greatest fixed-point solution J' to constant propagation on H' preserves the data-flow facts of J. Figure 80 depicts H'. Only the vertices F, $G_1 \ldots G_n$, and I from the original control flow graph C (Figure 78) are duplicated multiple times in H. Only the vertices in H that are copies of these vertices (*i.e.*, $F_1 \ldots F_n, F', G_{1a} \ldots G_{na}, G_{1b} \ldots G_{nb}, \text{and } I_1 \ldots I_n$) can be removed from H to create H'. Removing any other vertices results in a graph that is not path equivalent to H.

H' needs only one copy of the vertex I. For every vertex I_j in H, only the variable i is live in I_j and the solution for constant propagation on H states that i is mapped to $\perp$ (*i.e.*, $J(I_j) = \perp$). This implies that there are no paths in H that distinguish $I_1 \ldots I_n$ with respect to the important data-flow facts in J. Therefore, H' will have only one copy of I.

On the other hand, H' must have all of the vertices $G_{1a} \ldots G_{1n}$ and $G_{2a} \ldots G_{2n}$. Consider the duplicate vertices G_{1a} and G_{1b}. The data-flow solution states that at the use of x_1 in G_{1a}, the variable x_1 has the constant value 1. (This is a desirable data-flow fact must be preserved.) However, at the use of x_1 in G_{1b}, x_1 has the non-constant value $\perp$. Furthermore, because of the path $E \rightarrow F' \rightarrow G_{1b}$, there is no way to restructure H to alter this fact. This means that the vertices G_{1a} and G_{1b} must be distinguished if we wish to preserve the desirable data-flow fact in G_{1a}. An identical situation holds for every pair of (duplicate) vertices G_{ia} and G_{ib}.

Vertices in the set $\{F_1, F_2 \ldots F_n\}$ may be coalesced under certain circumstances (although F' may not). Every variable in a vertex F_i is live. However, only the constant value in the variable x_i reaches a use in a hot vertex, namely G_{ia}. Thus, only the data-flow value of x_i must be preserved in F_i. This means that F_i may be combined with a vertex F_j iff they both have the same constant values for x_i and x_j. We say that F_i and F_j are *collapsible* if this holds. F' is not collapsible with any vertex F_i, since it has a data-flow value of $\perp$ for every variable. In order to combine two collapsible vertices F_i and F_j of H into one vertex F_i of H', the following steps are taken:

1. Change the graph so that F_i and F_j have the same successors (by using the edge redirection

technique described above). This may be done while preserving desirable data-flow facts because F_i and F_j are collapsible: replace the edge $F_i \rightarrow G_{ib}$ with the edge $F_i \rightarrow G_{ja}$ and replace the edge $F_j \rightarrow G_{jb}$ with the edge $F_j \rightarrow G_i a$.

2. Change the edges that point to F_j to point to F_i.

3. Remove the vertex F_j.

In Figure 80, we have assumed that the vertices F_1 and F_n are compatible. Figure 80 shows the result of eliding the vertex F_n into the vertex F_1.

To minimize the number of copies of F in H', we must separate the vertices $F_1, F_2, \ldots F_n$ into a minimum partition such that the vertices in each partition are collapsible with one another. H' must contain one copy of F for each partition.

Two vertices F_i and F_j in the hot-path graph H are collapsible iff the vertices V_i and V_j in the undirected graph G are not adjacent. This follows from the encoding of the graph G in the statements in vertices $D_1 \ldots D_n$. Let $\mathcal{F} \subseteq \{F_1, \ldots, F_n\}$ be a set of collapsible vertices. Then there are no adjacent vertices in the set $\mathcal{V} = \{V_i : F_i \in \mathcal{F}\}$. Likewise, if $V \subseteq \{V_1, \ldots V_n\}$ has no adjacent vertices, then the set $\mathcal{F} = \{F_i : V_i \in \mathcal{V}\}$ contains collapsible vertices.

This implies that any partition of $F_1, \ldots, F_n$ into sets of collapsible vertices encodes a partition of $V_1, \ldots V_n$ that is a valid coloring of G. Any valid coloring of G encodes a partition of $F_1, \ldots, F_n$ into sets of collapsible vertices. It follows that a minimal partition of the vertices $F_1, \ldots, F_n$ such that each partition contains collapsible vertices must encode a minimal coloring of G: if there were a coloring with fewer partitions, then we could use that coloring to find a smaller partitioning of $F_1, \ldots F_n$ into sets of compatible vertices.

Thus, it is NP-hard to find a minimum-sized reduced hot-path graph H' such that the meet-over-all-paths J' to copy constant propagation preserves the hot data-flow facts of J. *QED* □

Theorem 11.0.2 *Given a hot-path graph H^*, a distributive data-flow framework $\mathcal{F}$, and the meet-over-all-paths solution J to $\mathcal{F}^*_H$, it is computable to generate a reduced hot-path graph H'^* such that H'^* has a minimal size and the meet-over-all-paths solution J' to $\mathcal{F}_{H'^*}$ preserves the valuable data-flow facts of J.*

Proof: The proof is informal. We give an algorithm for finding the minimal H'^* with the desired data-flow solution J', but not an actual Turing Machine. The algorithm for finding H'^* is as follows:

1. Enumerate the set $\mathcal{H}$ of all the graphs that are smaller than H^* and path-congruent to H^*.

2. Pick H'^* to be smallest graph in $\mathcal{G}$ such that the meet-over-all path solution J' for $\mathcal{F}_{H'^*}$ preserves the valuable data-flow facts of J.

For every $H' \in \mathcal{H}$ we must find the meet-over-all-paths solution J' to $\mathcal{F}_{H'}$ and check that J' preserves the valuable data-flow facts of J. Since $\mathcal{F}$ is distributive, J' is equivalent to the greatest-fixed-point solution; this means that there are iterative algorithms for finding J' [40]. To check if J' preserves the valuable data-flow facts of J, we can use a modified version of the Vertex Subsumption Algorithm. This is discussed in Appendix E. *QED* □

In contrast to the previous theorem, we have the following conjecture:

Conjecture 11.0.3 *Given a hot-path graph H, any monotonic data-flow framework $\mathcal{F}$, and the meet-over-all-paths solution J to $\mathcal{F}_H$, there is no algorithm for generating a reduced hot-path graph H'*

such that H' has a minimal size and the meet-over-all-paths solution J' to $\mathcal{F}_{H'}$ preserves the valuable data-flow facts of J.

The principal reason for believing this conjecture is that there are no general algorithms for computing the meet-over-all-paths solution for monotone frameworks [36, 39]. This means that given a reduced hot-path graph H', there is no algorithm for computing the meet-over-all-paths solution J' to $\mathcal{F}_{H'}$ from H' alone. (If J' cannot be computed, then we cannot check if J' preserves the hot data-flow facts of J.) It may be possible to compute J' from H, J and H'. In other words, it might be possible to find the meet-over-all-paths solution to $\mathcal{F}_{H'}$ by leveraging off the fact that we have the meet-over-all-paths solution, namely $J = \mathcal{F}_H$, for a graph H that is path congruent to H'. However, this seems unlikely: let v be a vertex in H and let p_1 and p_2 be paths in H from $Entry$ to v. Since H is path congruent to H', we know that there are paths p_1 and p_2 in H' that are congruent to p'_1 and p'_2, respectively. The problem is that p'_1 may end at v'_1 and p'_2 may end at v'_2 such that $v'_1 \neq v'_2$. This means that if we want to use the meet-over-all-paths solution at v for computing the meet-over-all-paths solution at v'_1 and v'_2, we would have to "un-meet" the solution at v to find the individual contributions of the paths p_1 and p_2.

Chapter 12

Experimental Results for Reducing the Hot-path Supergraph and for Program Optimization

This chapter presents experimental results for the hot-path supergraph-reduction algorithms presented in the previous chapters. As stated in previous chapters, all experiments were run on a 833 MhZ Pentium III with 256M RAM running Solaris 2.7. For a compiler, we used GCC 2.95.3 with the "-O3" option. When run-time numbers are given, they represent the average of three runs.

12.0.1 The Supergraph Partitioning Algorithm

In the first set of experiments, we considered "conditional branches that have been determined to have only one possible outcome" to be desirable data-flow facts. We used the Ammons-Larus Reduction Algorithm with the Supergraph Partitioning Algorithm to preserve the branch outcomes that were determined by range analysis. For example, consider a vertex v that ends with a conditional branch "`if (x < y)`": suppose that range analysis determines that $J(v)(x) = [2..5]$ and $J(v)(y) = [6..8]$; then the condition `(x < y)` must always evaluate to true in the vertex v. After hot-path graph reduction, we want it to still be the case that the expression `(x < y)` is always be true in v (or, if v is replaced by a vertex w in the reduced graph, we want `(x < y)` to always be true in w).

This definition of a desirable data-flow fact allows great latitude in creating the compatibility partition that is given as input to the Supergraph Partitioning Algorithm. Continuing the above example, suppose v' is a duplicate copy of v such that $J(v')(x) = [20..22]$ and $J(v')(y) = [26..28]$. Then, the expression `(x < y)` is always true in v'. This means that v and v' can potentially be collapsed, because they agree on the outcome of the branch "`if (x < y)`", even if they disagree on the data-flow facts for x and y. When we create the compatibility partition, the vertices v and v' are put in the same block; if the Supergraph Partitioning Algorithm does not separate them into different blocks, they will be collapsed to a single vertex.

Figure 81 shows the results of our first set of experiments. In these experiments, we created the hot-path graphs by duplicating enough paths to cover 99% of the dynamic execution paths (*i.e.*, $C_A = 99\%$; see Chapter 8). We then performed range analysis on the hot-path supergraph. Next, we marked as hot any vertex that had a "decided" branch — a branch whose outcome can be determined from range analysis. We formed the compatibility partition as described in Section 9.2.2 and in the example above. Then we ran the Supergraph Partitioning Algorithm. Next, we collapsed vertices that remained in the same partition block. Finally, we emitted source code in which decided branches were replaced by `goto` statements (*i.e.*, unconditional jumps), and instruction operands and instructions that had constant values were replaced by literals. Note that this approach uses the Supergraph Partitioning Algorithm to preserve decided branches, which means that other data-flow facts (*e.g.*, a range data-flow fact showing a variable to be constant) may (or may not) be destroyed by the Supergraph Partitioning Algorithm;

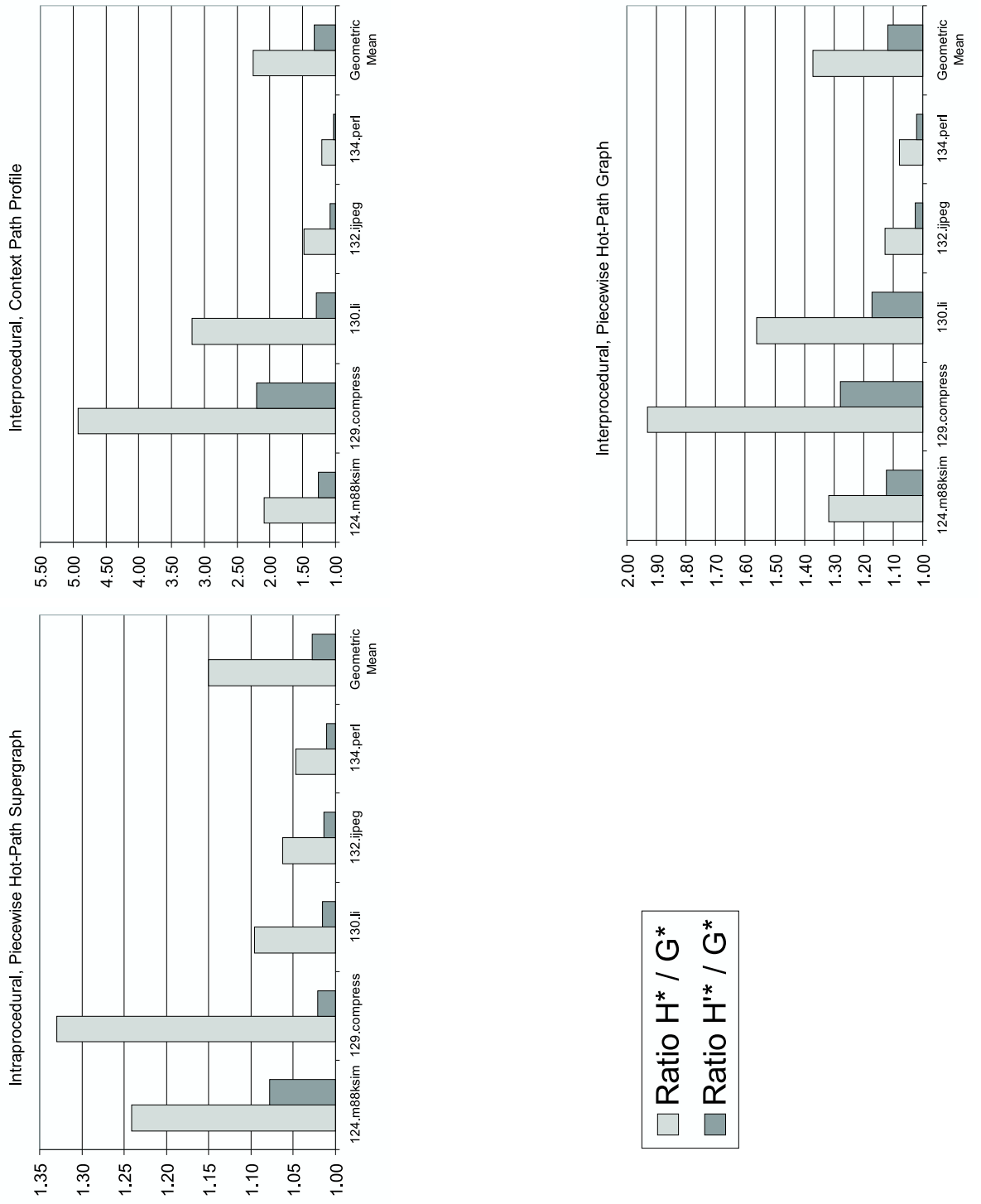


Figure 81: Charts showing how well the Supergraph Partitioning Algorithm does when it preserves all of the results for conditional branches in the range analysis.

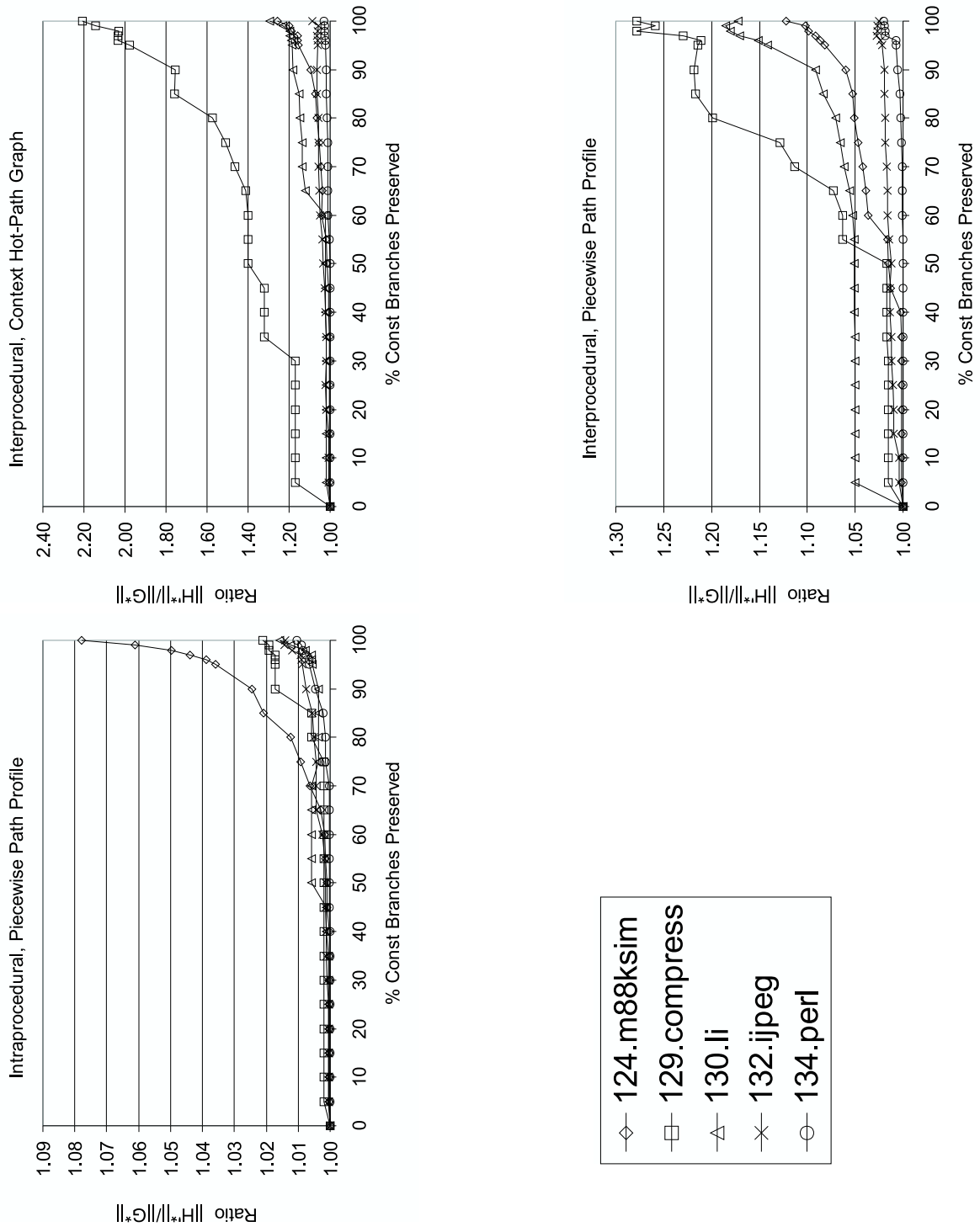


Figure 82: Plots of the amount of reduction done by the Supergraph Partitioning Algorithm versus the percentage of branch results that are saved.

however, when there is a data-flow fact in the reduced hot-path graph that shows that an instruction operand or an instruction result is constant, we make use of it.

Figure 81 shows that the Supergraph Partitioning Algorithm is quite effective in these experiments. The "average" interprocedural, context hot-path graph is 126% larger than the supergraph; after reduction, it is only 32% larger. (We are not actually comparing average sizes here: we compare the geometric mean of the ratios of the hot-path graph size to the supergraph size against the geometric mean of the ratios of the reduced-hot-path graph size to the supergraph size; see Figure 81.) The average interprocedural, piecewise hot-path graph is 37% larger than the supergraph; after reduction, it is only 12% larger. The average intraprocedural, piecewise hot-path graph is 15% larger than the supergraph; after reduction, it is only 3% larger.

The next set of experiments were identical to the previous experiments, except that we varied the number of vertices marked hot. Lowering the number of vertices marked hot lowers the number of vertices with data-flow solutions that must be preserved, which allows greater reduction. In these experiments, we marked as hot a certain percentage (weighted by execution frequency) of the vertices that contained decided branches. The results are shown in Figure 82. The fact that the lines in these graphs are not monotonic indicates that the Supergraph Partitioning Algorithm is sensitive to the input compatibility partition; changing the set of vertices that are marked hot changes the way the greedy algorithm for creating the compatibility partition will group vertices (see Section 9.2.2); this has an effect on the output of the Supergraph Partitioning Algorithm. Overall, Figure 82 shows that the fewer vertices that are marked hot, the greater the reduction in the hot-path supergraph.

12.0.2 Edge Redirection Algorithm

In this section, we present experimental results for reducing the hot-path supergraph using the Edge Redirection Algorithm. The Edge Redirection Algorithm is most effectively used with the Supergraph Partitioning Algorithm: the Edge Redirection Algorithm is too expensive to run unless the hot-path supergraph has already been partially reduced by the Supergraph Partitioning Algorithm; and the Edge Redirection Algorithm may make the graph more amenable to reduction by the Supergraph Partitioning Algorithm.

Figure 83 shows results from the first set of experiments with the Edge Redirection Algorithm. As in the previous section, we start by creating a hot-path supergraph by duplicating enough paths to cover 99% of the program's execution. Next, we perform range analysis on the hot-path supergraph. Then we use the Supergraph Partitioning Algorithm to reduce the hot-path supergraph while preserving every data-flow fact (by marking every vertex as hot and allowing vertices v and v' to be in the same block of the compatibility partition if and only if $J(v) = J(v')$ for any data-flow facts used in v). Recall from Section 10.7 that we must preserve every data-flow fact if we wish to follow the Supergraph Partitioning Algorithm by the Edge Redirection Algorithm.

After running the Supergraph Partitioning Algorithm, we run the Edge Redirection Algorithm and remove any vertices that become unreachable. Then we run the Supergraph Partitioning Algorithm again, and follow that by running the Edge Redirection Algorithm a second time. Finally, we run the Supergraph Partitioning Algorithm again, but this time, we only preserve decided branches, as we did in the last section (after which, it becomes unsafe to run the Edge Redirection Algorithm again).

These experiments show that the Edge Redirection Algorithm combined with the Supergraph Partitioning Algorithm can reduce the code growth even further when we wish to preserve all data-flow facts on the hot-path supergraph (see the fourth bar in the charts of Figure 83). For example, performing the interprocedural express-lane transformation on compress and then reducing the hot-path supergraph

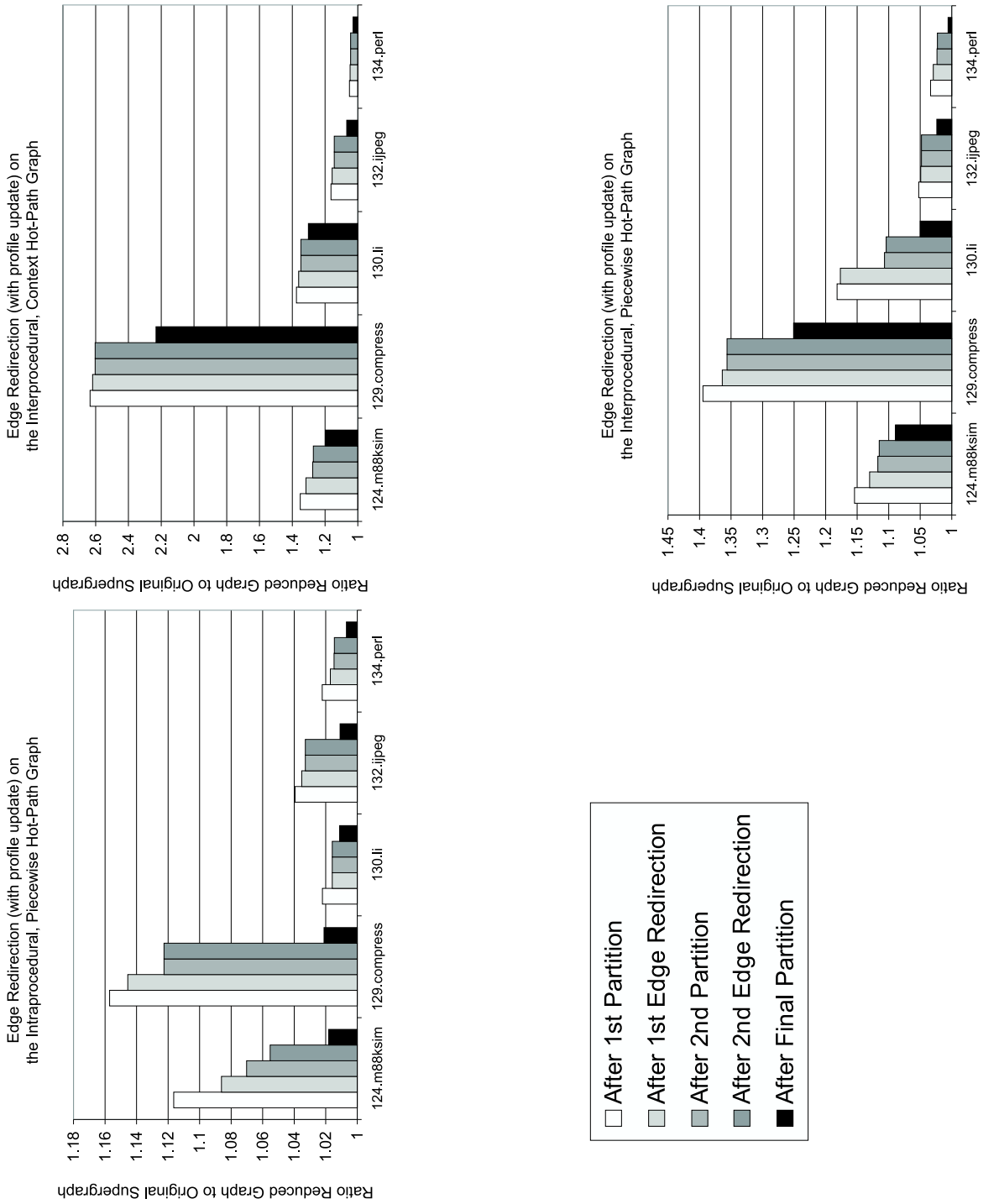


Figure 83: Plots of the amount of reduction done using successive iterations of the Supergraph Partitioning Algorithm and the Edge Redirection Algorithm. In these experiments, the Edge Redirection Algorithm is restricted by the need to translate the path profile on the hot-path supergraph into a path profile for the reduced graph.

Benchmark	Express-Lane Trans.	Time 1^{st} Partition	Time 2^{nd} Partition	Time 1^{st} Edge Redirection	Time 2^{nd} Edge Redirection
124.m88ksim	Inter., Context	23.05	7.67	21.28	18.26
	Inter., Piecewise	9.10	4.76	14.54	14.13
	Intra., Piecewise	7.46	4.61	13.58	12.93
129.compress	Inter., Context	1.25	0.48	0.63	0.61
	Inter., Piecewise	0.41	0.29	0.18	0.17
	Intra., Piecewise	0.27	0.19	0.09	0.09
130.li	Inter., Context	6.30	2.84	8.10	7.90
	Inter., Piecewise	3.67	2.43	6.87	5.58
	Intra., Piecewise	2.25	1.77	4.11	4.16
132.ijpeg	Inter., Context	10.73	8.83	1375.16	1419.20
	Inter., Piecewise	7.74	6.97	1168.59	1232.78
	Intra., Piecewise	6.76	6.50	1174.01	1236.21
134.perl	Inter., Context	20.26	8.68	78.52	79.32
	Inter., Piecewise	11.05	8.56	81.64	77.37
	Intra., Piecewise	10.50	7.81	74.30	75.85
Average	Inter., Context	12.32	5.70	296.74	305.06
	Inter., Piecewise	6.39	4.60	254.36	266.01
	Intra., Piecewise	5.45	4.18	253.22	265.85

Table 10: Table showing the time in seconds required to run the analyses in the first thru fourth columns of Figure 83

with the Supergraph Partitioning Algorithm (preserving all data-flow facts) results in a graph that is 1.39 times the size of the original supergraph. Using the Edge Redirection Algorithm and the Supergraph Partitioning Algorithm results in graph that is 1.2 times the size of the original supergraph. Below, we will discuss how much the Edge Redirection Algorithm helps in reducing the hot-path graph and preserving only decided branches.

Table 10 shows the runtimes for the first four phases of the first set of experiments. The second run of the Supergraph Partitioning Algorithm is much faster because it must collapse many fewer nodes. (The Supergraph Partitioning Algorithm spends the bulk of its time computing the meet of data-flow facts from vertices that it has decided to collapse.) For the most part, the Supergraph Partitioning Algorithm and the Edge Redirection Algorithm are cheap, especially when compared to the time required to perform interprocedural data-flow analysis. The one exception is when the Edge Redirection Algorithm is run on ijpeg.

Next, we consider the effect of removing the requirement that the Edge Redirection Algorithm translate a path profile for the hot-path supergraph to a path profile for the reduced hot-path supergraph. Recall from Section 10.6 that sometimes an replacing an edge $u \rightarrow v$ in a graph G with an edge $u \rightarrow v'$ to create a graph H may make it impossible to translate a path profile for G into a path profile for H. To avoid this problem, we restrict the set of edge redirections that the Edge Redirection Algorithm can perform. However, if we are not concerned with translating a path profile from G to H, then we can increase the opportunities for redirecting edges. Figure 84 shows the results of running the experiments

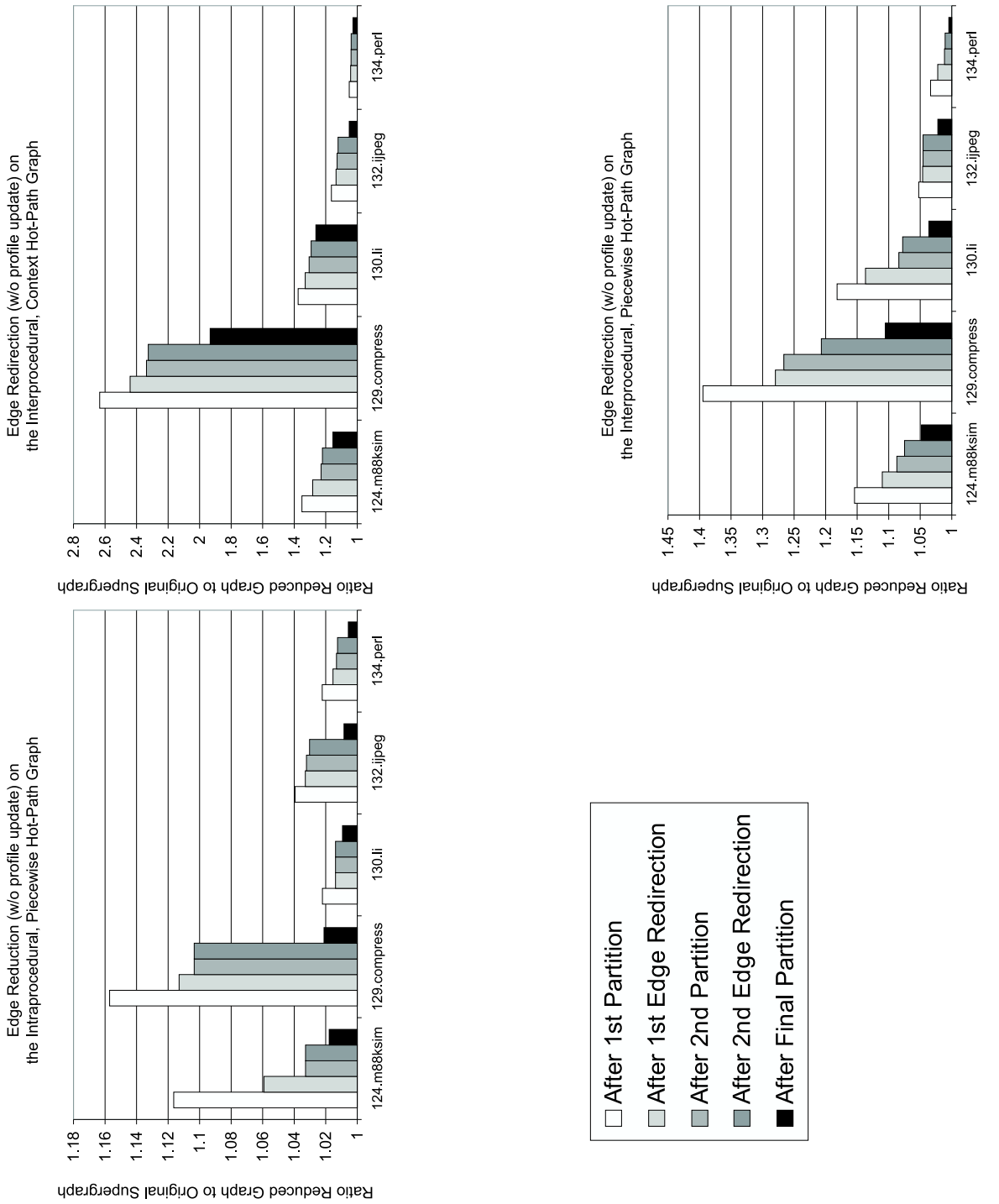


Figure 84: Plots of the amount of reduction done using successive iterations of the Supergraph Partitioning Algorithm and the Edge Redirection Algorithm. In these experiments, the Edge Redirection Algorithm is *not* restricted by the need to translate the path profile on the hot-path supergraph into a path profile for the reduced graph.

Benchmark	Express-Lane Trans.	Time 1^{st} Partition	Time 2^{nd} Partition	Time 1^{st} Edge Redirection	Time 2^{nd} Edge Redirection
124.m88ksim	Inter., Context	23.3476	6.87433	19.9231	16.887
	Inter., Piecewise	8.88762	4.81371	13.6085	12.7472
	Intra., Piecewise	7.5746	4.82251	12.9695	12.2203
129.compress	Inter., Context	1.2146	0.481806	0.447762	0.317464
	Inter., Piecewise	0.400633	0.1993	0.129242	0.0846311
	Intra., Piecewise	0.261307	0.166892	0.069011	0.059237
130.li	Inter., Context	5.75041	2.5138	6.292	5.87555
	Inter., Piecewise	3.51735	2.21667	5.37468	4.54208
	Intra., Piecewise	2.15271	1.73595	3.88901	3.90023
132.ijpeg	Inter., Context	10.307	7.42806	1304.09	1305.13
	Inter., Piecewise	7.52524	6.76588	1166.43	1229.05
	Intra., Piecewise	6.64826	6.32887	1164.78	1227.36
134.perl	Inter., Context	15.3893	8.26194	76.9664	77.5232
	Inter., Piecewise	10.8743	8.57494	73.311	73.5693
	Intra., Piecewise	10.3659	7.65246	80.4258	75.6512
Average	Inter., Context	11.201782	5.1119872	281.5438524	281.1466428
	Inter., Piecewise	6.2410286	4.5141	251.7706844	263.9986422
	Intra., Piecewise	5.4005554	4.1413364	252.4266642	263.8381934

Table 11: Table showing the time in seconds required to run the reduction algorithms in the first thru fourth columns of Figure 84

described at the beginning of this section (which are summarized in Figure 83), but without the requirement that we be able to update a path profile after performing the Edge Redirection Algorithm. Figure 84 shows that removing this requirement allows the Edge Redirection Algorithm to be much more effective. The disadvantage of not being able to translate a path profile is that the profiling information is not available for later passes of the compiler that may come after the express-lane transformation.

We have presented three strategies for reducing the hot-path supergraph while preserving decided branches: (1) use the Supergraph Partitioning Algorithm (by itself); (2) use the Supergraph Partitioning Algorithm together with the Edge Redirection Algorithm (Column 5 of Figure 83); and (3) use the Supergraph Partitioning Algorithm together with the Edge Redirection Algorithm, but without the requirement that the path profile on the hot-path supergraph be updated to a path profile for the reduced graph(Column 5 of Figure 84). Figure 85 compares the results of these three strategies and shows that the third strategy has the best results.

12.1 Using the Express-Lane Transformation for Program Optimization

Tables 12 through 18 show the results of using various forms of the express-lane transformation together with Range Analysis to optimize SPEC95Int benchmarks. Specifically, we followed the following steps:

1. Perform an express-lane transformation.

2. Perform interprocedural range analysis on the hot-path (super)graph.

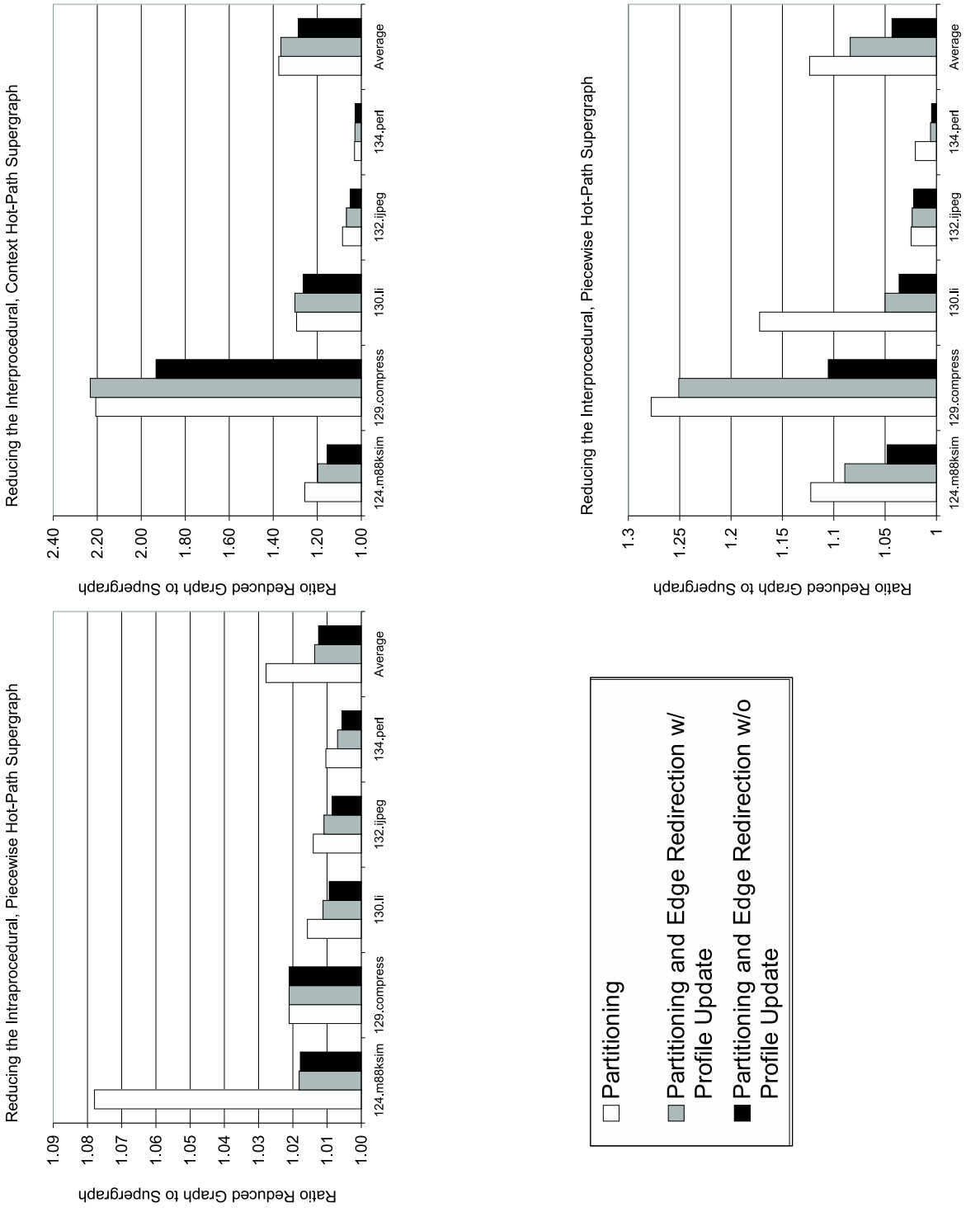


Figure 85: Comparison of strategies for reducing the hot-path supergraph while preserving decided branches.

Benchmark	Base run time (sec)
124.m88ksim	146.70
129.compress	135.46
130.li	125.81
132.ijpeg	153.83
134.perl	109.04

Table 12: Base run times for SPECInt95 benchmarks. The programs were optimized using interprocedural range analysis to remove decided branches and constant expressions (but without any express-lane transformation). Then they were compiled using GCC 2.95.3 -O3.

Benchmark	None	Partitioning	Partitionining, Kill Range Prop
124.m88ksim	-34.7%	-9.3%	-29.5%
129.compress	-14.0%	**1.0%**	-4.3%
130.li	-57.2%	-20.4%	-27.8%
132.ijpeg	-7.5%	-1.6%	-1.2%
134.perl	-21.3%	**4.9%**	**6.0%**

Table 13: Program speedups due to the **interprocedural, context** express-lane transformation. In Column II, no reduction algorithm was used on the hot-path supergraph. In Column III, the Supergraph Partitioning Algorithm was used, preserving decided branches. In Column IV, the Supergraph Partitioning Algorithm was used, preserving decided branches; then the results of range propagation were discarded.

Benchmark	Partitioning and Edge Redirection, Save all DFA	Partitioning and Edge Redirection, Save Branches
124.m88ksim	-13.1%	-11.4%
129.compress	**2.4%**	**2.0%**
130.li	-30.4%	-25.4%
132.ijpeg	-4.5%	-4.8%
134.perl	-3.1%	-3.0%

Table 14: Program speedups due to the **interprocedural, context** express-lane transformation. In Column II, the hot-path supergraph was reduced using the Supergraph Partitioning Algorithm and the Edge Redirection Algorithm such that all data-flow facts were preserved. In Column III, the hot-path supergraph was reduced using the Supergraph Partitioning Algorithm and the Edge Redirection Algorithm such that decided branches were preserved. In both Columns II and III, the Edge Redirection Algorithm was not inhibited by the requirement that the path profile be translated.

Benchmark	None	Partitioning	Partitionining, Kill Range Prop
124.m88ksim	-13.6%	-0.7%	-11.4%
129.compress	-14.0%	**0.5%**	-4.5%
130.li	-68.1%	-26.7%	-40.1%
132.ijpeg	-2.3%	-2.2%	-0.8%
134.perl	-19.4%	2.8%	2.7%

Table 15: Program speedups due to the **interprocedural, piecewise** express-lane transformation. In Column II, no reduction algorithm was used on the hot-path supergraph. In Column III, the Supergraph Partitioning Algorithm was used, preserving decided branches. In Column IV, the Supergraph Partitioning Algorithm was used, preserving decided branches; then the results of range propagation were discarded.

Benchmark	Partitioning and Edge Redirection, Save all DFA	Partitioning and Edge Redirection, Save Branches
124.m88ksim	**5.7%**	**5.4%**
129.compress	-0.2%	**2.0%**
130.li	-11.4%	**2.5%**
132.ijpeg	-2.2%	-4.2%
134.perl	**6.1%**	**3.6%**

Table 16: Program speedups due to the **interprocedural, piecewise** express-lane transformation. In Column II, the hot-path supergraph was reduced using the Supergraph Partitioning Algorithm and the Edge Redirection Algorithm such that all data-flow facts were preserved. In Column III, the hot-path supergraph was reduced using the Supergraph Partitioning Algorithm and the Edge Redirection Algorithm such that decided branches were preserved. In both Columns II and III, the Edge Redirection Algorithm was not inhibited by the requirement that the path profile be translated.

Benchmark	None	Partitioning	Partitionining, Kill Range Prop
124.m88ksim	**10.6%**+	**13.0%**	**1.2%**
129.compress	**6.4%**	**5.5%**	-2.1%
130.li	**8.1%**	**10.3%**	**7.2%**
132.ijpeg	**1.0%**	**0.7%**	-0.1%
134.perl	**9.7%**	**10.0%**	**6.3%**

Table 17: Program speedups due to the **intraprocedural, piecewise** express-lane transformation. In Column II, no reduction algorithm was used on the hot-path supergraph. In Column III, the Supergraph Partitioning Algorithm was used, preserving decided branches. In Column IV, the Supergraph Partitioning Algorithm was used, preserving decided branches; then the results of range propagation were discarded.

Benchmark	Partitioning and Edge Redirection, Save all DFA	Partitioning and Edge Redirection, Save Branches
124.m88ksim	**11.6%**	**7.4%**
129.compress	**2.1%**	**0.1%**
130.li	-1.7%	-0.6%
132.ijpeg	-1.6%	-2.0%
134.perl	**9.9%**	**5.4%**

Table 18: Program speedups due to the **intraprocedural, piecewise** express-lane transformation. In Column II, the hot-path supergraph was reduced using the Supergraph Partitioning Algorithm and the Edge Redirection Algorithm such that all data-flow facts were preserved. In Column III, the hot-path supergraph was reduced using the Supergraph Partitioning Algorithm and the Edge Redirection Algorithm such that decided branches were preserved. In both Columns II and III, the Edge Redirection Algorithm was not inhibited by the requirement that the path profile be translated.

3. Reduce the hot-path (super)graph.
4. Use the results of interprocedural range analysis to eliminate branches and to replace constant expressions with a literal.
5. Emit C source code for the transformed program.
6. Compile the C source code using GCC 2.95.3 -O3.
7. Compare the runtime of the new program with the runtime of the original program.

For a base case, we performed range analysis without any express-lane transformation and used the results to eliminate branches and replace constant expressions (Table 12). We ran experiments with the following express-lane transformations:

1. The interprocedural, context express-lane transformation with $C_A = 99\%$ (Tables 13 and 14).
2. The interprocedural, piecewise express-lane transformation with $C_A = 99\%$ (Tables 15 and 14).
3. The intraprocedural, piecewise express-lane transformation with $C_A = 99\%$ (Tables 17 and 18).

For each of the express-lane transformations, we tried the following hot-path graph reduction strategies:

1. None.
2. Using the Supergraph Partitioning Algorithm to preserve all decided branches (as was done in Figure 81).
3. Using the Supergraph Partitioning Algorithm and the Edge Redirection Algorithm to preserve all of the results of range analysis (which is equivalent to the fourth column of Figure 84); here, the requirement that the path profile be translated after edge redirection was lifted.
4. Using the Supergraph Partitioning Algorithm and the Edge Redirection Algorithm to preserve all decided branches (which is equivalent to the fifth column of Figure 84); as above, the requirement that the path profile be translated after edge redirection was lifted.

We also ran experiments where we performed an express-lane transformation, then used the second listed reduction strategy (partioning), and then discarded the results of range analysis. In all of the experiments, the reported run time is the average of three runs.

The best results were for the intraprocedural express-lane transformation (Tables 17 and 18). The intraprocedural express-lane transformation together with the range analysis optimizations has a benefit to performance even when no reduction strategy is used to limit code growth. In fact, aggressive reduction strategies can destroy the performance gains. There are several possible reasons for this:

1. GCC may be able to take advantage of the express-lane transformation to perform its own optimizations (*e.g.*, code layout [44]).
2. Reduction of the hot path graphs may result in poorer code layout that requires more unconditional jumps along critical paths [48].
3. The more aggressive reduction strategies seek only to preserve decided branches, and may destroy data-flow facts that show an expression to have a constant value.

4. The reduced graph may have a code layout that interacts poorly with the instruction cache.

The results for the interprocedural express-lane transformations are disappointing. Here the performance is usually degraded by the express-lane transformation. There are two likely reasons for this:

1. We use a poor implementation of entry and exit splitting.

2. There is significantly more code growth than in the intraprocedural express-lane transformation.

Unlike the intraprocedural express-lane transformation, using aggressive reduction strategies with the interprocedural express-lane transformations usually helps performance. In the interprocedural case, not only does graph reduction lessen code growth, it may also eliminate the need to perform entry and exit splitting. In fact, with the most aggressive reduction strategy, the interprocedural piecewise express-lane transformation consistently leads to modest performance improvements (Table 16).

It should also be noted that the interprocedural express-lane transformations combined with the range-analysis optimizations do have a strong positive impact on program performance, although it is usually not as great as the costs incurred by the transformations. This can be seen in the experiments where we discarded the range-analysis results (and did not eliminate branches and replace constants): the performance was consistently worse. In those few cases where performance showed a slight improvement, we assume there was a change in code layout that had positive instruction cache effects.

Chapter 13

RelatedWork

This chapter is divided into two sections. Section 13.1 discusses work that is related to our path-profiling techniques. Section 13.2 discusses work that is related to the interprocedural express-lane transformation.

13.1 Related Profiling Work

The path-profiling techniques presented here extend the Ball-Larus path-profiling technique of [12]. Section 2.1 summarizes the approach taken in [12]. Our approach generalizes their technique in several ways:

1. We present techniques for collecting interprocedural path profiles. This means that the observable paths (that may be logged in a profile) may cross procedure boundaries. We present algorithms for collecting both interprocedural context and interprocedural piecewise path profiles.

2. In context path profiling, each observable path corresponds to a pair $\langle C, p\rangle$, where p corresponds to a subpath of an execution sequence, and C corresponds to a context (*e.g.*, a sequence of pending calls) in which p may occur. The set of all p such that $\langle C, p\rangle$ is an observable path must cover every possible execution sequence. In our interprocedural, context path profiling technique, the context C may include not only the sequence of pending calls, but also the last acyclic intraprocedural path that was taken before each pending call. We also present an intraprocedural, context path profiling where the context may summarize the path taken to a loop header.

Ammons, Ball, and Larus also offer an interprocedural context path profiling method in [4]. In their method, each observable path is a pair $\langle C, p\rangle$ where C is (a summary of) a sequence of pending calls and p is a path from the Ball-Larus path profiling technique (*i.e.*, an intraprocedural, acyclic path). In contrast, in our interprocedural, context path-profiling technique, an observable path is a pair $\langle C, p\rangle$ where C is a (fragmented) interprocedural path that contains information not only about the sequence of pending calls, but also about the paths taken before each pending call, and p is an interprocedural path that may cross procedure boundaries.

Theoretically, our observable paths contain more information. However, in order to lower the overhead of our profiling machinery, we are often forced to break observable paths into smaller pieces. This means that the amount of calling context that can be captured by our profiling technique is bounded in practice. So, there is a trade-off between our profiling techniques and the one described in [4]. It is possible to combine the two methods to obtain a profiling technique where each observable path is a triple $\langle C', C, p\rangle$, where C' is the context summary used in [4], and C and p form an observable path in our interprocedural, context technique. Such a combination is beyond the scope of the thesis.

Young and Smith present a path-profiling technique that uses a sliding window that advances one edge at a time over the program trace; each time the window advances, they record the execution of the path that is found in the window [63, 65]. Their technique may be contrasted with ours in the following

ways: (1) in their technique, there is a great deal more overlap between observable paths; (2) the length of each observable path is fixed by the size of the sliding window; (3) they do not distinguish between cyclic and acyclic paths; and (4) they are only concerned with intraprocedural paths. One advantage of their technique is that they can handle cyclic paths. A possible disadvantage is that every path is of a fixed length: in [65], Young experiments with sliding windows that are as short as 1 and as long as 15; in our interprocedural, context path profiles, the average path length can be as long as 70 edges (see Table 4), though this comparison is not entirely fair since we are comparing an interprocedural, context path profiling technique with an intraprocedural path-profiling technique. Young has shown that with a careful implementation, the overhead of their profiling technique is reasonable (it is similar to our overhead for interprocedural path profiling) [65].

Bit tracing is an example of a piecewise path-profiling technique [12]. In bit tracing, the outcome of each n-ary branch is recorded using $\log n$ bits in a buffer. When the buffer is filled, a counter is incremented for the path in the buffer. In contrast to the Ball-Larus technique and to our techniques, bit tracing does not produce a dense numbering of observable paths [12]. This means that, in general, more bits may be required for a path name, which may increase the cost of the profiling machinery, or lower the average length of an observable path. The overhead for bit-tracing is also likely to be higher because (1) it must include repeated checks for whether the end of the buffer has been reached and (2) there must be some instrumentation code on each edge. In contrast, the Ball-Larus technique and our path-profiling techniques do not have to check for reaching the end of a buffer, and there is some flexibility in picking which edges to put instrumentation on [12, 10].

Recently, Larus developed a technique for gathering a *whole program path* [42]. A whole program path is an encoding of a program's entire execution sequence; that is, a whole program path for a program $\mathcal{P}$ encodes the sequence of all control decisions made during an execution of $\mathcal{P}$. (This is an improvement of Ball and Larus's earlier technique for collecting and representing an execution sequence [11].) The encoding has the following advantages: (1) it is relatively cheap to generate; (2) it is compact; and (3) it is easy it analyze. In particular, Larus shows how to analyze the whole program path in order to identify frequently executed paths; the paths identified by this technique may by cyclic, and they may cross procedure boundaries.

Zhang and Gupta developed a *timestamped whole program path* [66]. A timestamped whole program path is a variant of a whole program path; Zhang and Gupta claim that certain types of information (*e.g.*, intraprocedural path profiles) are easier to extract from a timestamped whole program path than from one of Larus's whole program paths.

Whole program paths and timestamped whole program paths were developed after our profiling techniques (see [46]) and they may obviate our approach. (There is also recent evidence that suggests that the information in edge profiles is sufficient for many optimization tasks [14, 54, 18, 17].) The information in a whole program path subsumes the information in a path profile. What is more, the hot paths that are extracted from a whole program path do not suffer from the limitations of paths in our profiling techniques: they may contain backedges and recursive calls. However, no matter what technique is used to select hot paths, our techniques for performing the interprocedural express-lane transformation and for reducing the hot-path graph are still applicable.

13.2 Related Path Optimization Work

Work that focuses on improving the performance of particular program paths dates back at least to Fisher's trace scheduling technique [28]. In this technique, frequently executed paths, called *traces*, are

optimized at the expense of infrequently executed paths. Code motion is used to improve (*i.e.*, shorten) the instruction schedule on a parallel machine; instructions are copied to CFG edges that branch into or out of the trace and the edges may be moved in order to preserve the program's execution behavior; the technique does not use path duplication.

Hwu et al. examined the *superblock transformation* [44, 35, 20]. A superblock is similar to an express-lane in that there is no control-flow into the middle of a superblock. The technique for creating superblocks estimates hot paths by using edge profiles. [44] focuses on improving the instruction schedule within a superblock. In [20], Chang, Mahlke, and Hwu investigate the benefits of superblock formation for dead-code elimination, common-subexpression elimination, and copy propagation. Hank combines function inlining with superblock formation, in effect duplicating interprocedural paths. He investigates the benefits for several data-flow problems [35]. None of these works include a stage for eliminating duplicated code that did not show improved data-flow results.

Another interesting example of using path duplication to improve data-flow analysis is found in Poletto's work on *Path Splitting* [52]. This work duplicates paths within a loop body in order to provide optimal reaching-definitions information along the duplicated paths. The improved information is used in several classic code optimizations, including copy propagation, common subexpression-elimination, dead-code elimination, and code hoisting. The path-splitting transformation is done statically: no profiling information is used. It also differs from our work on the interprocedural express-lane transformation in that it focuses on optimizing intraprocedural loop bodies and does not include a stage to collapse vertices with similar data-flow solutions.

In general, procedure cloning and procedure inlining can be considered to be a kind of bulk duplication of interprocedural paths; a highly abbreviated list of works that have examined procedure cloning or procedure inlining includes [24, 34, 60, 26, 19, 55, 25, 7]. Given a call-site c to a procedure P, creating a unique copy of P that is only called from c (or inlining a copy of P at c) has the effect of separating the interprocedural paths that enter P through c from the interprocedural paths that enter P through some other call-site. However, individual paths that enter P through c are not isolated from one another, as they can be in the interprocedural express-lane transformation.

Of the work done on procedure cloning, [60] is interesting because it attempts to use intraprocedural path profiles to guide procedure cloning in order to improve data-flow results along interprocedural paths. More specifically, in [60], for each call-site s to a procedure P, they collect a Ball-Larus path profile of P. If there is a significant difference in the profiles for a pair of call-sites that both call a procedure P, then they assume that the call-sites should each call different copies of P. In some cases, this technique has the effect of isolating interprocedural paths from one another. However, this approach, like all techniques based on inlining and cloning, cannot be used to separate interprocedural paths that contain the same call-sites.

Bodik also uses code duplication to improve the performance of specific program paths [18]. His approach differs from ours in that code duplication is the last step in his optimization strategy: first he creates a program representation that exposes possible optimization opportunities; then he uses data-flow analysis to identify paths that can benefit from duplication; and, finally, he duplicates paths. In the second step, Bodik uses profiling information to pick which paths to duplicate. Bodik also uses demand-driven analysis to focus his analyses on hot paths. In our optimization strategy, we start by duplicating hot-paths, then perform data-flow analysis on the entire hot-path supergraph, and finally eliminate duplicate code that does not show any benefit.

Young's thesis includes work on using path profiling to improve the accuracy of static branch prediction and on improving instruction scheduling [65, 63, 64]. As mentioned above, his work is based on intraprocedural path profiles.

There are many other works that are based on improving the performance of specific paths by using path duplication, including [9, 21, 48, 49, 16]. There has also been a great deal of work that uses techniques besides code duplication (such as code motion, predication, and speculation) to improve the performance of certain paths at the expense of others; included in this group are [31, 33, 32].

In summary, the express-lane optimization strategy differs from other path-specific optimization strategies for one or more of the following reasons:

1. we use interprocedural path profiles;
2. we duplicate interprocedural paths;
3. we perform path duplication before performing data-flow analysis;
4. and we attempt to eliminate duplication when there has been no benefit to the results of the data-flow analysis.

Chapter 14

Contributions and Future Work

We have shown how to extend the Ball-Larus path-profiling technique in many ways. In particular, we have developed several interprocedural path-profiling techniques. Our techniques are practical for most benchmarks, and our path profiles have several advantages over their intraprocedural counter-parts: (1) the paths may cross procedure boundaries; and (2) the paths (especially in interprocedural context path profiles) tend to be significantly longer. This means that our interprocedural path-profiling techniques contain a more accurate summary of a program's execution behavior. This is a positive result for any application of path profiling.

The thesis has examined in detail one application of path profiling: the express-lane transformation. We presented techniques for performing the interprocedural express-lane transformation, both for an interprocedural, context path profile and for an interprocedural, piecewise path profile. We have shown that the interprocedural express-lane transformations do improve the results of range analysis along the duplicated paths. Furthermore, the improvements for range analysis are usually better for one of our interprocedural express-lane transformations than they are for the intraprocedural express-lane transformation.

Unfortunately, we have not shown that the interprocedural express-lane transformation is an effective optimization strategy. One reason for this is that the benefits for range-analysis (and branch elimination) are not great enough to offset the costs of the express-lane transformation, namely, code growth and increased function-call overhead due to entry and exit-splitting.

Our implementation of entry- and exit-splitting is not very efficient. We implement entry-splitting by passing an extra parameter to each procedure and inserting a switch statement on this parameter at the beginning of the procedure; similarly, we implement exit-splitting by returning an extra value and in inserting a switch statement on this value in each return-site vertex. The advantage of this implementation is that it can be done as a source-to-source transformation using the SUIF toolkit, which simplified our development. However, it is not the most efficient approach for implementing entry and exit splitting. At this point, it is impossible to say if using a better implementation of entry- and exit-splitting would help make the interprocedural express-lane transformation a desirable program optimization.

We expect that the express-lane transformation will benefit other data-flow analyses in much the same way that it benefits range-analysis. One area of future research is to measure the effects of the express-lane transformation on other data-flow analyses such as common sub-expression elimination, dead-code elimination, copy propagation, and flow-sensitive pointer analysis. Our preliminary results with range analysis suggest that the express-lane transformation combined with multiple data-flow analyses will result in improved runtime performance.

The thesis also contains several contributions to the problems of reducing the hot-path graph and reducing the hot-path supergraph. This work may be applicable to the application of efficiently enforcing security policies (see below). We have shown that the problem of reducing a hot-path graph while preserving the valuable data-flow facts is NP-hard. We have also given some simple examples where the Coarsest Partitioning Algorithm does a poor job of reducing the hot-path graph. We presented a new technique, called the Edge Redirection Algorithm, that helps to reduce the hot-path graph further than

the Coarsest Partitioning Algorithm does. We have also shown that the Edge Redirection Algorithm can be a valuable tool when reducing the Hot-path Supergraph.

We have also shown how to adapt the Coarsest Partitioning Algorithm to reduce the hot-path supergraph. The Coarsest Partitioning Algorithm can be used to minimize a deterministic, finite automaton. Our modified version of the Coarsest Partitioning Algorithm, called the Supergraph Partitioning Algorithm, is used to reduce (though it does not minimize) the hot-path supergraph, an example of a pushdown automaton.

One area of future work is to find an efficient algorithm to replace the Supergraph Partitioning Algorithm that: (1) takes as input a hot-path supergraph H^* and a partition π of H^*'s vertices and (2) produces as output a *minimal* partition π' that respects the edges of H^* and the partition π. Alternatively, one could show that such an algorithm is unlikely to exist, by showing that the problem is NP-hard. (Note that an algorithm that produces the minimal partition π' that respects H^* and π still does not solve the problem of finding a minimal reduction of H^* that preserves valuable data-flow facts; this is because the choice of the input partition π may be suboptimal, and because finding a minimal reduction of H^* may require modifying the edge relation of H^*, not just partitioning H^*'s vertices.)

Another area of future work is to determine how much worse our reduction techniques are than an optimal reduction strategy. In other words, how much larger are the reduced hot-path graphs produced by our techniques than the reduced hot-path graph produced by an optimal hot-path reduction algorithm? Clearly, a reduced hot-path graph must be at least as large as the supergraph from which the hot-path graph was constructed, which puts a lower bound on how small a reduced hot-path graph can be. However, this does not establish a precise lower bound for the size of the reduced hot-path graph.

Another approach to reducing the hot-path supergraph is to use procedure extraction [41, 27, 23]. Procedure extraction attempts to find identical code segments that can be extracted into a procedure and then replaced with a call to the new procedure. This may make legacy code more understandable, and it may reduce the size of the program (which is an important goal for programs running on memory-constrained systems). Use of procedure extraction on a hot-path supergraph H^* has the potential to create a reduced hot-path supergraph that is smaller that the original supergraph G^*. Our reduction techniques cannot do this: we only eliminate hot-path supergraph vertices that we created during the express-lane transformation; we cannot "fold together" similar vertices that are not duplicates.

Procedure extraction also differs from our techniques for reducing the hot-path supergraph in the following sense: in our techniques, given duplicate vertices v and v', we can (potentially) eliminate v' if for every path p' from v', there is a path p from v that is equivalent or better than p'. In contrast, using procedure extraction, v and v' can be replaced by a single vertex if for every path p' from v' *that is in the region to be extracted*, there is a path p from v that is equivalent. This potentially gives procedure extraction many more opportunities to reduce the hot-path supergraph in size. Note, however, that procedure extraction may have a negative impact on the program's performance; in particular, it may add procedure-call overhead along hot paths.

Recently, there has been increased interest in combining security automata with a program's control-flow (super)graph in order to guarantee at runtime that the program does not violate certain security protocols (*e.g.*, see [22, 56]). The problem of combining a security automaton with a supergraph is similar to the problem of combining a hot-path automaton with a supergraph. Thus, enforcing security policies (efficiently at run-time) may be another application of the Hot-Path Tracing Algorithm and our algorithms for reducing the hot-path graph. This is another area of future work.

Bibliography

[1] A. V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[2] Alfred V. Aho. *Algorithms for finding patterns in strings*, chapter 5, pages 255–300. MIT Press, 1994.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[4] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI'97*, June 1997.

[5] G. Ammons and J. Larus. Improving data-flow analysis with path profiles. In *Proc. of the ACM SIGPLAN 98 Conf. on Program. Lang. Design and Implementation*, June 1998.

[6] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Univ. of Copenhagen, May 1994. (DIKU report 94/19).

[7] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno jvm. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2000.

[8] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM computing surveys*, 26(4), 1994.

[9] Vasanth Bala, Evelyn Dusterwald, and Sanjeev Banerjia. Transparent dynamic optimization: The design and implementation of dynamo. Technical report, Hewlett-Packard Company, 1999.

[10] T. Ball. Efficiently counting program events. In *Trans. on Prog. Lang. and Syst.*, 1994.

[11] T. Ball and J. Larus. Optimally profiling and tracing programs. *Trans. on Prog. Lang. and Syst.*, 16:1319–1360, 1994.

[12] T. Ball and J. Larus. Efficient path profiling. In *MICRO 1996*, 1996.

[13] T. Ball and J. Larus. Programs follow paths. Technical Report MSR-TR-99-01, Microsoft Research, 1999.

[14] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In *Symposium on Principles of Programming Languages*, New York, NY, January 1998. ACM Press.

[15] R. Bodik, R. Gupta, and M.L. Soffa. Interprocedural conditional branch elimination. In *SIGPLAN Conference on Programming Languages Design and Implementation*, 1997.

[16] R. Bodik, R. Gupta, and M.L. Soffa. Complete removal of redundant expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[17] R. Bodik, R. Gupta, and M.L. Soffa. Load-reuse analysis: Design and evaluation. In *SIGPLAN Conference on Programming Languages Design and Implementation*, 1999.

[18] Rastislav Bodik. *Path-sensitive, value-flow optimizations of programs*. PhD thesis, University of Pittsburg, 2000.

[19] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–369, May 1992.

[20] Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. Using profile information to assist classic code optimizations. *Software practice and experience*, 1(12), Dec. 1991.

[21] R. Cohn and P.G. Lowney. Hot cold optimization of large Windows/NT applications. In *MICRO-29*, 1996.

[22] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Symposium on Principles of Programming Languages*, 2000.

[23] K. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *SIGPLAN Conference on Programming Languages Design and Implementation*, 1999.

[24] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–118, April 1993.

[25] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *Conf. on Lisp and Functional Programming*, pages 273–282, June 1994.

[26] S. Debray. Profile-guided context-sensitive program analysis. draft report, October 1998.

[27] Soumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *Trans. on Prog. Lang. and Syst.*, 2000.

[28] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. In *IEEE Trans. on Computers*, volume C-30, pages 478–490, 1981.

[29] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman and Co., San Fransisco, 1979.

[30] David Gries. Describing an algorithm by hopcroft. *Acta Informatica*, 2:97–109, 1973.

[31] R. Gupta, D. Berson, and J.Z. Fang. Path profile guided partial dead code elimination using predication. In *International Conference on Parallel Architectures and Compilation Techniques*, 1997.

[32] R. Gupta, D. Berson, and J.Z. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *IEEE/ACM 30th International Symposium on Microarchitecture*, 1997.

[33] R. Gupta, D. Berson, and J.Z. Fang. Path profile guided partial redundancy elimination using speculation. In *IEEE International Conference on Computer Languages*, 1998.

[34] Mary Wolcott Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, 1991.

[35] Richard Eugene Hank. *Region-Based Compilation*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[36] Matthew S. Hecht. *Flow analysis of computer programs*. Elsevier North-Holland, New York, N.Y., 1977.

[37] L. Howard Holley and Barry K. Rosen. Qualified data flow problems. *IEEE Trans. on Softw. Eng.*, 1981.

[38] J. Hopcroft. An n log n algorithm for minimizing the states of a finite automaton. *The Theory of Machines and Computations*, pages 189–196, 1971.

[39] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–318, 1977.

[40] G.A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Languages*, pages 194–206, New York, NY, 1973. ACM Press.

[41] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *Symposium on Principles of Programming Languages*, 2000.

[42] J. Larus. Whole program paths. In *SIGPLAN Conference on Programming Languages Design and Implementation*, 1999.

[43] James R. Larus and Eric Schnarr. Eel: Machine-independent executable editing. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 291–300, 1995.

[44] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G.

Holm, and Daniel M. Lavery. The superblock: and effective technique for VLIW and superscaler compilation. *The Journal of Supercomputing*, pages 229–248, 1993.

[45] D. Melski and T. Reps. Interconvertibility of set constraints and contex-free language reachability. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Base Program Manipulation*, pages 74–89, June 1997.

[46] D. Melski and T. Reps. Interprocedural path profiling. In *International Conference on Compiler Construction*. Springer-Verlag, 1999.

[47] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoretical Computer Science*, 248:29–98, 2000. (Invited submission).

[48] F. Mueller and D. B. Whalley. Avoiding unconditional jumps by code replication. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 322–330, 1992.

[49] Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 56–66, 1995.

[50] R. Muth and S. Debray. Partial inlining. (Unpublished technical summary).

[51] Jason R. Patterson. Accurate static branch prediciton by value range propagation. In *SIGPLAN Conference on Programming Languages Design and Implementation*, 1995.

[52] M. Poletto. Path splitting: a technique for improving data flow analysis, 1995.

[53] G. Ramalingam. *Bounded Incremental Computation*. Springer-Verlag, 1996.

[54] G. Ramalingam. Data flow frequency analysis. In *SIGPLAN Conference on Programming Languages Design and Implementation*, New York, NY, May 1996. ACM Press.

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

[56] F.B. Schneider. Enforceable security policies. Technical report, Cornell University, 1998.

[57] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[58] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[59] B. Steensgaard. Points-to analysis in almost-linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[60] T. Way and L. Pollock. Using path spectra to direct function cloning. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 40–47, Oct. 1998.

[61] M.N. Wegman and F.K. Zadeck. Constant propagation with conditional branches. In *Symposium on Principles of Programming Languages*, New York, NY, 1985. ACM Press.

[62] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C.-W. Tseng, M. Hall, M. Lam, , and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12), 1994.

[63] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proc. of ASPLOS-VI*, 1994.

[64] Cliff Young and Michael D. Smith. Better global scheduling using path profiles. In *Proc. of the 31st Annual Intl. Sym. on Microarchitecture*, Dec. 1998.

[65] Reginald Clifford Young. *Path-based Compilation*. PhD thesis, Harvard University, 1998.

[66] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *SIGPLAN Conference on Programming Languages Design and Implementation*, June 2001.

Appendix A

Proof of Theorem 3.4.2

Before restating Theorem 3.4.2, we review some definitions.

Let the graph G and the context-free grammar *CF* be a finite-path graph. Let L be the language described by *CF*. Let the function *numValidComps* take an L-path prefix q in G and return the number of valid completions of q.

Let q be an L-path prefix in G from $Entry$ to a vertex v. Let $w_1, \ldots, w_k$ be the valid successors of the path q. Recall that *edgeValueInContext*$(q, v \rightarrow w_i)$ is defined as follows:

$$\textit{edgeValueInContext}(q, v \rightarrow w_i) = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{j<i} \textit{numValidComps}(q \parallel v \rightarrow w_j) & \text{otherwise} \end{cases} \quad (33)$$

Equation (33) is the same as Equation (5) and is illustrated in Figure 14.

Let p be an L-path through G. Recall that the path number for p is given by the following sum:

$$\sum_{[p' \| v \rightarrow w] \text{ a prefix of } p} \textit{edgeValueInContext}(p', v \rightarrow w) \quad (34)$$

Equation (34) is the same as Equation (6).

We are now ready to restate Theorem 3.4.2:

Theorem 3.4.2 (Dense Numbering of L-paths) *Given the correct definition of the function numValidComps, the Equations (33) and (34) generate a dense numbering of the L-paths through G. That is, for every L-path p through G, the path number of p is a unique value in the range $[0..(\textit{numValidComps}([\epsilon : Entry]) - 1)]$. Furthermore each value in this range is the path number of an L-path through G.* □

The Ball-Larus technique achieves a dense numbering by maintaining the following invariant when assigning values to edges:

Ball-Larus Invariant: For any vertex v, for each path q from v to $Exit$, the sum of the edges in q is a unique number in the range $[0..(\textit{numPaths}[v] - 1)]$.

A consequence of this invariant is that each path from $Entry$ to $Exit$ has a unique path number in the range $[0..(\textit{numPaths}[Entry] - 1)]$.

To prove Theorem 3.4.2, we show that the definition of *edgeValueInContext* given in Equation (33) maintains a similar invariant. We have the following lemma:

Lemma A.0.1 *The definition of edgeValueInContext given by Equation (33) satisfies the following invariant:*

Invariant 1: *For any nonempty L-path prefix p from $Entry$ to a vertex v, let setOfValidComps(p) be the (finite) set of valid completions of p. That is, for every path q in setOfValidComps(p), p concatenated with q (denoted by $p \parallel q$) is an L-path from $Entry$ to $Exit$. Note that numValidComps$(p) =$*

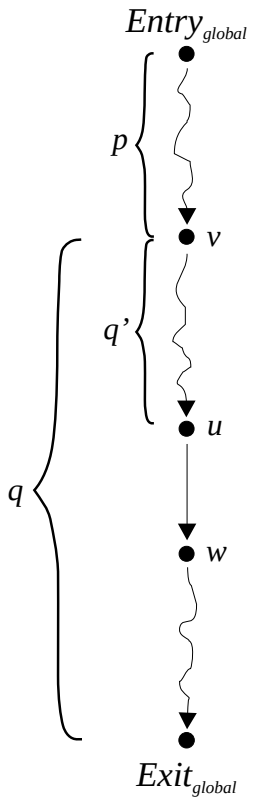


Figure 86: Schematic of the paths referred to in Equation (35). Roughly speaking, for a valid completion q of the path p, the value $\sum_{q' \| u \to w \text{ a prefix of } q}$ ***edgeValueInContext***$(p \parallel q', u \to w)$ is the sum of the "***edgeValueInContext***" values for the edges of q (where each edge e of q is considered with the appropriate context—part of which is supplied by p).

$|$*setOfValidComps*$(p)|$. *Then, for every nonempty path* q *in* *setOfValidComps*(p), *the sum*

$$\sum_{q' \| u \to w \text{ a prefix of } q} \textit{edgeValueInContext}(p \parallel q', u \to w) \tag{35}$$

is a unique number in the range $[0..(\textit{numValidComps}(p)-1)]$. *(Figure 86 shows the paths referred to in Equation (35).)*

That is, for each valid completion q *of the path* p, q *contributes a "unique" value* n *in the range* $[0..(\textit{numValidComps}(p)-1)]$ *to the path number associated with* $[p \parallel q]$. *The value* n *is unique in that for every valid completion* $s \neq q$, *the value that* s *contributes to the path number of* $[p \parallel s]$ *is different from the value that* q *contributes to the path number of* $[p \parallel q]$.

Proof: The proof is by induction on path length (from longest path to shortest path). Let the length of a path p be the number of edges in p, and let *maxLength* be the maximum length of an L-path through G. (Note that it is not possible to have an infinite L-path, since derivations under a context-free grammar must be finite; thus, the fact that there are a finite number of L-paths through G means that there is a bound on the length of L-paths in G.)

For the base case of the induction, we show that for any L-path prefix of length *maxLength*, Invariant 1 is satisfied. An L-path prefix p of length *maxLength* must be an L-path and hence must start at $Entry$ and end at $Exit$. An L-path that ends at $Exit$ has only the empty path as a valid continuation, and hence satisfies Invariant 1 vacuously. It follows that any path of length *maxLength* satisfies Invariant 1.

For the inductive step, suppose that Invariant 1 is satisfied by any L-path prefix of length n that starts at $Entry$, where $1 \leq n \leq$ *maxLength*. Consider an L-path prefix p of length $n-1$ that starts at $Entry$ and ends at a vertex v. If $v = Exit$, then p satisfies Invariant 1 vacuously. Otherwise, let $w_1, \ldots, w_k$ be the valid successors of p. By the inductive hypothesis, for any valid successor w_i of p, $[p \parallel v \to w_i]$ satisfies Invariant 1; that is, for an L-path prefix of the form $[p \parallel v \to w_i]$, for every valid completion q of the path $[p \parallel v \to w_i]$, q contributes a unique value in the range $[0..(\textit{numValidComps}(p \parallel v \to w_i)-1)]$ to the path number of $[p \parallel v \to w_i \parallel q]$. This fact, combined with the definition of *edgeValueInContext*, gives us that any valid completion $[v \to w_i \parallel q]$ of the path p will contribute to the path number of $[p \parallel v \to w_i \parallel q]$ a unique value in the range:

$$\begin{aligned}&[\textit{edgeValueInContext}(p, v \to w_i)..\\ &\quad (\textit{edgeValueInContext}(p, v \to w_i) + \textit{numValidComps}(p \parallel v \to w_i) - 1)]\end{aligned}$$

By Equation (33), this is equal to the following range:

$$\left[\left(\sum_{j<i} \textit{numValidComps}(p \parallel v \to w_j)\right) .. \left(\left(\sum_{j<i} \textit{numValidComps}(p \parallel v \to w_j)\right) + \textit{numValidComps}(p \parallel v \to w_i) - 1\right)\right]$$

Because this holds for each successor w_i, $1 \leq i \leq k$, every valid completion of p contributes a unique value in the range

$$\left[0.. \left(\left(\sum_{i=1}^{k} \textit{numValidComps}(p \parallel v \to w_i)\right) - 1\right)\right] = [0..(\textit{numValidComps}(p) - 1)]$$

(see Figure 14). It follows that Invariant 1 holds for the path p.

In other words, the definition of *edgeValueInContext* works for the same reason that the Ball-Larus edge-numbering scheme works—for each valid successor w_i of p, a range of numbers is "reserved" for valid completions of p that start with $v \to w_i$.

Consequently, the definition of *edgeValueInContext* in Equation (33) satisfies Invariant 1. □

Theorem 3.4.2 is a consequence of Lemma A.0.1.

Appendix B

Runtime Environment for Collecting an Interprocedural, Context Path Profile

In this section, we describe the instrumentation code that is introduced to collect an interprocedural path profile. The instrumentation for a program $\mathcal{P}$ is based on the graph G^*_{fin} that is constructed for $\mathcal{P}$ as described in Section 3.2. In essence, the instrumentation code threads the algorithm described in Section 3.5.5 into the code of the instrumented program. Thus, the variables `pathNum` and `numValidCompsFromExit` become program variables. There is no explicit stack variable corresponding to `NVCstack`; instead, the program's execution stack is used. The variable `pathNum` and procedure parameter `numValidCompsFromExit` play the following roles in the instrumentation code (see Figures 12 and 13 for a concrete example):

`pathNum`: `pathNum` is a local variable of *main* that is passed by reference to each procedure $P \neq$ *main*. It is used to accumulate the path number of the appropriate path in G^*_{fin}. As execution proceeds along the edges of the supergraph G^* of $\mathcal{P}$, the value of `pathNum` is updated. The profile is updated with the value in `pathNum` at appropriate places (*e.g.*, before an intraprocedural backedge of G^* is traversed).

`numValidCompsFromExit`: Each procedure $P \neq$ *main* is modified to take an additional parameter `numValidCompsFromExit` that is passed by value. This parameter is used to tell the instrumentation code in P the number of valid completions from $Exit_P$ for the path in G^*_{fin} that was used to reach P. The value in `numValidCompsFromExit` is used with the ρ functions to compute edge values for the edges of P.

Given a program $\mathcal{P}$ and the graphs G^* and G^*_{fin} that are associated with $\mathcal{P}$, the modifications described below are made to $\mathcal{P}$ in order to instrument it to collect an interprocedural context path profile. (As before, the ordered pair $\langle a, b\rangle$ denotes the linear function $\lambda x.a \cdot x + b$.) We use C++ terminology and syntax in the example instrumentation code.

1. A global declaration of the array `profile` is added to the program; `profile` is an array of `unsigned longs` that has *numValidComps*$([\epsilon : Entry_{global}])$ elements, *i.e.*, one for each unbalanced-left path through G^*_{fin}.[1]

2. Code is added to the beginning of *main* to initialize each element of `profile` to 0. Code is added just before *main* exits to output the contents of `profile`. This output constitutes the profile.

3. Declarations for the variables `pathNum`, `pathNumOnEntry`, `pathNumBeforeCall`, and `numValidCompsFromExit` are added to the beginning of procedure *main*. `pathNum` is an

[1] In practice it is likely that a hash table would be used in place of the array `profile`.

`unsigned long`[2] and is initialized to the value that is calculated for *edgeValueInContext*($[\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_{main}$); see Section 3.5.3. `pathNumOnEntry` is an `unsigned long` and is initialized to the same value as `pathNum`. `pathNumBeforeCall` is an `unsigned long` that is used to save the current value of `pathNum` before a recursive call is made. `numValidCompsFromExit` is an `unsigned long` initialized to 1. (Note that `pathNumOnEntry` and `numValidCompsFromExit` are somewhat redundant in *main*; they are added for consistency with the other procedures.)

4. For each procedure P such that $P \neq$ *main*, P is modified to accept the following additional parameters:

```
unsigned long &pathNum                /* passed by reference */
unsigned long numValidCompsFromExit   /* passed by value */
```

That is, a function prototype of the form

```
return_type func(...params...);
```

becomes

```
return_type func(...params...,
                 unsigned long &pathNum,
                 unsigned long numValidCompsFromExit);
```

5. For each procedure $P \neq$ *main*, the declarations

```
unsigned long pathNumOnEntry = pathNum;
unsigned long pathNumBeforeCall;
```

are added to the declarations of P's local variables.

6. Each nonrecursive procedure call is modified to pass additional arguments as follows: Let the vertices c and r represent the following nonrecursive procedure call:

```
t = func(...args...);
```
(36)

Let $\psi_r = \langle a, b \rangle$. Then the function call in (36) is replaced by the following call:

```
t = func(...args..., pathNum, a * numValidCompsFromExit + b);
```

7. Each recursive procedure call is modified as follows: Let the vertices c and r represent the following recursive procedure call from the procedure P to the function *func*:

```
t = func(...args...);
```
(37)

[2] In practice, it is possible that the number of bits needed to represent a path number will not fit in an `unsigned long`. In this case, instead of using `unsigned long`s it may be necessary to use a `Counter` class and `Counter` objects to represent path numbers. The `Counter` class must behave like an `unsigned long` but be able to handle arbitrarily large integers.

Let x denote the value of *edgeValueInContext*$([\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_{func})$ (see Section 3.5.3). Then the procedure call in (37) is replaced by the following code:

```
/* A: */ pathNumBeforeCall = pathNum;
/* B: */ pathNum = x;
/* C: */ t = func(...args..., pathNum, 1);
/* D: */ profile[pathNum]++;
/* E: */ pathNum = pathNumBeforeCall;
```

The line labeled "A" saves the value of `pathNum` for the path that is being recorded before the recursive call is made. The lines "B" and "C" set up the instrumentation in `func` to start recording a new path number for the path that begins with the edge $Entry_{global} \rightarrow Entry_{func}$; the line "C" also makes the original procedure call in (37). The line "D" updates the profile with the unbalanced-left path in G^*_{fin} that ends with the edge $Exit_{func} \rightarrow Exit_{global}$. The line "E" restores `pathNum` to the value that it had before the recursive call was made, indicating that the instrumentation process resumes with the path prefix $[p \parallel c \rightarrow r]$, where p is the path taken to c.

8. For each intraprocedural edge $v \rightarrow w$ that is not a backedge, code is inserted so that as the edge is traversed, `pathNum` is incremented by the value $\rho_{v \rightarrow w}$(`numValidCompsFromExit`). For example, consider the following if statement (v, w_1, and w_2 are labels from vertices in G^*_{fin} that correspond to the indicated pieces of code):

```
v :if( ... ) {
     w1 : ...
   } else {                    (38)
     w2 : ...
   }
```

   Let $\rho_{v \rightarrow w_1} = \langle a, b \rangle$ and $\rho_{v \rightarrow w_2} = \langle c, d \rangle$. Then the if statement given in (38) is replaced by

```
v :if( ... ) {
      pathNum += a * numValidCompsFromExit + b;
   w1 : ...
   } else {
      pathNum += c * numValidCompsFromExit + d;
   w2 : ...
   }
```

   Note that one of $\langle a, b \rangle$ and $\langle c, d \rangle$ will be the function $\langle 0, 0 \rangle$; clearly, no code needs to be added for an edge labeled with the function $\langle 0, 0 \rangle$.

9. For each intraprocedural edge $w \rightarrow v$ in procedure P that is a backedge, code is inserted that updates the profile for one unbalanced-left path and then begins recording the path number for a new unbalanced-left path. For a example, consider the following while statement (v and w represent labels from vertices in G^*_{fin} that correspond to the indicated pieces of code):

```
v :while( ... ){
     ...
   w : /* source vertex of backedge */      (39)
  }
```

In G^*_{fin}, the backedge $w \to v$ has been replaced by the edges $Entry_P \to v$ and $w \to GExit_P$. Let $\rho_{Entry_P \to v} = \langle a, b \rangle$. In this example, $\rho_{w \to GExit_P} = \langle 0, 0 \rangle$ (because the surrogate edge $w \to GExit_P$ is the only edge out of w in G^*_{fin}). The while statement in (39) is replaced by

```
v :while( ...  ){
     ...
   w : /* source vertex of backedge */
    /* A: */ profile[pathNum]++;
    /* B: */ pathNum=pathNumOnEntry+
        a*numValidCompsFromExit+b;
}
```

The line labeled "A" updates the profile for the unbalanced-left path of G^*_{fin} that ends with $w \to GExit_P \to Exit_{global}$, and the line labeled "B" starts recording a new path number for the unbalanced-left path p that consists of a context-prefix that ends at $Entry_P$ and an active-suffix that begins at v. (The context-prefix is established by the value of `pathNum` on entry to P, which has been saved in the variable `pathNumOnEntry`.)

For a second example, consider the following do-while statement (v, w and x represent labels from vertices in G^*_{fin} that correspond to the indicated pieces of code):

```
                          v :do {
                              ...
                  w : } while( /* test */ );
                          x :...
```
(40)

In G^*_{fin}, the backedge $w \to v$ has been replaced by the edges $Entry_P \to v$ and $w \to GExit_P$. Let $\rho_{Entry_P \to v} = \langle a, b \rangle$, $\rho_{w \to GExit_P} = \langle c, d \rangle$, and $\rho_{w \to x} = \langle e, f \rangle$. Then the do-while in (40) is replaced by

```
v :do {
    ...
   w :if( /* test */ ){
     /* A: */ ++profile[pathNum + c * numValidCompsFromExit + d];
     /* B: */ pathNum = pathNumOnEntry +
                          a * numValidCompsFromExit + b;
     /* C: */ continue;
    } else {
     /* D: */ pathNum += e * numValidCompsFromExit + f;
     break;
    }
  } while( 0 );
x :...
```

The line labeled "A" updates the profile for the unbalanced-left path of G^*_{fin} that ends with $w \to GExit_P \to Exit_{global}$. The line labeled "B" starts recording a new path number for the unbalanced-left path p that consists of a context-prefix that ends at $Entry_P$ and an active-suffix that begins at v. The line "D" updates `pathNum` using the function $\rho_{w \to x}$. Again, one of $\langle c, d \rangle$ and $\langle e, f \rangle$ will be $\langle 0, 0 \rangle$, and no code to update `pathNum` need be included for the function $\langle 0, 0 \rangle$.

Appendix C

Proofs for Theorems in Chapter 9

The Supergraph Partitioning Algorithm can be used by the Ammons-Larus Reduction Algorithm in place of the Coarsest Partitioning Algorithm. This results in a reduction algorithm for hot-path supergraphs. The steps of this new algorithm are (where H^*, *pp*, and J are the input hot-path supergraph, path profile, and data-flow solution, respectively):

1. Determine which vertices of the hot-path supergraph H^* are hot.
2. Create a compatibility partition π of the vertices of H^*.
3. Run the Supergraph Partitioning Algorithm on H^* and π to produce the partition π'.
4. Output a new graph H'^*, path profile pp', and data-flow solution J':
   - H'^* contains one vertex s_i for each block C_i in π'. H'^* contains an edge $s_i \rightarrow s_j$ if and only if H^* has an edge $u \rightarrow v$ such that $u \in C_i$ and $v \in C_j$. H'^* has a call-edge $s_i \rightarrow s_j$ labeled "$(_{s_i}$" if and only if H^* has a call-edge $c \rightarrow e$ labeled "$(_c$" such that $c \in C_i$ and $e \in C_j$. H'^* has a return-edge $s_i \rightarrow s_j$ labeled "$)_{s_k}$" if and only if H^* has a return edge $x \rightarrow r$ labeled "$)_c$" such that $x \in C_i$, $r \in C_j$, and $c \in C_k$.
   - pp' is *pp* translated onto $H'*$ by replacing each vertex v that appears in a path of *pp* with v's representative in H'^*; thus, if $v \in C_i$, it is translated to s_i.
   - J' for vertex s_i is defined by $J'(s_i) = \sqcap_{v \in C_i} J(v)$.

The output from this algorithm (H'^*, pp', and J') is valid output for a hot-path reduction algorithm and J' preserves the valuable data-flow facts of J. These statements are restated as Theorems 9.3.1 and 9.3.2 below, though we first state an observation and a lemma. In the following lemma and theorems, we will use H^*, *pp*, J, $\pi' = \{C_1, C_2, \ldots C_{n'}\}$, H'^*, pp' and J' as they are defined in the above algorithm.

Observation C.0.2 (Well-formedness Properties) *The hot-path supergraph H^* has the following well-formedness properties:*

1. *For any pair of duplicate vertices u_1 and u_2 that are not exit vertices, for every edge $u_1 \rightarrow v_1$, there is a congruent edge $u_2 \rightarrow v_2$.*
2. *Every call-edge $c \rightarrow e$ is labeled "$(_c$".*
3. *For every call-edge $c \rightarrow e$ labeled "$(_c$", for every exit vertex x where there is a same-level valid path from e to x, there is exactly one return-edge $x \rightarrow r$ labeled "$)_c$".*
4. *For every return-edge $x \rightarrow r$ labeled "$)_c$" there must be an entry vertex e such that there is a call-edge $c \rightarrow e$ labeled "$(_c$" and a same-level valid path from e to x. This means that the return-edge $x \rightarrow r$ labeled "$)_c$" implies that there is a summary-edge $c \rightarrow r$.*

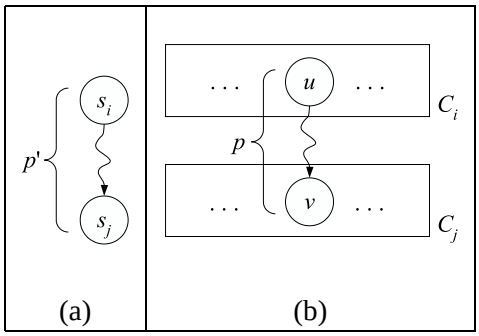


Figure 87: Visual interpretation of Lemma C.0.3. (a) shows an unbalanced-left path p' in H'^* that starts at s_i and ends at s_j. (b) shows an unbalanced-left path p in H^* that starts at $u \in C_i$ and ends at $v \in C_j$ where C_i is the block represented by s_i and C_j is the block represented by s_j. Lemma C.0.3 states that given the path p' and the vertex u in block C_i, the path p must exist.

5. *For every return-edge $x \to r$ labeled "$)_c$", r is a return-site vertex for the call vertex c.*

This observation is offered without proof, as the above properties follow directly from the construction of the hot-path supergraph.

Lemma C.0.3 *For any unbalanced-left path p' in H'^* from vertex s_i to vertex s_j, for any vertex $u \in C_i$, there is an unbalanced-left path p in H^* from u to v such that p is congruent to p' and $v \in C_j$. (See Figure 87.)*

Proof: The proof is by induction on the length (in edges) of p':

Base case: p' is the empty path from s_i to s_i (and has length 0). Let $v \in C_i$ be any vertex in C_i. Then the empty path p from v to v is an unbalanced-left path in H^* that is congruent to p' and ends at a vertex (namely v) in C_i.

Inductive step: Assume for the induction hypothesis that for any unbalanced-left path p' in H'^* of length less than n (where $n > 0$), the lemma is satisfied. Let q' be an unbalanced-left path in H'^* from s_i to s_j that is of length n. There are two possibilities:

1. The last edge of q' is *not* a return-edge. Then $q' = [a' \| s_k \to s_j]$ where a' is an unbalanced-left path from s_i to s_k and $s_k \to s_j$ is not a return-edge. Let u be an arbitrary vertex in C_i. We must show that there is an unbalanced-left path q in H^* from the vertex u to some vertex w such that q is congruent to q' and $w \in C_j$.

   The path a' has length $(n-1)$. By the induction hypothesis, there is a path a in H^* from $u \in C_i$ to $v \in C_k$ such that a is congruent to a'. If we can find an edge $v \to w$ such that $w \in C_j$, then the path $[a \| v \to w]$ is the desired path that is congruent to q'.

   Since H'^* contains the edge $s_k \to s_j$, it follows from the construction of H'^* that there is an edge $v_0 \to w_0$ in H^* where $v_0 \in C_k$ and $w_0 \in C_j$. The fact that $v, v_0 \in C_k$ means that v and v_0 are duplicate vertices. The vertices v_0 and v are not exit vertices, since $s_k \to s_j$ is not a return-edge. By the well-formedness of H^* (see Observation C.0.2), there must be an edge $v \to w$ that is congruent to $v_0 \to w_0$. Since $v, v_0 \in C_k$, it follows from the definition of the partition π' that $w, w_0 \in C_j$. Thus, the path $[a \| v \to w]$ is an unbalanced-left path in H^* from $u \in C_i$ to $w \in C_j$. Furthermore, $[a \| v \to w]$ is congruent to q'.

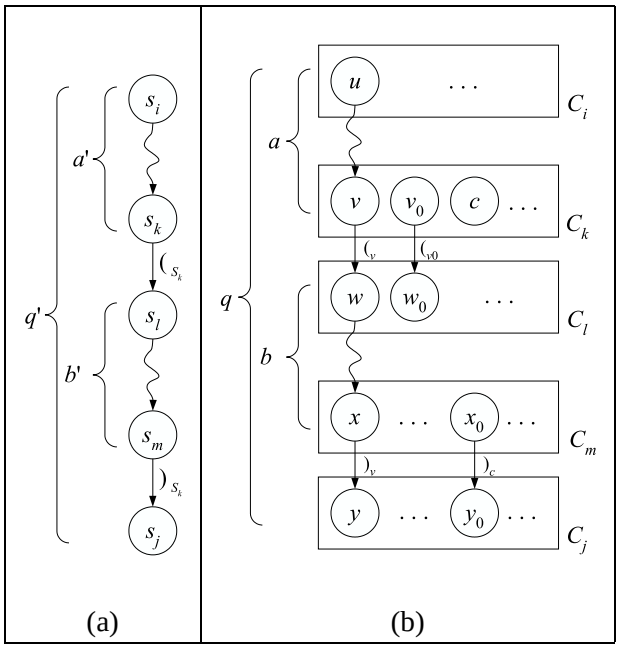


Figure 88: Visualization of Case II of the proof of Lemma C.0.3. (a) shows the unbalanced-left path $q' = [a' \| s_k \to s_\ell \| b' \| s_m \to s_j]$ in H'^*. (b) shows blocks C_i, C_j, C_k, C_ℓ, and C_m of partition π' and the unbalanced-left path $q = [a \| v \to w \| b \| x \to y]$ in H^*. The path q is constructed from the path q' as follows: the path a' yields a. The edge $s_k \to s_\ell$ yields the edge $v_0 \to w_0$, which gives the edge $v \to w$. The path b' implies the existence of the path b. The edge $s_m \to s_j$ labeled "$)_{s_k}$" yields the edge $x_0 \to y_0$ labeled "$)_c$", where $c \in C_k$; the edge $x_0 \to y_0$ labeled "$)_c$" implies the existence of the edge $x \to y$ labeled "$)_v$".

2. The last edge of q' is a return-edge. Then $q' = [a' \| s_k \to s_\ell \| b' \| s_m \to s_j]$ where a' is an unbalanced-left path from s_i to s_k, $s_k \to s_\ell$ is a call-edge labeled "$(_{s_k}$", b' is a same-level valid path from s_ℓ to s_m, and $s_m \to s_j$ is a return-edge labeled "$)_{s_k}$". (This follows from the definition of an unbalanced-left path; $s_k \to s_\ell$ and $s_m \to s_j$ are matching call and return edges.) Figure 88(a) shows a diagram of the path q'.

   Let u be an arbitrary vertex in C_i. We will show that there is an unbalanced-left path $q = [a \| v \to w \| b \| x \to y]$ in H^* from the vertex u to some vertex y such that q is congruent to q' and $y \in C_j$. Figure 88(b) shows a diagram of the path q' and summarizes the argument given below.

   Since the length of a' is less than n, by the induction hypothesis, there is an unbalanced-left path a in H^* from u to v such that a is congruent to a' and $v \in C_k$.

   Since H'^* contains a call edge $s_k \to s_\ell$, H^* must contain a call edge $v_0 \to w_0$ where $v_0 \in C_k$ and $w_0 \in C_\ell$. Since $v, v_0 \in C_k$, it follows from the well-formedness of H^* (see Observation C.0.2) that there is a call-edge $v \to w$ labeled "$(_v$". The definition of π' guarantees that w is in the same block as w_0, namely C_ℓ.

   By the induction hypothesis, there is an unbalanced-left path b in H^* from $w \in C_\ell$ to $x \in C_m$ such that b is congruent to b'. Since, b is congruent to b', b and b' must have the same number of open and closed parentheses, and the parentheses must occur in the same positions. Since b' is a same-level valid path, and b is an unbalanced-left path, it follows that b is also a same-level valid path. (Recall that the set of same-level valid paths is a subset of the set of unbalanced-left paths.)

   Given the call-edge $v \to w$ labeled "$(_v$" and the same-level valid path b from w to x, Observation C.0.2 states that H^* must have a return edge $x \to y$ labeled "$)_v$". The fact that H'^* has a return-edge $s_m \to s_j$ labeled "$)_{s_k}$" means that H^* must have a return-edge $x_0 \to y_0$ labeled "$)_c$" such that $x_0 \in C_m$, $y_0 \in C_j$, and $c \in C_k$. Since $x, x_0 \in C_m$ and $v, c \in C_k$, it follows from the definition of π' that $y, y_0 \in C_j$.

   Thus, the path $q = [a \| v \to w \| b \| x \to y]$ is an unbalanced-left path in H^* from $u \in C_i$ to $y \in C_j$. Furthermore, this path is congruent to q'. (See Figure 88.)

QED □

Theorem 9.3.1 *H^* and H'^* are unbalanced-left path congruent.*

Proof: We must show that for any unbalanced-left path in H^* there is a congruent unbalanced-left path in H'^* and vice-versa.

Let p be an unbalanced-left path in H^*. Define p' to be the path in H'^* formed by replacing each edge $u \to v$ of p with the edge $s_i \to s_j$, where $u \in C_i$ and $v \in C_j$. If the edge $u \to v$ of p is labeled "$(_c$" (or "$)_c$") where $c \in C_k$, then the corresponding edge $s_i \to s_j$ is labeled "$(_{s_k}$" (or "$)_{s_k}$"). Clearly, p' is congruent to p. This implies that p' must also be an unbalanced-left path. For any call-edge $c \to e$ labeled "$(_c$" and matching return-edge $x \to r$ labeled "$)_c$" that appear in p, it must be the case that the call-edge $s_i \to s_j$ that corresponds to $c \to e$ and return-edge $s_k \to s_m$ that corresponds to $x \to r$ are also matching: $s_i \to s_j$ is labeled "$(_{s_i}$" and $s_k \to s_m$ is labeled "$)_{s_i}$". Also, the number of call and return edges in p' is the same as in p. Thus, p' is unbalanced-left because it is congruent to p and p is unbalanced-left.

Let q' be an unbalanced-left path in H'^*. By Lemma C.0.3, there is a congruent, unbalanced-left path q in H^*. *QED* □

Theorem 9.3.2 *J' is a valid data-flow solution (i.e., it approximates the meet-over-all valid paths solution) for the graph H'^*.*

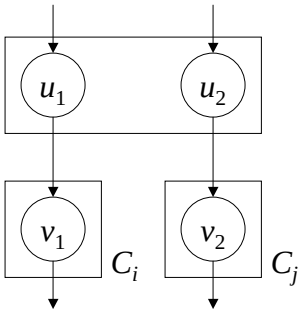


Figure 89: A violation of the second property of the Supergraph Partitioning Algorithm. The vertices u_1 and u_2 must be split.

Proof: Assume, on the contrary, that J' is not a valid data-flow solution for H'^*. Then there must be some vertex a_k and some unbalanced-left path $p' = Entry'_{global} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots a_{k-1} \rightarrow a_k$ such that

$$J'(a_k) \sqsupset T_{a_k}(T_{a_{k-1}}(\ldots T_{a_2}(T_{a_1}(\bot))\ldots)) \tag{41}$$

where T_i is the data-flow transfer function for a_i. Let C be the block of partition π' such that a_k represents C in H'^*. By Lemma C.0.3, there is some unbalanced-left path p from $Entry_{global}$ of H^* to a vertex u in H^* such that p is congruent to p' and $u \in C$. Since p is congruent to p', the sequence of data-flow transfer functions along p is the same as the sequence of data-flow transfer functions along p'. Thus, we have

$$J(u) \sqsubseteq T_{a_k}(T_{a_{k-1}}(\ldots T_{a_2}(T_{a_1}(\bot))\ldots))$$

However, we also have $J'(a_k) \sqsubseteq J(u)$, since by definition $J(a_k) = \bigsqcap_{v \in C} J(v)$. This implies that

$$J'(a_k) \sqsubseteq T_{a_k}(T_{a_{k-1}}(\ldots T_{a_2}(T_{a_1}(\bot))\ldots))$$

which contradicts (41). *QED* □

We now turn to proving the correctness of the Supergraph Partitioning Algorithm. We have the following theorem: **Theorem 9.3.4** *When the Supergraph Partitioning Algorithm is run on a hot-path supergraph H^* and a partition $\pi = B_1, B_2, \ldots, B_n$ of the vertices of H^*, the output partition $\pi' = \{C_1, C_2, \ldots, C_{n'}\}$ satisfies the properties of the Supergraph Partitioning Algorithm listed in Section 9.3.1.*

Proof: Suppose on the contrary that the partition π' violates one (or more) of the properties listed in Section 9.3.1. There are three possibilities:

1. Suppose the first property is violated. In this case, there must be some $C_i \in \pi'$ such that there is no $B_j \in \pi$ such that $C_i \subseteq B_j$. This cannot happen: every time the Supergraph Partitioning Algorithm creates a block C_i, that block is a subset of some block $B_j \in \pi$.

2. Suppose the second property is violated. In this case, there must be a pair of congruent edges $u_1 \rightarrow v_1$ and $u_2 \rightarrow v_2$ such that $u_1 \rightarrow v_1$ and $u_2 \rightarrow v_2$ are not return or summary-edges, u_1 and u_2 are in the same block of π', and v_1 and v_2 are in different blocks of π' (see Figure 89). There are two possibilities:

   (a) v_1 and v_2 start in the same block B of partition π. There must be some step of the algorithm where v_1 and v_2 are split. This means that at some point, the algorithm creates the blocks

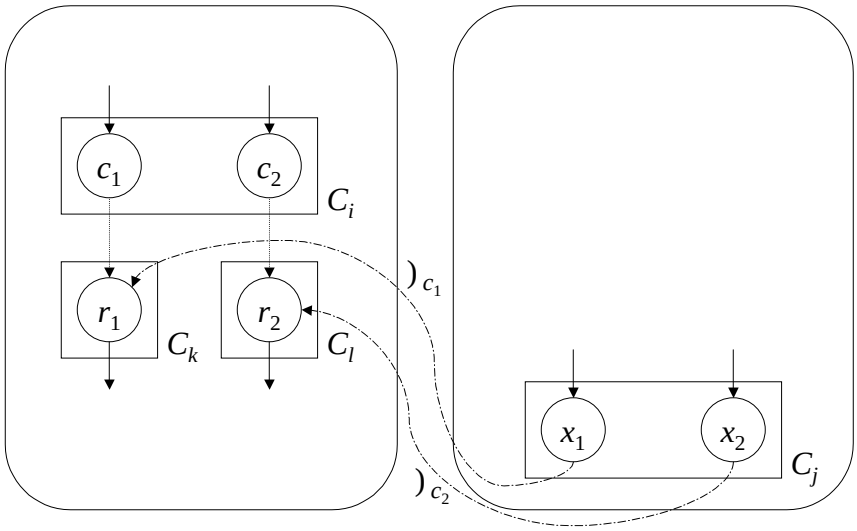


Figure 90: A violation of the third property of the Supergraph Partitioning Algorithm. The return-edges $x_1 \to r_1$ and $x_2 \to r_2$ are in conflict. Either c_1 and c_2 should be split or x_1 and x_2 should be split.

C_i and C_j where $v_1 \in C_i$, $v_2 \in C_j$, $C_i \subseteq B$, $C_j \subseteq B$, and $C_i \cap C_j = \emptyset$. This may happen in one of the functions $SplitPreds$, $RepartitionCallBlock$, or $RepartitionExitBlock$. No matter where it happens, one or both of C_i and C_j are put on the worklist $W_{SplitPreds}$. When either of these blocks is removed from $W_{SplitPreds}$ and processed by $SplitPreds$, the vertices u_1 and u_2 will be split into separate blocks.

(b) v_1 and v_2 start in different blocks of π. Let v_1 start in block $B \in \pi$. The block B is put on $W_{SplitPreds}$ at the beginning of the algorithm. When B is removed from $W_{SplitPreds}$ and processed by the function $SplitPreds$, the function $SplitPreds$ will determine that u_1 and u_2 must be split.

3. Suppose the third property is violated. In this case, there must be a pair of conflicting return edges. Let the return-edge $x_1 \to r_1$ labeled "$)_{c_1}$" and the return-edge $x_2 \to r_2$ labeled "$)_{c_2}$" be the return-edges in conflict, where $x_1, x_2 \in C_i$, $c_1, c_2 \in C_j$, $r_1 \in C_k$, and $r_2 \in C_\ell$ (see Figure 90). Recall that the edge $x_1 \to r_1$ labeled "$)_{c_1}$" implies that there is a summary-edge $c_1 \to r_1$. Similarly, the edge $x_2 \to r_2$ labeled "$)_{c_2}$" implies that there is a summary-edge $c_2 \to r_2$. We must consider two possibilities:

   (a) The return-site vertices r_1 and r_2 started in the same block $B \in \pi$. In this case, at some point the function $SplitPreds$ must split r_1 and r_2 into the distinct blocks C'_k and C'_ℓ where $r_1 \in C'_k$, $r_2 \in C'_\ell$, $C_k \subseteq C'_k \subseteq B$, and $C_\ell \subseteq C'_\ell \subseteq B$. When this happens, $SplitPreds$ will put either C'_k or C'_ℓ onto $W_{SplitPreds}$. Without loss of generality, let us assume that C'_k is put on $W_{SplitPreds}$. When $SplitPreds$ later takes the block C'_k from $W_{SplitPreds}$ and processes it, the return-edge $x_1 \to r_1$ causes $SplitPreds$ to put the block C'_i containing x_1 onto $W_{Repartition}$, where $C_i \subseteq C'_i$. Also, the summary-edge $c_1 \to r_1$ will cause $SplitPreds$to put the block C'_j containing c_1 onto $W_{Repartition}$, where $C_j \subseteq C'_j$. After both C'_i and C'_j have been removed from $W_{Repartition}$ and repartitioned, it must be the case that x_1 and x_2 have been split or c_1 and c_2 have been split, or both. This contradicts the above assumptions.

(b) The return-site vertices r_1 and r_2 started in different blocks of π. Let r_1 start in block B. The block B is put on $W_{SplitPreds}$ at the beginning of the algorithm. When $SplitPreds$ processes B, it will put the blocks C'_i and C'_j onto $W_{Repartition}$, where $x_1 \in C'_i$, $C_i \subseteq C'_i$, $c_1 \in C'_j$, and $C_j \subseteq C'_j$. This happens because of the return-edge $x_1 \rightarrow r_1$ and the summary-edge $c_1 \rightarrow r_1$. After the blocks C'_i and C'_j have been repartitioned, the return-edge conflict will be resolved. This contradicts the above assumptions.

Thus, the output partition must have the properties listed in Section 9.3.1. □

Appendix D

Proofs for Theorems in Chapter 10

To prove Theorem 10.4.5, we must first show the correctness of a modified version of the Vertex Subsumption Algorithm that does not add return-edges.

Lemma D.0.4 *Let the Simple Vertex Subsumption Algorithm be the same as the Vertex Subsumption Algorithm in Figure 67, but without the call to* $\mathrm{AddRtnEdges}$ *(line 15 of Figure 67). Then every vertex subsumption fact* $v' \succ v$ *output by the Simple Vertex Subsumption Algorithm is correct. However, the Simple Vertex Subsumption Algorithm may conclude that* $v' \not\succ v$ *when in fact* $v' \succ v$*.*

Proof: We begin by proving that if the Simple Vertex Subsumption Algorithm concludes that $v' \succ v$, then this is, in fact, the case. Suppose, on the contrary, that this is not the case; *i.e.*, in fact, $v' \not\succ v$. This means that there is some unbalanced-right path p from v to $Exit_{global}$ such that there is no path p' from v' that subsumes p ($p' \succ p$). If p is of length 0, then Observation 10.4.1 tells us that $J(v') \not\supseteq J(v)$. But if this were the case, then the Simple Vertex Subsumption Algorithm would conclude at line 3 (see Figure 67) that $v' \not\succ v$. This contradicts our assumption. So, p must have a length of at least 1.

Let $p = [q \| w \rightarrow y]$. Observation 10.4.1 states that there are three cases to consider:

1. There is a path q' from v' to w' such that $q' \succ q$, but there is no edge $w' \rightarrow y'$ congruent to $w \rightarrow y$. The vertices w and w' are duplicates. If they are not exit vertices, then the edge $w \rightarrow y$ implies that there must be a congruent edge $w' \rightarrow y'$. This follows from the well-formedness of H^* (see Observation C.0.2).

   Thus, it must be the case that w and w' are exit vertices and that $w \rightarrow y$ is a return-edge. Furthermore, the paths q and q' must be unbalanced-right paths. Suppose, to the contrary, that this were not the case. Then there must be some unbalanced-left suffix of q. This means that the label "$)_c$" must match the last unmatched call-edge $c \rightarrow e$ in the unbalanced-left suffix of q. Also, there must be an unbalanced-left suffix of q'. By the third well-formedness principle in Observation C.0.2,

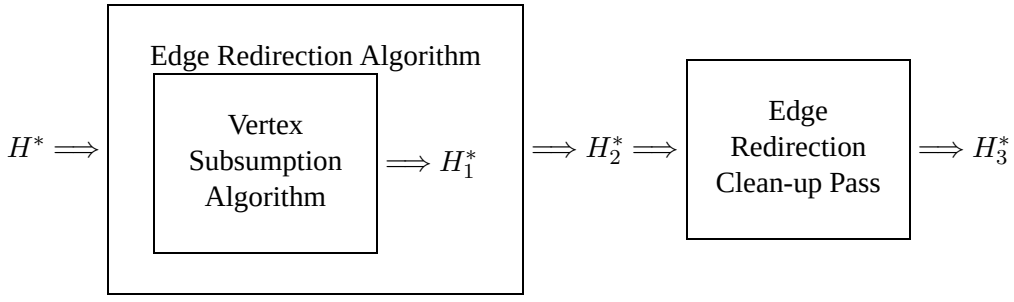


Figure 91: Stages used for minimizing a graph using edge redirection. The Edge Redirection Algorithm calls the Vertex Subsumption Algorithm as a subroutine, so the graph H_1^* is a temporary data structure used by the Edge Redirection Algorithm. (This Figure is reprinted from Chapter 10.)

there must be a return-edge $w' \rightarrow y'$ with a label "$)_{c'}$" that matches the last unmatched call-edge $c' \rightarrow e'$ in q'. By the fifth well-formedness principle in Observation C.0.2, it must be the case that y' is a return-vertex for c' and that y is a return-vertex for c. Since q and q' are congruent, c and c' must be duplicate vertices. This would imply that y and y' are congruent. Since w and w' are congruent, this means that $w \rightarrow y$ and $w' \rightarrow y'$ are congruent. So, q and q' must be unbalanced-right paths, otherwise there is an edge $w' \rightarrow y'$ that is congruent to $w \rightarrow y$.

We have that $w \rightarrow y$ is a return-edge and that there is no congruent return-edge $w' \rightarrow y'$. This means that the Vertex Subsumption Algorithm will conclude (in lines 31–32 of Figure 68) that $w' \not\succ w$. Since q and q' are congruent, unbalanced-right paths, the Vertex Subsumption Algorithm will propagate this non-subsumption fact (in lines 17–25 of Figure 67) to v' and v and conclude that $v' \not\succ v$, which contradicts our initial assumption.

2. There is a path q' from v' to w' such that $q' \succ q$, but $w \rightarrow y$ is a return-edge that is unmatched in p and there is no return-edge from w' with the same label as $w \rightarrow y$. In this case, the Vertex Subsumption Algorithm will conclude (in lines 31–32 of Figure 68) that $w' \not\succ w$. Since $w' \rightarrow y'$ is a return-edge that is unmatched in p, the paths p, q, and q' must all be unbalanced-right. (The path q is unbalanced-right because it is a prefix of p; the path q' is unbalanced-right because it is congruent to q.) This means that the non-subsumption fact $w' \not\succ w$ will be propagated by lines 17–25 of Figure 67 to the vertices v' and v. The Vertex Subsumption Algorithm will conclude that $v' \not\succ v$, which contradicts our original assumption.

3. There is a path p' from v' to y' such that p' exactly mimics p's control flow, but $J(y') \not\supseteq J(y)$. In this case, the Vertex Subsumption Algorithm will conclude at line 3 of Figure 67 that $y' \not\succ y$. Lines 8–14 and 17–25 of Figure 67 will propagate the non-subsumption fact $y' \not\succ y$ across the unbalanced-right-left paths p' and p and conclude that $v' \not\succ v$.

Thus, if the Simple Vertex Subsumption Algorithm concludes that $v' \succ v$ it must be the case that $v' \succ v$.

However, the Simple Vertex Subsumption Algorithm may conclude that $v' \not\succ v$ when in fact $v' \succ v$. This is because the Simple Vertex Subsumption Algorithm propagates non-subsumption facts across summary-edges (at lines 10–14 and 19–25). Given the summary-edges $c \rightarrow r$ and $c' \rightarrow r'$ and the non-subsumption fact $r' \not\succ r$, the Simple Vertex Subsumption Algorithm will conclude that $c' \not\succ c$. The summary-edge $c \rightarrow r$ implies that there is a same-level valid path q from c to r. Likewise, the summary-edge $c' \rightarrow r'$ implies that there is a same-level valid path q' from c' to r'. If the paths q and q' are congruent, then the non-subsumption fact $r' \not\succ r$ should be propagated to $c' \not\succ c$. However, q and q' need not be congruent. If for every same-level valid path from c to r there is no congruent path from c' to r', then the non-subsumption fact $r' \not\succ r$ should not be propagated across the summary-edges $c' \rightarrow r'$ and $c \rightarrow r$. (In fact, lines 12 and 23 of Figure 67 contain a check that $u_1 \neq u_2$; these checks are to prevent the case when a non-subsumption fact is incorrectly propagated across summary-edges to generate the non-subsumption fact $u_1 \not\succ u_1$, which is clearly false, since it violates the reflexivity of the $\succ$-relation.)

The Simple Vertex Subsumption is always correct when it concludes $v' \succ v$, but it may be incorrect when it concludes that $v' \not\succ v$. *QED* □

Theorem 10.4.5 *Let H^*be the hot-path graph input to the Vertex Subsumption Algorithm and let H_1^* be the transformed graph output by the Vertex Subsumption Algorithm. That is, H_1^* is $H^*$with the additional return-edges added by lines 39–49 of Figure 69. Every vertex subsumption assertion $v_1' \succ v_1$ output by the Vertex Subsumption Algorithm is correct for the graph H_1^*. However, the Vertex Subsumption Algorithm may conclude that $v_1' \not\succ v_1$ when in fact $v_1' \succ v_1$.*

Proof: The proof relies heavily on the proof of Lemma D.0.4. Suppose that we add any set of (randomly chosen) return-edges to $H^*$with the constraints that we never add a return-edge $x \rightarrow r$ labeled "$)_c$" if there is already a return-edge from x labeled "$)_c$" or r is not a return-site vertex for c. Call this graph $H^\bullet$. The graph $H^\bullet$ violates the fourth well-formedness principle stated in Observation C.0.2, but it obeys the other well-formedness principles. We can use the proof of Lemma D.0.4 to show that the Simple Vertex Subsumption Algorithm computes a conservative approximation of the $\succ$-relation for $H^\bullet$: if the algorithm concludes that $v' \succ v$ then this is the case. This is because the proof does not rely on the input graph obeying the fourth well-formedness principle: the extra return-edges in the graph $H^\bullet$ do not confuse the Simple Vertex Subsumption Algorithm. This means that we could run the Vertex Subsumption Algorithm to obtain H_1^* and then run the Simple Vertex Subsumption Algorithm to compute a $\succ$-relation for H_1^*.

We observe that the lines (1–14) of the Vertex Subsumption Algorithm that precede the call to $AddRtnEdges$ do not examine return-edges. This implies that that the $\succ$-relation computed by the Vertex Subsumption Algorithm for H_1^* is the same as the $\succ$-relation that the Simple Vertex Subsumption Algorithm computes for H_1^*. This means that if the Vertex Subsumption Algorithm concludes that $v' \succ v$ then in fact $v' \succ v$; however, if the Vertex Subsumption Algorithm concludes that $v' \not\succ v$, then it may be that $v' \succ v$. *QED* □

Theorem 10.4.6 *If J approximates ($\sqsubseteq$) the greatest fixed-point solution for $\mathcal{F}_H^*$, then J also approximates the greatest fixed-point solution for $\mathcal{F}_{H_3}$.*

Proof: Recall that if J is the meet-over-all valid paths solution to $\mathcal{F}_H^*$, then J does not necessarily approximate the greatest fixed-point solution for $\mathcal{F}_{H_3}$ (see Figure 66).

Note that before the Vertex Subsumption Algorithm adds a return-edge $x \rightarrow r$ to H^* (line 49 of Figure 69), it performs a check to make sure the data-flow facts at r will not be violated (line 46 of Figure 69). Similarly, before the Edge Redirection Algorithm adds an edge $u \rightarrow v'$ to H^*(line 7 of Figure 72) it performs a check to make sure that the data-flow facts at v' will not be violated (line 5 of Figure 72).

For every edge $u \rightarrow v$ in H'^*, whether the edge was added by the Vertex Subsumption Algorithm, the Edge Redirection Algorithm, or was in the original graph H^*, it must be the case that $T_u(J(u)) \sqsupseteq J(v)$. It follows that

$$J(v) \sqsubseteq \bigsqcap_{u \rightarrow v} T_u(J(u))$$

Thus, J must approximate the greatest fixed-point solution for $\mathcal{F}_{H'^*}$. *QED* □

Lemma 10.4.7 *Let H^* be the hot-path graph input to Edge Redirection Algorithm. Let H_1^* be the graph that results from running the Vertex Subsumption Algorithm on H^*. (H_1^* is H^* with extra return-edges.) Let H_2^* be the graph output by the Edge Redirection Algorithm. Rename each vertex v in H^* as v_1 in H_1^* and as v_2 in H_2^*. Let v_1' and v_1 be vertices in H_1^* and let v_2' be the same vertex as v_1' but in graph H_2^*. If $v_1' \succ v_1$, then $v_2' \succ v_1$. In other words, if $v_1' \succ v_1$, then for every unbalanced-right-left path p_1 from v_1 in graph H_1^*, there must be a unbalanced-right-left path p_2' from v_2' in graph H_2^* such that $p_2' \succ p_1$.*

Proof: The proof is by induction on the length (in edges) of the unbalanced-right-left paths in H_1^*. We will show that for every unbalanced-right-left path p_1 from vertex v_1 in H_1^*, for every subsumption fact $v_1' \succ v_1$, there is a unbalanced-right-left path p_2' from v_2' in H_2^* such that $p_2' \succ p_1$. It follows immediately that $v_2' \succ v_1$.

Base case: We need to show that for every unbalanced-right-left path p_1 of length zero from v_1, for every subsumption fact $v_1' \succ v_1$, there is an unbalanced-right-left path p_2' from v_2' such that $p_2' \succ p_1$. The path p_1 is the empty path from v_1 to v_1. Since $v_1' \succ v_1$ it must be that the path

p'_1 from v'_1 to v'_1 subsumes the path p_1. Since v'_1 and v'_2 are the same vertices but with different names, we can just take p'_2 to be the empty path from v'_2 to v'_2. It follows that $p'_2 \succ p_1$.

Induction step: For the induction hypothesis, we assume that for every unbalanced-right-left path p_1 of length i that starts at vertex v_1, for every subsumption fact $v'_1 \succ v_1$, there is an unbalanced-right-left path p'_2 from v'_2 such that $p'_2 \succ p_1$.

Let p_1 be an unbalanced-right-left path of length $i+1$ that starts at vertex v_1 in the graph H^*_1. Pick an arbitrary subsumption fact $v'_1 \succ v_1$. We must show that there is an unbalanced-right-left path p'_2 from the vertex v'_2 in the graph H^*_2 such that $p'_2 \succ p_1$. It follows from $v'_1 \succ v_1$ that there is a path p'_1 from v'_1 in the graph H^*_1 such that $p'_1 \succ p_1$. We will use the path p'_1 to find the path p'_2. The path p'_1 must have length $i+1$, since it is congruent to p_1 (this follows from the definition of $\succ$). This means that p'_1 has length at least 1. Therefore, there is an edge $v'_1 \rightarrow w'_1$ and a path q'_1 such that $p'_1 = [v'_1 \rightarrow w'_1 \| q'_1]$. There are two cases we must consider:

1. The $v'_1 \rightarrow w'_1$ also occurs in the graph H^*_2, renamed as $v'_2 \rightarrow w'_2$. The path q'_1 starts at w'_1 and has length i. The fact $w'_1 \succ w'_1$ follows from the reflexivity of the subsumption relation. It follows by the induction hypothesis that there is a path q'_2 from w'_2 such that $q'_2 \succ q'_1$. Since $v'_1 \rightarrow w'_1$ subsumes the first edge of p_1, $v'_2 \rightarrow w'_2$ (which is the same edge) must subsume the first edge of p_1. We now show that $p'_2 = [v'_2 \rightarrow w'_2 \| q'_2]$ is an unbalanced-right-left path that subsumes p_1. The only way p'_2 could not be unbalanced-right-left is if $v'_2 \rightarrow w'_2$ were a call-edge labeled "$(_c$" and the first unmatched return-edge of q'_2 were labeled "$)_d$", where $c \neq d$. This cannot happen because: (1) the path $[v'_1 \rightarrow w'_1 \| q'_1]$ is unbalanced-right-left; (2) the label on $v'_2 \rightarrow w'_2$ is the same as the label on $v'_1 \rightarrow w'_1$; and (3) the labels on the unmatched parentheses of q'_2 are the same as the labels on the unmatched parentheses of q'_1 (since $q'_2 \succ q'_1$). We have that q'_2 subsumes q'_1, and that q'_1 subsumes the suffix of p_1 that includes all but p_1's first edge. It follows from the transitivity of the subsumption relation that q'_1 subsumes the same suffix of p_1. It follows from the definition of subsumption that $p'_2 = [v'_2 \rightarrow w'_2 \| q'_2]$ must subsume p_1.
2. The $v'_1 \rightarrow w'_1$ does not occur in the graph H^*_2. This means that the Edge Redirection algorithm replaced the edge $v'_1 \rightarrow w'_1$ with the edge $v'_1 \rightarrow w''_1$. (The edge $v'_1 \rightarrow w''_1$ has been renamed as $v'_2 \rightarrow w''_2$ in the graph H^*_2.) It follows from the check on line 5 of the Edge Redirection Algorithm (see Figure 72) and the proof of Theorem 10.4.5 that the Edge Redirection Algorithm would not replace the edge $v'_1 \rightarrow w'_1$ with the edge $v'_1 \rightarrow w''_1$ unless $w''_1 \succ w'_1$. Thus, we have the path q'_1 of length i from the vertex w'_1 and we have the subsumption fact $w''_1 \succ w'_1$. By the induction hypothesis, there is a path q''_2 from w''_2 such that $q''_2 \succ q'_1$. As in the previous case, it follows that $p'_2 = [v'_2 \rightarrow w''_2 \| q''_2]$ is an unbalanced-right-left path that subsumes p_1.

Therefore, there is an unbalanced-right-left path p'_2 from the vertex v'_2 in the graph H^*_2 such that $p'_2 \succ p_1$.

QED □

Theorem 10.4.8 *Let H^*_3 be the graph the results from running the Edge Redirection Algorithm (see Figure 72) and the clean-up pass in Figure 74 on the graph H^*. Then, H^* and H^*_3 are unbalanced-left path congruent.*

Proof: First, we show that for every unbalanced-left path p in H^*, there is a congruent unbalanced-left path p_3 in H^*_3. Let v be the first vertex of p. Let H^*_2 be the graph that results from running the Edge

Redirection Algorithm on H^*. By reflexivity of subsumption, $v \succ v$. Thus, by Lemma 10.4.7, $v_2 \succ v$, where v_2 is the same vertex as v but renamed in H_2^*. This means that there is a path p_2 from v_2 in H_2^* such that $p_2 \succ p$. In particular, p_2 must be an unbalanced-right-left path and it must be congruent to p. This means that p_2 must be an unbalanced-left path that is congruent to p. The Edge Redirection Clean-up Pass (see Figure 74) only remove vertices and edges that do not occur in any unbalanced-left path. This means that the path p_2 must also appear in the graph H_3^*. Thus, if we take $p_3 = p_2$, we have an unbalanced-left path in H_3^* that is congruent to p.

Next, we show that for every unbalanced-left path p_3 in H_3^*, there is a congruent unbalanced-left path p in H^*. It is straightforward to construct the path p during a traversal of the edges of the path p_3. Let p_3 start at the vertex v_3. The path p starts at the vertex v, where v and v_3 are the same vertex, but v_3 has been renamed in the graph H_3^*. Suppose we have traversed some prefix q_3 of p_3 and have constructed a path q in H^* that is unbalanced-left and congruent to q_3. Let $m_3 \rightarrow n_3$ be the next edge after q_3. We need to find an edge $m \rightarrow n$ such that $[q \| m \rightarrow n]$ is an unbalanced-left path that is congruent with $[q_3 \| m_3 \rightarrow n_3]$. There are two possibilities:

1. The edge $m_3 \rightarrow n_3$ is not a return-edge. Recall that H^* has the well-formedness property that for any two duplicate vertices a and a' that are not exit vertices, for every edge $a \rightarrow b$, there must be a congruent edge $a' \rightarrow b'$. The Edge Redirection Algorithm preserves this property: every time it removes an edge $a' \rightarrow b'$, it replaces it with a congruent edge $a' \rightarrow b''$. Since m_3 (the last node of q_3) is a duplicate of m' (the last node of q) and m_3 and m' are not exit vertices (since $m_3 \rightarrow n_3$ is not a return-edge), it must be the case that there is an edge $m \rightarrow n$ that is congruent to $m_3 \rightarrow n_3$. Since q is unbalanced-left and $m \rightarrow n$ is not a return-edge, it must be that $[q \| m \rightarrow n]$ is an unbalanced-left path that is congruent with $[q_3 \| m_3 \rightarrow n_3]$.

2. The edge $m_3 \rightarrow n_3$ is a return-edge. In this case, there can only be one return-edge $m \rightarrow n$ such that $[q \| m \rightarrow n]$ is an unbalanced-left path: $m \rightarrow n$ is the return-edge that is labeled with the close parenthesis that matches the last unmatched call-edge in q (the existence of $m' \rightarrow n'$ follows from the well-formedness of H^*). To show that $[q \| m \rightarrow n]$ is congruent to $[q_3 \| m_3 \rightarrow n_3]$, we must show that $m \rightarrow n$ is congruent to $m_3 \rightarrow n_3$. We already know that the exit vertices m_3 and m are congruent (since they are the last vertices of the congruent paths q_3 and q, respectively). So, we must show that the return-site vertices n_3 and n are congruent. The vertex n_3 must be a return-site vertex for the call vertex c_3, where c_3 is the call vertex in the last unmatched call-edge $c_3 \rightarrow e_3$ of the path q_3. Similarly, the vertex n must be a return-site vertex for the call vertex c, where c is the call vertex in the last unmatched call-edge $c \rightarrow e$ of the path q. Since q and q_3 are congruent, c_3 and c must be congruent. This implies that n_3 and n are congruent. Hence, $[q \| m \rightarrow n]$ is congruent to $[q_3 \| m_3 \rightarrow n_3]$ are congruent.

QED □

Appendix E

Determining If J' Preserves the Valuable Data-Flow Facts of J

Let J be a data-flow solution for $\mathcal{F}^*_H$, where H^* is a hot-path graph. Let J' be a data-flow solution for $\mathcal{F}_{H'^*}$, where H'^* is a reduced H^*. To check if J' preserves the valuable data-flow facts of J, we need to know that for every unbalanced-left path p from the $Entry_{global}$ of H^*, for every hot vertex v in p, for the path p' in H'^* that is congruent to p, for the vertex v' in p' that corresponds to v, $J(v) \sqsubseteq J'(v')$. This condition is very similar to stating the $Entry' \succ Entry$, where $Entry'$ is the entry of H' and $Entry$ is the entry of H. We define *subsumes for valuable data-flow facts* as follows: for a path p' in H'^* and a path p in H^*, we say p' subsumes p for desirable data-flow facts (written $\succ_{val}$) iff

1. p' exactly mimics p's control flow and
2. for each hot vertex v in p and corresponding vertex v' of p', $J(v) \sqsubseteq J'(v')$ for all of the uses of desirable data-flow facts in v.

We say that $v' \succ_{val} v$ iff for every path p from v there is a congruent path from v' such that $p' \succ_{val} p$. To check if J' preserves the valuable data-flow facts of J, we need to check if $Entry' \succ_{val} Entry$.

To compute if $Entry' \succ_{val} Entry$, we can use a modified version of the Vertex Subsumption Algorithm. There are three changes that must be made to the algorithm:

1. The Vertex Subsumption Algorithm must change the way it propagates non-subsumption facts across summary-edges. As presented, the Vertex Subsumption Algorithm algorithm may conclude that $v' \not\succ v$ when in fact $v' \succ v$. This is due to the fact that the Vertex Subsumption Algorithm propagates non-subsumption facts across pairs of duplicate summary-edges (see the proof of Theorem 10.4.5).
2. Line 3 of the Vertex Subsumption Algorithm (see Figure 67) must be changed to:

   If $v2$ is hot **and** $J(v_1) \not\sqsupseteq J(v_2)$ for uses of desirable data-flow facts in v_2 **Then**

   This changes the Vertex Subsumption Algorithm to calculate the $\succ_{val}$-relation instead of $\succ$-relation.
3. The Vertex Subsumption Algorithm must be changed to compute the $\succ_{val}$-relation between the vertices of two graphs (H and H') rather than for the vertices of one graph.

The second and third modifications are trivial. The first is more difficult. For every non-subsumption fact $r_1 \not\succ r_2$, for every pair of congruent summary-edges $c_1 \rightarrow r_2$ and $c_2 \rightarrow r_2$, the Vertex Subsumption Algorithm concludes that $c_1 \not\succ c_2$. This conclusion is correct iff there is a same-level valid path p_1 from c_1 to r_1 and a same-level valid path p_2 from c_2 to r_2 such that p_1 and p_2 are congruent.

To compute the $\succ$-relation accurately, we must compute the set S of tuples $\langle c_1, c_2, r_1, r_2 \rangle$ such that

1. c_1 and c_2 are duplicate call vertices;
2. r_1 and r_2 are duplicate return-site vertices; and
3. there is a same-level valid path from c_1 to r_1 that is congruent to some same-level valid path from c_2 to r_2.

Given the set S, we change the Vertex Subsumption Algorithm so that for every non-subsumption fact $r_1 \not\succ r_2$, the algorithm generates the non-subsumption fact $c_1 \not\succ c_2$ iff $\langle c_1, c_2, r_1, r_2 \rangle$ is in S.

To compute the set S, we must simultaneously compute the set T of tuples $\langle v_1, v_2, x_1, x_2 \rangle$ such that

1. v_1 and v_2 are duplicate vertices;
2. x_1 and x_2 are duplicate exit vertices; and
3. there is a same-level valid path from v_1 to x_1 that is congruent to some same-level valid path from v_2 to x_2.

The sets S and T can be computed simultaneously as follows:

1. Initialize the set T to the set of all tuples $\langle x_1, x_2, x_1, x_2 \rangle$ where x_1 and x_2 are duplicate exit vertices (x_1 may equal x_2).
2. For every tuple $\langle v_1, v_2, x_1, x_2 \rangle \in T$, for every pair of intraprocedural, congruent edges (that are not summary-edges) $u_1 \to v_1$ and $u_2 \to v_2$ add the tuple $\langle u_1, u_2, x_1, x_2 \rangle$ to T
3. For every tuple $\langle e_1, e_2, x_1, x_2 \rangle \in T$, for every pair of congruent call-edges $c_1 \to e_1$ and $c_2 \to e_2$, for every return-edge $x_1 \to r_1$ labeled "$)_{c_1}$", for every return-edge $x_2 \to r_2$ labeled "$)_{c_2}$", add the tuple $\langle c_1, c_2, r_1, r_2 \rangle$ to S.
4. For every tuple $\langle r_1, r_2, x_1, x_2 \rangle \in T$, for every tuple $\langle c_1, c_2, r_1, r_2 \rangle \in S$, add the tuple $\langle c_1, c_2, x_1, x_2 \rangle$ to T.
5. Repeat steps 2, 3, and 4 until a fixed-point is reached.