University of Alberta

METHODOLOGIES FOR MANY-INPUT FEEDBACK-DIRECTED OPTIMIZATION

by

Paul Berube

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

Fall 2012
Edmonton, Alberta

for Amber
May 18, 2012

Abstract

This thesis makes improvement to the process of ahead-of-time feedback-directed optimization (FDO) in compiler design. It examines multiple aspects of FDO from profile collection and representation through to the performance evaluation of FDO code transformations. Two guiding principals knit the four components of the research together. The first principle is a recognition that program behavior frequently depends on the input to the program; the second is that FDO is a predictive modeling technique, and must be designed and evaluated as such. Performance-evaluation methodology can be improved to be sensitive to input-dependent program behavior by using cross-validation with a workload of diverse program inputs. In many cases, expert knowledge can be leveraged to select program workloads that are representative of expected program usage. However, even with expert knowledge, characterizing input similarity and determining both how many inputs and the specific inputs to include in a workload are challenging questions. A compiler-centric clustering approach selects a small, representative, evaluation workload from a large initial collection of program inputs. Previous work has not addressed the problem of representing and utilizing multi-run profiles. An FDO compiler should not simply add, or average, profiles from multiple runs because a profile obtained this way does not provide any information about the variations in program behaviors observed between different inputs. *Combined Profiling* (CP) merges the profiles from multiple runs into a distribution model that allows code transformations to consider cross-run behavior variations. An FDO-based function inlining transformation is implemented in the `LLVM` compiler to illustrate the use of combined profiling by a code transformation in a complex compiler and to provide a concrete example of the usage of a rigorous cross-validation evaluation methodology for FDO.

Acknowledgements

I must thank Nelson Amaral for his excellent supervision. Without his perfect blend of pressure and understanding this thesis would not have been completed. The many lunches spent with my mentor Bruce Wilkinson provided the hope necessary to stay the course. And to my friends and family, thank you for putting up with all my complaining. Claire, I cannot overstate how much your understanding and encouragement over the past $1\frac{1}{2}$ years have improved and maintained my mental health.

Funding for parts of this work have been provided by the Natural Sciences and Engineering Research Council of Canada, Alberta Innovates: Technology Futures (formerly iCORE and Alberta Ingenuity), and by the IBM Center for Advanced Studies at the IBM Toronto Software Laboratory.

Table of Contents

List of Tables

List of Figures

List of Symbols

Miscellaneous:

a, b	Miscellaneous subscripts.
`PROG`	A program.
j	Input or profile specifier/subscript, *e.g.*: i_j, p_j.
u	Training input/workload specifier/subscript, *e.g.*: $t_u(i)$.
x, y	Matrix indexes, *e.g.*, $D[x, y]$.

Workloads:

i	A program input.
$\mathcal{W}$	The full *evaluation workload* of inputs for a program.
n	The number of inputs in $\mathcal{W}$; $n = \|\mathcal{W}\|$.
Ω	A set of program inputs too large to use as $\mathcal{W}$.
ω	The number of inputs in Ω; $\omega = \|\Omega\|$.
$\mathcal{W}_{test}$	A *testing workload*.
$\mathcal{W}_{train}$	A *training workload*.
k	The size of a clustering; a k-clustering is a set of k disjoint partitions that cover the original set of inputs.
$\mathcal{C}$	A partition/cluster of similar inputs.
$C_{T,k}$	A k-clustering of Ω for transformation T: $C_{T,k} = \{\mathcal{C}_{T,1}, \mathcal{C}_{T,2}, \ldots, \mathcal{C}_{T,k}\}$
C_k	A k-clustering of Ω for transformation all transformation in $\mathcal{T}$: $C_k = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k\}$
$\overline{C}_k$	A performance-weighted k-clustering of Ω for transformation all transformation in $\mathcal{T}$: $\overline{C}_k = \{\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2, \ldots, \overline{\mathcal{C}}_k\}$

Profiling:

$\mathcal{P}$	The set of raw profiles generated by profiling each of the n inputs in $\mathcal{W}$: $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$
$p_j[l]$	The value of monitor R_l in profile p_j, or the distribution of R_l if p_j is a CProf.
l	A location in a program.

$\mathcal{L}$	A set of locations: $\mathcal{L} = \{l_1, l_2, \ldots, l_k\}$
k	The number of locations in a program (of transformation applications or of inserted monitors).
$\mathcal{B}$	A behavior.
M	A metric of a behavior.
$R(\mathcal{B}, l, M)$	A monitor of behavior $\mathcal{B}$ by metric M at location l.
R_l	The monitor at location l ($\mathcal{B}, M$ implicit).
H_n	The histogram for the combined profile of monitor R_n.

Transformations:

T, U	An FDO transformation.
$\mathcal{T}$	The set of code transformations that use profile information in an FDO compiler: $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$
m	The number of FDO transformations in a compiler: $m = \lvert\mathcal{T}\rvert$.
$\mathbf{V}_j$	A *transformation vector* for transformation T (implicit from context) using profile p_j.
D_T	The *difference matrix* for transformation T (pg. 27) (pg. 35). $$D_T[x, y] = Manhattan(\mathbf{V}_x, \mathbf{V}_y)$$
D	The *combined difference matrix* for all transformations (pg. 35). $$D = \sum_{T \in \mathcal{T}} \frac{D_T}{\lvert\mathcal{L}_T\rvert}$$
$\overline{D}$	The *performance-weighted combined difference matrix*; D point-wise weighted by the pair-wise performance weight PW(pg. 37). $$\overline{D}[x, y] = D[x, y] \times PW[x, y]$$
$S, S_T, \overline{S}$	The combined similarity matrix created from D, D_T, or $\overline{S}$, respectively. $$S[x, y] = \max(D[x, y]) - D[x, y]$$

Times:

$t_\emptyset(()i)$	The execution time of a non-FDO baseline version of a program on input i, measured as the average of an odd number of runs.
$t_u(i)$	The execution time of an FDO version of a program on input i when u is used as $\mathcal{W}_{train}$, measured as the average of an odd number of runs.
$\tau_u(i)$	The speedup of an FDO version of a program on input i versus a non-FDO baseline, when u is used as $\mathcal{W}_{train}$: $\tau_u(i) = \frac{t_\emptyset(i)}{t_u(i)}$.

$\tau_u^{-1}(i)$ The *normalized execution time* of an FDO version of a program on input i versus a non-FDO baseline, when u is used as $\mathcal{W}_{train}$: $\tau_u^{-1}(i) = \frac{t_u(i)}{t_\emptyset(i)}$.

Equations:

LNP A vector of *log-normalized performance* for each input in a workload (pg. 36):

$$\mathbf{LNP}[u] = \frac{\sum_{i \in \Omega - u} \left(\tau_u^{-1}(i) \times \log(t_\emptyset(i)) \right)}{\sum_{i \in \Omega - u} \log(t_\emptyset(i))}$$

PW The pairwise *performance weighting* factor (pg. 37):

$$PW[x, y] = \frac{max(\mathbf{LNP}[x], \mathbf{LNP}[y])}{min(\mathbf{LNP}[x], \mathbf{LNP}[y])} - 1$$

$Mismatch$ Clustering error (pg. 39):

$$Mismatch(C, D) = \sum_{\mathcal{C}_a \in C} \left(\sum_{i_x, i_y \in \mathcal{C}_a} D[i_x, i_y] \right)$$

$\Delta Mismatch$ Reduction in clustering error when k is increased (pg. 51):

$$\Delta Mismatch(k) = Mismatch(\mathcal{C}_{k-1}, D) - Mismatch(\mathcal{C}_k, D)$$

$CrossError$ Additional clustering error incurred by using an alternative clustering (pg. 44):

$$CrossError(C_{T,k}/C_{U,k}, D_U) = Mismatch(C_{T,k}, D_U) - Mismatch(C_{U,k}, D_U)$$

$\mu_g(\mathcal{W})$ The geometric mean of normalized execution times for $\mathcal{W}$, measured by 3-fold cross-validation (pg. 27):

$$\mu_g(\mathcal{W}) = \frac{1}{\textit{FDOPeak}} = \sqrt[|\mathcal{W}|]{\prod_{i \in \mathcal{W}} \tau_u^{-1}(i)}$$

$\sigma_g(\mathcal{W})$ The geometric standard deviation for $\mu_g(\mathcal{W})$ (pg. 27):

$$\sigma_g(\mathcal{W}) = \exp\left(\sqrt{\frac{\sum_{i \in W} (\ln \tau_u^{-1}(i) - \ln \mu_g(\mathcal{W}))^2}{|\mathcal{W}|}} \right)$$

List of Abbreviations

AOT	Ahead-of-Time, referring to any program-related processing that is done by the developer before a program is deployed and/or run by a user. AOT often refers to the compilation of source code to an architecture and operating-system specific executable binary. AOT contrasts with JIT techniques.
CP	Combined Profiling
CProf	A Combined Profile.
FDO	Feedback-Directed Optimization. When used as an adjective, indicates that the subject uses profile information for code optimization. (*e.g.*, an FDO transformation or an FDO compiler.)
FDI	Feedback-Directed Inlining. Function inlining that uses feedback information to guide its decisions.
IR	Intermediary Representation, a program representation used internally by a compiler to facilitate code transformation and native code generation. Many IRs attempt to be independent of the original source code language and the eventual target machine.
JIT	Just-In-Time, referring to processing to facilitate the execution of a program at the time it is run. JIT often refers to the (re)compilation and (re)optimization of machine-independent interpreted code to machine-specific binary code by a run-time system during the execution of the (initially interpreted) program.
LNP	Log-Normalized Performance (see symbols).
SCC	A strongly-connected component in a graph. In a call-graph, a set of functions that form a recursive cycle.
SPEC	Standard Performance Evaluation Corporation, a consortium of commercial and academic interests that produce the most widely used performance evaluation benchmark suites. Of particular note for compiler evaluation are the SPEC CPU suites of C, C++, and Fortran CPU-intensive programs released in 2000 and 2006.

Glossary

Evaluation Workload	($\mathcal{W}$) A set of program input that are used in the performance evaluation of a system. When cross-validation is used, the evaluation workload is the full set of all input that is then divided into training and testing sets.
Testing Workload	($\mathcal{W}_{test}$) In cross-validation, the subset of input used to evaluate the result of a training run. This set has no members in common with the training workload.
Training Workload	($\mathcal{W}_{train}$). In cross-validation, the subset of input used in a training run. This set has no members in common with the testing workload.
`LLVM`	A modular open-source compiler implemented in C++.
XLC	IBM's commercial compiler.
k-Fold	A cross-validation technique where $\mathcal{W}$ is partitioned into k equally-sized sets (folds). Each fold takes a turn as $\mathcal{W}_{test}$; $\mathcal{W}_{train} = \mathcal{W}/\mathcal{W}_{test}$.
Leave-One-Out	A cross-validation technique where $\mathcal{W}_{train}$ is all of $\mathcal{W}$ except a single element (the one left out). Each element of $\mathcal{W}$ takes a turn as $\mathcal{W}_{test}$.
Leave-One-In	An evaluation technique for situation where $\mathcal{W}_{train}$ is limited to a single element. It is the inverse of leave-one-out cross-validation: each element u of $\mathcal{W}$ takes a turn as $\mathcal{W}_{train}$, and is evaluated on $\mathcal{W}_{test} = \mathcal{W}/\{u\}$.
Coverage	(of a monitor) The proportion of runs represented in a CProf where the monitor executes at least once.
Occupancy	(of a monitor) The proportion of histogram bins containing weight out of the maximum possible number of bins that could contain weight given the number profiles that cover the monitor.
Span	(of a monitor) The ratio between a histogram's range and its maximum value.
Drift	(of a monitor) The proportion of weight that does not overlap between between the batch-constructed and incrementally-constructed versions of a histogram.
Source call site	A call site that is inlined, possibly copying original call sites in the callee to target call sites in the caller.

Original call site	A call site inside a callee that is copied to a target call site in a caller when a source call site in the caller is inlined.
Target call site	A call site copied into a caller from the original call site in the callee when a source call site in the caller is inlined.

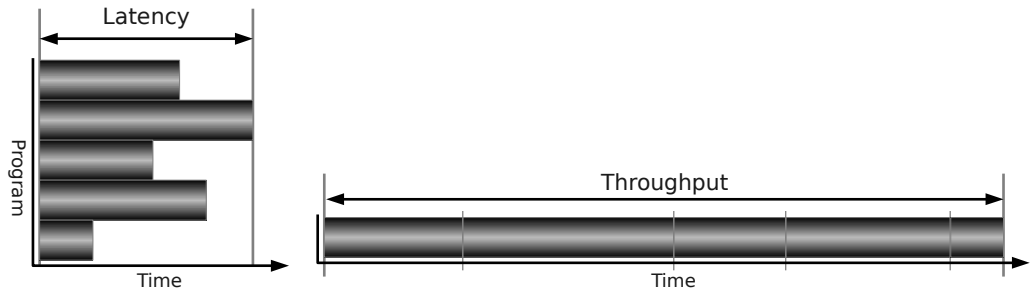
Chapter 1

Introduction

The field of compiler optimization is a combined effort of science and engineering which, like all scientific advancement, is an iterative process of analysis, exploratory implementation, and evaluation. For most useful programs, execution depends on the program's input. Consequently, there is no optimally-efficient executable representation of high-level source code that, for instance, executes in the least possible amount of time for every program input. Even given an optimization criteria, such as processing throughput, and a reasonable set of program inputs, creating an executable whose performance is optimal for that set of inputs is (provably) intractable. Moreover, it is infeasible to approach compiler design as a single problem. Instead, compilers apply a series of *transformations*[1] that attempt to improve the efficiency of a particular region of code in a specific way. For instance, Chapter 6 presents a function-inlining transformation that replaces a function call by a copy of the body of the called function. This transformation improves program efficiency by eliminating the overhead of making function calls, while improving the effectiveness of subsequent transformations.

Research in compiler transformations often demonstrates heroic efforts in both the identification and abstract analysis of opportunities to improve program efficiency, and in the concrete implementation of these ideas. However, standard practices at the evaluation stage of the scientific process are modest at best, perhaps because code transformations have a long history of providing significant benefits in practical, every-day situations. In most cases, compilers are evaluated using a collection of programs, with each program evaluated using a timing run on a single evaluation input. The deficiencies of this evaluation process are particularly prevalent, and especially disconcerting, when *feedback-directed optimization* (FDO) is used to guide a transformation. In this scenario, instrumentation is inserted into the program during an initial compilation in order to collect a profile of the run-time behavior of the program during one or more training runs. The profile is used in a second compilation of the program to help the compiler assess the benefit of code

[1] Transformations are frequently, but inaccurately, referred to as optimizations. Most transformations provide heuristic and/or partial solutions to NP-Hard problems, which are themselves abstracted from the full complexity of generating truly optimal code for a specific physical machine.



(a) Longest individual time (b) Total time

Figure 1.1: Measurements for workload evaluation using latency or throughput

transformation opportunities. The current standard practice for evaluating an FDO compiler uses the profile of a single training input to guide transformations, and evaluates the transformed program with a single evaluation input. These standard practices set program inputs as controlled variables. However, performance evaluation should be generalizable to real-world program workloads. Consequently, the program-input dimensions of a rigorous evaluation of compiler performance must be manipulated variables.

Performance evaluation is a challenging, multi-faceted problem. In this thesis, performance is always assessed in terms of program execution time[2] One dimension of this challenge is the choice between evaluating throughput or latency, as illustrated in Figure 1.1. Given a collection of tasks (*e.g.*, the programs in a benchmark suite or runs of a single program on a workload of inputs), throughput measures the total time required to complete all tasks sequentially. Conversely, latency considers the tasks in parallel and measures the task that takes the longest. Improving throughput means reducing average execution time; improving latency means reducing worst-case execution time. Both types of performance are important, and both approaches to evaluation are valid. The choice of focus in an evaluation must match the goals of the system's user, and as such is beyond the scope of this document. Fortunately, shifting focus is often as simple as changing the weighting used to combine measurements from the individual tasks. Different aspects of this work assume different performance goals and thus perform the weighting in different ways. Consider each of these approaches as one possible option for evaluation, independent of the specific evaluation in which they appear. A real-world application of any of the ideas presented here will have unique performance goals, and can mix-and-match these approaches as appropriate.

This work spans the scientific process as it relates to the design of FDO compilers, with two pervasive themes throughout. The first theme is a recognition that program behavior frequently depends on the input to the program; the second is that FDO is a predictive modeling technique, and must be designed and evaluated as such. Chapter 3 explains the current methodology used to evaluate compilers,

[2]Other measures of performance include power consumption and code size.

and how it can be improved for use with FDO. The fundamental idea is that an evaluation should be cross-validated; a workload of inputs should be used to test and train an FDO compiler. This methodology does not answer how the evaluation workload should be selected. In many cases, expert knowledge can be leveraged to select representative workloads of inputs. However, even with expert knowledge, characterizing input similarity and determining both how many inputs and which specific inputs to include in the workload are challenging questions.

Chapter 4 proposes a compiler-centric clustering methodology to select a small, representative, evaluation workload from an infeasibly-large initial collection of program inputs. The performance evaluation of an FDO compiler using the reduced workloads accurately predicts the performance evaluation results obtained using the full workload.

Previous work has not addressed the problem of representing and utilizing multi-run profiles. An FDO compiler should not simply add or average profiles from multiple runs, because such a profile does not provide any information about the variations in program behaviors observed between different inputs. Chapter 5 uses *Combined Profiling* (CP) to merge the profiles from multiple runs into a distribution model that allows code transformations to consider cross-run behavior variations. Experimental results demonstrate that meaningful behavior variation is present in the program workloads, and that this variation is successfully captured and represented by the CP methodology.

The FDO-based inliner presented in Chapter 6 demonstrates how a transformation can use the information stored in a combined profile. The feedback-directed inlining framework sorts inlining opportunities according to parameterized reward functions that query a combined profile using distribution quantiles. Chapter 7 brings these components together by performing a thorough cross-validated evaluation of the CP-informed inliner.

Chapter 2

Background and Motivation

This chapter presents some of the basic notation and terminology used through the rest of the document. In addition, it discusses two fundamental issues addressed in later chapters, namely the proper evaluation of FDO compilers, and the recognition of the existence of input-dependent program behavior and its impact on measured performance results. Finally, existing approaches to function inlining, the transformation investigated in Chapters 6 and 7, are reviewed. Further details on code analysis and FDO code transformations are left to Appendix A, while a description of the profiling techniques used to inform FDO transformations follows in Appendix B.

2.1 Terminology and Notation

Many aspects of this work require quite a bit of notation. Every effort has been made to ensure that this notation is clear and that every symbol has a single meaning throughout. However, with sets, vectors and matrices derived from behaviors, benefits, clusters, inputs, locations, metrics, parameters, partitions, multi-versioned programs, profiles, transformations, and all manner of times, all of which are used in several contexts, the notation can be burdensome. To help ease this burden, there is a list of symbols in the front matter of this document . Furthermore, to help keep the notation as clean and consistent as possible, most notation follows these guidelines:

- Sets are denoted by a capital letter in calligraphy: $\mathcal{S}$.
- An exemplar element from a set is a lowercase letter and is not subscripted with an index: $s \in \mathcal{S}$.
- Matrices are denoted by capital letters: M.
- Vectors use bold font: $\mathbf{V}$.
- Individual elements of matrices and vectors are specified using square brackets, never subscripts: $M[x, y], \mathbf{V}[z]$.

- Times and speedups indicate the input who's running time they measure in parenthesis, with the training set as a subscript. Non-FDO measurements use $\emptyset$ as the training set: $t_u(i), t_\emptyset(i)$.

2.2 Representations of Program Structure

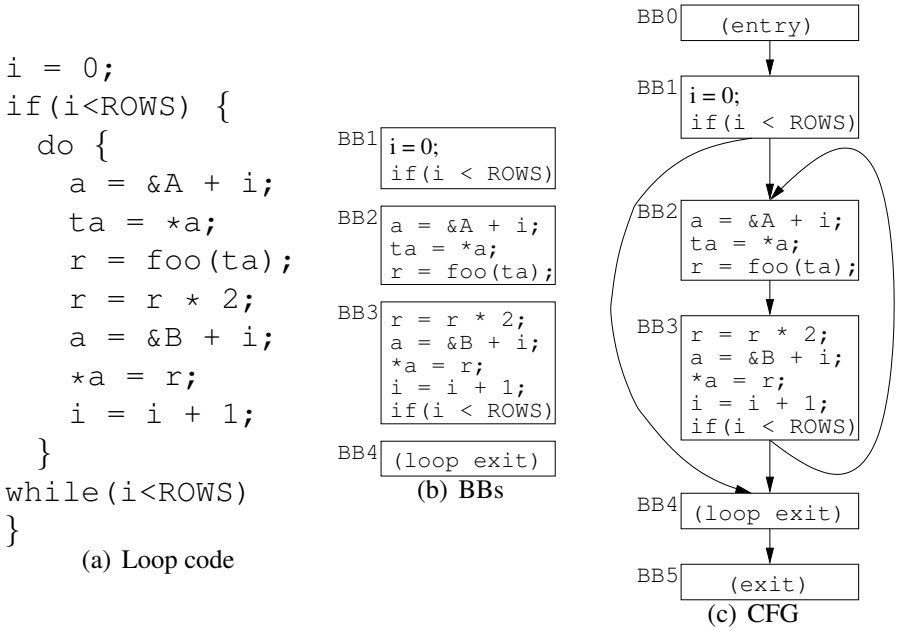


Figure 2.1: A loop and the corresponding CFG

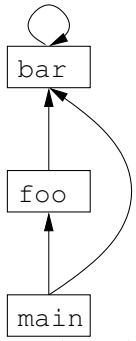
A major component of profiling is measuring program behavior to determine which parts of the code are hot. Most profiling techniques attempt to observe and record the control flow of a program during execution. Consequently, the standard representations of program structure and control flow used by compilers are also used to discuss profiling.

A *basic block* (BB) is a unit of program control flow consisting of a single-entry, single-exit sequence of instructions for which the following restriction holds: If program execution reaches the first instruction in the BB, every instruction in the BB must execute. Branches (conditional or unconditional), procedure calls[1], and returns from a procedure end a BB as they are a break in control flow and are thus the single exit from the BB. The first instruction in a procedure and the first instruction following the end of a BB start a new BB. Branch or jump targets (labels) start a new BB and end any block reaching the label since the label would break the single-entry property. Sub-figures (a) and (b) of Figure 2.1 show the

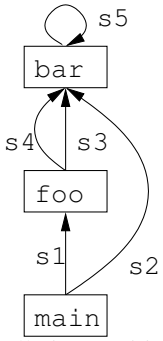
[1]Various programming constructs in, *e.g.*, FORTRAN (alternate returns), C (`setjump()`), and Pascal (out-of-procedure goto targets), allow execution to return from a procedure to a different location than the invoking call site [77].

```
main() {
  foo(x);//s1
  bar(y);//s2
}

foo(x) {
  if(cond)
    bar(x-1);//s3
  else
    bar(x*2);//s4
}

bar(x) {
  if(cond)
    bar(x-1);//s5
}
```

(a) Call sites `S1` – `S5`



(b) Context-insensitive CG



(c) Call-site sensitive CG

Figure 2.2: Call sites in a simple program, the context-insensitive CG, and the call-site sensitive CG

decompositions of a do-while loop into BBs.[2] `BB4` starts at the branch target used by the loop bounds tests in `BB1` and `BB3`, which is implied but not shown in the original code.

A procedure can be decomposed into a set of BBs. A *control flow graph* (CFG) represents a procedure, with each BB forming a node in the graph and the transfers in control between BBs forming the directed edges of the graph. Figure 2.1(c) shows the CFG for the loop in Figure 2.1(a). The entry point into a procedure marks the start of the first regular BB in the CFG. Each CFG starts with a special entry node which represents the entry point into the function and ends with an exit node, neither of which contain any instructions. For every BB BB_i in the CFG, there exists a directed path from the entry node to the exit node that contains BB_i.[3] In an extended CFG, BB execution may be interrupted by an exception [28].

The *call graph* (CG) of a program represents the calling relationships between the procedures of the program. Each procedure is a node in the graph. A call from one procedure to another is represented as a directed edge from the caller to the callee. A static CG is inferred from the program code and contains all procedure calls. A dynamic CG is constructed at runtime and contains only those calls that occurred during program execution. Procedure call statements in a program are referred to as call sites. A *call chain* exists during program execution and is the or-

[2]Compilers typically convert all loops to do-while loops to enable the use of the same internal representation for every loop.

[3]This constraint may be temporarily violated when code transformations render a BB non-executable. Dead code elimination will remove such BBs from the graph.

dered list of call sites on the call stack. That is, the call chain is the list of procedures that have been called (and where they were called from) but have not yet returned. A length l *calling context* of call site c consists of the l most recent call sites in the call chain when c is executed. A context-sensitive CG contains different edges for each distinct $\langle$*call site*, *calling context*$\rangle$ pair, while a context-insensitive CG will summarize all calls between a caller and a callee with a single edge. A call-site sensitive CG does not record calling contexts but does include a separate edge for each call site. A call-site sensitive CG is important for many inter-procedural transformations. Figure 2.2(a) presents the call sites (labeled `S1` through `S5`) of a simple program. Figure 2.2(b) is the static, context-insensitive call graph of the program. Notice that the two calls to bar() from foo() are represented by a single edge. In contrast, the call-site sensitive CG in Figure 2.2(c) contains two edges from foo() to bar() which correspond to the two different call sites.

2.2.1 Profiling Terminology

The profile of a program `PROG` records information about a set of *program behaviors*. A program behavior $\mathcal{B}$ is a (potentially) dynamic feature of the execution of a program. The observation of a behavior $\mathcal{B}$ at a location l of a representation of the program is denoted $\mathcal{B}_l$.[4] A behavior $\mathcal{B}$ is quantified by some metric $M(\mathcal{B})$ as a tuple of numeric values. A *monitor* $R(\mathcal{B}, l, M)$[5] is injected into a program at every location l where the behavior $\mathcal{B}$ is to be measured using metric M. At the completion of a *training run*, each monitor records the tuple $\langle l, M(\mathcal{B}_l)\rangle$ in a *raw profile* that contains unmodified metric values. The value (or distribution) of the metric of a monitor is simply called the value (or distribution) of the monitor. For example, in naive edge profiling, the locations l are the edges of the Control Flow Graph (CFG), the metric M is the execution count of each edge, the observation $\mathcal{B}_l$ of the behavior $\mathcal{B}$ is the traversal of the edge during program execution, and the raw edge profile contains a listing of $\langle \mathrm{edgeID}, \mathrm{count}\rangle$ pairs.

For simplicity, consider a program with a single monitor, R_1. When no program state is shared between executions, the raw profile from each training run i provides one independent sample,[6] $R_1[i]$, of the possible values of R_1. Thus, each $R_1[i]$ is an independent random variable identically distributed according to some unknown probability distribution which arises as the result of the interactions between a program and its inputs.

[4]For instance a location l can be a point or a single-entry-single-exit region in the Control Flow Graph of the program.

[5]A monitor can also be thought of as a Recorder, thus the use of the letter R to refer to a monitor.

[6]Execution independence is sufficient, but not strictly necessary for R_i and R_j ($i \neq j$) to be independent.

2.3 Performance Evaluation

The goal of most compiler research, and the result most reported by and most expected of compiler research, is reduced program execution time. It is therefore critical that the methodologies used to measure the impact of a compiler on program execution time, as well as the metrics used to summarize and draw conclusions from those measurements, are scientifically and statistically sound. Scientific soundness requires a recognition of controlled, manipulated, and responding experimental variables, along with any uncontrolled or unmanipulated variables and conflating factors that might limit the strength and/or generality of any conclusions. For FDO compilers, this means, in part, that both training and evaluation inputs, along with the evaluated programs, must be varied in the evaluation. Without varying both input sets, experimental results cannot be generalized to other training and/or evaluation inputs.

Data inputs are a key component of benchmarks. Evaluation inputs must be carefully selected to represent typical program usage while meeting benchmark requirements such as dynamic memory footprint size, CPU load, and running time. The use of FDO for benchmark programs enhances both the importance and selection difficulty of inputs. In particular, two intuitively similar inputs may not induce similar code transformations in a compiler, or may not present similar performance responses to compiler transformations [15, 16].

Training input selection presents several additional challenges beyond those presented by the selection of evaluation inputs. Training inputs must represent typical program use, without requiring long running times. Furthermore, in large applications, a single training-sized input cannot cover all the important use cases. Therefore, it can be difficult to provide a single representative training input.

Furthermore, there is disagreement in the benchmarking community over what is meant by a "representative training input." Some benchmark authors advocate that an effective way to ensure that the training input is representative of the evaluation input is to include a portion of the evaluation input in the training input. Thus, part of what the compiler sees during training is a perfect predictor of at least some portion of the evaluation input. Some studies use similar arguments to justify training and testing with the same input. However, others in the benchmarking community argue that this technique for training input creation makes the training and evaluation inputs too similar [101]. Using the same input(s) for both training and testing is neither a scientifically nor a statistically sound practice, and can generate very misleading results. For instance, as an argument for the combination of off-line and on-line profiling, Krintz shows that for a collection of Java benchmark programs, optimizing according to an off-line profile *improves* average performance by 13% when the training and testing inputs are identical, but *degrades* performance by 28% when the training and testing inputs differ [64]. When training inputs are repeated in the evaluation set, over-fitting is rewarded, which is counter to the goals of an effective evaluation. Is the purpose of FDO to maximize program performance

on a particular input, or to enhance program performance on the range of inputs likely to be seen in practice? If FDO is meant to improve performance in practice rather than to maximize benchmark scores, then training and evaluation workloads should not overlap. Nonetheless, overlapping training and testing inputs have been published by SPEC for SPEC CPU benchmarks, such as `gap` from CPU2000 and `hmmer` from CPU2006. Previously, there have been no precise guidelines or rules for input selection for SPEC CPU benchmarks, which has left individual benchmark authors to decide for themselves how these inputs should be devised.

Finally, an often overlooked issue in performance evaluation is the proper summarization of results. Fleming and Wallace provide an excellent discussion on this point that is perhaps best summarized by the paper's abstract [41]:

> *Using the arithmetic mean to summarize normalized benchmark results leads to mistaken conclusions that can be avoided by using the preferred method: the geometric mean.*

Nonetheless, arithmetic means are routinely used to summarize speedups both across the inputs to a single program and across the set of programs used in a study. All summary results presented here use carefully-considered summarization methods. In most cases, the technique of choice is the geometric mean. Furthermore, when presenting aggregated results, confidence intervals and/or extreme values are also reported to give a clear indication of the spread between the aggregated elements.

2.3.1 Evaluating Learning Systems

A training run to generate a program profile for a compiler is very similar to a training run in machine learning. The profile essentially provides a sample data point (of program execution) to which the compiler attempts to fit the program in order to maximize expected program performance. Thus, as with any predictive-modeling system, both the training and testing input dimensions must be considered by an evaluation methodology.

There are two classic characterizations of the amount of learning, or goodness of fit, achieved by a predictive model [53]. Underfitting is the problem where the system does not learn as much as possible from the training data, and consequently fails to achieve maximum performance on the evaluation data. Underfitting is not a significant problem for the reliability of an evaluation, but rather indicates that opportunities for improvement exist.

Conversely, over-fitting is the case where the learning system matches the training data too closely. Consequently, small variations from the training data in the evaluation data cause large errors by the system. Over-fitting can be detected by comparing the performance of the system when evaluated using the training data versus distinct evaluation data. Excellent performance on the training data, but poor performance on the evaluation data, suggests that over-fitting has occurred. Over-fitting is a significant problem that must be avoided. The upshot of the over-fitting-underfitting spectrum is that an evaluation must be careful to ensure that the

set of training inputs is fully distinct from the set of evaluation inputs. This way, if over-fitting does happen, it is detectable and does not inflate the evaluation results. Otherwise, evaluation could report significant performance gains that are not achievable in practice.

Over-fitting is particularly problematic when the number of training inputs is low. With few training examples, it is difficult to separate the peculiarities that make inputs non-identical from the commonalities between inputs that should be exploited. If a single training input is used, this distinction is impossible. Since discovering and exploiting the commonalities between inputs is a fundamental task of many learning systems, including FDO compilers, it is imperative that multiple inputs are used during the training process. The use of multiple inputs and cross-validation for robust performance evaluation are discussed in Chapter 3.

The statistical significance of results is very important when evaluating complex systems. System performance is not independent of the data used, and will vary according to the inputs used for training and evaluation. Changing the data may change the maximum possible level of performance. For example, the entropy in data used for a compression algorithm will change how much the data can be compressed, and consequently how significantly the performance of the compression routines impact the whole program. According to the central limit theorem, random samples from a population with finite variance are approximately normally distributed. Performance measures have finite variance, and are thus approximately normally distributed around their mean. Consequently, the sample standard deviation or confidence intervals can be calculated from the measurements to express the expected spread of individual performance measurements from the mean.

If the standard deviation is small compared to the measured gain, then the measured gain is meaningful, and will very likely be observed in any additional sample points (*i.e.*, other data inputs) that were not tested during evaluation. On the other hand, if the standard deviation (or confidence interval) is large compared to the mean, then the true mean may be significantly different than the measured mean, and the measured mean may not accurately represent the gain expected on other inputs. In other words, the gain observed during evaluation may simply be noise. For example, consider an evaluation that yields a mean speedup of 1.05 over a baseline. Two times the standard deviations is approximately a 95% confidence interval. Thus, if the standard deviation is 0.005, it can be stated, with high confidence, that the evaluated technique provided a 5% improvement, $\pm$ 1%. Alternatively, if the same evaluation produced a standard deviation of 0.05, then the technique provides a 5% improvement $\pm$ 10%. In this second case, the 5% improvement is not statistically significant, and it is uncertain that the technique improves performance. Thus, calculating standard deviations or confidence intervals provides a good measure of the robustness of the measured performance across the untested inputs that could be encountered in the field, provided that the sample of inputs used for evaluation is representative of those unseen inputs.

2.4 Static vs. Dynamic FDO

The work presented in this thesis is explicitly developed for, and evaluated using, FDO for ahead-of-time compilers (static FDO). Outside of this section, FDO refers only to static FDO. Although dynamic FDO is fundamentally different from static FDO in several ways, both use profile information to guide compilation decisions. Therefore, it is important to understand the differences between these techniques in order to distinguish between methodologies that are tightly-coupled with one approach and those that have more general applicability.

In dynamic FDO, both profiling and program optimization happen at runtime. Consequently, the profile is extremely likely to accurately predict program behavior in the near future. Since the program is re-optimized on each execution and even during execution, behavior diversity is not a significant issue for dynamic FDO. However, a system that operates only at runtime presents several challenges. First, the overhead of profiling and optimization must first be overcome before a performance benefit is incurred. As such, both the profiling mechanism and the program transformation candidates must be as light-weight as possible, thus excluding important classes of program transformations. For example, most transformations that require intra-procedural analysis are usually omitted from dynamic FDO systems. Furthermore, many opportunities for improvement are overlooked. In order to minimize the time required for optimization, only those transformation opportunities with the largest expected benefit are pursued. In order to ensure program progress, program optimization is usually allocated a time budget in proportion to the execution of the program. Consequently, the exploitation of opportunities, even those with large expected benefit, may be delayed while other opportunities consume the optimization budget. Additionally, a dynamic FDO system requires a warm-up period at the start of program execution while profiling gathers the initial data required to target the largest opportunities for performance improvement.

On the other hand, a static FDO system (the only FDO considered by this work) does profiling and optimization ahead of program execution time. Therefore, the compiler is free to use expensive program analysis and code transformation algorithms. All opportunities for improvement may be exploited and those code transformations improve program performance from the start of execution without delay. While offline training and optimization incur an overhead for the developer, this overhead need only be incurred once rather than during every execution of the program. Of course, since dynamic FDO requires the support of a runtime system to facilitate profiling and recompilation, static FDO is the only option for a very large body of applications written in languages such as C, C++ and FORTRAN, which currently execute without a runtime system.

However, offline FDO does not have a real-time measurement of program behavior. Instead, the premise of static FDO is that it should be possible to capture the general trends in program behavior by observing a fixed number of runs. That is, a summary of program behavior during past runs is assumed to predict the behavior

in future runs. In light of this premise, behavior diversity must be considered in the application and evaluation of static FDO. However, most uses of FDO assume (often implicitly) a negligible level of behavior diversity in the program workload. Unfortunately, behavior diversity is often present in practice, leading to conservative FDO-driven code transformations, unpredictable program performance across inputs, and unreliable FDO performance measurements in benchmark results and academic literature.

The Standard Performance Evaluation Corporation (SPEC) is a cooperative effort between system vendors, processor designers, compiler developers, and academics to produces benchmark suites that are relevant to current computing needs and provide reliable performance evaluation [37]. The SPEC CPU suite is designed to evaluate processor performance and has become the primary tool used for static compiler evaluation. SPEC CPU supports static FDO and provides both training and evaluation inputs for each benchmark program. However, usually a single training input is provided. Many programs use a single evaluation input, and none use a large collection of evaluation inputs. In fairness to SPEC, this methodology is reasonable when creating a set of representative programs for the purposes of evaluating *processor performance*. However, the adoption of the same methodology for *FDO compiler evaluation* ignores any issues of behavior diversity.

2.5 Variations in Behavior and Performance

Variation in program behavior is a central concern of FDO and code transformation in general. However, until recently, most studies assumed the representativeness of a profile and rarely investigated the existence or impact of behavior diversity.

Fisher and Freuenberger investigate the variability of branch probabilities in C and FORTRAN programs [39]. Hardware branch prediction is usually evaluated using the percent of dynamic branches correctly predicted. The authors argue that the average number of instructions executed before a break in control flow occurs is a more appropriate measure of the effectiveness of compiler branch prediction. This alternate metric is better correlated with the execution time of the program, since the cost of a mispredicted branch, unconditional jump, or procedure call impacts performance in accordance with the frequency of these events with respect to other instructions. For example, incurring an instruction cache miss on every branch would have little impact on the execution time of a program containing very little control flow compared to computation. Using this new metric, they find that control flow is very predictable and is stable across different input data sets.

Wall measures the accuracy of profiles across different data sets [99]. He compares a random profile, three statically-estimated profile techniques, and real profiles collected from multiple data sets. The profiles record BB, procedure, and call-site execution frequency, as well as global variable access frequency and procedure execution time. Each profile is sorted in decreasing order of frequency or time. A matching algorithm calculates how well the profile predicts the n most

frequent elements. BB execution frequency is the most difficult to predict. Static estimation methods perform almost as poorly as the random profile, while real profiles from alternate inputs are on average less than 50% accurate unless n is made large. Therefore, this result suggests that there exists significant variation in the most frequently executed portions of a program when the input is varied.

Perelman *et al.* introduce Variational Path Profiling (VPP), a technique that detects variations in the execution time of control-flow paths in a program [78]. They posit that paths with larger variations in execution time hold the greatest potential for optimization opportunities because it may be possible to make all executions of the path require time similar to the fastest execution of the path. In extensively hand-optimized programs they find that the paths with the most execution-time variation seldom correspond to the most frequently executed paths because those paths have already been heavily optimized. Investigation of the paths with the most execution time variation allows simple, but effective, hand optimization that reduces program execution time by 8% on average for three commercial applications.

Multiple inputs are used in attempts to scale input sizes (up or down). Bienia *et al.* focuses on micro-architectural features to scale the input sizes of PARSEC benchmarks [22]. When a reduced input has a large error in comparison with a reference input, they regard the reference input as "correct." Input-dependent program behaviors must become recognized as inherent characteristics of programs, not errors in workload selection.

Kim *et al.* compare FDO's simulated dynamic branch prediction accuracy on the diverge-merge processor using the MinneSPEC reduced program inputs against the same benchmarks using the SPEC training inputs [60]. They find that FDO is not sensitive to program input. Their evaluation is not sound because comparing program behavior between a reference input and an input that was specifically selected by experts on the criteria that it be representative of the reference is unlikely to predict the actual variations between inputs encountered after deployment.

Gove and Spracklen test how well the SPEC CPU 2006 training inputs represent the reference workloads [46]. They find that in almost every case, based only on the correspondence of function execution frequencies and branch behaviors, the training workload is highly representative of the reference workload. However, this comparison is only between the reference workload and the training input. The training input is explicitly selected to be representative of the reference workload. Significant dissimilarity between `ref` and `train` indicates a failure of the training-input selection process; similarity between these two inputs does not suggest that program runs on *all* inputs are similar.

Fursin *et al.*. collect 20 inputs for each program in the MiBench benchmark suite to evaluate iterative optimization over a workload of inputs [43]. While many of the optimized programs have similar performance across all inputs, some programs display significant variations in instructions per cycle (IPC) and execution time. However, the best optimization configuration usually produces stable performance results across the workload. Chen *et al.*. collect 1000 inputs for the

MiBench benchmark programs to perform a similar study [102]. As in the Fursin study, performance characteristics both during and after the iterative compilation process vary significantly between inputs for some programs. For instance, the best average-case execution time improvement of about 37% for adpcm_d also exhibits the greatest performance differences between inputs; individual performance improvements range from 10% to 70%. Iterative optimization is fundamentally different than FDO. Iteration allows the compiler to observe the impact of, and subsequently correct, poor decisions; a facility not available with FDO. Furthermore, in these studies, global compiler flags are tuned by the iterative process. FDO affects compilation at a much finer granularity, for example, informing the inlining decisions at individual call sites. Thus, FDO is potentially more sensitive to input variation.

2.6 Program Transformations

In the introduction to his 1971 study of FORTRAN programs, Knuth states "Our experience has suggested that frequency counts are so important they deserve a special name; let us call the collection of frequency counts the profile of a program." "There are strong indications that profile-keeping should become a standard practice in all computer systems." [63]

Sites provides a much stronger endorsement of profiling in 1978, saying "Statement counting is the single most useful tool that a programming system can provide to the user. Counting represents a simple, uniform, reliable mechanism which delivers a wealth of information." [93]

Indeed, we have been counting ever since; FDO is the result of automated collection and analysis of profile information. Many compiler analyses and code transformations have been reported to benefit from FDO, or require that profile information is available[7]. The literature contains (non-exhaustively) works using FDO for instruction set selection [66]; register allocation [98]; code placement [79, 45]; (speculative) partially-dead and (speculative) partially-redundant expression elimination [62, 74, 50, 51, 86]; superblock formation [56, 32, 103], hyperblock formation [71, 36] and other compilation-region enlargements [52]; hot-cold code splitting (partial inlining, outlining) [106, 6]; switch-case optimization [105]; call specialization for polymorphic and indirect calls [49]; and pointer and alias analysis [34, 92, 97]. The application of FDO to these transformations is discussed in detail in Appendix A.

Some compilers use profile information to guide code transformations for which no results have been reported. For instance, the IBM XL compiler uses value profiles to specialize the denominator of division instructions and to replace generic memory allocations by size-specialized, pooled custom allocators [58]. Loop fu-

[7]Compilers can estimate vertex and/or edge profile information using simple heuristics, *e.g.*, loops iterate 10 times, or more sophisticated analysis [10].

sion, loop unrolling, and loop peeling are also guided by profile information (see Chapter 4). GCC can use profile information for (at least) procedure placement[8] and loop transformations [81]. The Intel C Compiler [57], the Visual Studio compiler from Microsoft, the Open Research Compiler, and others also support FDO. These implementations of FDO in popular open-source and commercial products demonstrate that FDO is widely believed to improve a compiler's ability to generate efficient code.

Profiling is also used to guide a compiler when inserting instructions for software-controlled hardware adaptation to reduce power consumption. For instance, in multiple clock domain processors, the clock frequency of different processor components can be controlled independently. Magklis *et al.*. use context-sensitive, call-site sensitive call profiles that includes loops in order to find program sections that execute a sufficient number of instructions to warrant adjusting clock frequencies [70]. A detailed architectural simulation of the selected sections determines the use of functional units by the instructions in each section. A scheduling analysis of the dependence graph for the uses of the functional units finds a balanced execution-speed reductions for each use that consume the slack between instructions off the critical path. For each section and each clock domain, a histogram summarizes the distribution of the minimum frequency required by each use of each functional unit so that the critical path is not extended. A frequency for each domain is selected such that the critical path is not extended by more than a threshold. The use of single-input profiling in this work limit its application to the portions of the code executed by the profiled use case. Furthermore, the authors note that in some cases, whether or not a section is deemed "large enough" is dependent on the input used for profiling. The combined profiling and hierarchical normalization techniques presented in Chapter 5 would enhance the ability of clock-domain frequency reduction to save power without increasing execution time across a varied workload of inputs.

2.6.1 Inlining

The FDO code transformation discussed in Chapter 6 of this thesis, and evaluated in Chapter 7, is function inlining. Function inlining, or simply inlining, is well-studied both as a static code transformation and as an FDO code transformation. Procedure calls impose overhead on program execution, but, more importantly, they limit the effectiveness of many transformations. For example, in most cases an instruction scheduler cannot move instructions past a procedure call. The inliner developed in this work follows a classic design: call sites are evaluated to determine the expected benefit of inlining, weighted by call frequency from profiling, and placed in a sorted worklist. The call sites expected to most improve program execution time is inlined first, and the expected benefit of any affected call sites is re-evaluated. Inlining continues until a code-growth budget is spent.

[8] The -freorder-functions flag (§3.10 [1])

Early work by Chang proposes automatic (as opposed to manual) inlining. The inliner is guided by a context-insensitive call-graph profile identifying procedure execution frequency [31]. Functions are sorted by frequency and the most frequently executed function is selected for inlining first. A function selected for inlining is inlined at *all* call sites from which it is called, because the profile identifies the function as frequently-executed, but does not identify from where the function is called. Furthermore, a function is not inlined if it contains any calls. Inlining stops when a code-expansion budget is consumed. Hwu and Chang propose a similar approach that uses a call-site specific profile, and sorts call sites by expected inlining benefit instead of merely by frequency [30]. They identify the problems of estimating both the costs and benefits of inlining as very challenging, and use a constant value weighted by call frequency as the benefit term, and a crude code-size increase estimate as the cost term. Call sites are sorted for inlining by the difference between the estimated benefit and the estimated cost.

Arnold *et al.*. present an inlining strategy for Java that is similar to that used in modern C/C++ compilers [8]. They use a call-site sensitive call-graph profile, thus allocating procedure execution frequencies to individual call sites. Using code size expansion as the cost and call-site frequency as the benefit, call sites are inlined in decreasing cost/benefit order up to a code expansion limit. They find that a 1% code-size expansion limit accounts for 73% of dynamic calls and reduces execution time by 9% to 57%.

Zhao and Amaral investigate FDO inlining in the Open Research Compiler (ORC) [104]. The original benefit of inlining a call site in the ORC is measured by a *temperature* metric that takes the ratio of execution time spent in the callee compared to the total program execution time, weighted by the normalized frequency of the call compared to the invocation frequency of the caller. Temperature is intended to identify frequently-executed call sites to small callees, but infrequent calls to functions containing high trip-count loops will also result in a high temperature. An improved metric corrects this deficiency. A second improvement proposed in this work is adaptive inlining, which allows the temperature threshold required to inline a call site to vary with program size. Thus, smaller programs, which tend to benefit more from inlining, allow more aggressive inlining than larger programs, which tend to suffer more from the negative effects of excessive code growth.

Chakrabarti *et al.*. investigate the scalability of cross-module inlining for large applications [27]. They explore inliners that process the program call graph in both a top-down and a bottom-up order, as well as a worklist-based inlining order based on estimated benefit of inlining each call site. From these three alternatives, the worklist inlining order results in faster compilation that uses less memory. As well, the worklist approach produces faster application code because the orderings based on program structure frequently inline low-benefit call sites early in the process that suppress inlining of more beneficial call sites that occur later in the order. A further study by Chakrabarti and Liu presents a worklist-based inliner that updates the expected benefit of inlining candidates each time a candidate is selected for in-

lining [26]. In their implementation, the selection of all the call sites that will be inlined precedes any actual code transformation. Consequently, elaborate heuristics are required to predict the impact of inlining on the involved functions and to then update the expected benefit of inlining any associated call sites. They show that updating the expected inlining benefit of call sites to reflect previous inlining decisions improves program performance by up to 7% compared to inlining using the same worklist algorithm without such updates.

Lokuciejewsk *et al.*. use machine learning to optimize inlining heuristics to minimize the worst-case execution time of embedded kernels [69]. They use simulation to determine that inlining can degrade worst-case execution time by up to 60% when using a typical top-down static inliner. However, a learned heuristic based on (in order of importance) callee size, the number of calls in the caller, worst-case execution time of the callee and caller, and register pressure, provides an inlining order based on expected worst-case execution time that prevents these performance degradation.

Sewe *et al.*. enhance the Jikes RVM inliner by estimating the impact of subsequent transformation opportunities enabled by inlining [87]. Specifically, they predict whether or not call sites in an inlined callee will also be inlined. While the proposed technique produces accurate predictions of future inlining and often reduces compilation times compared to the default inliner, total program execution time for the programs and input sets in the DaCapo benchmark suite are largely unaffected. The best geometric-mean improvement across the workload of inputs for each program is an improvement in total execution of 7% for `bloat`; the next-best improvement is 5% for `fop`. Other results are within $\pm 2\%$ of the default inliner. However, scatter plots of the results for the individual inputs show significant overlap between the results from the two inliners; except for a small improvement for `chart`, the differences in total execution time are most likely not statistically significant (confidence measures are not reported). In the case of `fop`, the difference in workload-performance is due entirely to a single poorly-performing outlier for the default inliner.

2.7 Conclusion

This chapter reviews the terminology used throughout the rest of this document, and discusses the proper evaluation of FDO systems. A cross-validation technique that addresses this need is presented in Chapter 3. The results from the literature highlighted in this chapter demonstrate that program behavior varies between inputs, and that this variation can significantly change measured performance outcomes. At the same time, profile information is used to guide a plethora of code transformations. In particular, inlining has received significant attention. Chapter 5 presents *combined profiling*, a technique that allows FDO based on these inter-run behavior variations, while Chapter 6 describes a feedback-directed inliner that uses combined profiles.

Chapter 3

Performance Evaluation Using Many Inputs

The SPEC CPU benchmark suite is likely the most significant performance evaluation tool for computer systems and compilers available today. SPEC CPU is the standard used for performance comparison across platforms. SPEC scores are used both as targets for development teams and as a quantification of performance advantage for sales teams. The SPEC CPU programs are a component of the testing framework for compiler teams, both to ensure functionality and to detect performance bugs or new performance-enhancing transformation opportunities. These industrial uses of the SPEC CPU are the primary concern of SPEC's industrial partners, and SPEC CPU fulfills this role very well.

Another significant group of consumers of SPEC CPU are academic researchers. While researchers seldom run the suite as prescribed by SPEC, and hence seldom publish reportable SPEC performance scores, the benchmark programs are well-known to the research community and provide a common framework for the discussion and comparison of research results. Furthermore, SPEC CPU is endorsed by industry as representative of a wide range of important applications, and comes complete with program input sets. Thus, using the benchmark suite minimizes research time for experimental design while providing an implicit suggestion of the general applicability of experimental results. Therefore, in a very significant way, the SPEC CPU benchmark suite guides compiler and architecture research.

While the academic community is indebted to SPEC for the value provided by its benchmark suites, we also have an interest and responsibility to help ensure that the benchmark suites used in research, including SPEC CPU, maintain the highest possible levels of scientific utility and integrity. The particular concern of this work is with the evaluation of FDO compilers. FDO is not widely regarded as robust: it may not improve program performance (and may possibly hurt performance). Rightfully, users worry about the representativeness of the training inputs they might select for the learning phase of the FDO process. Although FDO is disallowed when reporting base SPEC scores for CPU 2006 (the most recent version of the CPU benchmark suite), our concern does not lie in whether or not FDO is

allowed when reporting these scores. Instead, we want FDO to be evaluated in a scientifically sound manner, both in industry and in academia. Therefore, while the proposals in this chapter are not specific to the context of performance evaluation using SPEC CPU, applicability in that context is the driving motivation behind them.

There is both industrial and academic demand for benchmarks with effective methodologies for the evaluation of FDO and related technologies. Robustness (or performance stability) is one of the key evaluation criteria for FDO. A benchmark suitable for FDO should asses performance along three dimensions:

1. The range of application domains
2. The typical program inputs used for evaluation
3. The input(s) selected for training

The current structure of the SPEC CPU, and of most other benchmark suites, is already designed to address the first dimension, since it is a classic concern independent of FDO. However, in most suites where FDO is allowed, even in SPEC CPU, the second and third dimensions are neglected. In the absence of FDO, training inputs are irrelevant, and diversity in the set of evaluation inputs is (arguably) a smaller concern. However, when FDO is used, the program is specialized according to the training input(s), and input sensitivity issues should be considered. Unfortunately, the academic community has generally suffered from the same deficiencies as SPEC CPU with regards to the use of sound evaluation methodologies for FDO.

Most of this chapter was previously reported [17]. However, the previous version erroneously computed program performance using arithmetic means of normalized times. This presentation corrects that oversight by computing all summarized statistics of normalized values using the geometric mean.

The following section discusses the current construction and implementation of SPEC CPU. Section 3.2 details the proposed evaluation methodology for FDO, while Section 3.3 discusses the practicality of the proposal. Section 3.4 offers some concluding remarks.

3.1 The SPEC CPU Benchmark Suite Methodology

As its name suggests, SPEC CPU is designed to evaluate the core computing capability of a computer system. Therefore, the benchmark programs are all CPU and/or memory intensive, with minimal interaction with the operating system or I/O devices. In order to improve the performance measured by the suite, a hardware or software solution must reduce the execution time of the benchmark programs. Hardware solutions increase the speed, capacity, and efficiency of the processor and/or memory system. Compilers reduce a program's need for computation or memory access, or improve the interleaving of those needs to increase instruction-level parallelism or hide memory latency.

The programs in SPEC CPU are divided into two categories: INT and FP. The FP programs spend a large portion of their execution doing floating-point arithmetic operations, and are often numerical or scientific applications such as physics or chemistry simulations. Ideally, an FP benchmark loops over large arrays of data with the loop bodies containing a large amount of (floating-point) computation but little control flow. In contrast to FP, the INT, or "integer", programs spend most of their time in non-floating-point instructions. They typically have complicated control flow and process data in linked data structures. Compilers, interpreters, and text or media processing applications fall into the INT category.

3.1.1 Programs

Most useful programs cannot be immediately incorporated into a benchmark suite. For portability and consistency, system-specific implementation details must be minimized. Furthermore, programs are frequently modified to better suit the evaluation objectives of the benchmark suite.

For SPEC CPU, regular programs are converted to compute and/or memory-intensive benchmarks by minimizing file I/O. For instance, `bzip` replicates input data in memory until a size threshold has been reached rather than reading in large files. The benchmark version of `bzip` also leaves the compressed data in memory and does not produce the expected output file.

`Bzip` is an instructive example of the benchmark-conversion process. As a compression and decompression program, `bzip` has several mutually-exclusive use cases: compression using one of the nine mutually-exclusive block sizes and decompression cannot be combined into a single program run. The benchmark version of `bzip` is modified to compress the (replicated) input data three times in memory, using block-size settings of 5, 7, and 9. After each compression pass, the data is decompressed (also in memory) to enable correctness verification. Thus, each run of SPEC `bzip` bundles together six mutually-exclusive runs of the non-benchmark version of the program.

The benchmark conversion of `gcc` removes the preprocessor, and has unforeseen complications. Without the preprocessor, new programs cannot be used directly as benchmark inputs. Furthermore, this `gcc` accepts only a single file. Creating a new input for `gcc` thus requires preprocessing the source files using the same (very old) version of gcc used for the benchmark, and then manually combining of these files into a single input file. Unfortunately, this combination process requires the manual resolution of all the ordering, name-mangling, and scoping issues that are usually taken care of by the compiler. The least-problematic approach involves concatenating all the source files together (destroying file scope), removing multiple header inclusions and macro definitions, and then identifying symbols that occur in multiple files but that have different definitions. These symbols are then manually renamed in their original source files to allow find-replace renaming and to ensure that the original scope is preserved. The process is then restarted, and

iterates until the pre-processed combined file compiles successfully. Consequently, it is very difficult to create new `gcc` inputs from any program with more than a few source files. While this concern is irrelevant for a typical user of the benchmark suite, it is problematic for researchers or future benchmark authors who need to modify or augment the `gcc` workload.

3.1.2 Program Inputs

SPEC provides three workloads for each benchmark: test, train, and ref. Each benchmark program has one test input that is not intended to be used for performance evaluation. Rather, it is a minimal program run used to demonstrate that the benchmark suite has installed properly on the target system, that the benchmark program has compiled successfully, and that no obvious problems exist to prevent the benchmark from running.

Each benchmark also has one train input, intended for use as the training run for FDO compilers. This input provides a short run that should be "representative" of the ref workload. As discussed in Section 2.3, there are few guidelines to instruct benchmark authors on how this input should (and should not) be chosen, resulting in several train inputs that sample the ref workload.

The ref workload for each program is usually also a single input. One long run instead of several shorter runs means that program loading time and any initialization activity is only included in the evaluation once.

The test inputs are not suitable for evaluation, and studies show them to be non-representative of typical (*i.e.*, train or ref) program execution [61]. Nonetheless, research, including work presented in this document, has used the test inputs for evaluation. Two reasons exist for this decision: the ref inputs are too large for simulation-driven architecture studies; and research often requires more than a single evaluation run. Indeed, proper evaluation of any system that uses profile information requires far more evaluation inputs than are currently provided with any benchmark in SPEC CPU. In the absence of a proper evaluation workload, any and all available inputs are used.

3.1.3 Measuring Performance

SPEC CPU performance scores are computed relative to a historical baseline machine. Larger scores are better, and many regulations govern how the performance evaluations can be done and how the results must be reported. Execution time is measured as the average of three runs on the ref workload. SPEC CPU reports two different scores: base and peak.

The base score is computed by using an identical system configuration for every benchmark in the suite. In particular, each program is compiled with the same set of compiler optimization flags, and FDO is not permitted. Base represents the performance available to users who cannot or do not tune their compiler settings for

individual applications.

On the other hand, the peak score allows each benchmark program to be treated individually. Each program can be compiled with a hand-picked set of compiler flags that produce the best results for that program. It is well known that aggressive code transformations may only be beneficial for some programs, and can hurt the performance of others; these transformations are usually not enabled by default. When reporting a peak score, these transformations can be selectively enabled for the programs they benefit. FDO is one such compilation option and has traditionally been allowed when computing peak scores[1]. The peak score represents the performance potential of the system for users who demand the best possible application performance and are willing to spend the time and effort to get it.

3.2 Proposed Benchmark Methodology

FDO is a different compilation technique than traditional non-FDO compilation, with different performance issues and consequently different requirements for a performance-evaluation methodology. Therefore, instead of considering FDO "just another optimization" and allowing FDO in reported peak scores, we propose that FDO be disallowed for peak scores, but also that along with base and (non-FDO) peak, an optional *FDOPeak* score be reported for FDO performance. The compilation rules governing *FDOPeak* should be the same as those for peak, simply extended to allow FDO.

The proposed FDO evaluation methodology is based on cross-validation, and incorporates the requirements of this technique with the constraints and concerns of performance evaluation for high-performance systems. However, since most compilers have, at best, limited support for multi-run profiling, allowances are made for single-run training workloads that nonetheless encourage the use of full multi-run training workloads.

3.2.1 Programs and Benchmark Conversion

It is appropriate for some program modifications to be made when constructing a portable benchmark suite. It is also appropriate to omit file I/O time from a CPU (as opposed to a full-system) benchmark. However, as much as feasible, converting a program into a benchmark should preserve the functionality of the original program. For example, in the case of `bzip`, the program should simply be run multiple times to cover the desired use cases, rather than modifying the program to operate in an unrealistic manner. In order to avoid file I/O, inserting timing points inside a benchmark program to exclude that I/O (or measure and report it) is a less drastic, and likely easier and more general, solution than making significant modification to the original program.

[1] Though CPU 2006 disallows using FDO for base scores, its use is allowed in peak scores ([48], rule 2.1.3).

3.2.2 Evaluation Workload

Cross-validation requires a workload of inputs. We call this set of inputs the *evaluation workload*, $\mathcal{W}$. The intuitive guideline for the evaluation workload is that the inputs in the workload should be as varied as possible and should attempt to cover or sample the space of program inputs, biased toward typical program inputs but not selecting these types of inputs exclusively. Rather than attempting to find one input that is universally representative of all program runs, the workload as a whole will be representative of program usage. For example, one input could correspond to a particular use-case of the program, and another input could correspond to a different use-case. These two inputs can be completely unrelated, and neither need try to represent all common uses of the program. Program behaviors that are common independent of input characteristics will be universally represented, while input-dependent program behaviors will be represented in proportion to their occurrence in the workload. Consequently, the more important a particular program behavior is to overall program performance, the more frequently it will be represented in the workload.

The inputs selected for the FDO workload should be independent of each other. Input independence is difficult to define precisely, but intuitively, two inputs should never be identical, and the data in two inputs should not overlap or have a common origin. For example, a particular chess position should not occur in multiple input files, images should not be subsections from a common source, or a matrix of data should not be sampled from another matrix used in a different input. Randomly generated data should be avoided because it is usually unrepresentative of real data, and multiple samples randomly generated in the same way will likely have very similar characteristics. If the program accepts multiple data formats, $\mathcal{W}$ should contain examples from as many of these formats as practical. If the data has implicit or explicit dimensionality, different elements of $\mathcal{W}$ should avoid repeating the same dimensionality.

Each input in $\mathcal{W}$ may potentially be used for both training and evaluation (but only in different evaluation contexts). Therefore, these inputs must run long enough to provide meaningful performance measures, but must also be small enough that training runs (possibly an order of magnitude slower than an optimized non-training run) complete in an acceptable time. Furthermore, these constraints should continue to be met as machine performance increases during the lifetime of the benchmark. Therefore, we suggest that inputs in $\mathcal{W}$ should initially have a (non-training) running time of approximately 2 minutes on recent systems.

3.2.3 Training and Evaluation

Evaluation of FDO uses a 3-fold cross-validation strategy in order to minimize the number of FDO compilations required for evaluation while still using every input in $\mathcal{W}$ for both training and evaluation. The evaluation workload is randomly split into three non-overlapping partitions, each containing an equal number of inputs.

Let A, B, and C be the names of the three partitions. These partitions may be specified as part of the benchmark design in order to provide determinism in the evaluation, or may be selected at random on each evaluation in order to prevent the exploitation (intentional or accidental) of any particular interactions within a predetermined partitioning.

One at a time, each partition (say A) is selected as the *testing workload*, $\mathcal{W}_{test}$. The remaining inputs ($B \cup C$) form the *training workload*, $\mathcal{W}_{train}$. A training run using one or more of the inputs in $\mathcal{W}_{train}$ produces the profile(s) used by the compiler to guide optimization. If the compiler cannot use profile information from all inputs in $\mathcal{W}_{train}$(*e.g.*, it can only train on a single input), then the subset of inputs that the compiler uses for training is selected from $\mathcal{W}_{train}$ in order, according to an ordered input list supplied with the benchmark. Let the version of the program where $\mathcal{W}_{test} = A$ be called P_A. P_A is run on each input in A. Likewise, P_B is run on B and P_C is run on C. Thus, each input in W has been used twice for training, and once for evaluation. However, since each profiling run is independent, only one profiling run per input is needed; the compiler (or an external tool) should combine those individual profiles as needed for FDO training.

A rule of thumb suggests that at least 5 independent measurements are required to make statistical significance tests worthwhile. More than 30 evaluations would typically be considered a large sample size, which enhances the reliability of statistical measures. However, benchmark running time constraints and benchmark author resources place practical limitations on how many inputs can be collected and used. In order to conveniently facilitate the cross-validation methodology presented here, the number of inputs should be a multiple of 3. To enable meaningful statistical measures of confidence, a minimum of 15 independent inputs are required, since $\frac{15}{3}$ provides 5 independent performance measures of the FDO version of the benchmark produced for each $\mathcal{W}_{test}$.

3.2.4 Reporting FDO Performance

FDO performance should be reported as a geometric mean of execution time speedups compared to the program used to report peak performance.

Program execution time on input i measured as the arithmetic mean running time of an odd number (at least 3) of executions on input i. The baseline execution time for input i, $t_\emptyset(i)$, is measured using the version of the program used to report the SPEC peak score. During the cross-validation process described above, each input i is used for evaluation exactly once. Input i belongs to only one partition of $\mathcal{W}$, and thus implicitly defines the training set u as the union of the two partitions that do not contain i[2]. Let $t_u(i)$ be the program execution time on i when FDO is informed by training on u. To simplify notation, the training set u is left implicitly defined by i. The speedup obtained by using FDO, as evaluated on i, is $\tau_u(i) = \frac{t_u(i)}{t_\emptyset(i)}$.

[2]Compilers that train on a single input select this input from u according to the ordered list of inputs.

The *FDOPeak* score is the geometric mean speedup compared to peak observed in all FDO evaluation, computed as the $|\mathcal{W}|^{th}$ root of the product of speedups:

$$\textit{FDOPeak}(\mathcal{W}) = \sqrt[|\mathcal{W}|]{\prod_{i \in \mathcal{W}} \tau_u(i)}$$

In addition to reporting *FDOPeak*, the standard deviation of the speedups must also be reported, using the *geometric* standard deviation:

$$\textit{FDODev}(\mathcal{W}) = \exp\left(\sqrt{\frac{\sum_{i \in \mathcal{W}} (\ln \tau_u(i) - \ln \textit{FDOPeak})^2}{|\mathcal{W}|}}\right)$$

Sample arithmetic means are normally distributed by $N(\mu_a, \sigma_a)$, leading to confidence intervals of the form $\mu_a \pm z\sigma_a$ where μ_a is the arithmetic mean, σ_a is the standard deviation, and z is the number of standard deviations needed to achieve the desired confidence level, determined from the standard normal's PDF. Using the geometric mean is the same as calculating the arithmetic mean after the data has been transformed by taking the logarithm of each data value. The geometric mean (μ_g) and geometric standard deviation (σ_g) are the parameters of the resulting log-normal distribution. Consequently, the usual confidence interval around the geometric mean is not symmetric. However, the bounds of this interval can be computed by first taking the logarithm of μ_g and σ_g, determining the limits of $\mathrm{CI}_a = \mu_a \pm z\sigma_a$ in this space, and then re-applying exponentiation to get CI_g in the original space. Let $\mathbf{A}$ represent a vector of the $\tau_u(i)$ data for $\mathcal{W}$:

$$\begin{aligned} \ln \textit{FDOPeak} &= \ln \mu_g(\mathbf{A}) = \mu_a(\ln(\mathbf{A})) \\ \ln \textit{FDODev} &= \ln \sigma_g(\mathbf{A}) = \sigma_a(\ln(\mathbf{A})) \\ \ln \mathrm{CI}_g &= \mu_a(\ln(\mathbf{A})) \pm z\sigma_a(\ln(\mathbf{A})) \end{aligned} \tag{3.1}$$

Now, solve for the end-points of CI_g in terms of $\mu_g(\ln \mathbf{A})$ and $\sigma_g(\ln \mathbf{A})$. The $\ln \mathbf{A}$ parameters are only required to connect the arithmetic and geometric means when

setting up Equation 3.1; these parameters are henceforth omitted:

$$\begin{aligned}
\ln \mathrm{CI}_g &= \left[\mu_a(\ln \mathbf{A}) - z\sigma_s(\ln \mathbf{A}) \quad , \quad \mu_a(\ln \mathbf{A}) + z\sigma_s(\ln \mathbf{A})\right] \\
\ln \mathrm{CI}_g &= \left[\ln(\mu_g) - z\ln(\sigma_g) \quad , \quad \ln(\mu_g) + z\ln(\sigma_g)\right] \\
&= \left[\ln(\mu_g) - \ln(\sigma_g^z) \quad , \quad \ln(\mu_g) + \ln(\sigma_g^z)\right] \\
&= \left[\ln\left(\frac{\mu_g}{\sigma_g^z}\right) \quad , \quad \ln\left(\mu_g \times \sigma_g^z\right)\right] \\
\mathrm{CI}_g &= \left[\exp\left(\ln\left(\frac{\mu_g}{\sigma_g^z}\right)\right) \quad , \quad \exp\left(\ln\left(\mu_g \times \sigma_g^z\right)\right)\right] \\
&= \left[\frac{\mu_g}{\sigma_g^z} \quad , \quad \mu_g \times \sigma_g^z\right] \\
&= \left[\mu_g \times \sigma_g^{-z} \quad , \quad \mu_g \times \sigma_g^z\right] \qquad (3.2)
\end{aligned}$$

Equation 3.2 gives the geometric analogue of the $\mathrm{CI}_a = \mu_a \pm z\sigma_a$ expression for the confidence interval of arithmetic means. The addition of a multiple of the standard deviation in the arithmetic form is promoted to multiplication by a power of the standard deviation, just as the geometric mean uses multiplication in place of addition and exponentiation in place of multiplication when compared to the arithmetic mean. Substituting in *FDOPeak* and *FDODev*:

$$\mathrm{CI}_g = \left[\mathit{FDOPeak} \times \mathit{FDODev}^{-z}, \mathit{FDOPeak} \times \mathit{FDODev}^{z}\right] \qquad (3.3)$$

The value of appropriate value of z for a particular confidence interval is still based on the normal distribution, so the bounds of a 95% confidence interval for *FDOPeak* can be computed with Equation 3.3 by setting $z = 1.96$ as usual.

Clearly, compiler designers will strive for a high *FDOPeak* score. In order to achieve this goal, FDO must improve the performance of the program over a large number of unseen inputs. Furthermore, the training phase does not guarantee that selecting a single input from the training set will provide a particularly representative training run. However, using more than one input during training (in particular, all of the training set) likely provides a more representative sample of program execution than any single training input could. Therefore, in order for an FDO compiler to achieve a high *FDOPeak* score, it must be both robust in the face of the possibility of a poorly chosen training input, as well as in the face of potentially varying behavior in evaluation inputs. Of course, a good way to avoid problems with an unfortunate training input is to use multiple training inputs. The nascent capability to use multiple training inputs is already present in several commercial compilers, including the IBM XL compiler [58] and the Intel C++ Compiler [57].

Additionally, the standard deviation is a critical metric, because it provides confidence for the *FDOPeak* speedup value. If the lower bound of CI_g in Equation 3.3 is greater than 1, we have confidence (determined by z) that FDO does improve

program performance. Even if *FDOPeak* shows a large speedup over peak, we cannot be confident that FDO improves performance for the program if CI_g is large compared to that difference. In essence, the standard deviation gives a measure of the robustness of FDO in improving performance, which is a key factor that must be considered when FDO is evaluated.

While SPEC scores use the geometric means of speedups over a baseline machine, performance evaluation frequently uses normalized execution time ($\tau_u^{-1}(i)$) instead. The geometric mean of normalized execution times for $\mathcal{W}$, measured by 3-fold cross-validation, and its geometric standard deviation are denoted:

$$\mu_g(\mathcal{W}) = \frac{1}{\textit{FDOPeak}} = \sqrt[|\mathcal{W}|]{\prod_{i \in \mathcal{W}} \tau_u^{-1}(i)}$$

$$\sigma_g(\mathcal{W}) = \exp\left(\sqrt{\frac{\sum_{i \in W}(\ln \tau_u^{-1}(i) - \ln \mu_g(\mathcal{W}))^2}{|\mathcal{W}|}}\right)$$

Ideally, the performance differences between alternative evaluations, such as between *FDOPeak* and peak, would be consistent and large enough to require little statistical analysis to establish the superior result. After all, a statistically-significant result that is inconsequential in practice has little value. However, the standard deviation of a data set is a measure of the spread of values from the mean. Thus, while useful for assessing the generality of average performance results, confidence intervals on the mean of a high-variance data set can obscure real improvements in mean performance. Statistical location tests can determine if the difference between the means of two data sets, such as the evaluations for *FDOPeak* and peak over a workload, is significant [100]. Many such *hypothesis tests* exist, such as the Student's t-test or the Wald test. Each test starts with a null and an alternative hypothesis. The test indicates if there is sufficient evidence in the data to reject the null hypothesis. Tests with higher *power* reduce the chance of false negatives, the situation where a real difference exists, but the test fails to reject the null hypothesis. In the typical setup for this case, the null hypothesis states that the the sample means for *FDOPeak* and peak are the same; the alternative hypothesis states that they are different. Both *FDOPeak* and peak are computed on the same workload of inputs. Thus, a paired difference test should be used to compare the two sets of evaluations. Not only does the paired difference approach account for the two data sets not being independent, but it also increases a test's power compared to a non-paired approach. Thus, a paired difference test may determine the statistical significance of an improvement in mean workload performance in spite of large performance variations across the workload.

Additional utility can be added to the results of hypothesis testing if the question of significant performance-impact is framed as an equivalence test, as suggested by Hoenig and Heisey [54]. Instead of taking the null hypothesis as the equivalence of *FDOPeak* and peak, the null hypothesis states that FDO produces a *practically*

significant performance impact, say, a 1% or 2% difference from peak. The alternative hypothesis states that the impact of FDO is not practically significant. The usual hypothesis testing methodology is applied to determine if the null hypothesis should be rejected.

3.2.5 Combining Workloads for *FDOPeak* and Peak

Traditionally, SPEC CPU benchmarks have included a ref input or workload of inputs used for the base and peak performance evaluations. In cases where ref is a workload, it is likely appropriate to let ref and $\mathcal{W}$ be the same set of inputs. This unification of ref and $\mathcal{W}$ not only reduces the input selection burden on benchmark authors, but also reduces the computational overhead required to get the running time measurements required to compute *FDOPeak*. Furthermore, even in the case where FDO is not used, evaluation on a workload of inputs provides statistical confidence measures that evaluation on one large input cannot. Thus, a transition to a model that always uses a workload of inputs for evaluation is beneficial in all cases.

However, in some rare cases it may be necessary for ref to consist of $\mathcal{W}$ plus one or more large inputs that are not suitable for inclusion in $\mathcal{W}_{train}$. In these cases, two options are available. The simplest option is to leave the evaluations of base/peak and *FDOPeak* completely independent. However, a potentially more informative approach is to let those large ref inputs be added to each $\mathcal{W}_{test}$ but never used for training. In this way, the performance of FDO on these large (and presumably more important) inputs is taken into account. If this method is used, it may be beneficial to reformulate the equation for *FDOPeak* to use a weighted mean instead of an unweighted mean, and to give more weight to those large inputs. These weights must take into account that these non-training inputs will always be part of $\mathcal{W}_{test}$, and will consequently be used for evaluation three times.

3.3 Practicality Considerations

A cross-validation approach to FDO evaluation provides performance measurements that are more reliable than traditional approaches. However, this improved evaluation does incur a cost, and it is important that the cost is not prohibitive for any stakeholders.

For compilers without multi-run profiling and FDO, an alternative approach to that presented here is a *leave-one-in* evaluation strategy. Under this methodology, each input is used in turn for training and FDO individually; each version of the program produced this way is evaluated on the entire evaluation workload except that training input. This approach provides a thorough evaluation of a single-profile FDO compiler across both the training input and evaluation input dimensions. However, if $|\mathcal{W}| = n$, n FDO compilations are needed, and evaluation runs on all $n(n-1)$ training-testing input pairings are required. Clearly, the cost of

this method is impractical for all but the smallest workloads. The proposed 3-fold cross-validation method keeps the number of compilations constant (3) and needs n evaluation runs.

3.3.1 Compiler Users

The performance results produced using benchmarks are intended to help end-users of computer systems estimate the performance of various systems for their application workloads. If multiple training inputs are required to reliably obtain the level of performance indicated by the proposed evaluation methodology, end users should also adopt a multi-input training policy. Despite the immediate impression that such a training methodology would increase the burden on FDO users, training on multiple inputs should not be a significant issue for the class of performance-sensitive users who are interested in FDO. Such users typically maintain sets of inputs to use for regression testing, both for program correctness and program performance. These input sets will include the important use cases of the program. Instead of requiring the user to develop a specific training input that attempts to cover all those use cases (in order to be representative), the user is freed to simply train on all the use cases they have already identified. While the training time may be extended, the training process can be be trivially parallelized. Meanwhile, the human effort required to create and maintain a representative training regiment is significantly reduced.

3.3.2 Compiler Developers

A primary objective of a compiler developer is the generation of the fastest possible programs from source code. For important clients, compiler developers may interact directly with the client in order to minimize program execution time. FDO is one option for improving program performance. When all parties are accustomed to using a rigorous training and evaluation methodology, compiler developers can immediately focus on analyzing program behavior and on the impacts of code transformations, rather than working to ensure that the client is using a good training input.

More importantly, during the development of FDO transformations, evaluation on a workload ensures that these transformations provide reliable performance gains despite variations in input data. Since FDO currently uses a single training input, compiler developers know that a profile is merely a hint, rather than a proper characterization of program behavior. A compiler that uses multiple profiling runs with FDO can place greater trust in the generalizability of that profile information and thus exploit it more aggressively, but only when using a sound evaluation methodology such as the one proposed here. Under these circumstances, the developer can recommend FDO compiler options to clients with greater confidence.

3.3.3 Benchmark Users

When properly incorporated into a benchmark suite, cross-validation should be nearly invisible to a benchmark user. The multiple training runs, evaluations, and performance measure calculations should all be performed automatically by the benchmark framework. The only task for the user should be to specify the correct arguments to the compiler to enable the creation and use of program profiles, and possibly to specify the maximum number of inputs to use for training, if this is a limitation in the profiling or FDO implemented by the compiler.

The proposed training and evaluation methodology is not significantly more expensive in terms of computation time than the traditional methodology. Assuming that the evaluation workload is unified with the ref workload, the baseline measurements for speedup comparison are taken care of by a standard non-FDO run of the benchmark. Training requires instrumented runs on at most $|\mathcal{W}|$ training-sized inputs, but only three additional program compilations. Evaluation requires one additional set of runs on each input.

3.3.4 Benchmark Authors

Benchmark authors face the largest burden from the proposed methodology. Each benchmark author must select the inputs used with the program. Authors have two options when selecting the training set. They may use their expert knowledge to carefully consider a number of inputs and select those that expose different program behaviors or present distinct program use cases. Alternatively, they may select a large collection of inputs and then use a clustering technique, perhaps as presented in Chapter 4, to determine redundancy in the original collection.

The ease with which many inputs can be gathered or generated is an important consideration when proposing the collection of a workload of inputs. Most applications in wide use collect sets of inputs for correctness and performance testing; these sets provide a good starting point to construct the evaluation workload. Based on the documentation of SPEC CPU 2006 benchmark programs [94], there should be little difficulty obtaining inputs for integer-style benchmark programs:

`perlbench`, `gcc`, `xalancbmk` There are large repositories on the web of C, C++, and Perl programs, and each compiler or interpreter maintains sets of example and testing inputs. XML documents should be easy to obtain. The ref workloads for these programs already consist of several inputs.

`bzip2` Any file can serve as a valid input for a compression program. The `bzip2` reference workload consists of several inputs.

`mcf`, `omnetpp`, `astar` The algorithms implemented by these programs work on the provided topology. Commodity flow graph, network topologies and path-finding maps may not be plentifully available in the particular formats required by these programs, but the use of format conversion scripts or other

input generators should allow for the collection of many inputs with moderate effort from the authors.

`gobmk`, `sjeng` Both go and chess board positions are easily generated. Furthermore, given that programs for both games compete at events such as the Computer Olympiad [3] and the World Chess Championship [4], collections of board positions in standard formats should be available.

`hmmer` Many online, publicly accessible databases for protein sequences and related information exist, such as the Swiss-Prot [5] database.

`libquantum` Number factoring requires only an integer as input, and an optional base for modular exponentiation.

`h264ref` Video streams are plentiful on the Internet. Furthermore, various video characteristics impact encoders, such as the amount of action in a scene, gray-scale ("black and white" movies) or color, and animated or live-action.

A scan of the SPEC 2006 floating-point programs presents modeling and simulation tools, equation solvers, a rendering engine and a speech-recognition program. Changing system parameters, data sizes, material properties, and/or the scenario presented by the input files should lead to the creation of collections of inputs, in the absence of real-world input sets. However, profile-guided optimizations are typically most beneficial to integer programs, and have far less impact on scientific codes.

In short, the burden of collecting a set of inputs for cross-validation does not appear to be significant in most cases, as represented by SPEC CPU 2006.

3.4 Conclusion

FDO requires a more robust evaluation methodology than traditionally used for the performance evaluation of computer systems. The methodology presented here provides a cross-validation approach to FDO evaluation that avoids the problem of over-fitting the training data while providing statistical confidence measures for the performance results. Furthermore, while the proposed methodology is effective for the current single-profile approach to FDO, it encourages FDO compilers to use a multi-run profiling approach to enhance the compiler's behavior modeling and behavior prediction capabilities.

Chapter 4

Selecting Workloads of Inputs

Performance evaluation is usually envisioned as a post-hoc characterization of a system; evaluation is performed once the system is completed. This view is incorrect. First, most software is never completed; versions are released at certain milestones, but development is continuous. Furthermore, the development of any performance-critical system will either continuously or periodically use performance estimation to guide development. For instance, compiler vendors routinely assist clients in using the compiler to improve the performance of the client's programs. In cases involving the most important customers, members of the compiler development team may be involved in this process, and may implement improvements in the compiler to better optimize the client's code. These changes will be evaluated both on the client's program as well as on the in-house performance evaluation suite.

Input selection is an important dimension of performance evaluation. Chapter 3 discusses how multiple inputs should be used when evaluating FDO. However, consider the case where a large number of inputs are available. How should a developer considering FDO transformations proceed if the client provides hundreds, or thousands, of inputs to the compiler team? Using such a large set of inputs is computationally prohibitive, and is not suitable for repeated performance evaluations as the compiler evolves. Each additional input used for training and/or evaluation increases the time required to evaluate performance, due to additional training runs and execution-time measurements. This problem is particularly pronounced for compilers limited to single-profile FDO, since leave-one-in evaluation requires a number of evaluation runs quadratic in the number of inputs. On the other hand, if the compiler supports the simultaneous use of multiple training inputs (perhaps as proposed in Chapter 5), the client cannot be expected to use such a large set of inputs for training in their build process.

Therefore, a minimal set of representative inputs is required to reduce the time consumed by performance evaluation, without compromising evaluation quality. The method used to reduce the workload should not rely on human intuition or the experience of experts. For large, complicated programs, predicting the interactions between the program, the compiler, data inputs, and the underlying computer archi-

tecture is likely impossible, even for an expert of all the involved components. Thus, an automatic workload-reduction technique is useful for both compiler designers, who may not be experts regarding the client's program or its inputs, as well as for the client, who may not be an expert regarding the compiler or architecture. Furthermore, the method should measure how representative the selected inputs are of the full workload, and thus provide a quantitative estimate of the trade-off between workload size and workload accuracy.

This chapter presents a compiler-centric methodology to reduce the size of the workload needed for proper evaluation of the performance improvements achieved by an FDO compiler for a given application. Input similarity is based on the code-transformation decisions made by the compiler according to the profile generated for each input. Inputs are clustered based on the variations they induce in code transformations. This clustering produces groups of inputs to which the compiler responds in a similar fashion, and thus identifies redundancy in the training, and testing, workloads.

Furthermore, we present a novel metric to compare different clusterings on related data. This metric allows for intuitive investigation of the correlation between individual transformations and the significance of differences between clusterings as development of the application and/or compiler advance.

Previous presentations of this work reported the impact of reduced workloads on performance evaluation in Section 4.4.4 using a summary metric called LWA [19]. However, LWA is flawed because it inappropriately weights the workload performance values for each FDO version of a benchmark using the execution time of the *training* input used to create that version. This problem is corrected by instead taking a geometric mean over the workload performance evaluations of each version of the program, without regards to the (irrelevant) execution time of the training input.

The next section details the clustering technique, ϵ-greedy spectral clustering, while Section 4.2 discusses how the clustering results can be used during the development of an FDO compiler. Sections 4.3 and 4.4 present the evaluation methodology and experimental results of applying clustering to SPEC CPU programs.

4.1 Clustering

Consider a program PROG with an impractically-large workload of inputs $\Omega = \{i_1, i_2, \ldots, i_\omega\}$, and an optimizing compiler with a set $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$ of profile-directed transformations. Each input $i \in \Omega$ is profiled to create a corresponding set of profiles, $\mathcal{P} = \{p_1, p_2, \ldots, p_\omega\}$. When the compiler uses p_a instead of p_b, it may potentially apply transformations from $\mathcal{T}$ to different locations of PROG, and/or apply such transformations with different frequencies. Clustering will group inputs in Ω based on how similarly the compiler applies the transformations in $\mathcal{T}$ to PROG when using each $p \in \mathcal{P}$. The clustering then enables the selection of a usable representative $\mathcal{W}$ from Ω. The calculated input similarities are

held in a similarity matrix, which is the input for the clustering algorithm.

4.1.1 Input Features and Similarity

For a given transformation T, the set $\mathcal{L}_T = \{l_1, l_2, \ldots, l_k\}$ is the union of all locations where T is applied when compiling using any $p \in \mathcal{P}$, or during a statically-optimized baseline compilation. A transformation vector $\mathbf{V}_j$ records how many times T is applied at each $l \in \mathcal{L}_T$ when the compiler is guided by profile p_j.

Collecting transformations vectors from each compilation essentially entails profiling the compiler. For each $T \in \mathcal{T}$, each location $l \in \mathcal{L}_T$ where T may be applied is identified, as uniquely as possible, using source-code line numbers and expression identifiers. A monitor is inserted into transformation T to collect $\langle location, value \rangle$ pairs that indicate how many times T is applied at each location. Most transformations are all-or-nothing transformations: when using the profile p_j, a 1 is recorded in $\mathbf{V}_j[l]$ if T is applied at l; a 0 is recorded if T is not applied at l. For loop unrolling, the unroll factor is recorded as the *value* for the pair in the transformation vector. Due to code replication, a transformation may be applied multiple times in indistinguishable locations. In these cases, the *values* from aliased locations are accumulated at the appropriate index of $\mathbf{V}_j$.

Distance metrics, such as the Euclidean[1] and Manhattan[2] distances, are often used to compute the similarity of vectors. In this work, similarity is based on transformation decisions. The difference between two inputs increases with the number of differing transformations decisions between their vectors. An appropriate similarity metric will count how many transformation decisions differ between two vectors. Therefore, the Manhattan distance is used as the similarity metric. The Manhattan distance between two vectors is the sum of the absolute value of their index-wise differences. Geometrically, the Manhattan distance is the length of the shortest path from two points if travel is restricted to movement along a unit grid. Thus, in the context of transformation vectors, the Manhattan distance directly counts the number of differences between two vectors. As discussed above, code transformations are either applied or not applied at a particular location; they cannot be partially applied. The Euclidean, or straight-line, distance is not restricted to the unit grid, and thus would essentially allows fractional decision differences between vectors. For instance, there are two decision differences between $v_1 = [0, 1, 1]$ and $v_2 = [1, 0, 1]$. The Manhattan distance between v_1 and v_2 is 2, but the Euclidean distance is only $\sqrt{2}$.

The *difference matrix* D_T for transformation T is an $\omega \times \omega$ symmetric matrix that encodes the pairwise Manhattan distances between the $\mathbf{V}_j$ vectors for T from each of the ω profiles:

$$D_T[x, y] = Manhattan(\mathbf{V}_x, \mathbf{V}_y)$$

[1] also called the L_2-norm

[2] also called the L_1-norm

	Training Input				
Data	**A**	**B**	**C**	**D**	$t_\emptyset$**(i)**
A	-	59.07	62.08	58.74	61.76
B	71.29	-	74.15	70.09	73.35
C	4.34	4.14	-	4.14	4.29
D	110.14	108.65	115.17	-	116.89

Table 4.1: Running-times (in seconds) for an example workload when using alternative training inputs for FDO, and for the non-FDO baseline ($t_\emptyset$(i))

In order to account for all transformations during clustering, a *combined difference matrix* D includes the data from all $T \in \mathcal{T}$. Each D_T is normalized by dividing its elements by the size of $\mathcal{L}_T$. The combined difference matrix D is created by point-wise summing the normalized matrices:

$$D = \sum_{T \in \mathcal{T}} \frac{D_T}{|\mathcal{L}_T|}$$

Normalization gives each transformation equal weight in D, even if the number of transformation sites differ by orders of magnitude between transformations. Thus, a single transformation will not dominate the combined difference.

4.1.2 Performance Weighting

Compilers make a large number of transformation decisions while compiling a program. The interactions between these decisions is complex and can result in unexpected performance results. A single decision may have a large impact on performance, while many others may not make any measurable difference. In order to take program performance into account, and to attempt to filter out the inconsequential differences in transformation decisions, the elements of the difference matrix are weighted by a pair-wise performance factor. A running example will illustrate how the weights are determined. Table 4.1 presents a small set of actual execution times. The input names are omitted for clarity since this example has no relation to the results in Section 4.4. Each table row gives execution times on the listed input. Each column indicates the training input. The baseline, $t_\emptyset$(i), indicates that FDO is not used.

Table 4.2 presents the matrix D for the example inputs, using data from the seven transformation discussed later in Section 4.3.1. Training on `A` produces significantly different transformation decisions than training on the other inputs, while training on `B` or `D` results in very similar decisions.

The non-FDO baseline program is run on each input $i \in \Omega$ to provide reference time measurements ($t_\emptyset(i)$). Similarly, each of the FDO-optimized programs are run on Ω, excluding the training input for that program. A log-weighted normalized

	A	B	C	D
A	0	86.55	91.11	90.05
B	86.55	0	9.38	0.06
C	91.11	9.38	0	9.14
D	90.05	0.06	9.14	0

Table 4.2: Combined difference matrix D for the example workload

	Training Input				
Data	**A**	**B**	**C**	**D**	$log(t_\emptyset(i))$
A	-	0.96	1.01	0.95	4.12
B	0.97	-	1.01	0.96	4.30
C	1.01	0.97	-	0.97	1.46
D	0.94	0.93	0.99	-	4.76
LNP	0.96	0.94	1.00	0.96	

Table 4.3: The upper portion of the table lists the normalized run-times ($\tau_u^{-1}(i)$) and log-weights ($\log(t_\emptyset(i))$) computed from Table 4.1. The bottom row lists the per-input $\mathbf{LNP}$ values computed from each column.

workload running time is calculated for each copy of the program. Given a training input u and an evaluation input $i \neq u$, let $t_u(i)$ be the average running time of three[3] runs of the program trained on input u executing using input i. For example, the value of $t_B(D)$ in Table 4.1 is 108.65s. A log-weighted normalized performance ($\mathbf{LNP}$) vector summarizes the workload performance using each training input u in Ω:

$$\mathbf{LNP}[u] = \frac{\sum_{i \in \Omega - u} (\tau_u^{-1}(i) \times \log(t_\emptyset(i)))}{\sum_{i \in \Omega / \{u\}} \log(t_\emptyset(i))}$$

We take a throughput-oriented approach to performance weighting, assuming that Ω is representative of the real workload with respect to the relative execution times of the inputs. Thus, workload performance should be a weighted average, such that the weight assigned to relative performance compared to the baseline on any individual input is in relation to the execution time of that input. Furthermore, the performance measure should follow a cross-validation methodology, as discussed in Chapter 3. Thus, $\mathbf{LNP}[u]$ excludes values where the training and evaluation input would be the same. In the situation of acquiring many inputs from a client, there is no control over the running-times of the inputs. Consequently, the execution times for different inputs may vary significantly. A long-running input should not unduly influence the workload performance metric (a concession toward latency-oriented evaluation). Log-weighting addresses this issue, while normalizing by the weights ensures comparability between different $\mathbf{LNP}$ values. Table 4.3 shows the speedup in the execution time from Table 4.1 relative to the baseline

[3]Good experimental practice uses multiple runs to measure execution times; averages should be over an odd number of runs.

	A	B	C	D
A	0	19.82	37.21	9.31
B	19.82	0	57.77	10.41
C	37.21	57.77	0	46.87
D	9.31	10.41	46.87	0

Table 4.4: Performance Weight matrix PW (x1000), computed from the **LNP** values in Table 4.3

($\tau_u^{-1}(i)$), along with $log(t_\emptyset(i))$, to illustrate the logarithm's effect on the weights. `C` still has a smaller weight than the other inputs, but the other three inputs are assigned similar weights, even though processing input `D` takes nearly twice as long as input `A`. The final **LNP** values are listed in the last row of Table 4.3.

The definition of **LNP** presumes that all running times will be longer than 1 second. Shorter times are highly susceptible to significant perturbation by system noise and timing imprecision; the offending inputs should be categorically removed from the workload. However, if these very short times are unavoidable, adding 1 to each time keeps the logarithm positive.

LNP is used to calculate the performance weight matrix PW, which is used to weigh the transformation-vector differences between inputs in a difference matrix. Weighing these differences based on performance helps to identify when the differences in transformation decisions impact program performance, and filters out cases where different decisions have little practical effect. Clustering requires a symmetric matrix, thus PW must also be symmetric.

$$PW[x,y] = \frac{max(\mathbf{LNP}[x], \mathbf{LNP}[y])}{min(\mathbf{LNP}[x], \mathbf{LNP}[y])} - 1$$

As the difference between the **LNP** scores for a pair of programs reduces to 0, so does the weight assigned to their difference scores. Table 4.4 shows the PW matrix for the example. The largest PW values correspond to `C`, as expected. Unlike training on `C`, training on the other inputs results in a performance improvement. Therefore, some portion of the decisions the compiler made differently when using `C`'s profile compared to the other profiles have a significant (negative) impact on performance. Similarly, the differences between the other inputs are less important, but still impact performance. Complex interactions between transformation decisions make inferring the performance impact of any individual transformation difficult. Therefore, when individual transformations are investigated, D_T is not weighted by PW.

The combined difference matrix D is pointwise-weighted by PW to generate the *weighted difference matrix*, $\overline{D}$. For $0 < x \leq \omega$ and $0 < y \leq \omega$:

$$\overline{D}[x,y] = D[x,y] \times PW[x,y]$$

	A	B	C	D
A	0	1.716	3.391	0.838
B	1.716	0	0.542	0.001
C	3.391	0.542	0	0.428
D	0.838	0.001	0.428	0

Table 4.5: The weighted difference matrix $\overline{D}$, computed as the pointwise product of PW matrix from Table 4.4 and the difference matrix from Table 4.2

	A	B	C	D
A	3.39	1.86	0.00	2.55
B	1.68	3.39	2.85	3.39
C	0.00	2.85	3.39	2.96
D	2.55	3.39	2.96	3.39

Table 4.6: Similarity matrix $\overline{S}$, computed from the weighted difference matrix $\overline{D}$ from Table 4.5

$\overline{D}$ for the example is shown in Table 4.5. Consider the columns for input `A` in Table 4.2 and Table 4.5. The differences between `A` and `C` and between `A` and `D` in D are almost the same. When these differences are weighed by PW to create $\overline{D}$, the relative differentiation between `A` and `D` is reduced, but the relative differentiation between `A` and `C` is maintained. This change indicates that while both `D` and `C` had a similar number of transformation differences when compared to `A`, the differences between `A` and `D` have less impact on performance.

Thus far, input similarity data has been presented in a difference matrix. However, the clustering problem is formulated in terms of similarity, and clustering algorithms require a similarity matrix as input. As implied by its name, a similarity matrix measures input similarity rather than difference. Any difference matrix D can be converted to a similarity matrix S by subtracting each element of D from the maximum element in the D. For $0 \leq x < n$ and $0 \leq y < n$:

$$S[x, y] = \max(D[x, y]) - D[x, y]$$

$\overline{S}$ denotes the similarity matrix for $\overline{D}$. Table 4.6 shows $\overline{S}$ for the example. The lowest values in the similarity matrix indicate that the strongest combination of transformation differences and performance differences occur between `A` and `C`. In a manual study this result would indicate that the decisions made by the compiler using FDO from inputs `A` and `C` warrant closer examination. However, a complete analysis should consider the similarity between every pair of inputs, which is what clustering provides.

4.1.3 Clustering

The goal of clustering is to group inputs to which the compiler responds similarly. By comparing each $\mathbf{V}_j$, $\overline{D}$ leverages the expertise and experience built into the

compiler to indirectly identify which aspects of the profiles are important. Thus, clustering $\overline{S}$ will group together inputs predicted by the compiler to have similar important runtime behaviors.

For clarity and simplicity, further discussion will use a graph interpretation of matrices. Each data input is a vertex in a complete, undirected graph. The difference matrix D_T for a transformation T contains the edge weights for the graph. Clustering can be interpreted as breaking the graph into several disconnected maximal cliques by removing edges. The goal is to minimize the sum of the weights on the remaining edges. The quality of a clustering is measured by calculating that sum.

A k-clustering C_k of D is a set of k disjoint vertex partitions (*clusters*) that covers the set of program inputs Ω as presented in the difference matrix D. Denote the a^{th} cluster of C_k by $\mathcal{C}_a$. Likewise, $C_{T,k}$ is a k-clustering of the similarity matrix S_T. When used in general, without a specific k, the k subscript is omitted. Formally:

$$\begin{aligned}
&\forall\, \mathcal{C}_a \in C : \mathcal{C}_a \subset \Omega \\
&\forall\, \mathcal{C}_a, \mathcal{C}_b \in C, a \neq b : \mathcal{C}_a \cap \mathcal{C}_b = \emptyset \\
&\bigcup_{\mathcal{C}_a \in\, C} \mathcal{C}_a = \Omega
\end{aligned}$$

Given a clustering C and a difference matrix D, the clustering error $Mismatch(C, D)$ is defined as the sum of the edge weights in D for all clusters of C:

$$Mismatch(C, D) = \sum_{\mathcal{C}_a \in C} \left(\sum_{i_x, i_y \in \mathcal{C}_a} D[i_x, i_y] \right)$$

$Mismatch$ is applicable to clusterings based on any difference matrix: D, $\overline{D}$, and D_T are all valid parameters, provided that the clustering used S, $\overline{S}$, or S_T, respectively. For the combined matrices D or $\overline{D}$, the clustering will be denoted C or $\overline{C}$, respectively.

4.1.4 ϵ-Greedy Spectral Clustering

Spectral clustering conveniently relies entirely on the similarity matrix, and does not use the raw vectors to recompute the similarity matrix as partitioning progresses [91]. We modify the original recursive spectral clustering algorithm to parameterize the number of clusters generated, and mitigate the sub-optimality of a purely greedy algorithm.

As in the original formulation, cuts are selected in a best-first order. Each cut splits one of the current partitions in two. To select the best cut, a local similarity matrix is created using the rows/columns of the nodes in the partition. The elements of the partition are ordered by their projection onto the 2^{nd} eigenvector of the local similarity matrix. The partition is cut at each point along the ordered list of partition

Algorithm 1: ϵ-Greedy Spectral Clustering

```
1  C = randomPartitioning(S);
2  for i ← 1 to N do
3      Partition.length = 1;
4      Partition[0] = 𝒲;
5      while Partition.length < k do
6          maxCutValue = 0;
7          for i ← 0 to Partition.length do
8              if rand() < ε1
9                  cut[i] = SpectralCut(Partition[i]);
10             else
11                 cut[i] = RandomCut(Partition[i]);
12             endif
13             if cut[i].NcutValue > maxCutValue
14                 maxCutValue = cut[i].NcutValue;
15                 maxCut = i;
16             endif
17         end
18         if rand() < ε2
19             p = Random(Partition.length);
20         else
21             p = maxCut;
22         endif
23         [partA,partB] = Partition[p].applyCut(cut[p]);
24         Partition[p] = partA;
25         Partition.add(partB);
26     end
27     if Mismatch(Partition, D) < Mismatch(C, D)
28         C = Partition;
29     endif
30 end
31 return C;
```

elements, and a cut value is determined. The cut corresponding to the smallest cut value is selected.

Cut values are calculated using Ncut. Ncut solves a relaxed version of the NP-Complete minimum cut problem. Cuts selected based on the Ncut value are therefore not guaranteed to minimize $Mismatch$. Moreover, greedy algorithms can result in very sub-optimal solutions unless specific conditions are met. For graph partitioning, a greedy algorithm has no optimality guarantees. In particular, when purely greedy 2-way partitioning is used, the $Mismatch$ does not always monotonically decrease as the number of clusters increases.

Therefore, we employ the classic search technique of injecting a random component to the greedy partitioning, and then iterate the search. Simulated annealing is not appropriate because the solution space is not smooth and has many local min-

ima. We use a fixed number of iterations with a constant probability of making a random choice in each iteration. As shown in Algorithm 1, the clustering result C is initialized randomly (line 1). A fixed, pre-determined, number of iterations, N, is used to search for the best clustering (line 2). *Partition* is a vector, thus *Partition.length* is the number of partitions.

Each iteration of partitioning proceeds as follows: Initially, *Partition* has a single partition containing every vertex of S (line 4). Two-way partitioning is then iterated to produce k partitions. In order to select which of the $Partition.length < k$ current partitions to split, and how to split it, a cut is proposed for each partition (lines 7-17), and the Ncut value recorded (lines 9, 11). However, a random cut will be proposed with probability ϵ_1, and a greedy cut calculated by spectral clustering with probability $(1-\epsilon_1)$ (line 8). The partition with the lowest Ncut value is selected to be cut with probability ϵ_2, while a random partition is cut with probability $(1-\epsilon_2)$ (line 18). Applying the selected cut (from line 9 or 11) creates two new partitions (line 23), which replace the split partition in the *Partition* vector (lines 24,25). Partitioning continues until there are k partitions. At this point, the *Partition* vector is one possible k-clustering of S.

$Mismatch$(*Partition*,D) is calculated at the end of each partitioning. If $Mismatch$ for *Partition* is less than the previous best clustering, the clustering result is updated (line 27). After the N iterations of partitioning are complete, the best clustering, C, is reported.

A direct method for spectral clustering using multiple eigenvectors of the similarity matrix is presented in conjunction with the recursive version [91]. The direct method is preferable in most cases because it is computationally efficient and takes the desired number of clusters as an input. However, to produce k clusters, the similarity matrix must have k distinct eigenvalues. Unfortunately, the direct method is unsuitable for the exhaustive clustering used in this study because an $m \times m$ similarity matrix frequently does not have m distinct eigenvalues, thus necessitating the use of an iterative approach.

4.1.5 Unimplemented Refinements

The ϵ-greedy algorithm presented above is inefficient if clusterings for multiple values of k are desired. Multiple clusterings are needed, for instance, in order to choose the most appropriate k for a particular data set. The recursive splitting process produces intermediate clusterings for every size smaller than k. Furthermore, the quality of each clustering is computed in order to determine the greedy option. Therefore, it would be trivial to keep the best clustering for each size.

The partitioning process is essentially a search in a lattice with an often large, non-constant, branching factor. With each iteration, Algorithm 1 walks one full path to depth k in the lattice, thus producing one possible k-clustering. A depth-first search is more efficient because the earlier steps in the path would need to be selected and evaluated less frequently. However, since an exhaustive search is in-

feasible, some form of search-space pruning is required. An adequate discussion of efficient search techniques is beyond the scope of this document. However, one simple approach is to keep the basic ϵ-greedy algorithm, but to convert it to a recursive depth-first search and move the iteration component inside the recursion. Rather than doing N walks to depth k, each intermediate partitioning (recursive call) would try M cuts, using the same ϵ-greedy approach to select which cuts to try.

4.2 Using Clustered Workloads

Grouping inputs in Ω by similarity allows a smaller $\mathcal{W}$ to be created for efficient performance evaluation. However, in the context of an FDO compiler under development, input similarity may also be constantly changing. The use of clustering results during compiler development should take this fact into consideration. In addition, a metric is needed to evaluate differences between alternative clusterings to help developers estimate the frequency with which Ω should be re-clustered.

4.2.1 Using Reduced Workloads

Clustering a large set of inputs provides a (much) smaller set of clusters, each containing a subset of Ω to which a single-profile FDO compiler responds in a similar manner. A representative evaluation workload of reduced size, $\mathcal{W}$, must be created by selecting inputs from each cluster. A naive approach to using the clustering results involves picking one representative input from each cluster for inclusion in $\mathcal{W}$, and using this $\mathcal{W}$ throughout the compiler development process. This approach can be significantly improved. If the compiler supports multi-profile FDO a slightly better approach is to select one such reduced workload to use for $\mathcal{W}_{train}$, and another to use for $\mathcal{W}_{test}$.

However, both these approaches neglect two factors. First, the original workload is a tremendous resource, and using only a small, fixed subset of inputs throws away most of that resource. Furthermore, the compiler, and possibly the application, are constantly evolving. Any fixed, initially-representative subset of inputs may potentially become less and less representative as development progresses. Eventually, a new clustering should be computed, but in the mean time, dynamically selecting representative inputs by random sampling can mitigate the divergence between performance evaluations using $\mathcal{W}$ instead of Ω. Therefore, $\mathcal{W}$ should be created by randomly selecting a small set of representative inputs from each cluster. Moreover, $\mathcal{W}$ should be routinely re-selected, perhaps on a weekly or bi-monthly basis. Additionally, a compiler development team has many members, each modifying and testing the compiler simultaneously. $\mathcal{W}$ need not be shared by the team. Instead, each developer can randomly select (or be provided with) a new $\mathcal{W}$ on a routine basis. Consequently, a large portion of Ω can be in simultaneous use

across the team, although no individual developer is using more than a small sample. Furthermore, variance between the simultaneous performance evaluations of multiple developers can serve as a heuristic indicating that a complete re-clustering is required. Inter-developer performance variance is only a heuristic, and must be tempered by an understanding of each developer's private modifications to the compiler.

4.2.2 $CrossError$: Comparing Clusterings

When both the compiler and the application are evolving, how often should the input clustering be recomputed? The more significant the gradual changes due to development work are on the clustering, the more frequently the clustering process should be repeated to ensure that the clustering remains relevant. Also, if a transformation T may generate clustering results that are representative of several transformations. In that case, generating transformation vectors only for T can reduce compilation time, and save time and effort invested by developers into analyzing and interpreting clustering data. To address both of these issues, the clustering methodology must provide a measure of clustering similarity based on different data-sets.

The clustering similarity measure cannot simply compare error curves or cluster members. Comparing error curves is only useful for different clustering methods using the same data. Different data sets are not equally difficult to cluster — a good k-clustering of one data set may also be a good k-clustering of another data set, even if the $Mismatch$s are different. Comparing the members of the resulting clusters may be more informative, but does not indicate the importance of the observed differences. Several near-optimal clusterings may exist, with cluster composition differing only for (possibly many) insignificant elements. We propose a novel metric, $CrossError$, to allow quantitative clustering comparison.

$CrossError$ requires that each matrix contain the same amount of potential error, so that all $Mismatch$ measurements share the same range. Transformation vector dimensions depend on the transformation, the compiler, and the program, leading to different ranges of possible Manhattan distances. The original matrices result in error values without context, which are both incomparable and difficult to interpret. Therefore, each similarity matrix is normalized by point-wise dividing it by the sum of its elements:

$$\widetilde{S}[x,y] = \frac{S[x,y]}{sum(S)}$$

Normalization makes each edge weight proportional to the total weight in the graph. Consequently, $Mismatch$ and $CrossError$ values are a percent of the total possible error, a more intuitive metric that enables comparisons between error values. Clustering uses relative edge weights and is not influenced by this uniform scaling. Henceforth, normalization always precedes clustering, but we omit the $\widetilde{S}$ symbol to streamline the notation.

The $CrossError$ metric quantitatively measures the differences between two k-clusterings of the same workload, using different edge weights. If the $CrossError$ is low, one similarity matrix, and consequently the clustering based on that matrix, may be representative of the other.

Given code transformations T and U, with difference matrices D_T and D_U, and their clusterings $C_{T,k}$ and $C_{U,k}$, the $CrossError$ metric is computed:

$$CrossError(C_{T,k}/C_{U,k}, D_U) = Mismatch(C_{T,k}, D_U) - Mismatch(C_{U,k}, D_U)$$

$CrossError$ evaluates $C_{T,k}$ using $C_{U,k}$ as a baseline. If $C_{U,k}$ is an optimal clustering of S_U, then $Mismatch(C_{U,k}, D_U)$ is the minimum error for D_U. This property of D_U and k is invariant with respect to the clustering. Thus, for any other k-clustering of D_U (*e.g.*, $C_{T,k}$), $Mismatch(C_{T,k}, D_U) \geq Mismatch(C_{U,k}, D_U)$. With effective but sub-optimal clustering, $C_{U,k}$ estimates the minimum error for D_U. Therefore, $CrossError(C_{T,k}/C_{U,k}, D_U)$ measures (or estimates, for sub-optimal clustering) the *extra* error incurred by using the alternate clustering $C_{T,k}$.

Consider $Mismatch$ over the possible range of k. For small k, clustering separates the greatest differences in D. As k increases, less significant differences are separated until each partition contains identical elements. Therefore, regardless of k, if C_U is a good clustering of S_U, $CrossError(C_{T,k}/C_{U,k}, D_U)$ will be low, implying that transformation T provides input similarity data that is representative of the input similarity data provided by transformation U. The curve for $CrossError(C_{T,k}/C_{U,k}, D_U)$ over the range of k provides a quantitative measure of the strength of the representativeness relationship.

$CrossError$ is not symmetric. As such, if T is judged to be representative of U, the reciprocal relationship is not implied. For example, transformation U might have little impact on the program, and consequently expose few differences between inputs.

Alternately, $CrossError$ can be used to compare clusterings for similar programs.[4] In this case, the transformation T and workload $\mathcal{W}$ are held fixed, and the programs p and q are compared:

$$CrossError(C^p_{T,k}/C^q_{T,k}, D^q_T) = Mismatch(C^p_{T,k}, D^q_T) - Mismatch(C^q_{T,k}, D^q_T)$$

4.3 Evaluation Methodology

We investigate workload clustering for a range of programs. Clustering is performed for both individual transformations and combined multi-transformation data. These clusterings are analyzed to identify the impact of performance filtering, the

[4] A practical use for such comparison is to evaluate if two versions of the same program produce a similar clustering of inputs.

correlations between transformations, and the significance of the input-processing source code on input similarity.

All programs are compiled using a development snapshot of the IBM XL 8.0 compiler that is instrumented to output transformation vectors. Performance evaluation uses a dedicated machine running AIX on a POWER4 processor. Five runs are used for each program on each input, and the average of these runs is used as $t_j(i)$ when calculating $\mathbf{LNP}[j]$.

All experiments are performed using `gzip`, `bzip2`, `VPR`, `crafty`, `MCF`, and `GAP` from the SPEC CPU 2000 suite, as well as `gcc` from the SPEC CPU 2006 suite [37]. `VPR` performs two digital design tasks, logic placement and circuit routing, which take different input files and exercise different portions of the code. Therefore, `VPR` is used for these two tasks separately and identified as `vpr.place` and `vpr.route`.

Each program uses a workload of inputs. The provided inputs from both CPU 2000 and CPU 2006 are used when possible (`MCF`, `bzip2`, and `gcc`). Furthermore, the CPU 2000 program workloads are augmented with inputs that we collected or generated [14], and described in more detail in our previous work [16]. The `gcc` workload is augmented with source code from the CPU2000 benchmark programs `gzip`, `mesa`, `parser`, and `twolf`. `Gzip` and `bzip2` use a common workload that is the union of their available inputs.

4.3.1 Transformations

During the training process, transformation vectors are collected for the transformations that are the primary consumers of profile information. Inlining and loop transformations use profile information to order the transformation opportunities by expected profitability (*i.e.*, hottest first). The following transformations are instrumented:

Early Inlining Inlining at the beginning of the optimization phase focused on removing calls to small functions to enable subsequent transformations.

Late Inlining Inlining after high-level transformations such as loop nest optimizations and function-pointer specialization, focused on removing function call overhead.

Loop Unrolling The loop unrolling factor is based on the number of memory streams in the loop, the number of hardware-supported memory streams, and the number of instructions in the unrolled loop. Ideally, unrolling should activate all the hardware memory stream prefetching units without overflowing the instruction cache.

Loop Unroll-and-Jam Loop unrolling, loop peeling and loop fusion for the inner loops in loop nests, to create perfectly-nested loops.

Specialization transformations use value profiling to replicate code segments, replacing the use of variables with constants. A test ensures that the run-time variable value matches the specialized value:

Memory Allocation Specialization Memory allocation library calls with variable memory block sizes are specialized with a constant memory size. These specialized allocations use a pooled memory allocator that increases the spatial locality of memory accesses and reduces the overhead of memory allocation and deallocation.

Function-Pointer Specialization Indirect calls are converted to direct function calls. Removing indirection enables other transformations, *e.g.*, inlining. Furthermore, function call overhead is reduced on architectures where a branch and direct call is less expensive than an indirect call.

Value Specialization Integer division and modulus operations with a variable denominator are replaced with constant-denominator versions for frequently observed denominator values. Constant denominators allow for the generation of more efficient code using various architecture-dependent instruction-level transformations.

4.3.2 Clustering Comparison

ϵ-Greedy clustering is applied to these similarity matrices:

- The similarity matrix S_T for each transformation T from Section 4.3.1, without performance-weighting, produces clustering C_T.
- The combined, performance-weighted similarity matrix $\overline{S}$ produces clustering $\overline{C}$, as discussed in Section 4.1.2.
- The combined, but not performance-weighted, similarity matrix S produces clustering C

Clusterings are compared in several ways. These comparisons are not intended to support strong claims about particular transformations, benchmark programs, or data inputs. Rather, they serve as single-point case studies that illustrate the types of questions that cluster comparison can help answer. Three variables that influence clustering results are investigated:

Performance Performance-weighting using PW serves as a filter to remove the impact of distinct transformation decisions that have little impact on program running time. Comparing the clustering with and without performance weighting indicates the impact of this filtering.

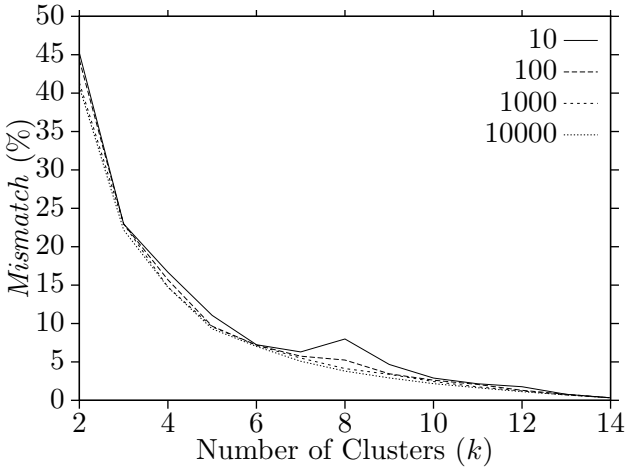


Figure 4.1: Clustering error for varied iterations of ϵ-greedy spectral clustering when using similarity matrix S from `gcc`

Transformations Clustering based on one transformation may predict the clustering based on another, particularly if the transformations are closely related. If the clustering for T can be predicted by the clustering for U, then T could be omitted from the analysis.

Algorithms `Bzip2` and `gzip` share an identical workload, but are very different algorithms. Comparing the clusterings for the two programs suggests the degree to which input similarity can be considered independently of the code processing the input, and thus the feasibility of manual clustering without extensive detailed analysis. This information may also be significant when using a reduced workload for evaluation in the case of rapid program development with frequent and significant code changes.

4.4 Clustering Evaluation

Reliable clustering depends on setting the clustering algorithm parameters to appropriate values. Once the ϵ-greedy clusterer has been tuned, the workload for each benchmark program is clustered for each transformation listed in Section 4.3.1, along with the combined matrices D and $\overline{D}$, for each possible number of clusters.

4.4.1 ϵ-Greedy Parameters

The two ϵ parameters for ϵ-greedy clustering control the amount of randomization in the search process. In practice, selecting the partition to cut at random (ϵ_2) provides significant improvement to clustering results. The addition of random cuts (ϵ_1) provides a small additional improvement. The ϵ values provide exploration away from the greedy choice. Given a particular k-clustering, the value of all possible cuts to form cluster $k+1$ are fixed. Since the greedy choice is always the same for the

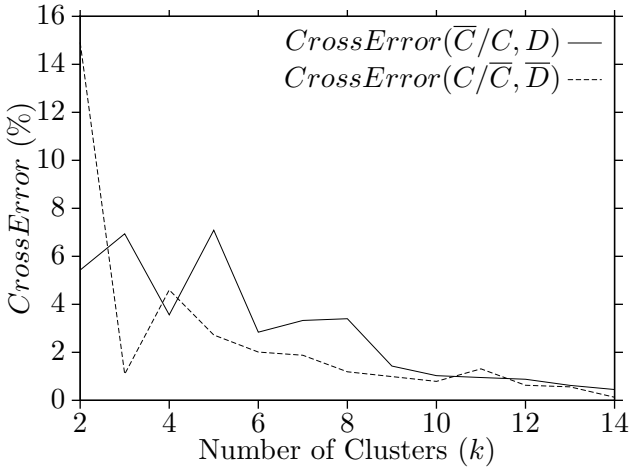


Figure 4.2: Comparison of $CrossError$ using weighted (D) and unweighed ($\overline{D}$) difference matrices from `gcc`

same point in the recursive splitting process, there is no local benefit to selecting it multiple times. However, any subsequent splitting decisions present a space that is predicated by selecting the greedy option, but may need many iterations to explore. Thus, the ϵ values must balance selecting greedy option with exploration. Manual tuning suggests that 0.5 is a reasonable value for both ϵ values for the transformations, benchmark programs, and input sets used in this experimental study.

Each iteration of ϵ-greedy clustering increases the amount of the clustering space explored. However, each additional iteration increases the computational cost of clustering, while providing diminishing returns with respect to error reduction. Empirically, 1,000 iterations of clustering produces good results, with very little improvement when the number of iterations is increased to 10,000, as demonstrated in Figure 4.1. Using fewer than 1000 iterations does not always allow ϵ-greedy to find a good clustering, and produces significant variation in the $Mismatch$ across different clustering runs. The ϵ-greedy results presented henceforth set $\epsilon_1 = 0.5$ and $\epsilon_2 = 0.5$, with 1000 iterations.

4.4.2 Impact of Performance Weighting

Section 4.1.2 justifies the use of performance weighting as a means of taking program performance into account when clustering inputs. Compiler decisions with a larger impact on the program's performance should have more weight when clustering inputs than decisions with little effect on performance. But does performance weighting change the resulting input clusters? Comparing the $CrossError$ between unweighted and weighted clusters should answer this question. The results in Figure 4.2 are typical of this comparison: D and $\overline{D}$ are, respectively, the weighted and unweighted combined difference matrices for the seven transformations described in Section 4.3.1 for `gcc`, and C and $\overline{C}$ are their corresponding clusterings.

$CrossError(\overline{C}/C, D)$ evaluates the weighted clustering with the unweighted

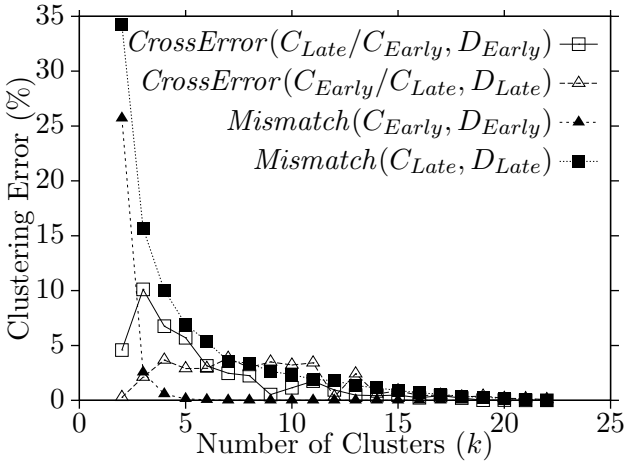


Figure 4.3: Clustering error comparison between the early-inlining and late-inlining clusterings from `VPR` routing

difference matrix. As illustrated by the evaluation of `gcc` in Figure 4.2, an unweighted clustering C evaluated with a weighted difference matrix $\overline{D}$ generally results in less *CrossError* than $\overline{C}$ evaluated with D. This result indicates that D is more representative of $\overline{D}$ than vice-versa. Thus, in this case, performance-weighting filters out performance-irrelevant differences in the data that would otherwise influence the clustering. The additional error under the *CrossError* curves illustrates that $\overline{D}$ is not a scaled version of D; in that case, the clustering results would be equivalent, and both *CrossError* curves would be near 0. Instead, the performance-weighting has changed which pairs of inputs are most similar or most different from each other, thus changing the clustering in a meaningful way.

4.4.3 Clustering Comparison

An important application of *CrossError* is to evaluate the representativeness of previous clustering when the compiler and application program change. Benchmarks only provide a single snapshot of application code and consequently make evaluating this use of *CrossError* difficult. However, the placement and routing tasks for `VPR` use related data sets with the same application, and `bzip2` and `gzip` share a common workload. An investigation of the *CrossError* in these cases indicates that the clustering results are not similar across such large differences in programs or workloads.

Section 4.3.2 discussed the possibility that several code transformations in the compiler could yield similar clusterings, and thus some of these transformations could be eliminated from future input clusterings. This section illustrates how to evaluate the similarity of two transformations in relation to the clustering of inputs.

We generated *CrossError* graphs for each possible pairing of clusterings from the transformations listed in Section 4.3.1 plus the combined matrices D and $\overline{D}$, for each of the 7 benchmark programs. Careful examination of these graphs sug-

gests that transformations are generally not representative of others. Furthermore, the combined matrices tend not to be good representatives of any individual transformation, nor is any single transformation representative of either combined case. The greatest correlation exists between the two inlining transformations, but even here, the correlation is usually weak.

The $Mismatch$ between a clustering, such as C_{Early}, and its difference matrix, D_{Early}, is a measurement of the differences that exist within nodes that C_{Early} clusters together. Figure 4.3 plots $Mismatch(C_{Early}, D_{Early})$ and $Mismatch(C_{Late}, D_{Late})$ as a function of the number of clusters k for the `VPR` routing benchmark. This plot is typical of such curves between early and late inlining for most benchmarks. While four clusters are sufficient to separate virtually all the differences between inputs with respect to early inlining, the $Mismatch$ curve for late inlining has a long tail, indicating that many more differences exist amongst the inputs when late inlining is considered. Thus, early inlining would be a poor representative of late inlining.

Figure 4.3 also plots the $CrossError$ curves that evaluate how well the clustering based on late inlining, C_{Late}, represents that data in the early inlining difference matrix D_{Early}, and vice-versa. $CrossError(C_{Early}/C_{Late}, D_{Late})$ stays level in the 3%-3.5% range from 4 clusters to 11 clusters. Over this range, $CrossError$ is on average 97% of $Mismatch$, so applying C_{Early} to the late inlining data results in about twice as much error as using C_{Late}. The level $CrossError$ curve through this range indicates that even with additional clusters, early inlining does not provide any information to enable the extra clusters to better separate the inputs.

On the other hand, since late inlining has more information regarding input dissimilarity, perhaps this information is a superset of the information provided by early inlining. However, from 4 to 8 clusters, $CrossError(C_{Late}/C_{Early}, D_{Early})$ tracks just below $Mismatch(C_{late}, D_{Late})$. Even at 8 clusters, the $CrossError$ is only slightly less than $Mismatch(C_{late}, D_{Late})$ at 3 clusters. The late-inlining clustering does not separate the few differences that do exist in the early-inlining data. Therefore, in this case, late inlining is not a good representative of early inlining, despite the conceptual similarity of the transformations. We observed similar patterns between early and late inlining for most benchmarks.

An alternate way to try to test the correlation between these two transformations is to take each pair of inputs as a data point and use the early-inlining Manhattan distance between the inputs for one axis and their late-inlining Manhattan distance for the other axis. The coefficient of correlation calculated this way is 0.99, which indicates a very high degree of correlation. The large coefficient of correlation can be attributed to data points falling into two clusters, with one cluster occurring very far from the others. Consequently, at the full scale of the data, the clusters become two points and display a linear relationship. However, looking at each of the two groups of data individually, the data does not exhibit any linear or recognizable non-linear relationship. A systematic study of the same form of data across all transformation pairings and all programs suggests that the coefficient of correlation

is usually *not* a good indicator of a representativeness relationship between a pair of transformations.

4.4.4 Clustering for Workload Reduction

The goal of clustering is to group similar inputs so that one of them can be selected as the representative of that cluster in a reduced workload. The overall performance estimated using the evaluation workload $\mathcal{W}$ should predict the real performance evaluation of the actual workload Ω. As suggested in Chapter 3, the FDO workload performance of a program is summarized by taking the geometric mean of the performance computed for each training input. In this case, per-training-input performance is given by $\mathbf{LNP}$, thus workload performance is:

$$\mu_g = \sqrt[|\mathcal{W}|]{\prod_{i \in \mathcal{W}} \mathbf{LNP}[i]} \tag{4.1}$$

An evaluation of reduced workloads selected from Ω according to clustering suggests that clustering can effectively group similar inputs, and that evaluation of a reduced workload can provide reasonable estimates of full-workload performance evaluation. Selecting an appropriate number of clusters from which to select a reduced workload is a problem beyond the scope of this document. The automatic method used here is based on the reduction in $Mismatch$ observed when k is increased by one. This quantity, denoted $\Delta Mismatch$, indicates the benefit to clustering quality obtained by incurring the increased evaluation cost of a larger k:

$$\Delta Mismatch(k) = Mismatch(\mathcal{C}_{k-1}, D) - Mismatch(\mathcal{C}_k, D)$$

The results of performance evaluation when using the full workload Ω, as well when using reduced workloads selected based on clustering results from combined difference matrixes, are presented in Table 4.7. The table contains five rows for each benchmark. The first row, in bold, is the full-workload evaluation where $\mathcal{W} = \Omega$. The average and confidence interval on this line correspond to the geometric mean, as calculated by Equation 4.1. The column labeled "Range" displays the range of the per-training-input $\mathbf{LNP}$ values.

The remaining table rows for each benchmark give results when the value of k for $\mathcal{W}$ is chosen as the largest k where $\Delta Mismatch(k) > \delta$. For each δ, the k column gives the number of clusters selected, and M column gives the $mismatch$ of that clustering. The two values of $\delta = 0.75$ and $\delta = 0.05$ are chosen for demonstration.

Table 4.7 contains two rows for each δ. The first row reports on 100 samples of random-representative selection, as proposed in Section 4.2: one representative is chosen at random from each cluster. Each sample provides one workload evaluation using Equation 4.1. The averages and confidence intervals in these rows correspond to the *arithmetic* average of these 100 workload evaluations, while the

Program	δ	k	M	AVG	95% CI	Range	± 1
bzip2	Ω = **23**			**97.4**	$[\mathbf{93.6}, \mathbf{101.4}]$	$[\mathbf{95.5}, \mathbf{105.3}]$	
	0.75	6	1.73	98.5	± 1.4	[96.5, 99.5]	42%
				99.3	$[94.0, 104.9]$	$[97.6, 104.6]$	
	0.05	11	0.24	97.7	± 0.4	[97.2, 98.1]	100%
				97.4	$[93.4, 101.6]$	$[95.7, 102.8]$	
crafty	Ω = **8**			**89.5**	$[\mathbf{87.9}, \mathbf{91.0}]$	$[\mathbf{87.8}, \mathbf{90.6}]$	
	0.75	4	0.33	89.3	± 0.4	[89.0, 89.7]	100%
				89.1	$[87.1, 91.1]$	$[87.9, 90.1]$	
	0.05	6	0.01	89.2	± 0.2	[89.0, 89.3]	100%
				89.3	$[87.8, 90.8]$	$[88.0, 90.3]$	
gap	Ω = **11**			**74.8**	$[\mathbf{64.5}, \mathbf{86.8}]$	$[\mathbf{67.1}, \mathbf{88.6}]$	
	0.75	5	0.70	77.3	± 4.6	[73.6, 80.1]	34%
				84.9	$[72.6, 99.3]$	$[75.3, 93.4]$	
	0.05	6	0.10	80.4	± 1.2	[79.6, 81.3]	0%
				80.1	$[70.6, 90.8]$	$[71.9, 89.7]$	
gcc	Ω = **15**			**91.7**	$[\mathbf{89.8}, \mathbf{93.5}]$	$[\mathbf{90.1}, \mathbf{94.1}]$	
	0.75	5	1.92	91.6	± 1.4	[90.1, 92.9]	81%
				92.8	$[89.1, 96.6]$	$[90.1, 95.1]$	
	0.05	12	0.05	91.8	± 0.2	[91.7, 91.9]	100%
				91.9	$[89.9, 94.0]$	$[90.5, 94.3]$	
gzip	Ω = **23**			**85.7**	$[\mathbf{83.9}, \mathbf{87.6}]$	$[\mathbf{83.6}, \mathbf{87.4}]$	
	0.75	4	0.73	87.1	± 4.0	[82.0, 92.4]	27%
				89.1	$[86.5, 91.8]$	$[87.4, 90.3]$	
	0.05	9	0.02	86.1	± 2.4	[83.7, 88.7]	53%
				83.7	$[81.7, 85.8]$	$[81.9, 85.3]$	
mcf	Ω = **16**			**104.1**	$[\mathbf{96.4}, \mathbf{112.5}]$	$[\mathbf{97.2}, \mathbf{107.3}]$	
	0.75	4	0.28	103.4	± 1.9	[101.1, 104.8]	69%
				103.3	$[92.1, 115.8]$	$[96.7, 107.1]$	
	0.05	5	0.02	104.0	± 1.3	[102.5, 105.2]	86%
				102.9	$[92.5, 114.5]$	$[98.3, 107.7]$	
vpr.place	Ω = **23**			**89.7**	$[\mathbf{87.5}, \mathbf{91.8}]$	$[\mathbf{88.1}, \mathbf{92.7}]$	
	0.75	3	0.70	90.4	± 2.1	[88.4, 92.8]	52%
				87.7	$[83.8, 91.8]$	$[86.1, 89.4]$	
	0.05	7	0.08	90.6	± 0.8	[89.8, 91.4]	52%
				91.5	$[89.7, 93.4]$	$[90.6, 93.5]$	
vpr.route	Ω = **23**			**92.9**	$[\mathbf{90.6}, \mathbf{95.4}]$	$[\mathbf{90.9}, \mathbf{94.9}]$	
	0.75	4	0.41	92.9	± 0.9	[91.8, 93.8]	92%
				93.5	$[89.7, 97.6]$	$[91.3, 95.7]$	
	0.05	7	0.05	93.2	± 0.6	[92.2, 93.6]	100%
				92.3	$[89.4, 95.2]$	$[90.4, 94.7]$	

Table 4.7: Workload performance evaluated using the **full workload (bold)** and clustering-based reduced workloads consisting of either the *best representative (italics)* of each cluster, or 100 samples of randomly-selected representatives.

range indicates the minimum and maximum values of those samples. The "± 1" column reports the proportion of samples where the reduced-workload evaluation is within 1% of the full-workload evaluation. The second line for each δ, in italics, is a single evaluation directly comparable to the full-workload row. These rows are calculated on a $\mathcal{W}$ created by selecting the best candidate from each cluster of the selected k-clustering. Determining the best candidate is straight-forward: the best representative is the input corresponding to the column of the difference matrix with the lowest column-wise sum of differences.

The full-workload evaluations demonstrate that FDO does, in general, improve program performance. Based on the average **LNP**, all programs except `MCF` benefit from FDO: from a 2.6% improvement for `bzip2` to a 25.2% improvement for `GAP`. However, the confidence intervals and ranges show how much variation in workload performance is possible depending on the selection of training input. For `MCF` and `bzip2`, FDO could be judged to be either beneficial ($\mu_g < 1$) or detrimental ($\mu_g > 1$), depending on which input is used for training. Any variation in performance measured between different *testing* inputs is hidden within the **LNP** for the training input; the variations in performance measured pair-wise between individual testing and training inputs may be much greater than the variation between **LNP** values presented here. This evidence further supports the need for the cross-validated evaluation of FDO proposed in Chapter 3.

Selecting cluster representatives at random to create $\mathcal{W}$ may not provide reliable results if this practice results in large evaluation variations between different instances of $\mathcal{W}$. Recall that a reduced workload reduces not only the set of training inputs, but also the set of testing inputs. Thus, even keeping the training input constant, the **LNP** for that training input in a reduced workload will not be the same as in the full workload. Table 4.7 presents promising results in this respect: most $\mathcal{W}$ selected in this way predict the full-workload evaluation results quite accurately. The evaluation results for `GAP` are inconsistent with the results of the other benchmarks, and are discussed separately.

The range of the evaluation results produced by random representative selections is usually small. Furthermore, that range contains the full-workload evaluation in all cases except for `crafty` when $\delta = 0.05$. However, in that case, the range of sampled evaluation results is only 0.3, and no sample is more than half a percent from the full-workload evaluation. Furthermore, when a larger k is selected, the spread of the sampled results becomes tighter, as seen in both the confidence intervals and ranges. Even the small workloads (3–6 inputs) selected when $\delta = 0.75$ provide good predictions of full-workload performance. The last column in Table 4.7 directly measures the proportion of randomly-selected workloads that produce evaluation results within 1% of the full-workload value. In most cases, the majority of samples fall within this window, and the proportion increases as δ decreases. However, this pattern does not apply to the placement task for `VPR`. While there is less variation between the samples when $\delta = 0.05$ than when $\delta = 0.75$, the full-workload performance value is slightly outside the range covered by the

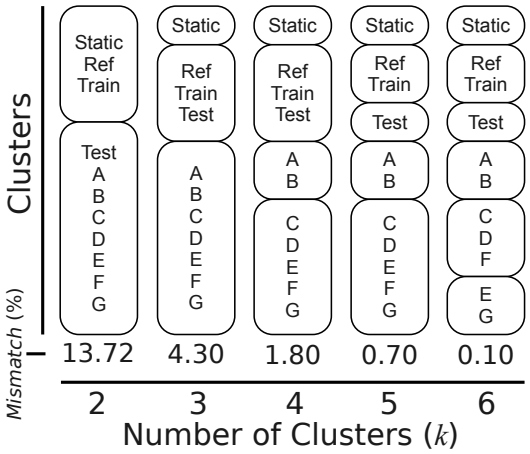


Figure 4.4: Clustering results for GAP as k increases, using the combined difference matrix

samples at $\delta = 0.05$. Performance evaluation using these reduced workloads are nonetheless very accurate: all the samples span a range of only 1.6%, and all fall within 1.7% of the full workload average.

The clustering-based evaluations of GAP contradict the preceding discussion. Most significantly, both reduced-workload evaluations significantly underestimate the benefit of FDO. This result is explained by particularly favorable pairings of testing and training inputs being placed into the same clusters, and thus never evaluated in the reduced workload. The workload for GAP is poorly constructed. GAP is an interpreter for a domain-specific language for mathematics. All but the SPEC inputs are the same numerical algorithm, simply called with an increasingly large input-parameter value. Larger parameter values invoke longer-running computations that may shift the relative frequency weighting between different components in the interpreter, but the underlying computation, and thus the overall behavior of the interpreter, is essentially unchanged. Details of the clustering for GAP are illustrated in Figure 4.4. The SPEC ref, test, and train inputs, along with the static non-FDO compilation, are labeled accordingly. The remaining inputs are labeled A through G by increasing parameter value. As indicated by the proportion of $Mismatch$ remaining between the clustered inputs, the most significantly different inputs have been separated by 4 or 5 clusters. As expected, the static compilation is unlike the FDO compilations. At 3 clusters, the SPEC inputs are separated from the additional inputs, and test is separated from ref and train at 5 clusters. As noted in Section 2.3, the train input is a subset of the computation performed by ref, thus these two inputs are expected to be very similar. The A and B inputs execute significantly more quickly than the other additional inputs; at 4 clusters, these short-running inputs are separated from their peers.

Consider two inputs, such as ref and train. If the runs on ref and train are very similar, using them both in an evaluation workload is liable to reward over-fitting. Clustering naturally identifies ref and train as similar and groups them together. A

reduced workload will thus never contain both ref and train, and the evaluation will not benefit from over-fitting. Consequently, the reduced workloads likely provide a *more realistic* evaluation of GAP than the full-workload evaluation.

4.4.5 Qualitative Clustering Evaluation

Section 4.4.4 uses a threshold on δ-$Mismatch$ to select the number of clusters for a reduced workload. While such an automatic technique is helpful for the practical application of clustering, tuning the process and evaluating the quality and implications of the resulting clusters can be enhanced using qualitative assessment.

$Mismatch$ encourages splitting large clusters, since this action removes the most edges from the graph. However, when the edge weights are not similar, the benefit of splitting smaller clusters or splitting a cluster into unequally-sized parts increases. Thus, partitioning that does not split the largest cluster suggests that significant differences are being separated in the graph. For example, the combined 2-clustering of gzip places the baseline compilation in its own cluster[5]; the 3-clustering places a single FDO compilation alone in another cluster. These two singleton clusters persist as k increases, and identify very significant differences in code transformations for these two compilations. This observation explains the large confidence interval on the average LWA for gzip in Table 4.7 when $\delta = 0.5$: only these two very significant differences have been identified by clustering; the less extreme differences within the rest of the workload remain.

There is no strict relationship between the clusters of a k-clustering and a (k+1)-clustering, though large differences will keep inputs from being clustered together, while small differences will tend to keep inputs clustered together. However, consider a subset of inputs that all have similar pair-wise differences. Partitioning this subset will reduce $Mismatch$ by nearly the same amount regardless of exactly which inputs end up clustered together, as long as the same number of edges are removed. Thus, cluster membership within such a subset often appears to randomly change as k increases. This re-distribution of inputs among clusters occurs in two situation. On one hand, when $Mismatch$ is large, not enough clusters are available to separate out the many similarly-large differences. On the other hand, when $Mismatch$ is near zero, the number of clusters required to separate meaningful differences has already been exceeded.

For a lower-level analysis, a developer can scan the values in the combined difference matrix to identify where the largest differences exist, and use this information either to evaluate a clustering, or to direct investigation within the compiler. In the case where the combined difference matrix does not present strong evidence, the difference matrices of the individual transformations may be consulted. Finally, investigating the individual transformation locations that differ in the transforma-

[5]Baseline compilation is expected to be significantly different than FDO compilation because the compiler is forced to estimate profile information from the source code to inform transformation decisions.

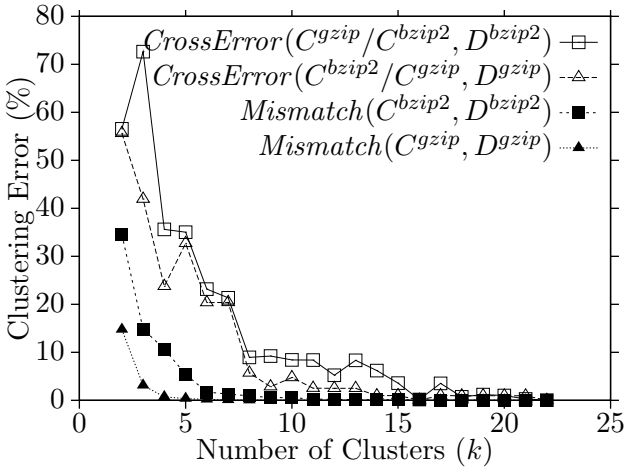


Figure 4.5: *Mismatch* and *CrossError* using the combined clusterings of `bzip2` and `gzip`

tion vectors may not only provide important information on how much inputs differ from each other, but also offer clues about why those differences occur.

4.4.6 Algorithm-Independent Input Similarity

Human intuition for input similarity is largely algorithm-independent. Both `bzip2` and `gzip` are lossless data compression programs, and use the same set of inputs for clustering, but the algorithms implemented by the two programs are completely different. Nonetheless, human intuition likely would groups inputs together the same way regardless of the algorithm used. The grouping would be done based on the types of data in the files: jpeg images, or plain-text files for example. However, intuition also expects that clustering inputs without regard for the algorithm using them is ill-conceived.

Figure 4.5 highlights the dangers of relying on such manual clustering by showing the *Mismatch* curves for `bzip2` and `gzip`, along with the *CrossError* curves when those clusterings are swapped between the programs. By 6 clusters, the *Mismatch* for both programs is very small. However, at the same point, the *CrossError* for both programs is more than 18%. Furthermore, even as the number of clusters increases, the *CrossError*s reduce slowly. This evidence indicates that, from the compiler's perspective, input similarity cannot be adequately measured outside of the context of the algorithm processing the inputs. Therefore, an automated approach such as the methodology presented here is required to adequately asses the differences between inputs within the complex context of their interactions with a program. Furthermore, switching one algorithm in a program for another (as opposed to incremental refinements) during development will likely necessitate the reevaluation and reclustering on its inputs.

4.5 Conclusion

FDO is an important tool for program optimization. Unfortunately, the standard single-training/single-evaluation practice of FDO can be sensitive to input diversity, and thus a cross-validation strategy is required for performance evaluation. However, selecting an appropriate workload for cross-validation is challenging: The workload must cover all the important aspects of the program, while also minimizing the number of inputs in the workload. This chapter illustrates a clustering technique to select this small subset of inputs from the large number of inputs available to a compiler designer. A similarity matrix is constructed from transformation vectors, information extracted directly from the compiler regarding differences between inputs. This matrix is weighted by a cross-validation-based performance metric in order to filter out those differences that do not impact performance. Once the workload has been clustered, the $Mismatch$ curve presents a quantitative measure of the tradeoff between the number of clusters and how well a selection of representatives from these clusters represents the full workload. Finally, $CrossError$ provides a means to investigate correlations between transformations, and the significance of differences between clustering done at different points in compiler or application development.

Chapter 5

Combined Profiling: Multi-Run Behavior Modeling

Capturing behavior variations across inputs is important in the design of an FDO compiler. A number of speculative code transformations are known to benefit from FDO, including speculative partial redundancy elimination [35, 51], trace-based scheduling and others [24, 33]. Several open questions remain about the use of profiles collected from multiple runs of a program. How should the multiple profiles be combined? Is it sufficient to simply average the multiple measurements? Is it necessary to compute the parameters for an assumed statistical distribution of the measurements? Or is there a simple technique to combine the measurements and provide useful statistics to FDO?

This chapter addresses these questions by arguing that the behavior variations in an application due to multiple inputs should be evaluated by FDO decisions. It also argues that a full parametric estimation of a statistical distribution is not only unnecessary, but it may also mislead FDO decisions if the wrong distribution is assumed or there is insufficient data to accurately estimate the parameters. Instead, it proposes the use of a non-parametric empirical distribution that makes no assumptions about the shape of the actual distribution.

A major challenge in the use of traditional single-training-run FDO is the selection of a profiling data input that is representative of the execution of the program throughout its lifetime. For large and complex programs dealing with many use cases and used by a multitude of users, assembling an appropriately representative workload may be a difficult task. Picking a solitary training run to represent such a space is far more challenging, or potentially impossible, if use-cases are mutually-exclusive. While benchmark programs can be modified to combine such use-cases into a single run (Section 3.1), this approach is obviously inapplicable to real programs. Moreover, user workloads are prone to change over time. Ensuring stable performance across all inputs in today's workload prevents performance degradation due to changes in the relative importance of workload components.

The *Combined Profiling* (CP) statistical modeling technique presented in this chapter produces a *Combined Profile* (CProf) from a collection of traditional single-

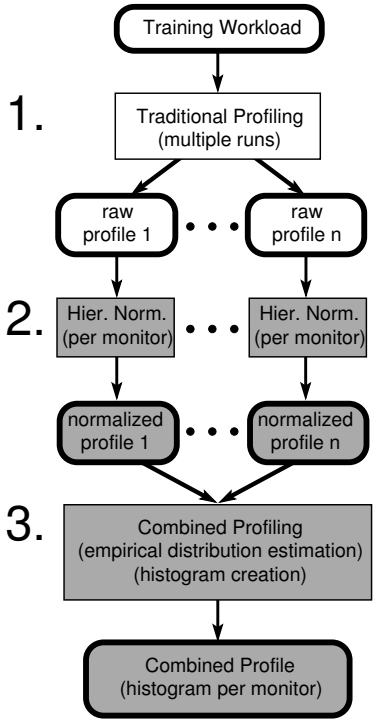


Figure 5.1: Three phases of combined profiling: 1) profile each input, 2) normalize each profile, and 3) combine the profiles into a distribution model.

run profiles, thus facilitating the collection and representation of profile information over multiple runs. The use of many profiling runs, in turn, eases the burden of training-workload selection and mitigates the potential for performance degradation. There is no need to select a single input for training because data from any number of training runs can be merged into a combined profile. More importantly, CP preserves variations in execution behavior across inputs. The distribution of behaviors can be queried and analyzed by the compiler when making code-transformation decisions. Modestly profitable transformations can be performed with confidence when they are beneficial to the entire workload. On the other hand, transformations expected to be highly beneficial on average can be suppressed when performance degradation would be incurred on some members of the workload.

Combining profiles is a three-step process, as illustrated in Figure 5.1. Shaded components of the figure identify the combined-profiling work-flow:

1. Collect raw profiles via traditional profiling.

2. Apply *Hierarchical Normalization* (HN) to each raw profile.

3. Apply CP to the normalized profiles to create the combined profile.

CP and HN have been presented in previous work [20, 18]. However, this presentation clarifies and expands on previous versions, particularly the description of CP's histograms in Section 5.2 and the discussion of queries in Section 5.3.3.

Section 5.1 discusses the design of CP, and the details of the technique are presented in Section 5.2. CP is widely applicable; Section 5.3.4 briefly discusses the use of CP with additional forms of profiling.

5.1 Design Considerations

To facilitate the use of CP with existing FDO compilers, CP should offer a semantic "drop-in replacement" for raw profiles. In particular, a CP created from a single raw profile should be as informative as the original raw profile. This goal is at odds with parametric models, which need many data points to accurately estimate their parameters. As a matter of practicality, the distribution model should have a (small) bounded size because it competes with the rest of the compiler for memory during compilation.

5.1.1 Model Properties

We refer to traditional single-run profiles, such as edge or path profiles, as *raw profiles*. The simplest technique to maintain information about many profiling runs is to keep all the raw profiles and provide them to the compiler. However, not only does such a representation require space linear in the number of profiles, but querying such data (*e.g.*, within code transformation heuristics) incurs an associated computational cost. CP aims to represent an unbounded number of profiles in a compact, fixed-size representation in order to bound such costs by a small constant.

In a batch environment, optimization minimizes (weighted) average execution time, and consequently an average of program behavior over a workload is a sufficient statistic. However, more typically, program optimization across a workload of inputs is not a batch-execution scenario: the execution time on each individual input is significant. Thus, average-case performance is *not* the metric that an FDO compiler should maximize. Rather, for a given program, each transformation should attempt to minimize the execution time for each input in the program workload; average execution time is merely a convenient aggregate statistic. Thus, an FDO transformation decision is a multi-objective optimization problem with the dual goals of maximizing both the worst-case and average-case improvements in program execution time across the workload. A single-run profile, or even an aggregated profile using sums or averages across multiple runs, is not adequate to meet these goals because it only allows for the assessment of the average case.

Similarly, there is no reason to assume that the amount of computation performed on a given training input is related to the importance of such an input in a user's workload. The relative weights of profiles being combined can only be assigned by the user. A CProf is a weighted combination of profiles, but in the absence of user specification, all profiles are assumed to be equally important.

5.1.2 Parametric Models

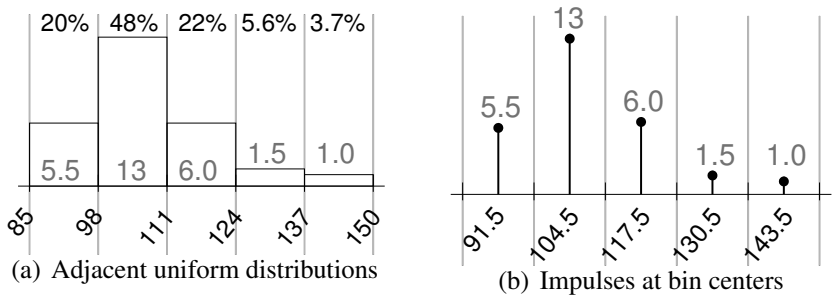
The core of CP is the distribution model associated with each monitor. CP is based on the empirical distribution and histograms because we believe this approach to be both effective and efficient. An alternative would be to create parametric probability models.

The empirical distribution is a non-parametric model that makes no assumptions about the shape of the data. Parametric probability models assume that data comes from a family of distributions characterized by a fixed set of parameters. Building the model entails estimating the values of those parameters. For instance, a normal distribution is parameterized by the mean and standard deviation of the data. While those two parameters are easily estimated and have a small space requirement, we have no justification to assume that monitor values are distributed according to any particular distribution. In fact, preliminary data contradicts this assumption [20]. More flexible parametric models can better estimate arbitrary distributions, but require a larger number of parameters. Unfortunately, accurately estimating many parameters necessitates many data samples in order to constrain each degree of freedom in the model. For example, the generalized lambda distribution can approximate a large number of well-known distributions, but is parameterized by the first four moments of the data [67]. Thus, the model may be very different from the real distribution of the data when the number of raw profiles collected is small.

5.1.3 Statistical Considerations

Any execution profile is a statistical model of program behavior; FDO uses these models to predict future program behavior. It is therefore important to identify the assumptions that limit the prediction accuracy of the model. Compared to traditional FDO, CP makes this statistical modeling explicit and replaces point statistics with probability distributions. By using raw profiles to build a CP, the CP inherits the statistical assumptions of those raw profiles. An edge profile does not model the correlations in execution frequency between edges in a CFG. Thus, when an edge profile is used to estimate CFG edge frequencies, the estimate is made under the assumption that each edge frequency is statistically independent. This limitation of edge profiles inspired path profiles. A *combined edge profile* (CEP) built from multiple edge profiles cannot remove this assumption. However, at the time of combination, the model could measure cross-run edge correlations. That model would be a joint distribution across all edges, and would require space exponential in the number of edges. Furthermore, a vast number of input profiles would be needed to estimate all of the joint probabilities.

Hierarchical normalization, presented in Section 5.3.1, models the correlations between a monitor and its immediate dominator, and assumes independence for other pairings. This assumption allows the size of the model to grow linearly with the number of monitors, and furthermore allows queries to the combined profile to be computed in constant time.



(a) Adjacent uniform distributions

(b) Impulses at bin centers

Figure 5.2: Alternative interpretations of a monitor's histogram

The inter-run independence assumptions of hierarchically-normalized combined profiles are analogous to the intra-run assumption made by the underlying profiling technique. An edge profile assumes all edge probabilities are independent; a CEP assumes that edge probabilities remain independent across runs. A path profile models the (in)dependence between edges within a path, but assumes that all paths are independent within a run. A *combined path profile* (CPP) maintains the correlations between edges in a path, but extends the assumption of path independence across multiple runs.

5.2 Approximating the Empirical Distribution

A simple method to create a model is to build the empirical distribution, where the data *is* the distribution. This approach requires the storage and analysis of all existing profiles. However, in the context of compiler decisions, a coarse-grained distribution model is sufficient because small variations in a distribution have no impact on decision outcomes. Therefore, the empirical distribution can be approximated by storing quantized monitor values in histograms.

A monitor's histogram can be interpreted in two ways, as illustrated in Figure 5.2. In both interpretations, the underlying histogram has five bins, each 7.0 units wide, over the range $[85, 150]$. The first bin contains 5.5 units of weight, the second 13 units of weight, and so on. Assuming that a monitor is uniformly distributed within a bin, as in a Riemann sum, its histogram forms a contiguous n-step probability distribution, with a well-defined and piece-wise continuous CDF and inverse CDF. This interpretation is shown in Figure 5.2(a). FDO's limited precision requirements make this assumption reasonable. The probability that a monitor's value for a run will be in the range covered by the i^{th} histogram bin is the proportion of the histogram's total weight falling in that bin, and is shown at the top of the figure. For instance, the 5.5 units of weight in the first bin in Figure 5.2(a) account for 20% of the total histogram weight, and thus there is a 20% probability that the monitor will have a value between 85 and 98.

Alternatively, a histogram can be interpreted as a set of impulses centered at the midpoints of each bin, as in Figure 5.2(b). Under this interpretation, the 5.5

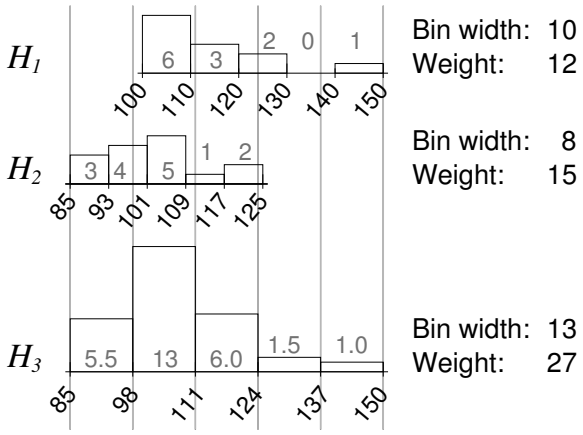


Figure 5.3: Combining histograms: $H_1 + H_2 = H_3$

units of weight in the first histogram bin are a single impulse at exactly 91.5, the mid-point of the bin. Similarly, an individual monitor value from a raw profile, or monitors where all observations have the same value, can be seen as degenerate *point histograms* where all the weight is contained in a single point (single impulse).

5.2.1 Building Histograms

For the purposes of combined profiling, a histogram H is a structure containing the following fields:

bins	The number of bins used by the histogram.
$\min, \max$	The histogram covers the range $[H.\min, H.\max]$. Zero-valued samples are not added to bins, so $H.\min > 0$.
$\mathrm{bin}[b]$	The weight in the b^{th} bin of an array of weights. Bins are counted from 1. The lower bound of $H.\mathrm{bin}[1] = H.\min$, and the upper bound of $H.\mathrm{bin}[H.\mathrm{bins}] = H.\max$.
W	The total *non-zero* weight added to H, *i.e.*, the sum of weights in the bins.
TW	The total weight added to H. TW is the sum of W plus the weight assigned to zero, which is not stored in any bin.
S	The sum of all values added to H.
SS	The weighted sum of squared deviations from the histogram mean ($\sum weight(value - H.\mu)^2$)

The values of W, TW, S, and SS allow the weighted mean and standard deviation of the values added to H to be updated incrementally. Tracking both W and TW allows queries to the histogram to choose whether or not to include the raw profiles where a monitor is never executed. The weighted average value, $H.\mu$,

is computed by either $H.\mu = \frac{H.S}{H.W}$ (zeros not included) or $H.\mu = \frac{H.S}{H.TW}$ (zeros included). The weighted standard deviation $H.\sigma$, is computed $H.\sigma = \frac{H.SS}{H.W}$ (zeros excluded). To include zeros in the standard deviation, a copy of SS, SS', is incrementally updated to include $H.TW - H.W$ weights-worth of zero values, and then $H.\sigma = \frac{H.SS'}{H.TW}$.

The histogram of a combined profile may be updated in a batch, incrementally, or by a hybrid approach. The update method is unaffected by the choice of update frequency. In general, updating produces a new histogram in 4 steps, with details to follow:

1. Determine the range of the combined data. Create a new histogram with this range.

2. Proportionally weight the bins of the new histogram according to their overlap with the bins of the original histogram.

3. Add the new data by increasing the weight in the appropriate bins.

4. Calculate new values for the mean and variance.

Usually, data added to a histogram comes from raw profiles, and takes the form of one $\langle weight, value \rangle$ pair from each new profile. Weights are assigned on a per-profile basis, and default to 1.0. If both the weight and value are non-zero, $weight$ is added to the bin who's range encloses $value$, and $weight$ is added to both W and TW. Otherwise, the only change to the histogram is the addition of $weight$ to TW.

The combination, or addition, of two histograms H_1 and H_2 to form a new histogram H_3 is illustrated in Figure 5.3. The range of H_3 is simply the minimum encompassing range of the ranges of H_1 and H_2: $H_3.\min = min(100, 85)$ and $H_3.\max = max(150, 125)$. This range ($[85, 150]$) is divided into the same number of bins as were present in H_1 (5), giving H_3 a bin width of 13. The weight of a bin $H_3.\text{bin}[b]$ is given by the weights of the bins of H_1 and H_2 that overlap the range of $H_3.\text{bin}[b]$, multiplied by the proportion of overlap. Take for example $H_3.\text{bin}[3]$ in Figure 5.3. In H_1 the bin width is 10, and in H_2 the bin width is 8. The weight in $H_3.\text{bin}[3]$ is calculated as follows:

$$
\begin{aligned}
overlap(H_1, H_3.\text{bin}[3]) &= 3\left(\tfrac{120-111}{10}\right) + 2\left(\tfrac{124-120}{10}\right) = \tfrac{27+8}{10} = 3.5 \\
overlap(H_2, H_3.\text{bin}[3]) &= 1\left(\tfrac{117-111}{8}\right) + 2\left(\tfrac{124-117}{8}\right) = \tfrac{6+14}{8} = 2.5 \\
H_3.\text{bin}[3] &= 3.5 + 2.5 = 6.0
\end{aligned}
$$

Updating the sample mean and variance of a combined profile uses a weighted version of the parallel algorithm due to Chan *et al.*. [29]. Given two bags of $\langle weight, value \rangle$ pairs, A (*i.e.*, the original histogram H_1) and B (*i.e.*, the new data,

or another histogram H_2), the W, TW, S, and SS from each set are combined to calculate the values of these fields for the new histogram H_3:

$$\begin{aligned} H_3.W &= H_1.W + H_2.W \\ H_3.TW &= H_1.TW + H_2.TW \\ H_3.S &= H_1.S + H_2.S \\ H_3.SS &= H_1.SS + H_2.SS + \frac{H_1.W \times H_2.W}{H_1.W + H_2.W} \left(\frac{H_1.S}{H_1.W} - \frac{H_2.S}{H_2.W} \right)^2 \end{aligned}$$

5.2.2 Multiplication of Histograms

In the next section, the ability to multiply together histograms (or monitors) is necessary. However, multiplication is not a well-defined operation on histograms. Recalling that these histograms are probability distributions, multiplication intuitively represents the conjunction of the events represented by the two monitors; the resulting histogram is for the situation where both monitored behaviors occur. As well, the *coverage* ratio $\frac{H.W}{H.TW}$ represents the (weighted) probability that a monitor is executed in one run of the program.

Consider first the simplest case where a histogram H_1 is multiplied by a point-histogram H_p to produce H_3. H_p is essentially a single $\langle weight, value \rangle$ pair. The value range of H_3 should be scaled by $value = H_2.\min = H_2.\max$. Since the bin ranges are computed from min and max, this scaling automatically moves and resizes the bins:

$$\begin{aligned} H_3.\min &= H_1.\min \times H_2.\min \\ H_3.\max &= H_1.\max \times H_2.\max \end{aligned}$$

Additionally, the weight of H_1 must be scaled by the weight of H_2 to produce the expected coverage ratio. Since we require that:

$$\frac{H_3.W}{H_3.TW} = \frac{H_1.W}{H_1.TW} \times \frac{H_2.W}{H_2.TW}$$

the weights can be set in the expected way:

$$\begin{aligned} H_3.W &= H_1.W \times H_2.W \\ H_3.TW &= H_1.TW \times H_2.TW \end{aligned}$$

Finally, the weights in the bins of H_1 must be similarly scaled to maintain the condition that $H.W$ is the sum of weights:

$$H_3.\text{bin}[b] = H_1.\text{bin}[b] \times H_2.W$$

Now consider the case where H_1 and H_2 are both full-fledged histograms. Under the probability-distribution interpretation, each histogram bin is a uniform distribution. The probability distribution function of $H_1 \times H_2$ is the normalized sum

of the probability distributions of the products of each pair-wise combination of bins between H_1 and H_2. For illustrative purposes, consider two *discrete* uniform distributions, $X \in [1, 10]$ and $Y \in [2, 11]$. The product $X \times Y$ must have the range $[1 \times 2, 10 \times 11]$. However, the resulting distribution will not be uniform: the pair-wise products of the possible values for X and Y will be biased toward smaller values. Likewise, the products of the *continuous* uniform distributions of histogram bins will have non-constant range sizes and will not be uniformly distributed.

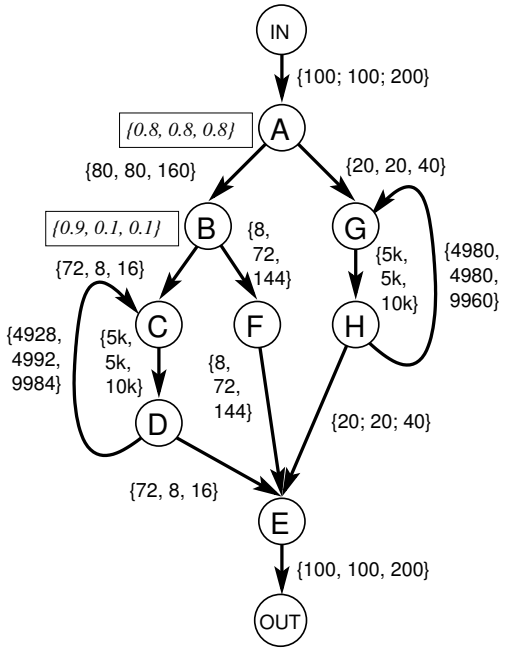
Histograms are already a fairly coarse approximation of monitor distributions. Instead of dealing with $H_1.\text{bins} \times H_2.\text{bins}$ overlapping, variable-width, and non-uniformly-distributed[1] bin×bin products, the multiplication of histograms assumes that each histogram bin is an impulse at the bin's midpoint, as illustrated in Figure 5.2(b). Using a larger number of narrower bins reduces the modeling error induced by this assumption. Consequently, the product of two bins is a $\langle value, weight \rangle$ pair, where the *value* is the product of the bin midpoints, and *weight* is the product of the bin weights. The point-wise product of two histograms uses all pairings between the b_1 bins of H_1 and the b_2 bins of H_2 to produce a bag B of $b_1 b_2$ weighted-value pairs. First, the range and weight of H_3 are computed as above. Then, H_3 allocates the same number of bins as H_1, and the pairs in B are added to H_3.

5.3 Unifying and Using Profile Information

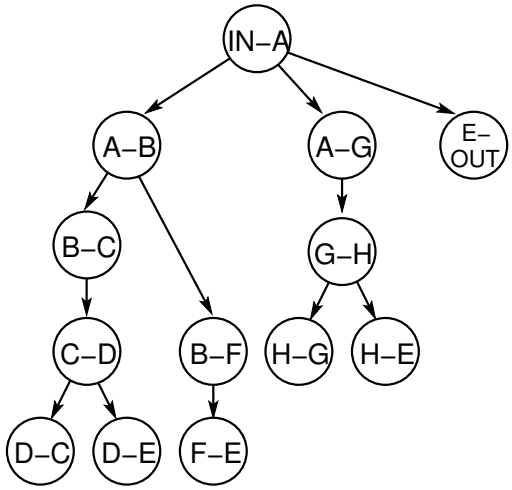
CP provides a data representation for profile information, but does not specify the semantics of the information stored in the combined profile. Raw profiles cannot be combined naively. To illustrate this point, Figure 5.4(a) presents a CFG and the table in Figure 5.4(c) provides the edge frequencies observed for three profiles: P1, P2, and P3. The numbers within the rectangles are the probabilities for edges A→B and B→C. First note that averaging values across profiles is misleading because it can easily characterize behavior in a way that does not correspond to any individual profile; The average branch probability at B is 0.37, hiding its strongly biased behavior. In all three profiles, the probability of entering the G-H loop from A is 0.2. The loop trip counts for P1 and P2 are identical, but the probability of entering the C→D loop from B is 0.9 in P1 and 0.1 in P2. P3 is identical to P2, except that all edge counts are doubled. Therefore, P2 and P3 are essentially the same profile; if they were combined, the resulting profile should not show any variation in program behavior. However, if the two raw frequencies for an edge such as G→H were combined into a histogram, the values 5,000 and 10,000 would not suggest this consistent behavior.

On the other hand, the raw frequencies for edge C→D in P1 and P2 are both 5,000, but P1 enters the loop much more frequently than P2 due to the 0.9 vs 0.1 branch probability at B. Therefore, the average trip count of the loop in P1 is much

[1] Glen *et al.* provide a case-based algorithm on geometric regions to compute the piece-wise distribution function for the product of continuous random variables [44].



(a) A control flow graph. Edges are labeled with the raw frequencies for {P1, P2, P3}. The probabilities that the left branch is taken from nodes A and B are listed in the adjacent boxes.



(b) The edge-dominator tree for Figure 5.4(a).

		Raw			Normalized		
Edge	**Dom**	**P1**	**P2**	**P3**	**P1'**	**P2'**	**P3'**
IN→A		100	100	200	1.0	1.0	1.0
A→B	IN→A	80	80	160	0.8	0.8	0.8
A→G	IN→A	20	20	40	0.2	0.2	0.2
G→H	A→G	5,000	5,000	10,000	250	250	250
H→G	G→H	4,980	4,980	9,960	249	249	249
H→E	G→H	20	20	40	4.0e-3	4.0e-3	4.0e-3
B→C	A→B	72	8	16	0.9	0.1	0.1
C→D	B→C	5,000	5,000	10,000	69.4	625	625
D→C	C→D	4,928	4,992	9,984	1.0	1.0	1.0
D→E	C→D	72	8	16	1.4e-2	1.6e-3	1.6e-3
B→F	A→B	8	72	144	0.1	0.9	0.9
F→E	B→F	8	72	144	1.0	1.0	1.0
E→OUT	IN→A	100	100	200	1.0	1.0	1.0

(c) Profiles P1, P2 and P3 show raw edge frequency counts. P1', P2', and P3' are hierarchically-normalized profiles suitable for combined profiling.

Figure 5.4: The CFG and edge-dominator tree of a procedure, with three possible edge profiles

lower (69.4) than in P2 (625). In this case, histogramming the raw frequencies suggests consistent behavior for the loop, which is misleading.

5.3.1 Hierarchical Normalization

The problem in both of the examples presented above is that the pairs of measurements were taken under different conditions. Thus, when combining these measurements, all values recorded for a monitor must be normalized relative to a common fixed reference. *Hierarchical normalization* (HN) is a profile semantic designed for use with CP that achieves this goal by decomposing a CFG into a hierarchy of dominating regions. The results of using HN for the profiles in Figure 5.4(c) are shown in the right portion of the table. As desired, P2 and P3 are identical, and the differences in loop trip count between P1 and P2 are identified.

HN is presented for edge profiling. Vertex profiles are treated identically, but use the domination relationships between vertexes instead of edges. Domination is usually defined in terms of vertexes. In order to use an existing implementation of a vertex dominator-tree algorithm with edge profiles, use the line graph of the CFG instead of the CFG itself. The line graph contains one vertex for each edge in the CFG, and edges in the line graph correspond to adjacencies between the edges of the CFG.

Decomposing a CFG into a hierarchy of dominating regions to enable HN is achieved by constructing its dominator tree. Each edge in the CFG is represented by a node in the dominator tree. Denote the immediate proper dominator of CFG edge e by $dom(e)$. Each non-leaf node $n(e)$ in the dominator tree is the head of a region G_e, which, by construction, encompasses any regions entered through descendants of e. To prepare a raw profile for combination with other profiles, the frequency f_e of each non-root node $n(e)$ is normalized against the frequency of its immediate proper dominator, $f_{dom(e)}$. The ratio of these two frequencies is invariant when a branch probability or loop iteration count is (dynamically) constant. Along with the issues illustrated in Figure 5.4, this process also prevents variable behavior in an outer loop from masking consistent behaviors within the loop. Normalization proceeds in a bottom-up traversal of the dominator tree, so that the head of a region is normalized to its immediate dominator only after all of its descendents have been normalized. The root of the dominator tree, *i.e.*, the edge representing entry into the procedure, is assigned a "normalized" value of 1. The HN for the example is shown in the left portion of the table in Figure 5.4(c).

In order to understand the capabilities and limitations of a statistical model incorporating HN, and to use it correctly, the model must be precisely defined. Therefore, let $\mathcal{F}_e$ and $\mathcal{F}_{dom(e)}$ be random variables for the raw frequencies of e and $dom(e)$, respectively. Define a new random variable $Y_e = \frac{\mathcal{F}_e}{\mathcal{F}_{dom(e)}}$, which is the frequency of edge e with respect to its dominator. The raw profile from run 1 of the program records $f^1_{dom(e)}$ and f^1_e, the observed frequencies of the two nodes over that run. One sample of Y_e, $y^1_e = \frac{f^1_e}{f^1_{dom(e)}}$ is calculated as the hierarchically normal-

ized value for e. Over k runs, k samples $y_e^1, y_e^2, ..., y_e^k$ are added to the histogram of R_e. Thus, the histogram of monitor R_e is an approximation for the true probability density R_e^*:

$$\mathbb{P}(R_e \leq \theta) \approx \mathbb{P}(R_e^* \leq \theta) = \mathbb{P}\left(Y \leq \theta\right) = \mathbb{P}\left(\frac{\mathcal{F}_e}{\mathcal{F}_{dom(e)}} \leq \theta\right)$$

5.3.2 Denormalization

The properties of a monitor R_a can only be directly compared to those of a monitor R_b when $dom(a) = dom(b)$. However, more generalized reasoning about R_a may be needed when considering code transformations. Similarly, when code is moved by a transformation, its profile information must be correctly updated. *Denormalization* reverses the effects of hierarchical normalization to lift monitors out of nested domination regions by marginalizing-out the distribution of the dominators above which they are lifted. Denormalization is a heuristic method rather than an exact statistical inference because it assumes statistical[2] independence between monitors.

Consider first the hierarchically-normalized raw profiles in Figure 5.4. Intuitively, the expected execution count of node F for a single execution through the graph is calculated:

$$\begin{aligned}
\mathrm{BP}_l(A) &= \left(\frac{f_{A\leftarrow B}}{f_{A\leftarrow B} + f_{A\leftarrow G}}\right) \\
\mathrm{BP}_r(B) &= \left(\frac{f_{B\leftarrow F}}{f_{B\leftarrow C} + f_{B\leftarrow F}}\right) \\
\mathbb{E}[f_F] &= \mathbb{E}[R_{IN\leftarrow A} \times \mathrm{BP}_l(A) \times \mathrm{BP}_r(B)] \\
\mathrm{P1} : \mathbb{E}[f_F] &= 1.0 \times 0.80 \times 0.90 = 0.72 \\
\mathrm{P2, P3} : \mathbb{E}[f_F] &= 1.0 \times 0.80 \times 0.10 = 0.08
\end{aligned}$$

where $\mathrm{BP}_d(n)$ is the probability of a branch going in direction d (either (l)eft or (r)ight) from node n. However, even a single raw profile is a statistical model. Thus, the calculation above assumes that the edge frequencies are independent.

With the same assumption, the same approach can be used with a CP. Thus, for the CP built from P1, P2 and P3:

$$\mathbb{E}[f_F] = 1.0 \times 0.8 \times \left(\frac{0.9 + 0.1 + 0.1}{3}\right) = 0.29$$

which is the average of the expected frequencies.

The mean is a special case of marginalization; independence allows the joint distribution to be broken into the product of individual distributions, where the expectation associates over the product, simplifying the calculation to the product of

[2] R_i,R_j are independent iff $\forall i, j : \mathbb{P}(R_i = i, R_j = j) = \mathbb{P}(R_i = i)\mathbb{P}(R_j = j)$. Control-flow equivalence implies independence. Independence does not hold in most other cases.

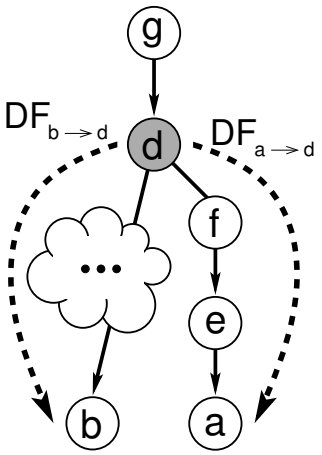


Figure 5.5: Denormalization of R_a and R_b with respect to their least-common dominator R_d. Dashed lines show the path over which the marginalized histograms are computed.

means seen above. Thus, to recover an "absolute" expected execution count from an HN CProf, multiply the means of each monitor up the dominator tree to the procedure entry. Then, multiply by the expected invocation frequency of the procedure (possibly using this technique over a CG CProf). Denormalization is this process of multiplying monitors along a path in the dominator tree. The mean is a special case of denormalization because it does not require the distribution of monitor values. The general denormalization technique is formally presented in the remainder of this section.

Let R_a and R_b be monitors from the same CFG. Let $dom^i(R_a)$ be the i^{th} most-immediate proper dominator of R_a. The least-common dominator of R_a and R_b is $R_d = dom^j(R_a) = dom^k(R_b)$, where there is no monitor R_n such that R_d properly dominates R_n, and R_n dominates both R_a and R_b. Denormalizing R_a from the region dominated by $dom(R_a)$ to the region dominated by R_d is achieved by walking up the dominator tree. Let $\widehat{R_n^{-i}}$ be the denormalized distribution when R_n is lifted above $dom^i(n)$. $\widehat{R_n^{-1}}$ is created by multiplying together the histograms H_n and $H_{dom(n)}$. Denormalization can be applied to R_a and R_b recursively to produce the desired $\widehat{R_a^{-j}}$ and $\widehat{R_b^{-k}}$, which can be compared.

The computation of $\widehat{R_n^{-i}}$ takes $O(ib^2)$ time, assuming that all histograms use b bins. The number of bins is chosen by the user. There is a tradeoff between accuracy and precision on one hand and memory space and computation time on the other.

Figure 5.5 shows a dominator tree containing the nodes a and b and their least common dominator d, which is shaded. The dashed lines illustrate the paths followed, in the dominator tree, to compute the denormalization. Nodes f and e are in the path from d to a and there might be other nodes in the path from d to b. Thus,

$R_d = dom^3(R_a)$, and the histogram for $\widehat{R_a^{-3}}$ is calculated:

$$\widehat{H_a^{-3}} = H_a \times H_e \times H_f \times H_d$$

5.3.3 Queries

In an AOT compiler, profiles are used to predict program behavior. Thus, raw profiles are statistical models that use a single sample to answer exactly one question: *"What is the expected frequency of X?"* where X is an edge or path in a CFG or a Call Graph (CG). A CP is a much richer statistical model that can answer a wide range of queries about the measured program behavior. The implementation of CP used in this work provides the following statistical queries as methods of a monitor's histogram:

$H.\min, H.\max$:
: The maximum and non-zero minimum monitor value observed, as in Section 5.2.1.

$H.\text{mean}(incl0s)$:
: The true weighted average of observed monitor values. If *incl0s* is true, count raw profiles where the monitor did not execute as 0-valued observations, as in Section 5.2.1.

$H.\text{stdev}(incl0s)$:
: The true weighted standard deviation of observed monitor values. If *incl0s* is true, count raw profiles where the monitor did not execute as 0-valued observations, as in Section 5.2.1

$H.\text{estProbLessThan}(v)$:
: Estimates the value of the monitor's CDF at v, *i.e.*, $\mathbb{P}(R \leq v)$. The estimation is based on the assumed uniform distribution of bins.

$H.\text{quantile}(\text{q})$:
: For $0 \leq q \leq 1$, estimates the (minimum) value v at the point where the monitor's CDF equals q, *i.e.*, v when $\mathbb{P}(R \leq v) = q$. The estimation is based on the assumed uniform distribution of bins.

$H.\text{applyOnRange}(F(w,v), vmin, vmax)$:
: Computes the sum of applying the function $F(w,v)$ to the impulses of the histogram in the range $[vmin, vmax]$, as with the multiplication of histograms in Section 5.2.2. For bins that partially overlap the range, the impulse's weight is proportional to the overlap of the bin with the range, and the impulse's value is the midpoint of the overlapping range.

$H.\text{applyOnQuantile}(F(w,v), qmin, qmax)$:
: Like applyOnRange, but the range is set using the values associated with the quantile points *qmin* and *qmax*.

$H.\text{coverage}$:

The probability of the monitor executing in a run of the program, computed as $\frac{H.W}{H.TW}$.

$H.\text{span}$:

The ratio between the range of the histogram and its maximum value, computed as $\frac{H.\max - H.\min}{H.\max}$.

The mean value of a monitor is analogous to the value provided by a single raw profile, and provides the desired substitutability of a CProf for a raw profile in existing FDO transformations.

The additional statistical information provided by a CProf allows an FDO heuristic to quantify the expected trade-offs between various workload-performance measurements, such as between the impact on the 5%-quantile (nearly worst-case) or the average impact on the 5%–95%-quantile range (omitting potential outliers). In some transformations, the order in which candidates are considered is important [26]. CP allows a sorting function to use, for example, both the mean and the standard deviation of the candidates in order to prioritize low-variance opportunities. Behavior variation should not, by itself, inhibit optimization. Rather, CP enables the accurate assessment of the potential performance impact of transformations informed by variable-behavior monitors in a variety of ways, and with adjustable confidence in the result. Concrete examples of this kind of analysis are provided by the implementation of an FDO inliner using CP described in Chapter 6.

5.3.4 Extensions and Alternative Usage

The empirical-distribution methodology of CP is orthogonal to the techniques used to collect raw profiles. CP is applicable whenever multiple profile instances are collected, including intra-run phase-based profiles, profiles collected from hardware performance-counter, and sampled profiles. The main issue when combining profiles is how normalization should be done in order to preserve program-behavior characteristics.

CFG Paths

An algorithm that collects path profiling in a program that contains loops must break cycles. The most commonly used technique to break such cycles is due to Ball and Larus [12]. Given a simple loop, the main idea is to replace the back edge with a set of sub-paths that include (a) a path from a point outside the loop to the end of the first iteration, (b) a path from the loop entry point to a point outside the loop, and (c) a path from the entry point to the exit point in the loop. Figure 5.6 illustrates some of the paths inserted to replace the two back edges in a double-nested loop.

Hierarchical normalization must be adapted to work with paths because there are no dominance relationships between paths. Consider two runs of the double-nested loop of Figure 5.6, where the outer loop L_1 iterates a total of $k = 10,000$

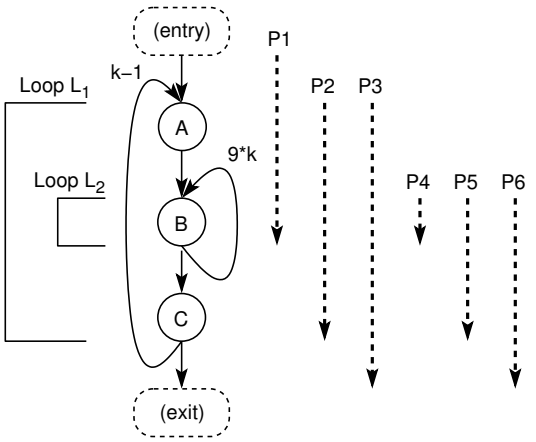


Figure 5.6: Some sub-paths through a nested loop. The outer loop L_1 iterates a total of k times; the inner loop L_2 iterates 10 times per iteration of L_1.

times in the first run, and $k = 100$ times in the second run. In both cases the inner loop L_2 iterates 10 times per iteration of L_1. A combined profile should identify the path of execution within L_1 as consistent across runs, but should indicate that the frequency of the paths into and out of L_1 vary significantly from run to run. The solution is to normalize path frequencies with respect to the frequency of the vertex that starts the path. For instance, P4 should be normalized to the frequency of L_2 to factor out k and preserve the constant nature of the inner loop.

Program Call-Graphs

Combined profiling can easily be extended to call-graphs (CG)[3]. Profiling a CG gathers information about the frequency of inter-procedural calls. A CG can be represented in multiple ways. For instance, a single edge may represent all calls from a procedure foo() to a procedure bar(). Alternatively, there may be a separate edge for each call-site in foo() that targets bar(). If context-sensitivity is included, there are several alternatives to keep track of the execution path that leads to a call from foo() to bar(). A common solution is to keep track of the k most recent calls on the stack when the call from foo() to bar() occurs [76]. This sequence of calls is called a *call string*.

Unlike a CFG, a CG is not a well-structured graph. Consequently, the dominator tree is often very wide and shallow, which limits the utility of applying HN to the full CG. Instead, we propose that CG monitors are normalized with respect to the invocation frequency of the procedure where the behavior originates. In the case of CG profiles that do not use context sensitivity, call frequencies are normalized against the caller's frequency. Likewise, when context-sensitivity is used to collect a CG profile, call-string frequencies are normalized against the frequency of the caller of the first call in the string. The combined profile then provides a conditional

[3]We do not attempt to extend CP to inter-procedural paths [75].

distribution describing the expected frequency of following a call or call string, given that the start of the call string has been reached.

Value Profiling

A monitor R for value profiling observes the run-time values of a variable at a specific program point in order to enable specialization transformations [25]. Each profiling run produces a histogram of the frequency of observed values of the variable. However, since a variable could potentially take very many different values over a program run, a caching technique is used to estimate the frequency of the n most frequent values. Since a value profile is completely local to a single program point, there is no hierarchy over which to normalize; normalization simply requires converting the frequency for each value v, $f(v)$, into a proportion of the total number of observations, $\mathbb{P}(v)$ (*i.e.*, the probability of observing v). The CProf for value profiling then creates a histogram for each frequently-observed v over the $\mathbb{P}(v)$ of each run. Thus, the CProf identifies the frequent values of R, and the distribution of the likelihood of observing each value. If the set of frequent values is not consistent across runs, less-likely values may need to be pruned from the CProf, or the variable may simply be marked as unsuitable for specialization.

Profiling Granularity

This work assumes that CP will combine input profiles from complete, single, program runs. However, the input profiles can have arbitrary granularity. For instance, CP could combine CFG profiles from each separate invocation of a function (a finer temporal granularity). Similarly, each thread in a concurrent application could contribute a separate raw profile for combination into a multi-threaded CProf for a single run (thread-level granularity). In conjunction with phase detection, a CProf could be build to represent behavior variation between fine-grained program phases. Conversely, long-running server applications could periodically commit profile information to a CProf to model program behavior variation over different times of day or even different days of the week.

5.4 Characterizing Combined Profiles

Combined profiling is a data representation for profile information collected over multiple runs, and is motivated by the observation that program behavior is input dependent and varies from run to run. Thus, a CProf supports queries to allow the variation present in a program workload to be assessed. These metrics are calculated on a per-monitor basis for edge and path profiles for SPEC CPU 2006 integer C benchmarks. All the inputs provided by SPEC are used; when the SPEC 'ref' run uses multiple inputs, each of these inputs is treated separately. Additional inputs for `MCF` are taken from Berube [16]. `Bzip2` uses the 1000-input workload

from kDataSets [102]. All combined edge profiles employ HN. The figures for path profiling are very similar to their edge profiling counterparts, and are omitted. All experiments are performed, and all metrics are calculated for combined profiles using 10, 20, 30, 40, and 50 histogram bins. We present detailed results for 50 bins. Section 5.4.8 discusses the impact, which is modest and anticipated, of the number of bins on the reported metrics.

The figures in this section, such as Figure 5.7, use violin plots, which are probability densities drawn vertically with the weight centered horizontally: the width of the shaded area represents the probability mass at the corresponding y-axis value. A uniform distribution would appear as a vertical band with constant width, while a normal distribution would have a bulge at its mean and thin to a vertical line. Gaussian smoothing transforms the discrete experimental data into a continuous distribution. A black dot is placed at the mean of the data. The values listed at the tops of the figures identify the number of unique monitors represented in the plot.

5.4.1 Histogram Breakdown

Table 5.1 summarizes the characteristics of the combined profiles for each benchmark. *Runs* indicates the number of program inputs in the workload. The *Monitors* column lists the number of unique (E)dge or (P)ath monitors executed at least once across all runs. All results exclude unexecuted monitors.

The "%" column for *Histograms* indicates the proportion of monitors who's non-zero values vary across the workload and thus require a histogram for accurate modeling. Monitors executed in every run have *Full* coverage. Otherwise, they have *Partial* coverage. No less than 10% of monitors require histograms, demonstrating the presence of input-dependent program behavior in all benchmarks. The following subsections examine this variation in detail.

The column labeled *Points* list the proportion of monitors that are *Point distributions* in the CP. Points arise when all non-zero values for a monitor are equal: all the probability mass occurs at a single point on the real number line. In these cases, no histogram bins are required to represent the monitor in the CP. Point histograms at 1.0 are uninteresting because they indicate that the monitor always executes the same number of times as its dominator. For example, in Figure 5.4, the F→E edge would have a point distribution at 1 because it is immediately dominated by, and control-flow equivalent to, B→F, and thus must always have an HN frequency of 1. For edges, most of these monitors can likely be proved redundant by static analysis and then removed from the CP to reduce file size with no loss of information. For paths, these point histograms identify paths that execute exactly once each time their procedure executes. In most cases, such a path is the only non-loop path executed in a function. Point distributions at values other than 1 are more interesting. These points arise is cases such as (dynamically) constant loop trip counts or branch probabilities. CP allows a compiler to evaluate potential code transformations involving these monitors with confidence that the analysis is applicable to all program

			Histograms			Points (%)	
Name	**Runs**	**Monitors**	**%**	**Partial**	**Full**	$\neq 1.0$	$= 1.0$
bzip2	1,000	2,182 E	35	366	399	3	61
		1,295 P	90	904	265	3	6
gcc	11	93,748 E	43	15,973	24,703	3	52
		41,276 P	81	18,972	14,724	13	4
gobmk	20	29,858 E	49	12,807	2,030	4	45
		64,436 P	82	51,051	2,011	16	1
h264ref	5	8,846 E	26	1,023	1,297	8	65
		4,857 P	76	2,398	1,325	13	9
hmmer	4	1,534 E	11	1	179	5	82
		390 P	46	8	174	21	31
lbm	3	188 E	10	1	18	28	61
		99 P	15	3	12	56	28
libquantum	3	585 E	27	8	150	1	71
		220 P	63	13	126	5	31
mcf	12	491 E	42	22	187	1	56
		249 P	83	39	170	1	14
milc	3	1,933 E	11	12	202	16	72
		750 P	32	23	217	44	23
sjeng	3	3,778 E	58	127	2,077	2	38
		36,111 P	41	9,913	4,971	58	0
sphinx3	3	3,278 E	10	9	342	14	74
		1,147 P	35	30	374	36	28

Table 5.1: Characteristics for batch-combined (E)dge and (P)ath profiles.

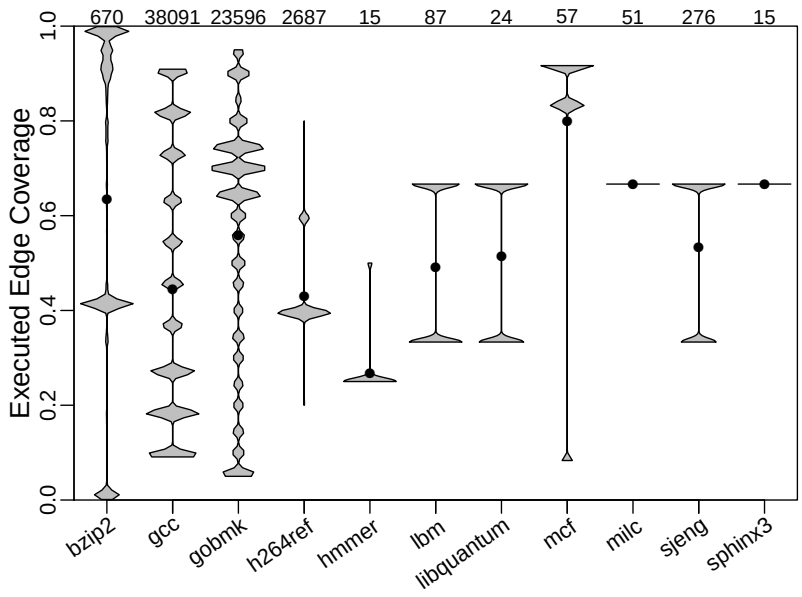


Figure 5.7: Edge coverage, excluding fully-covered monitors

runs.

5.4.2 Coverage

One benefit of using multiple profiling runs is that these runs might exercise more of the program code than any individual run. The dichotomy between a monitor being executed or unexecuted in a run is perhaps the most obvious indicator of behavior variation. We report the *coverage* of a monitor as the proportion of runs where the monitor executes.

Figure 5.7 shows the distribution of monitor coverage across the workload, excluding fully-covered monitors. The coverage value is normalized to the number of runs. For example, 670 of the 2182 executed edges in `bzip2` are not executed in every run. On average, those edges are executed by about 65% of the 1000 runs. However, the small bulge at the bottom of the plot indicates that several edges are covered by very few runs; a large group of edges are covered by slightly more than 40% of the runs, and another group of edges are covered by more than 85% of the runs. An FDO compiler would be oblivious to the execution of any or all of these monitors using a single-input profile, and may consequently make suboptimal decisions from a whole-workload perspective.

`Hmmer`, `libquantum`, `milc`, `sjeng`, and `sphinx3` have more than 90% of their executed edges covered by every run, which may be due to a lack of diversity in their very small workloads. At least 30% of the edges in the other benchmarks are not executed in every run, up to 79% for `gobmk`. Furthermore, the distribution of coverage for these benchmarks shows that the number of runs that did not execute some edge is spread across the range, indicating that these differences in

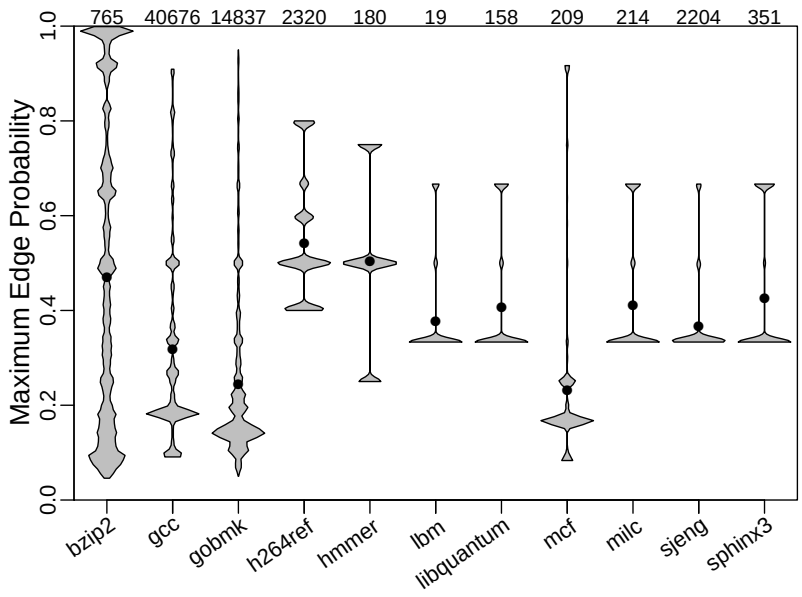


Figure 5.8: Maximum edge likelihood with 50 bins (no points)

coverage are unlikely due to a small number of large-scale control-flow alternatives. Particularly for `gcc` and `gobmk`, the set of executed edges varies significantly from run to run. In contrast, for both `milc` and `sphinx` the ref and train inputs cover identical sets of edges, while the test input misses a handful of edges, producing their distinctive "point violins" at 66%.

5.4.3 Maximum Probability

An FDO compiler uses profile information to predict future program behavior. In the case of point histograms, this prediction can be made correctly from any non-zero sample. The prediction is more complicated when behavior varies from run to run. However, if the behavior is consistent for most runs, then perhaps the most frequently observed behavior is a good predictor. It may be sufficient for transformations to only consider this *dominating behavior*. A CP histogram is a probability distribution: the probability of the monitor taking on a value within the range of a histogram bin is equal to the proportion of the histogram's total weight found in that bin. Thus, the most likely behavior of a monitor can be estimated by finding the bin containing the most weight. The *maximum probability* of a monitor is the proportion of weight in the heaviest bin out of the weight in the histogram. Histogram weight only includes weight from raw profiles that cover the monitor.

Unfortunately, most monitors do not have overwhelmingly dominant behaviors. Figure 5.8 shows the proportion of histogram weight that occurs in the heaviest bin, *i.e.*, the probability of the most likely behavior. For the four benchmarks with more than 10 runs, this probability tends to be low: there is no dominant behavior for these monitors. No single run, and no point statistic, is a good representative of

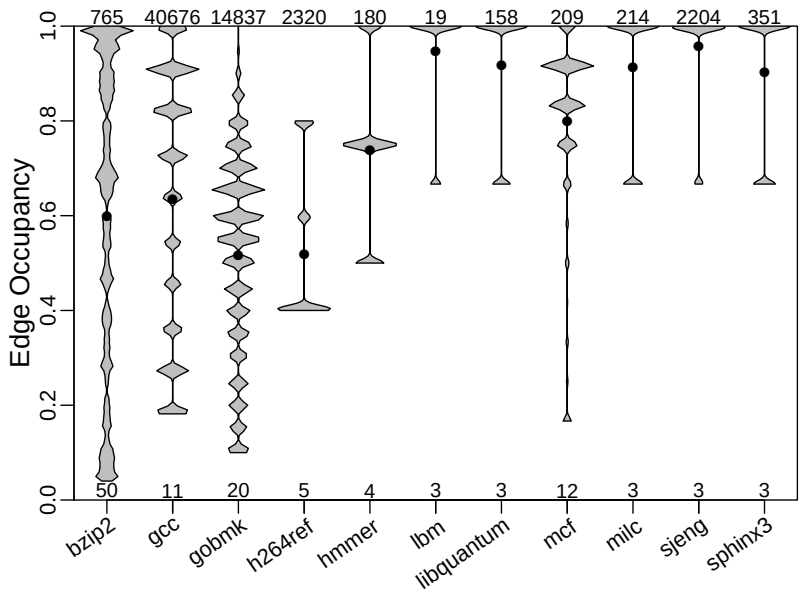


Figure 5.9: 50-bin histogram occupancy for edges (no points)

such monitors. A distribution model is needed to evaluate transformations involving these monitors, as discussed in Section 5.3.3. Redundancy in `bzip2`'s very large workload allows for dominant behaviors in some monitors, as exemplified by the bulge near 100%.

5.4.4 Occupancy

The *occupancy* of a histogram refers to the proportion of bins that contain non-zero weight, and thus indicates how weight is distributed within the histogram. If the weight is distributed across the histogram, many bins will be used, but if weight is concentrated at a few points, then most histogram bins will be empty. The number of non-empty bins is limited by the number of raw profiles combined. This evaluation reports the number of non-empty bins as a proportion of the maximum possible number of non-empty bins.

Figure 5.9 presents bin occupancy: a histogram with weight in many bins increases the violin width toward the top of the figure. The maximum number of occupied bins is listed at the bottom of the figure, which is, with the exception of `bzip2`, the number of runs. The average proportion of bins used is over 50%, indicating that when variation is present, monitor values are not limited to a small number of possibilities. The maximum probabilities discussed above indicate that, in most cases, none of the bins contain a majority of the histogram weight. Consequently, the weight must be distributed across many bins. Visual investigation of the individual histograms for `bzip2` reveals that no single simple parametric model (*e.g.*, uniform, normal) matches the shape of a majority of the histograms [21]. In contrast, CP's histograms match the shape of the data automatically.

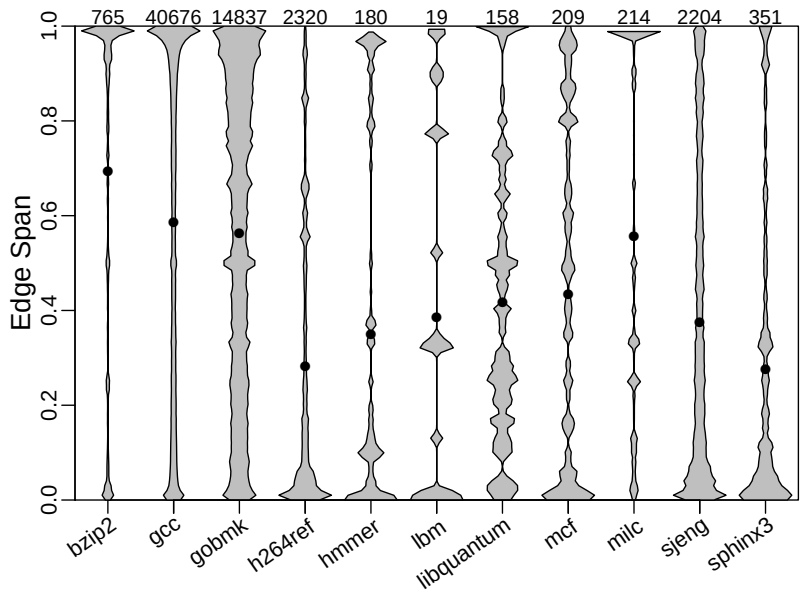


Figure 5.10: Span of edge histograms (no points)

5.4.5 Span

Variation of program behaviors is practically relevant only if the variation is significant compared to typical monitor values. The *span* of a histogram is the ratio between its range and its maximum value. The lower-bound on the range is the smallest non-zero value in the histogram. Figure 5.10 presents histogram span. Recall that monitors use HN to keep behavior variation local to the monitor where it occurs. Practically relevant behavior variation should widen the violin plot toward the top of the figure. Figure 5.10 suggests that all the benchmarks contain monitors that exhibit practically significant behavior variation across the workloads. An FDO compiler must take this variation into account when proposing code transformations by, for example, calculating expected benefit for the worst case or a low-quantile point as well as the average, weighting by coverage, or considering span in sorting functions.

5.4.6 Drift

Ideally, building a combined profile incrementally should yield the same result as building it from a batch of raw profiles. However, when histograms are combined in the incremental construction, weight is proportionally allocated to the overlapping bins in the new histogram. This weight-distribution process can cause histogram weight to shift away from the observed value. *Drift* measures the difference between a combined profile built as a single batch versus one built fully incrementally from the same raw profiles.

Drift is due to histogram ranges growing during incremental construction. However, the final range, and thus the bin boundaries, of both histograms will be iden-

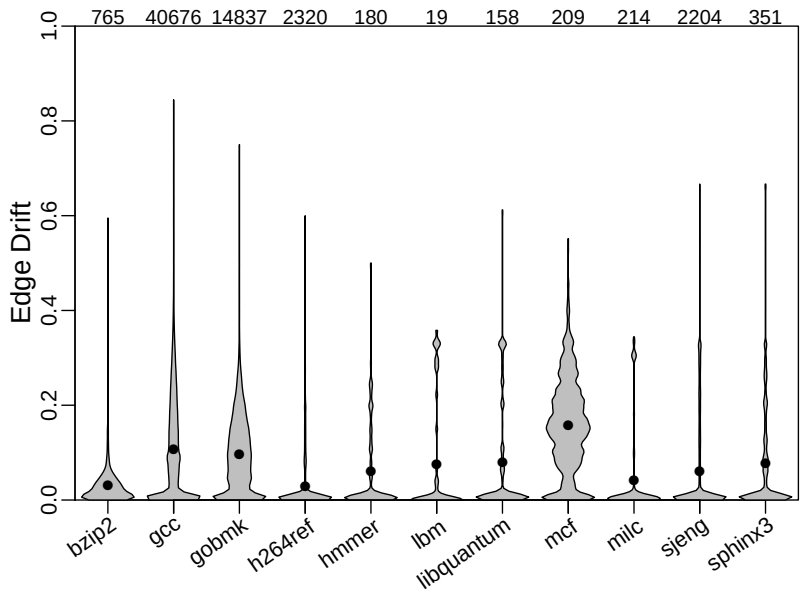


Figure 5.11: Edge-weight drift using 50-bin histograms (no points)

tical because the extreme values in the data are fixed. The difference in weight between corresponding bins is due to drift. Summing these differences for all pairs of bins double-counts total drift: drift is half this sum, reported as a proportion of total histogram weight. This study reports drift using the merged results of 5 different randomly-selected incremental combination orders. Drift is maximized by combining profiles one at a time. Drift is calculated between these CProfs and a batch-combined profile and presented in Figure 5.11. The figure merges all five comparison results. A drift of 0 indicates that the batch and incremental histograms are identical, while a value of 1 is impossible because it requires that the weight in the two histograms not overlap at all.

For benchmarks with few runs, there is very little drift because the histograms do not have a range until the second profile is added; a third profile will only change the histogram range if the new value is not between the first two. The infrequent large drift values occur when the range expansion from the third point causes one of the existing end-point bins to be split near that endpoint's value, causing a large proportion of that bin's weight to be distributed to an adjacent bin.

From the benchmarks with a greater number of runs, `gcc`, `gobmk`, and `MCF` show much more drift. This drift is due to two factors: histogram ranges have been changed more frequently in the incremental construction, causing more bin weights to be split; and more bins contain weight that can drift when the range changes.

However, `bzip2` does not display much more drift than the benchmarks with few runs. The larger the number of raw profiles that have already been added to the CProf, the lower the probability that an additional raw profile will change one of the extreme values. The large number of runs for `bzip2` allows the histogram ranges to expand to approximately their final size before most of the weight is added to

Name	Runs		Raw		Batch		Incremental	
			Single	Total	Size	%	Size	%
bzip2	1,000	E	14	14,392	489	0.03	530	0.03
		P	9	8,062	664	0.08	741	0.09
gcc	11	E	1,047	11,517	9,793	0.85	11,859	1.02
		P	251	2,552	5,853	2.29	7,413	2.90
gobmk	20	E	178	3,557	4,127	1.16	5,417	1.52
		P	432	4,916	10,436	2.12	13,456	2.73
h264ref	5	E	99	495	592	1.19	604	1.22
		P	35	146	420	2.86	433	2.95
hmmer	4	E	59	238	94	0.39	96	0.40
		P	4	15	32	2.13	34	2.25
lbm	3	E	1	4	11	3.17	12	3.20
		P	1	2	6	2.78	6	2.82
libquantum	3	E	5	16	40	2.42	41	2.48
		P	2	7	19	2.80	20	2.98
mcf	12	E	3	31	60	1.91	79	2.53
		P	3	33	45	1.36	63	1.93
milc	3	E	19	56	118	2.10	119	2.13
		P	7	21	54	2.61	56	2.71
sjeng	3	E	32	95	313	3.29	325	3.42
		P	272	451	2,570	5.69	2,584	5.72
sphinx3	3	E	30	91	199	2.18	202	2.21
		P	11	33	83	2.50	87	2.61

Table 5.2: File sizes, in KB, of raw, batch-combined, and incrementally-combined (E)dge and (P)ath profiles. The % column gives the overhead factor for the CProf vs. the collected raw profiles.

the histogram. Thus, the vast majority of the weight in those histograms is subject to very little drift. Therefore, in order to minimize drift, a CProf should initially be batch-constructed from a collection of raw profiles. Subsequent incremental additions to the CProf then have a greatly reduced probability of causing drift.

5.4.7 Space Requirements

Edge profiles grow linearly with the number of edges in a program; path profiles grow linearly with the number of executed paths. In raw profiles, each monitor is represented by a 4-byte counter. Like combined profiles, path profiles only store executed monitors, but add a 4-byte identifier. In a CP, each monitor maintains the true mean and variance of all samples along with the histograms; an entry for

a single monitor is 45 bytes[4], plus a 1-byte bin index and an 8-byte weight per non-empty histogram bin. Thus, even point distributions requires 11x (edge) or 5.5x (path) more space than the same monitor in a raw profile. Table 5.2 presents profile file sizes. Raw edge profiles and batch-combined profiles always have the same size. The sizes of raw path profiles and increment CPs are taken as the largest file across all runs or combination orders, respectively. Comparing batch and incremental combination, the drift observed in Figure 5.11 causes more bins to be non-empty, resulting in larger files. This effect is most visible for the benchmarks with several runs: `gcc`, `MCF`, `gobmk`, and `bzip2`. Likewise, these benchmarks illustrate that the file size grows slowly: as more profiles are combined, it becomes less likely that an additional profile will place weight in an empty histogram bin. `Bzip2` illustrates how CP can dramatically reduce storage requirements for profiles as the total number of profiles becomes large, *e.g.*, systems using continuous profiling.

5.4.8 Number of Bins

The appropriate number of histogram bins is dependent on the precision requirements of the profile's consumers and is independent of the number of raw profiles available. We present data from 50 bins because it is likely a (loose) upper-bound on the required precision.

Coverage, span, and the results in Table 5.1 are properties of program behaviors and are thus independent of the number of histogram bins, while maximum likelihood, occupancy, and drift depend on the number of bins. Results from 10, 20, 30, and 40 bins are consistent with the discussion and conclusions presented above for 50 bins.

Maximum likelihood increases by roughly 0.1 with 10 bins instead of 50 for benchmarks with more than 10 runs. Occupancy decreases slightly with fewer bins, even for benchmarks with only 3 runs: the reduced precision changes some 3-bin monitors into 2-bin monitors. `Bzip2` is the exception. With 10 bins, nearly all monitors have 100% occupancy. Increased precision allows the CP to identify ranges where monitor values are *not* observed, resulting in empty bins. The difference in drift between 10 and 50 bins is negligible for all benchmarks except `MCF`, where drift is reduced by about 0.05 with 10 instead of 50 bins.

File size increases with more bins, though this effect is small or negligible for all benchmarks except `gobmk` and `MCF`, and `bzip2`, where it is most pronounced. As expected from occupancy, `gcc`, `gobmk`, and `MCF` exhibit modest file size increases all the way up to 50 bins despite having 20 or fewer runs. The rate of growth decreases as the number of bins increases. For example, the batch-combined edge profile for `bzip2` is 219 KB with 10 bins, and grows by 75, 70, 65, and 60 KB to reach 489 KB with 50 bins.

[4]Fields are 8-byte doubles; floats would roughly halve the size

5.5 Conclusion

Combined profiling is a practical and statistically sound methodology to model behavior variation across multiple data inputs. Histograms provide a simple representation with bounded space requirements that nonetheless provide the rich set of queries associated with a distribution model. Hierarchical normalization localizes behavior variation to the monitor where it occurs, while providing a consistent frame of reference that separates the characterization of behavior variation from the inevitable, but uninformative, run-to-run variations in raw profile counter values. The experimental evaluation of CP shows that behavior variation is present both in simple programs such as `bzip` and in programs with more complex control flow like `gcc` and `gobmk`. This variation can be captured and queried by the CP statistical model. Chapter 6 demonstrates the use of CP to inform function inlining.

Chapter 6

Function Inlining

Function inlining, or simply inlining, is a classic code transformation that can significantly increase the performance of many programs. A compiler pass that decides which calls to inline, and in which order, is referred to as an inliner. The basic idea of inlining is straightforward: rather than making a function call, replace the call in the originating function with a copy of the body of the to-be-called function. Nonetheless, many inliner designs are possible; Section 6.2 describes the existing inliner in `LLVM`, while Section 6.3 describes the alternative approach used by a new feedback-directed inliner (FDI) that uses CP. All inlining discussed in this chapter is implemented in the open-source `LLVM` compiler [68].

The use of combined profiles to guide FDI allows FDI to consider the impact of inlining decisions across the variations in calling behavior observed in the training workload. Existing FDO systems rely on the single values recorded in raw profiles to inform code transformations. Consequently, FDI is the first feedback-directed transformation to make decisions weighted by more than a single value. Therefore, instead of developing a single inlining heuristic, FDI provides a framework upon which several parameterized families of inlining heuristics are built. These families, described in Section 6.3, demonstrate the versatility provided to transformations by CP's distribution model.

Some terminology is required to identify the various functions and calls involved in the inlining process. The function making a call is referred to as the *caller*, while the called function is the *callee*. The representation of a call in a compiler's *internal representation* (IR) is a *call site*; in `LLVM`, a call site is an instruction that indicates both the caller and the callee. Thus, inlining replaces a call site by a copy of that call site's callee. When a call is inlined, the callee may contain call sites, which are copied into the caller to produce new call sites. The call site where inlining occurs is called the *source* call site. A call site in the callee that is copied during inlining is called an *original* call site, and the new copy of the original call site inside the caller is called the *target* call site.

6.1 Inlining Considerations

There are several issues that any inliner must consider when choosing call sites for inlining. This section examines the conditions that make a call site ineligible for inlining, along with the costs and benefits of inlining a call.

6.1.1 Barriers to Inlining

Not every call site can be inlined. Indirect calls use a pointer variable to identify the location of the called code, and arise from function pointers and dynamically-polymorphic call dispatching. These calls cannot be inlined, because the callee is unknown at compiler time. External calls transfer execution outside of the current compilation unit, such as into different modules (source files) or to statically linked library functions, and cannot be inlined before link time because the source representation of the callee is not available to the compiler. Calls to dynamically-linked libraries can never be inlined. Moreover, if a callee uses a `setjump` instruction or indirect branch (computed goto), that callee cannot be inlined.

Recursion presents a challenge to inlining because care must be taken to prevent infinite inlining of the recursive call chain. The inlining chain of call site c represents the sequence of calls removed from the (dynamic) call chain between the entry into c's caller and the entry into c's callee because of inlining. For instance, if `foo` is inlined into `bar`, any target call sites created by this inlining will append `bar` to their inlining chain; inlining has removed `bar` from the call chain originally required to eventually execute the original call site starting from `foo`. If the callee of call site c is already in c's inlining chain, then c's inlining chain already contains a recursive cycle. For some inliners and some inlining heuristics, inlining a function a second time in an inlining chain may potentially lead to infinite inlining, because the set of target call sites from this inlining are equivalent to those in the initial set of call sites that lead to the creation of c.

The inliner implemented in `LLVM` never inlines directly-recursive call sites (identical caller and callee). Functions that form a multi-call recursive cycle can be inlined into other functions in the cycle, but a function can appear at most once in any inlining chain. The new FDO inliner presented in this chapter uses the same policy to control recursive inlining.

6.1.2 Benefits of Inlining

Inlining a call has a small direct benefit. Removing the call reduces the number of executed instructions. The `call` instruction in the caller is unnecessary, as is the `return` instruction in the callee. Furthermore, any parameters passed to the callee and any values returned no longer need to be pushed onto the stack[1].

[1] Some calling conventions allow values to pass between the caller and callee in registers.

```
fib5() {
  result = iterative( &fibStep(), 5);
  return(result);
}

iterative( *Step(), n) {
  A = array[n+1];
  for(i=1; i<=n; i++)
    A[i] = *Step(A, i);
  return(A[n]);
}

fibStep(A[], n) {
  if( (n==1)||(n==2) ) return(1);
  return(A[n−2] + A[n−1]);
}
```

(a) Original code fragment

```
fib5() {
  Step() = &fibStep();
  n=5;
  A = array[n+1];
  for(i=1; i<=n; i++)
    A[i] = *Step(A, i);
  ret_iterative = A[n];
  result = ret_iterative;
  return(result);
}

fibStep(A[], n) {
  if( (n==1)||(n==2) ) return(1);
  return(A[n−2] + A[n−1]);
}
```

(b) after inlining iterative

```
fib5() {
  A = array[6];
  for(i=1, i<=6; i++)
    A[i] = fibStep(A, i);
  return(A[6]);
}

fibStep(A[], n) {
  if( (n==1)||(n==2) ) return(1);
  return(A[n−2] + A[n−1]);
}
```

(c) after constant and copy propagation

```
fib5() {
  A = array[6];
  for(i=1, i<=6; i++) {
    if( (i==1)||(i==2) ) A[i] = 1;
    else A[i] = A[i−2] + A[i−1];
  }
  return(A[6]);
}
```

(d) after inlining fibStep and simplification

```
fib5() {
  A0 = 1;
  A1 = 1;
  A2 = A0 + A1;
  A3 = A1 + A2;
  A4 = A2 + A3;
  A5 = A3 + A4;
  A6 = A4 + A5;
  return(A6);
}
```

(e) after loop unrolling and scalar promotion

Figure 6.1: A sequence of transformations on a code fragment that computes Fibonacci numbers, illustrating the code-simplification opportunities enabled by inlining

However, the greatest potential benefit of inlining comes from additional code simplification it may enable by bringing the callee's code into the caller's scope. Figure 6.1 presents a running example demonstrating the transformations that become possible due to inlining. Many code analysis algorithms work within the scope of a single function; inter-procedural analysis is usually fundamentally more difficult, and always more computationally expensive than intra-procedural analysis, because of the increased scope. A function call inhibits the precision of analyses and is a barrier to code motion because the caller sees the callee as a "black box" with unknown effect.

Upon inlining, the formal parameters in the callee are replaced by the actual parameters used in the call. Thus, constants passed as parameters can be propagated within the inlined code and any expressions using these values can be simplified, as demonstrated by the transformation from Figure 6.1(b) to Figure 6.1(c). When a branch test simplifies to a constant expression, the branch outcome is known at compile time. The branch is removed, along with the now-dead code for the not-taken branch. Similarly, constant propagation can limit the possible target of an indirect call to a single function. In that case, the indirect call can be converted to a direct call, as illustrated by the conversion of $*$Step to fibStep in Figure 6.1(c).

Pointers frequently impede the ability of a compiler to transform code because the exact memory location(s) accessed through a pointer are usually obscured or unknowable at compile time. Alias[2] analysis attempts to prove which memory locations a pointer must never, must always, or may possibly point to, and the sets of pointers that may point to the same location(s). If a local variable is passed to a call by a pointer, an alias is created. Inlining the call allows the parameter to remain a local variable referring to a non-ambiguous memory location. Thus, the alias analysis within the inlined code is more precise than it is in the callee.

6.1.3 Estimating Inlining Benefit

As discussed above, inlining a call directly reduces the number of executed instructions. Both the `LLVM` and FDI inliners estimate these savings as 10 instructions for the call, plus one instruction per parameter. The analyses used to determine the additional benefits of inlining a call site do so by estimating the number of instructions in the inlined code that will be eliminated after inlining. Therefore, the estimate is for the static number of instructions, or code size, rather than the dynamic number of instructions executed at run time. Code size reduction is estimated for constant parameters and stack-allocated arrays passed by pointer, on a per-parameter basis.

The same straight-forward analysis is used by both inliners to estimate the impact of each parameter. When analyzing a parameter, the `LLVM` inliner counts the number of instructions eliminated, and adds bonus "instruction-equivalent" savings for beneficial situations that do not directly eliminate instructions. Instead of

[2]Multiple names (*e.g.*, pointer variables) for a single memory location are *aliases*.

adding bonuses to a grand total during the analysis, the FDI inliner maintains separate counts for each indirectly-beneficial situation.

The analysis for stack-allocated arrays passed by pointer is essentially the same as the constant-parameter analysis. The base address of the array becomes a constant that can be propagated, while the array data is known to reside on the stack. This additional information enables transformations that treat the array locations as scalar values[3]. For example, after loop unrolling and scalar promotion, the code in Figure 6.1(d) is transformed into the code in Figure 6.1(e). Constant propagation in that final version of the code allows the compiler to replace the entire initial computation of fib5 by the constant value 13. However, the analysis does not consider the potential impact of such transformations, but only count instructions directly eliminated because the array's base address is a constant.

The impact of a constant parameter is determined for each formal parameter of each function in advance, by assuming that it takes a constant, but unknown, value. LLVM's IR uses a *single static assignment* (SSA) representation, where each `value` produced is defined exactly once. Data flow is represented by directly linking each `value`[4] to the instructions that use it. Each IR instruction u using parameter p is examined, assuming that p takes a constant value. If u is neither a branch nor a call instruction, then if all of u's inputs are constants, it can be eliminated by constant folding. When p is the only non-constant input to u, u is counted as an eliminated instruction, and the analysis continues recursively to the uses of u.

When u is an indirect call for which the callee becomes constant, the call is resolved to a direct call. LLVM awards a large bonus for this conversion; FDI counts the conversion separately from other eliminated instructions.

If u is a branch or switch instruction who's test condition becomes constant, the control-flow outcome of the branch is restricted to a single possibility. However, since the value of p is unknown, the actual outcome of the branch is also unknown. Each of the n possible outcomes are considered equally likely. The average size of the blocks corresponding to the possible outcomes, $\overline{s}$, is determined. Only one outcome is possible, thus $\overline{s}(n-1)$ instructions are expected to be eliminated. LLVM does not include the (now-unconditional) branch instruction in the count of eliminated instructions. FDI counts eliminated branch instructions separately from other eliminated instructions. The analysis does not continue to subsequent successor blocks.

In the evaluation of the inlining benefit of a particular call site, the benefit precomputed for the callee's formal parameters is retrieved, as appropriate, for each actual parameter that is a constant or a pointer to a stack-allocated array. The impact of each parameter is accumulated to estimate the total code-size reduction enabled by inlining the call site. The LLVM inliner simply adds these values together. FDI adds the counts for each category it measures and then computes a weighted sum of those values, as explained in Section 6.3.

[3] *e.g.*, scalar promotion, loop-invariant code motion

[4] Formal parameters and IR instructions are both `values`.

6.1.4 Costs of Inlining

Inlining non-profitable call sites can indirectly produce negative effects. The increased scope provided for analysis by inlining also increases the costs of these analyses. Most algorithms used by compilers have super-linear time complexity. Extremely large procedures may take excessively long to analyse; some compilers will abort an analysis that takes too long. Furthermore, a program must be loaded into memory from disk before it can be executed. A larger executable file size increases a program's start-up time. Finally, developers eschew unnecessarily large program binaries because of the costs associated with the storage and transmission of large files for both the developer and their clients. Therefore, inlining that does not improve performance should be avoided.

Both inliners discussed in this chapter use estimated code growth as the cost metric for inlining. The basic technique for code-growth estimation is to count the number of instructions in the callee. Each inliner then adjusts this base cost as explained in their respective sections.

6.1.5 Inlining-Invariant Program Characteristics

While inlining a call causes a large change in the caller's code, it has a minimal direct impact of the use of memory system resources at run time. Ignoring the subsequent simplifications the inlining enables, inlining proper has no appreciable impact on register use, or data or instruction cache efficiency. Regardless of inlining, the same dynamic sequence of instructions must process the same data in the same order to produce the same deterministic program result.

Inlining should have negligible impact on register spills. The additional variables introduced into the caller by inlining place additional demands on the register allocator, and may increase the number of register spills introduced into the caller. However, without inlining, the register values in the caller must be preserved across the call. The architecture-specific calling convention requires the caller and/or callee to save register values on the stack before execution in the callee begins, so that register values needed by the caller are not overwritten by the callee. The original register values must be restored before resuming execution in the caller. Thus, inlining merely shifts the responsibility for register management from the calling convention to the register allocator.

Similarly, inlining does not change the data memory accesses of a program. Whether in the caller or the callee, the same loads and stores, in the same order, are required for correct computation. Subsequent transformations may reorder independent memory accesses to better hide cache latency, or eliminate unnecessary accesses altogether, but this is not a direct consequence of inlining. Thus, data cache accesses do not change with inlining, and nor does the cache miss rate.

Likewise, the same instruction sequence, minus the call and return instructions, is executed regardless of inlining. Given a fully-associative cache with sufficiently small lines, instruction cache activity will be identical in either case. However,

caches are set associative, and have lines that hold many instructions. Therefore, instruction cache efficiency can change if frequently-executed instructions are more often adjacent to infrequently-executed instructions in the inlined code, and this adjacency is contained within a cache line. A block placement algorithm linearizes the basic blocks in a function in order to place it in the linear memory address space. Block placement attempts to follow a block in the linear sequence by its most likely successor. Given the same block placement algorithm for both the inlined and non-inlined versions of the code, there is no reason to believe that inlining will cause the algorithm to be less effective at separating hot and cold code. Furthermore, even if hot code more frequently shares a cache line with cold code, this situation will only cause a small increase in the number of cold misses in the instruction cache. Unless the hot code no longer fits within the instruction cache, the steady-state miss rate will not change. On the other hand, the potentially reduced total code size in the case of inlining can only increase the effectiveness of instruction caching.

6.2 Static inlining in LLVM

The default, or *static*, inliner in LLVM is focused on eliminating calls to small functions. It does not use any feedback information, nor does it make any attempt to estimate the execution frequencies of call sites. The CG is decomposed into strongly-connected components (SCC). Since the edges in a CG are function calls, an SCC in the call graph represents a recursive cycle. Each component is processed as a unit, and each function is processed exactly once during the entire inlining pass.

Inlining decisions for the call sites within a function are considered using a worklist. The worklist initially contains all call sites in the function, with calls to other functions in the same SCC moved to the end of the list. New call sites created by inlining are added to the end of the list. Thus, the inliner usually requires a single pass over the worklist. However, if the target of an indirect call is resolved, inlining becomes possible at that call site and it must be re-considered in an additional pass over the worklist.

The decision of whether or not to inline a call is made by comparing the cost of inlining to a threshold. Costs are measured in terms of code size. The base cost of inlining is determined for a callee independently of the call site. This base cost is computed as the number of instructions in the callee, plus a penalty for each call site in the callee. When making the inlining decision for a source call site, the base cost is modified by several factors. First, the cost is reduced by a large constant if the source is the only call to the callee; in that case, the original copy of the callee will be dead code after inlining, and will be deleted. The cost is also reduced using the method described in Section 6.1.3 if any of the actual parameters at source are constants and/or arrays allocated on the caller's stack. If the adjusted inlining cost is greater than the threshold, the source call site will not be inlined.

However, if the cost is less than the threshold, an additional check is performed to determine if inlining the source will prevent the caller from itself being inlined

at other *outer* call sites[5]. The *initial cost* of each outer call site is evaluated to see if it would be inlined under the condition that the source call site is *not* inlined. The outer call site is also evaluated assuming that the source call site *is* inlined. This *increased cost* of the outer call site is (roughly) the sum of its initial cost plus the cost of inlining source. If the outer call site would be inlined using the initial cost but would not be inlined using the increased cost, its initial cost is added to a *total outer cost*. After all outer call sites have been evaluated, the total outer cost is compared to the inlining cost of source. Source is inlined only if there are no outer call sites or its inlining cost is less than the total outer cost.

6.3 A New CP-Driven Feedback-Directed Inliner for `LLVM`

The feedback-directed inlining (FDI) evaluated in this work is fundamentally different than the existing static inliner in `LLVM`. The static inliner inlines small calls to remove call overhead with minimal increases in code size. FDI attempts to minimize the dynamic number of instructions executed by the program by inlining the most frequent calls. While the static inliner considers call sites on a function-by-function basis, FDI considers the set of inlining opportunities present at global scope in the current state of the program.

6.3.1 Worklist Algorithm

Algorithm 2 presents an outline of the worklist algorithm used by FDI. The algorithm uses several data structures:

candidates The worklist is a sorted list of candidates. A call site is an inlining candidate if it is a direct call, and if the callee does not contain a `setjump` nor has any previous attempt to inline the callee failed. Furthermore, the call site must have executed at least once during profiling.

ignored A list of call sites that are not inlining candidates. This list is maintained to enable correct and efficient bookkeeping, and to allow any copies of these call sites created by inlining their caller to be immediately ignored.

callers A mapping from functions to the call sites that call them. This map allows for the re-scoring of call sites on the event that a call is inlined into their callee. That inlining will change the callee's size, and may change the expected simplifications possible if the callee is inlined.

inlineResult A structure returned by inliner that provides summary information regarding the transformation. In particular, it indicates if the attempted inlining

[5]Some details are omitted from this description

Algorithm 2: FDI worklist

```
   input  : Module M: Whole-program IR
   input  : File cpFile: Combined profile
   Data: List<call site> candidates, ignored
   Data: Map<Function → List<call site> > callers
 1 initialize(M, cpFile);
 2 budget = computeCodeGrowthBudget();
 3 candidates.sort();
 4 while budget > 0 AND NOT candidates.empty do
 5     source = candidates.popBest();
 6     if source.score ≤ 0
 7         break;
 8     endif
 9     if source.callee.cannotInline
10         ignored.add(source);
11         continue;
12     endif
13     if source.expectedCodeGrowth > budget
14         ignored.add(source);
15         continue;
16     endif
       // Try to inline the candidate...
17     inlineResult = LLVM.inlineIfPossible(source);
18     if inlineResult.failed
19         source.callee.setCannotInline();
20         ignored.add(source);
21         continue;
22     endif
       // Inlining succeeded
23     budget -= inlineResults.codegrowth;
24     callers[source.getCallee].delete(source);
25     for caller ∈ callers[source.caller] do
26         caller.calcScore();
27     end
28     for i ← 1 to inlineResults.numInlinedCalls do
29         target = inlineResults.inlinedCall[i];
30         original = inlineResults.originalCall[i];
31         callers[target.getCallee].insert(target);
32         if ignored.contains(original) > 0
33             target.histogram = 0;
34             ignored.add(target);
35         else
36             target.histogram = source.histogram × original.histogram;
37             target.calcScore();
38             candidates.insert(target);
39         endif
40     end
41 end
```

failed. FDI enhances the default LLVM structure with co-indexed lists identifying the new call sites created in the caller by inlining, and their originating call sites in the callee. This information is required so that profile information can be estimated for the new call sites.

At the start of FDI, the CProf is read in, and the histograms are associated with the appropriate call sites (line 1). Every call site is inserted into the callers list of their callee. During initialization, each call site is evaluated, and added to either the candidates or ignored list, as appropriate. When a call site is rejected for inlining, it is immediately and permanently moved from the list of candidates to the ignore list. Transformations such as constant propagation or alias analysis can resolve the callee of an indirect call to a single possibility, thereby making it a direct call. However, if the call is indirect when the call site is first discovered by the inliner, it is placed on the ignore list in spite of the possibility of future inlining resolving the call. Calls to libraries and compiler built-in functions are also immediately ignored because they cannot be inlined.

The initial size of each function, in LLVM IR instructions, is computed; the total program size is used to determine the code growth budget (line 2). The computation of the code-growth budget is explained in Section 6.3.2. Each successful function inlining reduces the code growth budget by the resulting increase in code size (line 23). The worklist algorithm iterates until either the budget is expended, or no candidates remain.

In each iteration of the algorithm, the best candidate (largest score) is removed from the candidates list as a potential inlining source call site (line 5). Scoring candidates is explained in detail in Section 6.3.3, but the basic idea is that the score represents the expected benefit of inlining the candidate. Higher scores are better, while a zero score represents no expected benefit. Before inlining is attempted for the source, several checks are performed. If the source's score is not positive, then no more candidates exist with positive scores (line 6). Inlining call sites with no expected benefit is counter-productive, so the algorithm terminates. If it has already been determined that the callee of the source cannot be inlined, attempting to inline this candidate is futile (line 9). Finally, if the code growth incurred by inlining the source is expected to exceed the remaining code-growth budget, the candidate is skipped (line 13). However, other candidates with lower scores but also smaller size may still exist in the candidates list.

If all the initial checks succeed, an attempt is made to inline the source call site (line 17). If inlining fails, the callee is marked "cannotInline" and the source is ignored. Successful inlining leads to several subsequent tasks. First, the remaining code-growth budget is reduced by the actual increase in code size caused by inlining (line 23). If the inlined callee becomes dead code, the size of the callee is deducted from the increase in size of the caller. In such cases, the net code growth due to inlining is frequently *negative* due to simplification of the inlined code, and the remaining code-growth budget *increases*. Next, since the source no longer exists, it is removed from the callee's list of callers (line 24). The caller has changed due

to inlining, thus every candidate that calls the caller must have its score reevaluated (line 25).

Finally, inlining may copy original call sites from the callee to target call sites in the caller; they must be added to their callee's callers list. These target call sites may be new inlining opportunities. If the original call site was ignored, the target should be ignored as well (line 32). Otherwise, a combined profile for the target must be estimated. This estimation is in fact denormalization; the original call site has effectively been moved "above" the source call site and into the caller, just as CFG monitors were moved above branches in Chapter 5.3.2. The histogram for the target is thus the product of the histograms from the source and original call sites (line 36). The call site profiling is not context sensitive, and therefore assumes that the distributions of the original and source call sites are independent. This independence in turn allows the distribution of the target call site to be estimated by simply taking the product of the source and original histograms.

6.3.2 Code-Growth Budget

Section 6.1 identified that excessive increases in code size should be avoided. *Code growth* is the ratio of the size of the transformed program to the original size of the program. Size can be measured in many ways; the selected method is highly dependent on the context in which the measurement is done. The "real" measure of code growth comes from comparing the final executables. However, it is impossible to know these file sizes in the middle of compilation. In addition, the exact impact of a transformation (*e.g.*, inlining) on code size can only be estimated. More significantly, the actions of one transformation can change the actions taken by subsequent transformations. Therefore, both code size and code growth are estimated at inlining time by counting `LLVM` IR instructions.

As shown in Algorithm 2, inlining is controlled by a code-growth budget. Each successful inlining operation reduces the remaining budget by the increase in size of the caller function. If the callee becomes dead after inlining, *i.e.*, there are no other direct calls to the callee, and the callee cannot be called indirectly (is not address-taken), the callee code will be removed by inter-procedural dead-code elimination. In this case, the inlining budget is also credited with the size of the callee.

Zhao showed that smaller programs benefit from proportionally larger inlining budgets than larger programs [104]. He defines a sequence of code-size thresholds that determine the allowable code-growth. This work takes an alternative approach of defining a continuous function to calculate the code-growth budget. Consider the requirements of such a function. First, it must be a decreasing function. More specifically, it must be a concave function that has a slope that is initially negative and increases to 0: small programs can have a very large budget, but the budget quickly reduces to a small value as program size increases. The $\frac{1}{x}$ family is a good initial candidate for the code-growth function. However, these functions reduce too quickly initially. Therefore, a convex function is used in the denominator. Experi-

mentation suggests that a function of the form $\frac{1}{\sqrt{x}}$ has the desired shape. Given the initial program size, s, the final code-growth budget function, $\mathsf{Budget}(\mathrm{s})$, computes the proportional growth appropriate for a program of the given size. The allowed total code size after inlining is thus $s(1 + \mathsf{Budget}(s))$. Budget is fine-tuned with the following parameters:

g_{max}	The maximum limit on code growth.
g_{min}	The minimum limit on allowed code growth.
s_{max}	The maximum program size; the point where all larger programs simply use g_{min} as the code growth limit.
s_{min}	The minimum program size; the point where all smaller programs simply use g_{max} as the code-growth limit.

A scaling factor, denoted A, allows the budget computed when the program size is s_{min} or s_{max} to equal g_{max} or g_{min} respectively, keeping Budget continuous at the endpoints of the specified size range. Thus, A is defined by the g and s parameters:

$$A = g_{max}\left(\sqrt{s_{min}} - \sqrt{s_{max}} + g_{min}\right)$$

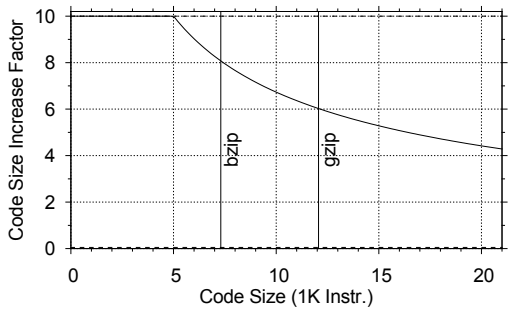
The complete code-growth function is:

$$\mathsf{Budget}(s) = \begin{cases} g_{max}, & s <= s_{min} \\ A\left(\frac{1}{\sqrt{s}} - \frac{1}{\sqrt{s_{max}}} + g_{min}\right), & s_{min} < s < s_{max} \\ g_{min}, & s >= s_{max} \end{cases}$$

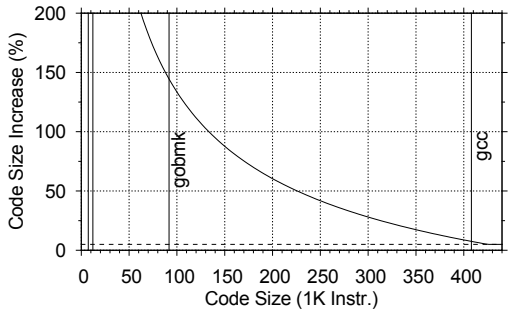
This implementation sets the program size range $[s_{min}, s_{max}]$ to [5,000, 425,000] `LLVM` IR instructions, and the code-growth range $[g_{min}, g_{max}]$ to [0.05, 10.0]. Figure 6.2(b) displays the resulting code-growth function, with dashed lines showing g_{min} and g_{max}, and vertical lines identifying benchmark sizes at the start of inlining. The total code size obtained if the code-growth budget is consumed exactly is presented in Figure 6.2(c).

6.3.3 Candidate Scoring

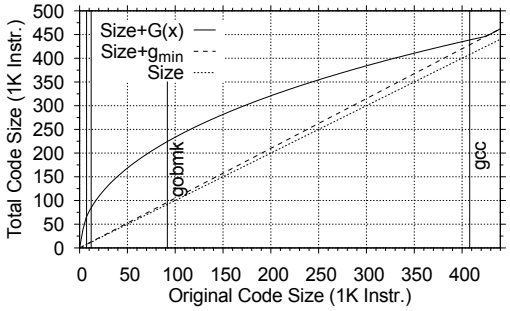
The `LLVM` inliner makes inlining decisions at each call site by comparing the expected code growth to a fixed threshold. FDI takes a more directly execution-time-oriented approach to inlining and attempts to achieve the greatest reduction in executed instructions for the least amount of code growth. Therefore, FDI breaks the evaluation of a call site into three components: the expected inlining benefit, the expected code growth, and execution frequency of the call site. Given a call site, CS, the inlining candidate scoring function, $\mathsf{Score}(\mathrm{CS})$, combines these three elements so that Algorithm 2 can select the best (highest score) candidates for



(a) Code-growth budget: small programs



(b) Code-growth budget: large programs



(c) Total code size if inlining budget is exhausted

Figure 6.2: Allowable code-growth budget for FDI inlining in terms of `LLVM` IR instructions. The initial size of `bzip2`, `gzip`, `gobmk`, and `gcc` are indicated with vertical lines.

inlining first. CP provides a rich characterization of execution frequency. Making use of that information is described in detail in Section 6.3.4; for now, let $\mathsf{Reward}(\mathsf{Benefit}(\mathrm{CS}), R_{\mathrm{CS}})$ represent some function of the estimated (execution-frequency independent) inlining benefit at call site CS and R_{CS}, that call-site's CP monitor. Given the benefit function $\mathsf{Benefit}(\mathrm{CS})$ and a cost function $\mathsf{Cost}(\mathrm{CS})$ described in this section, an inlining candidate's Score is conceptually computed:

$$\mathsf{Score}(\mathrm{CS}) = \frac{\mathsf{Reward}(\mathsf{Benefit}(\mathrm{CS}), R_{\mathrm{CS}})}{\mathsf{Cost}(\mathrm{CS})}$$

The exact definition of $\mathsf{Score}(\mathrm{CS})$ is given at the end of this section. Both Cost and $\mathsf{Benefit}$ consider the code simplification potential of a call site. Given a call site, CS, and its set of actual parameters, $\mathcal{A}$, the following terms contribute to those functions:

$\mathcal{A}$	The number of parameters passed at CS.
iConst	The **total** estimated number of non-branch instructions eliminated, for each constant parameter $a \in \mathcal{A}$.
$\mathsf{brConst}$	The **total** estimated number of branch instructions eliminated, for each constant parameter $a \in \mathcal{A}$.

iCall	The **total** estimated number of indirect call targets resolved over each constant actual parameter at CS.
iArray	The **total** estimated number of instructions eliminated over each local stack-allocated array parameter at CS.

The true code-size impact of multiple constant parameters is unlikely to be the additive combination of the impact of each constant parameter considered in isolation. However, the straight-forward implementation of this analysis does not propagate the impact of branch removal beyond the immediate successor blocks of a removed branch. Thus, the potential overlap between the sets of instructions eliminated for multiple constant parameters is negligible. Therefore, while the estimated amount of code reduction for a single constant parameter may be significantly under-estimated, the impact of multiple constant parameters is essentially additive. Improvements to the constant-parameter analysis are suggested in Section 6.3.5.

Benefit

The inlining benefit function $\mathsf{Benefit}(\mathrm{CS})$ predicts the benefit of inlining call site CS. This benefit is expressed as the estimated number of instructions (or equivalent) saved at execution time for *each* execution of R; Reward will use the CProf to incorporate the execution frequency of R_{CS} into the final score.

When estimating the number of instructions saved due to a constant parameter, FDI counts the number of non-branch instructions eliminated separately from the number of branches eliminated. Even though they will not become inlining candidates, FDI also counts the number of indirect calls resolved to direct calls. The increased analysability of the function may benefit subsequent transformations.

The underlying analysis for constant and local-array parameters is shared by FDI and the `LLVM` inliner. However, unlike the `LLVM` inliner, FDI does not assume that the benefits of constant arguments are additive. Instead, the maximum savings for any individual constant parameter is used as the estimated instruction savings. The number of indirect calls resolved to direct calls is additive, because the resolved target must, by definition, be a single constant value; a target resolved using one constant parameter must be independent of all other parameters.

In addition, parameter analysis in FDI is done lazily instead of eagerly. Inlining into a function invalidates any pre-computed parameter benefits in either approach, while the estimates are not required unless scoring a candidate call site that actually passes a constant or local array. There is no point calculating the impact of a constant parameter if no call sites have a constant for that parameter.

Unlike the static inliner, FDI does not penalize a call site when additional call sites exist in the callee. $\mathsf{Benefit}(\mathrm{CS})$ is computed:

$$\mathsf{Benefit}(\mathrm{CS}) = 10 + |\mathcal{A}| + \mathsf{iConst} + \mathsf{iArray} + (4 \times \mathsf{brConst}) + (2 \times \mathsf{iCall})$$

Cost

The code-growth budget limits the amount of inlining that FDI can do; when the total, real, increase in code size exceeds the budget, FDI terminates. Thus, the notion of cost in this section does not pertain to a negative impact of inlining on execution time, but rather is used to normalize the estimated Benefit of inlining according to the corresponding budget consumption. Consider, for instance, a single-instruction accessor function. The Benefit of inlining such a function is only the default 10 instructions saved for eliminating the call. However, the inlined instruction can likely be eliminated entirely; once the identity of the returned value is no longer obscured by the function call, uses of the returned value can access it directly. Thus, the Benefit of inlining the accessors, *per unit cost*, should be large.

The code-growth estimate for inlining CS is initially the size of the callee, $\mathsf{Size}(\mathrm{CS})$. Similar to the computation of Benefit, one instruction is subtracted from the callee's size for each instruction (branch or otherwise) estimated saved due to constant parameters or local-array parameters. If the callee has a single basic block, the code in this block is inserted directly into the caller's block in place of CS. Consequently, it is much more likely that the expressions in the callee can be folded into expression in the caller, thus eliminating those instructions. If the callee contains no conditional branches, only a single block will be inlined[6], and therefore the code-growth estimate is reduced by 5 instructions. Conditional branches are counted while the size of a function is calculated. Cost is thus defined:

$$\mathsf{Cost}(\mathrm{CS}) = \mathsf{Size}(\mathrm{CS}) - \mathsf{iConst} - \mathsf{brConst} - \mathsf{iArray} - (\mathrm{CS.branches}\ ?\ 0 : 5)$$

The above code-growth estimate may be negative for very small callees. Using a negative Cost value for normalization in the definition of Score given at the beginning of this section would result in a negative score (an unprofitable candidate), when it should instead indicate a highly profitable candidate for inlining. As well, a cost of 0 results in an undefined score. These problems are solved by interpreting a negative Cost [7] as a bonus to benefit. Thus, [8] a negative Cost is subtracted from Benefit, making it larger:

$$\mathsf{Score}(\mathrm{CS}) = \begin{cases} \dfrac{\mathsf{Reward}(\mathsf{Benefit}(\mathrm{CS}), R_{\mathrm{CS}})}{|\mathsf{Cost}(\mathrm{CS})|}, & \mathsf{Cost}(\mathrm{CS}) > 0 \\ \mathsf{Reward}(\mathsf{Benefit}(\mathrm{CS}), R_{\mathrm{CS}}), & \mathsf{Cost}(\mathrm{CS}) = 0 \\ \mathsf{Reward}(\mathsf{Benefit}(\mathrm{CS}), R_{\mathrm{CS}}) - \mathsf{Cost}(\mathrm{CS}), & \mathsf{Cost}(\mathrm{CS}) < 0 \end{cases}$$

[6] Function entry and exit blocks are eliminated by inlining.

[7] Intended method: as a multiplicative inverse rather than an additive inverse

[8] Intended method: Score is computed by dividing the Reward by positive Costs, but multiplying by the absolute value of negative Costs

6.3.4 Frequency Estimation with Combined Profiles

The execution frequency of a call site is a direct indicator of the *exploitability* of the inlining opportunity presented by that call site. A *reward function* combines this notion of exploitability with the expected benefit of inlining. The previous section presented the abstract reward function, $\mathsf{Reward}(\mathsf{Benefit}(\mathrm{CS}), R_{\mathrm{CS}})$, and detailed the frequency-independent calculation of Benefit. FDI provides several increasingly-sophisticated categories of reward functions that combine Benefit with the exploitability indicated by R's frequency distribution. The reward function desired for a particular compilation is selected, by name, through a command-line parameter.

Frequency estimation from a raw profile is straight-forward: the value in the profile *is* the frequency estimation. A CProf provides much richer data, and rather than reasoning about execution frequency values, inlining decisions must consider execution frequency distributions. The combined call profiling used to inform FDI uses the hierarchical-normalization approach suggested in Chapter 5.3.4: the execution frequency of each call site is normalized with respect to the invocation frequency of the caller. In order to improve execution time for all inputs in a program's workload, FDI should likely consider inlining as a multi-objective optimization problem and use an appropriate reward function. This section describes the framework provided by FDI for the construction of reward functions. Specific reward functions are proposed and evaluated in Chapter 7.

Static Rewards

Static reward functions are comparable to the default `LLVM` inliner, and do not use any profile information. FDI provides the static Benefit reward, which simply returns the result of $\mathsf{Benefit}(\mathrm{CS})$. Using this reward function allows the FDI inlining algorithm presented in Algorithm 2 to be directly compared to the default inliner. Any performance difference between these two inliners is not due to the use of FDO or CP but rather to the the use of a code-growth budget instead of an allowable callee-size threshold; a global, sorted, worklist; and/or the small differences in the computation of the impact of constant arguments.

Simple Point Rewards

Simple Point (SP) reward functions are comparable to traditional FDO approaches because they estimate call-site execution frequency using point summary statistics that can be computed exactly by simple incremental algorithms. The histograms described in Chapter 5 provide three such metrics: the minimum (non-zero) observed value, the maximum observed value, and the average of all observed values. FDI computes the min, max, and mean rewards by simply multiplying Benefit by

the corresponding statistic:

$$\begin{aligned}
\mathsf{min}(\mathrm{CS}) &= \mathsf{Benefit}(\mathrm{CS}) \times H_{\mathrm{CS}}.\mathrm{min} \times H_{\mathrm{CS}}.\mathrm{coverage} \\
\mathsf{max}(\mathrm{CS}) &= \mathsf{Benefit}(\mathrm{CS}) \times H_{\mathrm{CS}}.\mathrm{max} \\
\mathsf{mean}(\mathrm{CS}) &= \mathsf{Benefit}(\mathrm{CS}) \times H_{\mathrm{CS}}.\mathrm{mean} \times H_{\mathrm{CS}}.\mathrm{coverage}
\end{aligned}$$

Weighting Benefit by coverage in the min and mean rewards is appropriate because the minimum and average benefit of inlining should also consider how often (in terms of program runs) this reward is available. However, max is estimating the maximum benefit in *any* run, so the proportion of runs where the call site is executed is not relevant.

Quantile Point Rewards

Quantile Point (QP) reward functions use the estimated frequency of a monitor corresponding to one or more quantile values to compute reward, and thus cannot be computed without a distribution model. FDI accepts, as command line parameters, a list of quantile points, Q. For each point $q \in Q$, $\mathsf{QPPart}(\mathrm{CS}, q)$ computes the product of Benefit and the estimated frequency value for $\mathrm{quantile}(q)$. These products can be combined linearly, or, to support multi-objective evaluation, non-linearly:

$$\begin{aligned}
\mathsf{QPPart}(\mathrm{CS}, q) &= \mathsf{Benefit}(\mathrm{CS}) \times H_{\mathrm{CS}}.\mathrm{quantile}(q) \\
\mathsf{QPLinear}(\mathrm{CS}) &= \sum_{q \in Q} \mathsf{QPPart}(\mathrm{CS}, q) \\
\mathsf{QPSqrt}(\mathrm{CS}) &= \sum_{q \in Q} \sqrt{\mathsf{QPPart}(\mathrm{CS}, q)}
\end{aligned}$$

When $Q = \{0.5\}$, the QP reward acts like the mean SP reward, but weight Benefit by the estimated median value of the distribution rather than the average. If $Q = \{0.25, 0.75\}$, the quantile point metrics use the value of the first and third quartiles. QPLinear weights Benefit by the sum of these values; a lower frequency at $q = 0.25$ can be directly compensated for by a higher frequency at $q = 0.75$. On the other hand, QPSqrt takes the square root before summing the weighted Benefits, and thus balances the dual objectives of greater benefit at both the $q = 0.25$ and $q = 0.75$ quantile points.

Neither the QP nor QR quantile-based reward functions weight by coverage. The selection of multiple quantile points for consideration is a case-based analysis; selecting one low and one high quantile point could be seen as balancing a pessimistic estimation against an optimistic estimation. Weighting by coverage may be appropriate for a pessimistic estimation, but is not appropriate for an optimistic estimation (*e.g.*, the max SP reward)[9].

[9]Weighting only some quantiles by coverage is a problem not dealt with in this work, but is a straight-forward extension of the current implementation; weighting all quantiles by coverage can be trivially implemented.

Quantile Range Rewards

Quantile Range (QR) reward functions compute reward based on the histogram weight contained in one or more histogram segments (ranges of monitor values), where the endpoints of these segments are determined by pairs of quantile points. The QR rewards use the same command-line list, Q, of quantile points as the QP rewards, but group the q values into a sequence of $\langle qmin, qmax \rangle$ pairs. Recall from Chapter 5 that the histogram query $\mathrm{applyOnQuantile}(F(w,v), qmin, qmax)$ applies the function F to the $\langle weight, value \rangle$ pairs of the set-of-impulses interpretation of a histogram between the specified quantile points. The QR rewards define F to be a simple product, thus $\mathrm{applyOnRange}$ computes the weighted average value of the monitor in the specified range. $\mathsf{QRPart}(\mathrm{CS}, \langle q_{\mathrm{LB}}, q_{\mathrm{UB}} \rangle)$ weights Benefit by the average monitor value for each segment. As with the QP rewards, the QR rewards can be computed using either a linear or non-linear combination of the histogram-weighted Benefit of each segment:

$$
\begin{aligned}
F(w,v) &= w \times v \\
\mathsf{QRPart}(\mathrm{CS}, \langle q_{\mathrm{LB}}, q_{\mathrm{UB}} \rangle) &= \mathsf{Benefit}(\mathrm{CS}) \times H_{\mathrm{CS}}.\mathrm{applyOnQuantile}(F, q_{\mathrm{LB}}, q_{\mathrm{UB}}) \\
\mathsf{QRLinear}(\mathrm{CS}) &= \sum_{\langle q_{\mathrm{LB}}, q_{\mathrm{UB}} \rangle \in Q} \mathsf{QRPart}(\mathrm{CS}, \langle q_{\mathrm{LB}}, q_{\mathrm{UB}} \rangle) \\
\mathsf{QRSqrt}(\mathrm{CS}) &= \sum_{\langle q_{\mathrm{LB}}, q_{\mathrm{UB}} \rangle \in Q} \sqrt{\mathsf{QRPart}(\mathrm{CS}, \langle q_{\mathrm{LB}}, q_{\mathrm{UB}} \rangle)}
\end{aligned}
$$

6.3.5 Potential Improvements

The existing parameter analysis could be improved for FDI in several ways. The existing analysis tries to estimate the impact of parameter attributes on the amount of code inlined into the caller. While code-size impact is important to compute an accurate Cost, it provides a poor estimate of inlining's execution-time impact, which is needed to compute Benefit.

One improvement would be the use of a more sophisticated dead-code analysis when a branch is eliminated. Eliminating a branch can potentially cause an entire sub-graph of the CFG to become dead, not just the blocks that are the immediate successors of the branch. Consider, for instance, the common practice of early-exit tests that skip nearly the entire function when little or no work needs to be done. The current analysis estimates that the average size of the two immediate successor blocks of the test will be eliminated, regardless of the parameter value. A more accurate analysis would distinguish between the case where most of the function is eliminated because the early exit is taken, versus the case where the exit is not taken and only the early `return` and the branch are saved. However, in this case the benefit of multiple constant parameters would not be additive. One solution would be to compute the set of eliminated basic blocks for each individual

parameter, and then compute the combined impact using the union of these sets for all constant parameters. Furthermore, each analysis result would be tied to a specific parameter value. However, constant parameters tend to take values from a small set (*e.g.*, -1, 0, 1, 2; enumerated types) or widely-used compile-time constants (*e.g.*, `#define BLOCKSIZE 128`), so caching the analysis results should be very effective.

Additionally, FDI could significantly improve the estimation of the reduction in executed instructions by using CFG profiling to provide an execution-frequency estimate for eliminated instructions. In fact, even the current straight-forward analysis could be greatly enhanced; constant-folding an instruction in a hot loop would be estimated to provides a large reduction in executed instructions, while the same opportunity within an unexecuted error-handling path would be expected to have no impact on execution time.

6.4 Conclusion

This chapter presents the new FDI inlining framework, a versatile set of inliners for `LLVM` guided by combined profile information. Each family of inliners uses a parameterized Reward function to query the combined profile for points or ranges of expected execution frequencies in terms of distribution quantiles. FDI differs from the default inliner not only in its use of profile information to include expected execution frequencies in the estimated benefit of inlining a call site, but also because it employs a worklist to select the most profitable inlining opportunities first, stopping only when a code-growth budget is exceeded or when no profitable opportunities remain. In contrast, the default inliner investigates each call site once during a bottom-up traversal of the call graph, selecting call sites for inlining if the estimated benefit of inlining exceeds a threshold. Therefore, the FDI framework allows flexibility in the order in which inlining is performed. While the default inliner includes heuristics to suppress the inlining of a call site if this inlining will likely inhibit a more profitable inlining opportunity later in the call-graph traversal, the FDI worklist automatically prioritizes the most profitable opportunities regardless of where they occur in the program. A thorough evaluation of both the default inliner and FDI inliners follows in Chapter 7.

Chapter 7

Evaluation: CP in the LLVM Compiler

Many aspect of FDO for ahead-of-time compilation have been explored in the previous chapters. Chapter 3 proposed a cross-validation evaluation methodology for FDO. Chapter 4 presented evidence of the often large variations in FDO performance resulting from the choice of a different training input in the traditional single-profile approach to FDO. Chapter 5 introduced the combined-profiling technique that provides an FDO compiler with a distribution model characterizing the inter-run variations in program behavior. Chapter 6 explained FDI, an inlining framework for LLVM informed by combined profiles.

This chapter brings the elements of the preceding chapters together by performing a proper cross-validated evaluation of FDI and static inliners. Afforded with the parametrized families of inliners offered by FDI, this first evaluation of CP-driven FDO includes several instantiations of inliners from each family. Section 7.1 explains how FDI is integrated into the LLVM compilation process. The details of the experimental methodology used for this study, including concrete instantiations of FDI's quantile-based reward functions, are presented in Section 7.2. Results from a case-study evaluation of FDI using the real, non-benchmark versions of bzip2 and gzip, as well as the gcc and gobmk benchmarks from SPEC CPU 2006, follow in Section 7.3.

7.1 LLVM Implementation

Chapter 6 described FDI in insolation. However, FDI must be integrated into LLVM in order to be used. For this study, two issues must be resolved: the position of FDI in the sequence of code transformations, and the design of a build process that allows all inliners, including the default inliner, to be applied in a consistent and comparable fashion. Section 7.1.2 presents a high-level overview of the compilation process used to generate each version of the program used in the evaluation of FDI. Section 7.1.3 then details how FDI is incorporated into the LLVM transformation

sequence. The details of the evaluation of each inlining alternative, including the specific program versions created, the way profiles are used with FDO, and the concrete instantiations of FDI reward functions are left to Section 7.2.

7.1.1 Loading Combined Profiles

Combined profiling is available to `LLVM` transformations in two ways. An external tool named `llvm-cprof` creates a CProf using histograms with the specified number of bins. One or more profiles, in any combination of raw or combined, are given as command-line parameters. However, because CP applies HN to raw profiles, the whole-program bitcode file from which the instrumented executable was created must be available and specified as a required command-line parameter. Alternatively, FDI allows the set of raw and/or combined call profiles it should use to be specified as parameters in the compilation command-line that invokes FDI.

Both approaches use a CProf factory class to transparently load, normalize (if applicable), and combine the input profiles. The `LLVM` profile file format is structured as a sequence of profile information blocks. This format allows multiple raw and/or combined profiles for any combination of the profiling types supported by `LLVM` to exist in the same file. Thus, if instrumentation for both edges and call profiling is inserted into the program, a training run will generate a profile file containing one profile information block for the call profile, and another for the edge profile. If such a file is provided to the factory class, it will load all available profile blocks, and store each in a list according to the type (edge, path or call) of profile it represents. When all the input profile files have been processed, the profiles in the list for each type of profiling are combined in a CP batch operation, producing one combined edge profile, one combined path profile, and/or one combined call profile. These profiles are then available to FDI and `llvm-cprof`. FDI uses only the combined call profile, and will abort if this profile fails to build or is empty. `llvm-cprof` will write each non-empty combined profile as a profile information block in a new output profile file.

In this study, FDI is always provided with a single profile file containing one call-profiling information block. If the required profile is a raw profile, it is listed on the command-line as-is. However, if FDI will use a truly combined profile from multiple training runs, `llvm-cprof` is used to create the combined profile in advance.

7.1.2 Compiling with `LLVM`

An overview of the compilation process used in this study is presented in Figure 7.1. Light-gray shapes are intermediate program representations, black shapes are data inputs and profiles, and white shapes are executable programs. Thin white arrows represent conceptual data flow without processing, such as re-using preinline.bc; or un-grouping or regrouping the sets of inputs and profiles. The large, shaded,

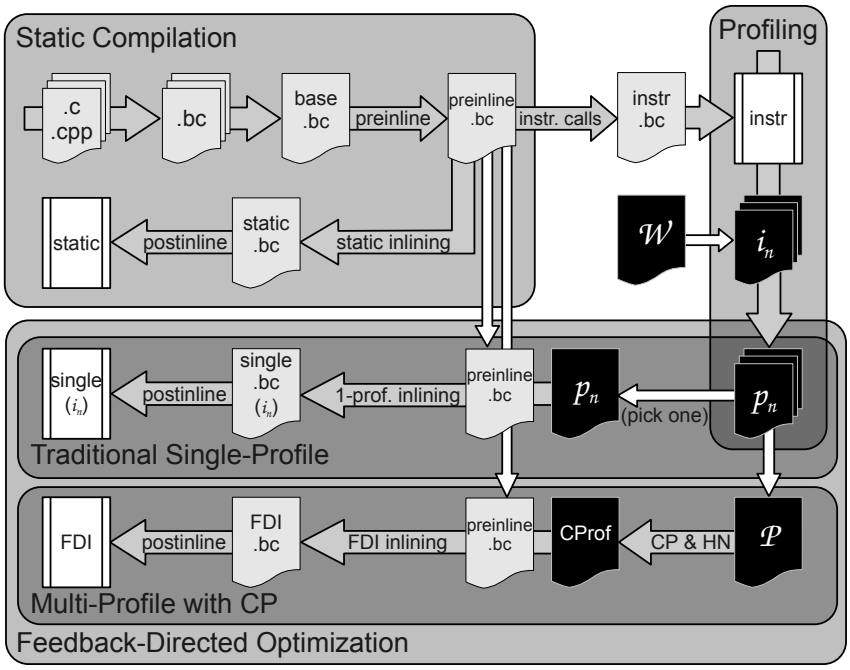


Figure 7.1: Overview of FDO program compilation with LLVM for static and FDI inlining. Postinline .bc files are omitted for simplicity.

boxes identify the three main components of the build process: static compilation, call profiling, and FDO; FDO contains sub-components for traditional single-input FDO and for the multi-profile FDO enabled by CP.

Compilation with LLVM begins with the translation of each source file into an LLVM bitcode file (.bc). Bitcode is the on-disk representation of LLVM's IR language; compilation can be stopped at any point and the intermediate form of the program written to disk. A bitcode file can be an input to the compiler in order to resume compilation or to combine several bitcode files. Bitcode can thus serve the same function as object files, and are used in their stead. A whole-program base bitcode file is constructed before any code transformations are applied.

LLVM provides the opt program, which allows compiler users to directly apply one or more transformations to a bitcode file. The results produced by opt are written to a new bitcode file. Each labeled transition between bitcode files in Figure 7.1 represents a separate invocation of opt. For instance, applying the preinline transformations to base.bc, or inserting the call profiling instrumentation (implemented as a transformation) into preinline.bc, are each done using opt.

The static compilation component of the build process applies the default inliner, as well as FDI using the static Anti and Benefit reward functions. After inlining, the postinline transformations are applied, and an optimized version of the program is produced for each of the three static inlining alternatives.

The profiling component of the build process inserts call profiling instrumentation into the preinline bitcode to produce the instr version of the program. The call

profiling is a vertex profiling, limited to the basic blocks containing call sites that FDI considers inlining candidates, and to the entry block of each function. This profiling allows hierarchal normalization to normalize the frequency of each call site by the frequency of the caller. The set of raw call profiles, $\mathcal{P}$, is generated by running instr on each input in $\mathcal{W}$.

The FDO component of the build process uses FDI. Under the traditional approach to FDO, a single raw profile is used. FDI internally produces a CProf from this profile, and uses a simple point reward function to guide inlining on the preinline bitcode. Since the CProf contains a single value for each monitor, all simple point metrics are equal. Alternatively, FDI is informed by a multi-profile CProf and uses one of the concrete reward functions described in Section 7.2.4. In both cases, compilation then proceeds as in the static inlining case: the postinline transformation group is applied and the native executable is created. The details of which profiles are used to create program versions follows the methodology of Chapter 3 and are presented in Section 7.2.

7.1.3 Transformation Sequence

Selecting the correct sequence in which to apply code transformations is a challenging problem. Iterative compilation techniques often search for the best ordering, which has been demonstrated to depend on both the program being compiled and the input on which the program is run. This problem arises because each transformation applied may create or inhibit opportunities for later transformations. In this study, the position of inlining in `LLVM`'s transformation sequence is selected to match its position during a typical compilation. However, because FDI uses profile information, it must follow the creation of the instrumented version of the program. Since profiling imposes an execution time overhead, as much optimization as possible should precede the profiling step in order to minimize profiling time.

Inlining alternatives are injected into `LLVM`'s standard `-O3` code transformation sequence by creating two subsets of `-O3` to apply before and after inlining. Transformations applied before inlining are referred to as the preinline transformation group; those applied after inlining are referred to as the postinline transformation group. Every compilation, regardless of inliner, first applies preinline, then the selected inlining, and finally postinline. In this way, each inliner, including the default inliner, is applied to the identical initial program, and followed by the same sequence of subsequent transformations. Therefore, a fair comparison of code size, inlining transformation time, and program execution times is possible between alternative inliners. The instrumented version of the program is created from the preinline version, as illustrated in Figure 7.1.

Inlining (with the rest of `-O3`) happens twice in a typical compilation using `LLVM`. The first inlining is done with the initial processing of the individual source files, where any calls to functions in a different file cannot be inlined. The second inlining happens at link time, and can thus inline those previously cross-module

calls. The compilation process followed in this work does not do any code transformation until after all the source files have been combined into a single bitcode file. Thus, the inlining evaluated here is essentially the link-time inlining.

The preinline and postinline groups are both large subsets of `-O3`. Each group maintains the original ordering of the transformations. postinline is all of `-O3` except for library-call simplification, which is done in preinline and should not be affected by inlining. Because preinline occurs before profiling, it focuses on code simplification (*e.g.*, constant propagation, global variable elimination, dead code elimination) and excludes transformations that modify function calls (*e.g.*, argument promotion, argument scalarization), replicate code (*e.g.*, loop unrolling, scalar promotion), or explicitly change control flow (*e.g.*, jump threading, loop unstitching). The application of preinline before profiling and inlining, and postinline afterward, is similar to a typical compilation where `-O3` is applied first on a file-by-file basis, and then again at link time.

Inlining is the first transformation applied in `LLVM`'s `-O3` transformation group. This approach is consistent with the default inliner's focus on code size. In comparison to the FDO transformations in the IBM XL compiler listed in Chapter 4.3.1, the default `LLVM` inliner is an aggressive early inliner. The `LLVM` inliner must be aggressive, because it is not followed by a late inliner focused on execution time improvement. In contrast, FDI is a late inliner, as discussed in Chapter 6. Ideally, FDI would be accompanied by an FDO-based early inliner that targets tiny, frequently-executed functions. However, this early inliner does not exist. While additional performance gains might be available if the default inliner was applied after FDI, this approach would conflate the inlining decisions of the two inliners, and hinder the evaluation and analysis of FDI. Therefore, the modified transformation sequence applies inlining only once.

7.1.4 Detecting Equivalent Inlining Outcomes

Performance evaluation can tell a compiler designer which inliner is most effective for a workload of programs and inputs. However, execution time alone does not enlighten the designer as to *why* one inliner is more effective than another. In order to answer this question, the inlining decision made by each inliner must be compared, and the differences identified. In Chapter 4, the outcome of a a decision made at a call site is associated with the source-code file and line number of the caller, and with the name of the callee. However, this approach suffers from significant imprecision, because multiple calls to the same callee can exist on a single line of source code. More significantly, inlining replaces a call site by the entire body of the callee. When an original call site is copied into the caller, the target call site could be associated with the line number of the source call site, because that is where it is created; with the line number of the original call site, because that is where it comes from; or with no line number at all, because the call does not exist in the original code. This problem is exacerbated when target call sites are

subsequently inlined.

Additionally, inlining call sites in a different order can produce the same final result. An *inlining chain* is similar to a call chain. When a new call site is created by inlining, that target call site retains the inlining chain of the original call site, and appends the source call site to the chain. In the absence of additional inlining into the functions along an inlining chain, the result of inlining the chain is independent of the order in which the call sites along the chain are inlined.

This work uses Zobrist hashing to provide unique identifiers to call sites that are independent of inlining-chain orderings, very similarly to how the technique is used to detect equivalent states arrived at by different paths in a search tree [2]. The implementation is straight-forward: when the initial set of inlining candidates is determined, each is assigned a random integer identifier called a zID. The random number generator is seeded with a constant, and the candidates are detected in a deterministic order. Thus, the assignment of identifiers to call sites is constant across all compilations that use the same initial program source. A target call site is assigned a zID by taking the bitwise exclusive OR of the zIDs of the source and original call sites. Thus, the inlining chain provides a history describing how a call site came to exist, and the zID summarizes this history in an order-independent manner. The same source call site will not be added to the inlining chain or XORed into a zID twice because recursive inlining is disallowed.

When the inlining pass is finished, a zID for each function can be created by adding together the zIDs of all the call sites in the function. Addition is used instead of XOR because using XOR would cause call sites inlined an even number of times through different inlining chains to cancel themselves out. If the transformed function after inlining is identical for two inliners, then the IDs computed for that function under either inliner will also be the same. Finally, a global identifier can be created by XORing the function IDs for all functions in the program. If two inliners transform a program in exactly the same way, their global zIDs will match.

Thus, a compiler developer can identify the transformation differences between inliners by following the zID computations in reverse order. The global zIDs are compared first to ensure that the transformed programs are in fact different. Then, the functions with non-matching zIDs can be identified. The zID of a function that does not contain a call site is 0, so this comparison will work even when all the call sites in a function have been inlined. Finally, difference in the zIDs of the call sites in a function indicate which inlining chains are different, and thus where the significant differences in inlining decisions occur.

7.2 Experimental Methodology

This study evaluates 14 instantiations of FDI reward functions, along with the Benefit static-reward function, single-profile FDO, and the default `LLVM` inliner. The version of a program transformed by an FDI inliner is referred to simply by the FDI reward function used by the inliner. For example, the mean version of a pro-

gram refers to the version of the program created by applying the FDI inliner using the mean reward function. This section outlines how performance is measured and evaluated for these inliners, the concrete FDI reward functions evaluated, and the set of programs and inputs used for the evaluation.

7.2.1 Measuring Single-Run Performance

Before an inliner's workload performance can be calculated, execution times for runs on individual inputs must be determined. In this study, the execution time of one version of a program on a particular input is computed as the minimum of the execution times of three runs for this pairing. Henceforth, any reference to execution time assumes this three-run measurement.

The baseline execution time used in this study is obtained using the Never static FDI reward function, which unconditionally returns -1.0. Thus, Never will never inline any call site where $\mathsf{Cost}(\mathrm{CS}) > -2$. Given the definition of $\mathsf{Cost}(\mathrm{CS})$ from Chapter 6.3, inlining is thus limited to callees containing a single basic block, that are expected to increase code size in the caller by no more than 3 instructions.

The performance of a particular version of a program on one input is computed as the execution time of that version of the program on that input, normalized by the execution time of Never on that input.

7.2.2 Static Inlining

Two static inliners are evaluated: Benefit, and the default `LLVM` inliner, Static. These inliners are evaluated by simply taking the geometric mean of their normalized execution times over all inputs in $\mathcal{W}$.

7.2.3 Single-Profile FDO

Traditional single-profile FDO is referred to in this evaluation as the Single inliner. Inlining is performed using a simple point reward function with FDI, informed by a single profile. Since the profile contains a single value for each monitor, all simple point reward functions are equal. The evaluation uses the leave-one-in methodology in a manner similar to that of Chapter 4. Each input $u \in \mathcal{W}$ is used for training, and then evaluated using $\mathcal{W}/\{u\}$. A performance score for u is calculated by taking the geometric mean of those normalized times:

$$\mathbf{gm}[u] = \sqrt[|\mathcal{W}-1|]{\prod_{i \in \mathcal{W}/\{u\}} \tau_u^{-1}(i)}$$

The final performance for traditional FDO is computed as the geometric mean of each $\mu_{\mathbf{g}}[u]$:

$$\mu_g = \sqrt[|\mathcal{W}|]{\prod_{u \in \mathcal{W}} \mathbf{gm}[u]}$$

7.2.4 FDI Reward Functions

The goal of CP-informed inlining is to increase performance across all future program executions by considering the cross-input behavior variations captured in the CProf. In particular, it strives to minimize the worst-case negative impact on the performance of any run. Performance degradation (versus an alternative inlining algorithm) is caused by a failure to inline call sites that are important in one or more program runs. These missed candidates are opportunity costs: the potential benefit of inlining is not realized in those runs because the inlining budget was spent on other candidates. It is the responsibility of the reward function to prioritize inlining candidates such that this opportunity cost is minimized. However, opportunity cost cuts both ways. Inlining a call that is important in only a small proportion of program runs may consequently expend the code-growth budget before more generally-beneficial inlining opportunities can be exploited.

> **Hypothesis:** Most inlining candidates that significantly impact the performance of any run also improve performance across most of the workload. The number of inlining opportunities that are particularly important for only a minority of runs is a small fraction of the total number of inlining opportunities that significantly impact program performance.

This hypothesis guides the selection of the example reward functions evaluated in this study. If correct, this hypothesis implies that scoring inlining candidates by their maximum expected benefit will identify both those few candidates that have a large impact in rare cases, as well as the candidates that provide benefit for the majority of the workload. Furthermore, weighting candidates by coverage is not necessary because the number of rarely-beneficial candidates is small. While inlining these candidates will consume some of the inlining budget, this merely prevents inlining a small number of calls that are expected to be the least beneficial among the set of calls that would otherwise be inlined.

As explained in Chapter 6.3.4, FDI provides three classes of reward functions that use combined profiles. Results are presented for the mean and max *simple point* rewards, seven concrete instantiations of *quantile point* rewards, and five concrete instantiations of *quantile range* rewards. The quantile points selected for these example reward functions are listed in Table 7.1. The type column, **T**, indicates if the reward functions uses (P)oints or (R)anges; this aspect of the reward function is also evident in the Quantiles column, where ranges are listed in angled brackets. The combination column, **C**, indicates how multiple points or ranges are combined: (L)inearly or (N)on-(L)inearly. Evaluation name provides the notation used to identify each inliner in the figures in Section 7.3.

The first resward functions sample the quartile boundaries, and represent somewhat pessimistic, median and somewhat optimistic rewards. The remaining point metrics represent cases that combine a high and a low frequency estimate. Non-

T	C	Quantiles (%)	Description	Evaluation Name
P	–	25	first quartile	QPointQ=25
P	–	50	estimated median	QPointQ=50
P	–	75	third quartile	QPointQ=75
P	L	50, 75	average and optimistic	QPLinearQ=50,75
P	NL	50, 75		QPSqrtQ=50,75
P	L	5, 95	worst and best w/o outliers	QPLinearQ=5,95
P	NL	5, 95		QPSqrtQ=5,95
R	–	$\langle 50, 100 \rangle$	top half: optimistic	QRangeQ=50,100
R	–	$\langle 25, 75 \rangle$	"central" average	QRangeQ=25,75
R	–	$\langle 5, 95 \rangle$	average w/o outliers	QRangeQ=5,95
R	L	$\langle 0, 25 \rangle$, $\langle 75, 100 \rangle$	pessimistic and optimistic	QRLinearQ=0,25,75,100
R	NL	$\langle 0, 25 \rangle$, $\langle 75, 100 \rangle$		QRSqrtQ=0,25,75,100

Table 7.1: Concrete quantile-based reward functions

linear combination (sqrt) uses a multi-objective approach that balances the importance of the two measurements. The first two range-based rewards consider the weighted average frequency over half of the histogram's weight. The third range-based reward uses the weighted average over the whole histogram, excluding any high or low outliers. The final two reward metrics combine the weighted average frequency from the top and bottom quartiles, balancing optimistic and pessimistic frequency predictions.

The inliners using these reward functions are evaluated according to the 3-fold cross-validation methodology proposed in Chapter 3. The testing and training sets are determined once from a random ordering of the inputs in $\mathcal{W}$. Identical testing and training sets are used in each fold by each inliner. Each input in $\mathcal{W}$ is in exactly one testing set, thus the workload performance of an inliner is determined by taking the usual geometric mean:

$$\mu_g = \sqrt[|\mathcal{W}|]{\prod_{i \in \mathcal{W}} \tau^{-1}(i)}$$

7.2.5 Programs and Inputs

This study evaluates the inliners described above using four programs: `bzip2`, `gzip`, `gcc`, and `gobmk`. Each program is evaluated using a 15-input workload, as suggested in Chapter 3. `Gcc` and `gobmk` are taken from the SPEC CPU 2006 benchmark suite. SPEC provides 11 inputs for `gcc`. In spite of the challenges involved in creating new inputs for this benchmark, four[1] of the SPEC 2000 benchmark programs were converted to the single pre-processed file format. The converted programs are `bzip2`, `LBM`, `MCF`, and `parser`. For `gobmk`, SPEC provides 20 inputs. However, only 5 of these inputs come from the `ref` workload;

[1] of seven attempts

the `train` workload contains 8 inputs, and the `test` workload contains 7 inputs. Many of the inputs from `test` and `train` have very short execution times: 4 inputs take less than 1 second, 6 take 2–9 seconds, 4 take 12–19 seconds, and 1 takes longer than 1 minute. Execution times of less than a few seconds are subject to large proportional timing imprecision, because the Linux `time` command reports times with a resolution of $1/100^{th}$ of a second. Therefore, the 15 longest-running inputs are chosen for $\mathcal{W}$. This set is composed of the `ref` and `train` SPEC workloads, plus connect and dniwog from `test`. The shortest baseline running time in $\mathcal{W}$ is 2.3 seconds, for connect.

The other two programs used in the case study are `bzip2` and `gzip`. However, rather than using the SPEC benchmark versions of these programs, the fully-functional "real" versions are used. Using the real versions of the compressor programs eliminates the unrealistically-simplified profiling situation where mutually-exclusive use cases are combined into a single program run. Consequently, these programs cannot do decompression and compression, or multiple levels of compression, within the same run. These distinct use-cases must be covered by different inputs in the program workload. Both `bzip2` and `gzip` share the same workload of inputs. This workload is split in half into a compression set and a decompression set. Several inputs in the compression set have an analogue in the decompression set. However, the file format is usually different, and the source of the data is never the same. For instance, revelation-ogg in the compression set and sherlock-mp3 in the decompression set are both audio books, but the audio is recorded in different formats, and the books themselves are different.

Both compressors use a numeric command-line flag to control the tradeoff between compression speed and compression quality. The flags take integer values between 1 (fastest, least compressed) and 9 (slowest, most compressed). The seven inputs in the compression set each use a different compression level, from 3 to 9. Most inputs are collections of files. Each collection is archived (uncompressed) so that the input and output of each run is a single file. In order to minimize the impact of disk access, the output of each run is redirected to `/dev/null`.

The compression set contains the following inputs, with the compression level shown in parentheses:

avernum (-3) The installer for the demo version of the game "Avernum: Escape from the Pit" from Spiderweb Software.

cards (-4) A collection of greeting card layouts in the TIFF (uncompressed) image format.

ebooks (-5) A collection of ebooks, with and without images, and in a variety of formats, from Project Gutenberg[2].

[2]http://www.gutenberg.org

potemkin-mp4 (-6) The 1925 movie "Bronenosets Potyomkin (Battleship Potemkin)" in MP4 format, from the Internet Archive[3].

proteins-1 (-7) A sample of 33 proteins from the RCSB Protein Data Bank database. 6 files for each protein, each stored in a different text-based format, provide different characteristics of the protein's structure[4].

revelation-ogg (-8) The audio book "The Revelation of Saint John" in OGG format, from Project Gutenberg[5].

usrlib-so (-9) A collection of shared object (.so) files from `/usr/lib/` of a 32-bit gentoo-linux machine.

The decompression set for each compressor uses the same base set of files, precompressed by the appropriate compressor at the default compression level. The decompression set is composed of:

auriel The "Auriel's Retreat" land-mass addition mod by lance4791 for the game "The Elder Scrolls IV: Oblivion" from Bethesda Softworks[6].

gcc-453 The source-code archive of the `gcc` compiler, version 4.5.3[7].

lib-a A collection of library files (.a) from `/lib/` of a gentoo-linux machine. As per the gentoo development guide, a library will be installed in `/lib` (boot critical) or `/usr/lib` (general applications), but not both[8].

mohicans-ogv The 1920 movie "Last of the Mohicans" in OGV (ogg video) format, from the Internet Archive[9].

ocal-019 The Open Clip Art Library archive, version 0.19. The images are primarily in vector-graphics formats[10].

paintings-jpg A collection of watercolor paintings, in JPG format.

proteins-2 A completely different sample of 157 proteins from the RCSB Protein Data Bank database, each in 6 different file formats.

sherlock-mp3 The audio book "The Adventures of Sherlock Holmes" in MP3 format, from Project Gutenberg[11].

[3] http://archive.org/details/BattleshipPotemkin
[4] http://www.rcsb.org
[5] http://www.gutenberg.org/ebooks/22945
[6] http://planetelderscrolls.gamespy.com/View.php?view=OblivionMods.Detail&id=5949
[7] http://gcc.gnu.org/gcc-4.5
[8] http://devmanual.gentoo.org/general-concepts/filesystem/index.html
[9] http://archive.org/details/last_of_the_mohicans_1920
[10] http://openclipart.org/collections
[11] http://www.gutenberg.org/ebooks/28733

7.3 Results

The results presented in this section are computed using Redhat Linux 2.6.32 running on quad-core AMD Opteron^{TM} Processor 2350 running at 2 GHz with 512 KB of L1 cache and 8 GB RAM. A single timing run executes on the machine at any time; no performance measurements execute in parallel.

Overall, the results presented in the remainder of this section suggest that when evaluated on a workload of inputs, none of the studied inliners are statistically different from the baseline or from each other. This result is due to large per-input variations in performance. Chapter 3 suggests the use of hypothesis testing to potentially establish statistically-significant differences in mean workload performance in spite such variation. While this approach may be effective when comparing two alternatives, comparisons between multiple alternatives must be performed with the utmost caution. Using the typical significance level $\alpha = 0.05$, 5% of test are expected to incorrectly reject the null hypothesis. The full pair-wise comparison for one program between the 14 FDI inliners requires $14^2 = 196$ tests, 10 of which are expected to indicate a statistically-significant result *by chance*. Sophisticated techniques that account for this increased risk of false positives among multiple comparisons in order to potentially expose statistically-significant (though not necessarily practically-significant) performance differences between the inliners could be employed. Instead, this section focuses on presenting the actual data. The figures found in the following sections summarize performance data-sets using the geometric mean with 95% confidence intervals. These confidence intervals describe the *mean*, not the data, thus the maximum and minimum values in each data set are also marked to indicate the range of observed values. Furthermore, performance measurements are presented along two dimensions: by inliner, and by data input. Cutting the results along the input dimension provides insight into the inliners' workload performance. Finally, ranking inliner performance across the workload investigates consistency in the relative per-input ordering of inliner performance.

However, from the collected results it is also clear that Benefit is a very poor inliner. Recall that the Benefit reward function is simply the unmodified Benefit(CS) static estimate of inlining utility discussed in Chapter 6.3.3. Benefit inlines more calls than other inliners, significantly increasing compilation time. At the same time, Benefit seldom improves program performance, but frequently causes large performance degradation. The FDI inliners are based on Benefit, but do not increase compilation time and often manage to improve on Benefit's performance. On the other hand, the default inliner is almost always better than Benefit. However, even this production-quality inliner causes some performance degradation compared to the baseline, and seldom improves performance by even 5%.

Compilation Step		bzip2	gzip	gcc	gobmk
Never	inlining	0.1	0.3	23.6	18.3
	postinline	0.6	2.8	18.9	20.7
	native	1.2	4.0	45.0	63.4
Static	inlining	0.1	0.5	6.9	18.6
	postinline	0.9	3.8	44.3	24.6
	native	1.4	5.2	62.2	115.1
Benefit	inlining	0.8	3.4	108.5	23.6
	postinline	18.6	36.0	20.2	26.8
	native	13.4	29.7	45.8	90.1
Single	inlining	0.1–0.1	0.3–0.4	16.0–21.3	18.2–20.1
	postinline	0.6–0.8	2.9–4.0	20.1–23.0	20.6–22.7
	native	1.2–1.4	4.1–5.8	47.2–47.5	63.1–70.3
FDI	inlining	0.1–0.1	0.4–0.8	17.5–25.4	19.4–21.9
	postinline	0.6–0.6	3.5–5.5	20.0–22.6	22.0–24.5
	native	1.2–1.3	4.6–6.8	47.0–47.8	77.9–83.1

Table 7.2: Time (in seconds) for each step of compilation from inlining to linking and generation of the native executable. Ranges are listed for the collections of Single and FDI inliners.

Measurement		bzip2	gzip	gcc	gobmk
Code	size	12,064	7,301	407,976	91,778
	growth	605%	812%	7.5%	145%
	budget	73,040	59,252	30,648	133,209
Calls inlined	candidates	348	302	45,296	8,640
	Never	29	2	1,162	566
	growth:	*-2*	*2*	*3,853*	*146*
	Benefit	354	353	9,047	3,697
	growth:	*642%*	*812%*	*7.5%*	*145%*
	Single	46–61	18–48	811–1,250	580–861
	growth:	*3%–59%*	*3%–25%*	*7.5%–7.5%*	*0.2%–10%*
	FDI	59–59	72–101	977–1,605	1,253–1,592
	growth:	*4%–4%*	*30%–72%*	*7.5%–8.1%*	*23%–37%*
Exec size	instr	88 K	73 K	3.7 M	3.9 M
	Static	93 K	73 K	3.6 M	3.9 M
	Never	81 K	65 K	2.9 M	3.7 M
	Benefit	305 K	238 K	3.0 M	4.4 M
	Single	81–88 K	66–75 K	3.0–3.0 M	3.7–3.7 M
	FDI	82–82 K	72–84 K	3.0–3.1 M	3.8–3.9 M

Table 7.3: Initial code size, inlining statistics, code growth and executable file sizes for each class of inliner

7.3.1 Compilation Time and File Size

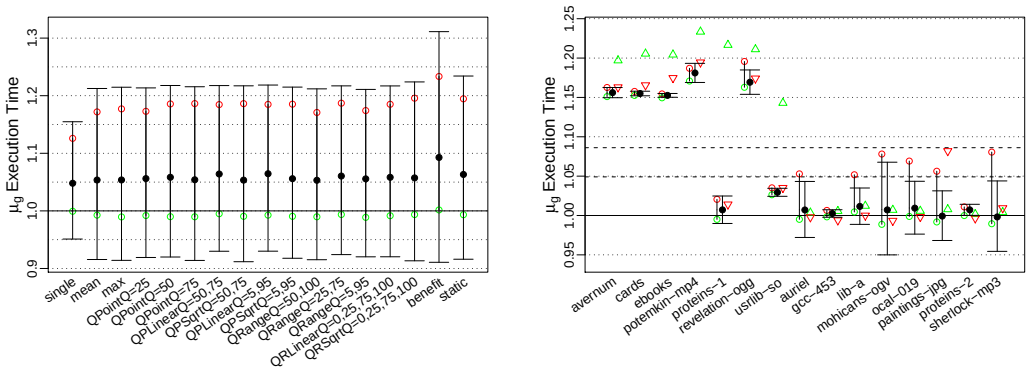
As illustrated in Figure 7.1, all versions of a program share a common compilation path until the point where inlining is applied. Table 7.2 presents a breakdown of the remaining compilation time for each program. The rows labeled "inlining" report the time required for `opt` to apply the inlining transformations, those labeled "postinline" report the time taken by `opt` to apply the postinline transformation group, and the lines labeled "native" report the time taken by the final invocation of `LLVM` used to generate executable native code from the postinline.bc bitcode files. The lines for Single give ranges over each version of the program created by leave-one-in; the lines for FDI give ranges over all program versions created by each of the three folds of cross validation over all FDI inliners.

Compilation time is similar across most inliners for most programs. The primary exception is Benefit, which takes significantly longer than other inliners for all compilation steps for `bzip2` and `gzip`, and dramatically increases inlining time for `gcc`. Conversely, for `gcc` the Static inliner spends very little time inlining, but compilation spends much more time on postinline and native than is required after applying the FDI inliner.

Table 7.3 provides additional details about the inlining performed by each inliner, and the sizes of the resulting executable files. The "Code" section of the table lists the initial size of the program in `LLVM` IR instructions, the proportional code growth allowed by Budget, and the code growth budget in IR instructions. The "Calls inlined" section lists the number of call sites inlined by each inliner, and the resulting code growth. The "candidates" row indicates the initial number of call sites that are inlining candidates at the beginning of inlining; additional inlining candidates are created during inlining when new target call sites are created. Growth is given as a number of IR instructions for Never, but as a proportion of the initial number of IR instructions for the other inliners. Finally, the "Exec size" section gives the size of the executable files.

As expected, Never provides the fastest compilation and smallest file sizes. However, for `gcc` with Never, inlining takes more than 3x longer than with Static, and longer than with many of the Single and FDI inliners. This observation is explained in Table 7.3 by the 1162 call sites inlined by Never. The resulting code growth of 3,853 instructions averages to 3.3 instructions per inlined call, an excess of 10% over Never's allowed limit of 3 instruction of predicted growth per inlined call. Thus, Benefit(CS) must be over-estimating the code-simplification opportunities available when inlining some call sites in `gcc`.

Benefit inlines many more calls than the other inliners. For `bzip2`, `gzip`, and `gobmk`, the increased number of call sites inlined is the result of Benefit exhausting the inlining budget while the other inliners do not. Profile information suppresses inlining for un-executed call sites; the FDI inliners are informed by a multi-run CProf, and thus tend to inline more calls than the Single inliners. Consequently, the postinline and native compilation steps after Benefit inlining take longer than after other inliners using the FDI framework, because subsequent analyses and trans-



(a) over all inputs in $\mathcal{W}$, for each inliner

(b) over all FDI inliners, for each input in $\mathcal{W}$

Figure 7.2: Geometric-mean performance: real `bzip2`

formations act over a larger program. However, for `gcc`, all inliners consume the entire code-growth budget. Thus, the increased inlining by Benefit indicates that each call site inlined contributes less code growth on average than those inlined by the other inliners, a result of the `LLVM` code simplification heuristics upon which Benefit is built begin focused on code size. The cost of inlining more call sites and of re-evaluating inlining candidate scores and re-sorting the list of candidates more frequently results in the increased time spent by Benefit on inlining. However, because the total code size after inlining is similar between inliners, the time taken by the rest of the compilation also remains similar.

Although code size measured in IR instructions does not directly predict the size of the executable file for a program, comparing code growth and executable file sizes in Table 7.3 suggests a reliable correlation between increases in the number of IR instructions in a program and the resulting size of the executable file.

The code-growth budget function presented in Chapter 6 has not been tuned for this work or tested on other programs. The code growth results in Table 7.3 suggest that the code growth limit allowed for small programs is significantly larger than required, while the limit for `gcc` may be too restrictive. In the case of `gcc`, the inlining done by Never suggests that an early inliner might inline many low-cost call sites; computing the inlining budget after early inlining will slightly increase the inlining budget of large programs, but will also prevent the inlining of these calls from consuming the inlining budget, which is mostly intended to control code growth due to inlining larger callees.

7.3.2 Execution Time

The execution time performance of the inliners described in Section 7.2 are presented for the four case-study applications in Figures 7.2 through 7.5. As described in Section 7.2, performance is summarized using a geometric mean of normalized execution times. Thus, values smaller than 1.0 represent speedups, while values greater than 1.0 represent slowdowns. Each figure displays two charts. In both, the

	Rank		
Reward	μ	σ	**Weighted**
QRangeQ=5,95	5.3	4.4	5.0
QPSqrtQ=50,75	6.3	3.1	6.1
QPointQ=50	6.4	3.5	6.5
max	6.5	3.4	6.2
QRangeQ=50,100	6.6	4.3	6.6
QPointQ=75	7.1	4.2	5.8
QPointQ=25	7.6	4.3	7.5
QRLinearQ=0,25,75,100	7.7	4.5	7.5
QPLinearQ=50,75	7.7	4.5	9.9
mean	7.7	3.2	8.3
QPSqrtQ=5,95	8.0	3.8	8.1
QRangeQ=25,75	8.6	4.8	8.8
QRSqrtQ=0,25,75,100	9.3	3.7	8.7
QPLinearQ=5,95	10.2	3.3	10.0

Table 7.4: Workload ranking of FDI inliners for `bzip2`

	Rank		
Reward	μ	σ	**Weighted**
QPLinearQ=50,75	5.3	4.2	6.4
QPointQ=50	6.5	4.2	5.2
max	7.0	4.2	8.5
QPSqrtQ=50,75	7.4	4.5	6.9
QPLinearQ=5,95	7.5	4.3	9.0
QPointQ=25	7.6	4.0	8.7
QRangeQ=25,75	7.6	3.5	7.9
QRangeQ=5,95	7.6	4.3	7.2
QRangeQ=50,100	7.6	3.9	6.8
QRLinearQ=0,25,75,100	7.7	3.0	8.1
mean	7.7	4.7	6.3
QRSqrtQ=0,25,75,100	8.2	4.8	7.1
QPSqrtQ=5,95	8.3	4.0	6.7
QPointQ=75	8.8	3.4	10.3

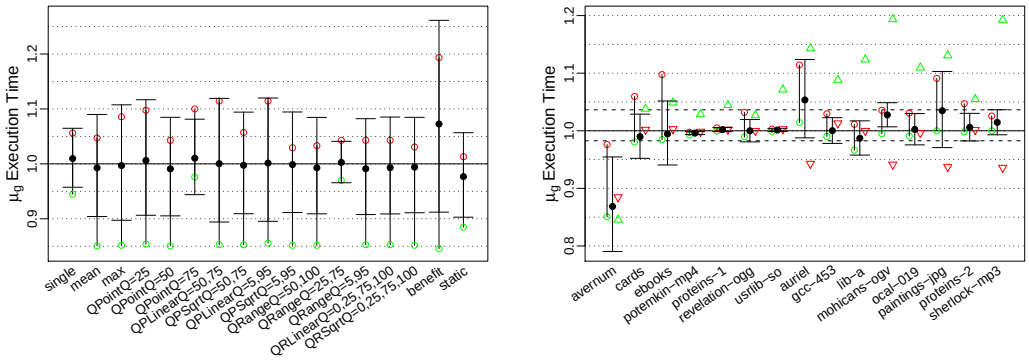
Table 7.5: Workload ranking of FDI inliners for `gzip`

	Rank		
Reward	μ	σ	**Weighted**
QPSqrtQ=5,95	5.9	4.4	5.4
QRangeQ=25,75	6.6	5.3	6.6
QPointQ=50	7.1	4.8	6.3
QPLinearQ=50,75	7.1	4.8	7.2
QRLinearQ=0,25,75,100	7.4	6.7	7.7
max	7.6	5.8	7.7
QPointQ=25	7.7	3.4	7.4
QPointQ=75	8.3	4.8	8.7
QPSqrtQ=50,75	8.5	4.6	8.9
QRangeQ=5,95	9.7	4.5	9.7
QRangeQ=50,100	9.7	5.0	9.1
mean	10.2	6.1	8.7
QRSqrtQ=0,25,75,100	10.6	5.6	10.5
QPLinearQ=5,95	10.8	6.2	10.5

Table 7.6: Workload ranking of FDI inliners for `gcc`

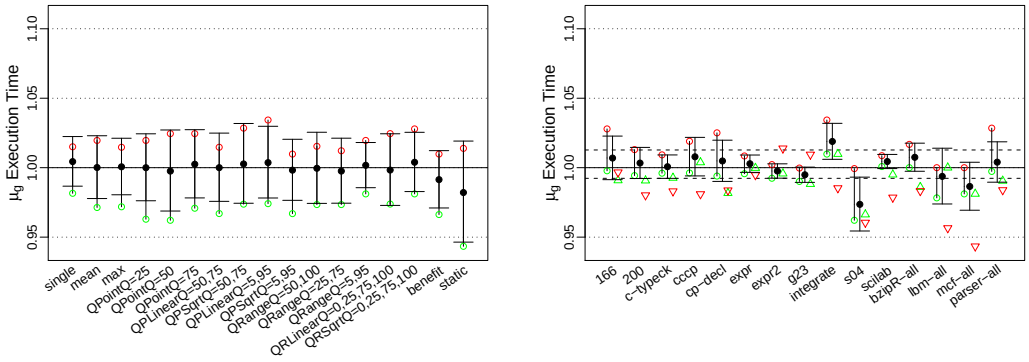
	Rank		
Reward	μ	σ	**Weighted**
QPSqrtQ=50,75	2.9	2.0	2.8
QPSqrtQ=5,95	3.4	3.7	3.2
QRangeQ=25,75	4.5	2.4	4.7
QRSqrtQ=0,25,75,100	4.9	2.6	4.9
QPLinearQ=50,75	6.9	2.9	6.8
QRangeQ=5,95	7.7	2.6	7.9
QRangeQ=50,100	8.1	1.9	8.2
QPointQ=25	8.4	3.7	8.7
QRLinearQ=0,25,75,100	8.6	3.7	8.4
QPointQ=50	8.7	1.6	8.7
QPLinearQ=5,95	9.3	5.2	9.3
QPointQ=75	9.6	4.1	9.1
max	9.7	4.2	9.8
mean	12.5	1.1	12.5

Table 7.7: Workload ranking of FDI inliners for `gobmk`



(a) over all inputs in $\mathcal{W}$, for each inliner (b) over all FDI inliners, for each input in $\mathcal{W}$

Figure 7.3: Geometric-mean performance: real `gzip`



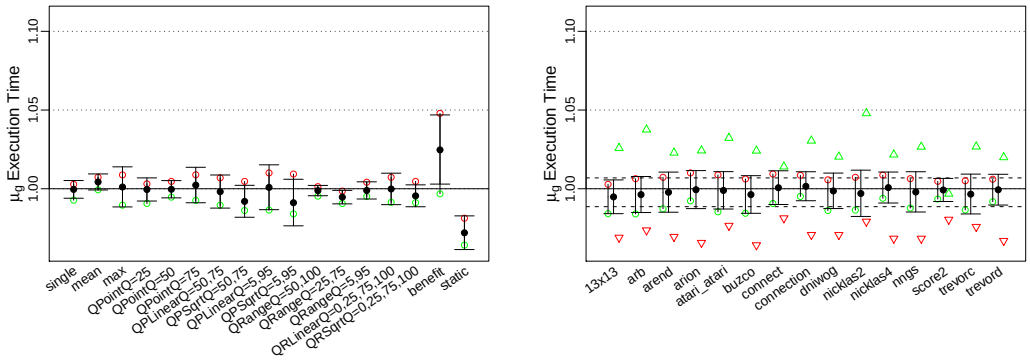
(a) over all inputs in $\mathcal{W}$, for each inliner (b) over all FDI inliners, for each input in $\mathcal{W}$

Figure 7.4: Geometric-mean performance: SPEC 2006 `gcc`

black dot represents the geometric mean of normalized execution times. The error bars show a 95% confidence interval for that mean. Another set of error bar, tipped with small open circles, indicate the best and worst measurements included in the mean.

In subfigure (a) on the left, each inliner is evaluated on $\mathcal{W}$. The results presented for the Single inliner refer to traditional single-profile FDO, and are computed as the geometric mean of workload performance from each version of the program produced by the leave-one-in method. The FDI inliners are evaluated by 3-fold cross-validation.

In subfigure (b) on the right, the performance of the FDI inliners is summarized for each input by taking the geometric mean of the normalized execution time of each FDI inliner for that input. For `bzip2` and `gzip`, the inputs are listed in alphabetical order, with the compression set on the left and the decompression set on the right. The dashed horizontal lines indicate the geometric mean of the best and worst normalized times, and thus indicate the bounds on possible workload performance if a different inliner could be chosen for each input. In addition, the results for Benefit are marked by $\triangle$, and results for Static are marked by $\bigtriangledown$. Note



(a) over all inputs in $\mathcal{W}$, for each inliner

(b) over all FDI inliners, for each input in $\mathcal{W}$

Figure 7.5: Geometric-mean performance: SPEC 2006 `gobmk`

that Single is not present in these charts. To reduce symbol overlap, the triangles are shifted slightly to the right, and the best-worst error bar is shifted slightly to the left, compared to the error bar for the mean.

The inlining performance presented in Figure 7.2(a) shows variations from 0.98 to 1.23 in the per-input performance of all inliners across the workload. Consequently, there is no statistically-significant difference between the inliners. Figure 7.2(b) shows a clear distinction between the compression and decompression sets of inputs. All FDI inliners perform quite similarly on the compression set of inputs. However, for 5 of the 7 compression inputs, all FDI inliners, as well as Benefit and Static, degrade performance by more than 15% compared to the Never baseline. Benefit also significantly degrades performance on the remaining 2 compression inputs; in these cases, FDI and Static avoid most of that degradation. Throughout the compression set, Benefit is at least 3% slower than Static.

On the other hand, the inliners cause at most an 8% degradation for the decompression set, and Benefit and Static are almost equivalent for all inputs except paintings-jpg, where Static is 7% worse than Benefit. However, there is significant variation between the FDI inliners on 6 of the 8 inputs in the decompression set.

A more detailed examination of FDI inliner performance is provided by the rank analysis presented in Table 7.4, and by the matched-pairs pairwise workload evaluations presented in Table 7.8. The rank analysis investigates relative inliner performance without regard to the magnitude of performance impacts. For each input, the inliners are sorted by execution time to determine each inliner's rank, from 1 (fastest) to 14 (slowest). Table 7.4 lists the FDI inliners, sorted by their average rank (μ) over $\mathcal{W}$, along with the standard deviation (σ). If the ranks were randomly assigned, an average ranks of 7.5 would be expected for all inliners. For `bzip2`, not only are the average ranks clustered around the expectation, but the variation in the ranks of each inliner across the workload is significant; none of the differences in average rank between inliners are statistically-significant. The "weighted" column computes a weighted average rank for each inliner. The difference in normalized execution time between the best and worst FDI inliner is used as the weight for each

input. Thus, the ranks assigned for inputs where little performance variation exists between the inliners will have little impact on the inliners' weighted average rank. However, this weighting has a negligible impact on the average ranks.

The complete pairwise performance comparisons of FDI inliners reported in Table 7.8 further supports the assessment that for `bzip2`, the FDI inliners are indistinguishable. Each inliner is selected as the baseline, in place of Never. Each baseline is a column on the table. The inliners on each row of the table are evaluated on $\mathcal{W}$ against each baseline inliner, using the same methodology used to compute workload performance compared to Never. The table gives the geometric mean and 95% confidence interval, as percentages, for each pairwise evaluation. Thus, the value 0.2 ± 1.1 in the first row of Table 7.8 in the QPointsQ=25 column indicates that, for the tested workload, mean produces 0.2% ($\pm$1.1%) faster code than QPointsQ=25. Negative values indicate that the evaluated inliner (row) is slower than the baseline (column). In Table 7.8, almost none of the pairwise differences are larger than 1%, and none are statistically significant.

The results for `gzip` in Figure 7.3 also show significant variation between inliners for the decompression set. While Static provides a 5% performance improvement for half the decompression set, and no impact for the other half, Benefit degrades performance by 5–20%. On the compression set, `gzip` does not suffer from the large degradations seen with `bzip2`. However, all inliners improve execution time for avernum. Avernum is also the only input where Benefit produces a better result than Static. Furthermore, this is also the only case over the entire `gzip` workload where Benefit, as well as every FDI inliner, improves performance. Recall that the FDI inliners essentially weight the inlining utility predicted by Benefit(CS) by frequency estimates from the profiles. Thus, FDI can amplify the expected inlining utility of frequently-executed call sites, or suppress the inlining of un-executed call sites. FDI cannot correct errors in the basic utility estimates computed by Benefit(CS). The results in both Figure 7.2(b) and Figure 7.3(b) suggest that FDI performance can be dramatically degraded when the inlining utility predicted by Benefit(CS) produce poor inlining decisions. The `gzip` results for avernum also suggest that if Benefit provides good utility predictions, the FDI inliners can exploit this information.

The results of rank analysis and pairwise evaluation for `gzip` are presented in Tables 7.5 and 7.9, respectively. As with `bzip2`, neither analyses provides any evidence of statistically-significant differences between the FDI inliners.

Inlining has very little impact for `gcc`, as shown in Figure 7.4(a). At worst, performance is degraded by less than 4%, and at best performance is improved by slightly more than 5%. The average impact of the FDI inliners tracks the impact of Benefit quite closely. In particular, the greatest improvement from FDI inlining corresponds to the best impact of Benefit, for s04. Similarly, the worst result of FDI inlining corresponds to the worst impact of Benefit, for integrate. The average rankings in Table 7.6 do not suggest any significant ranking differences between the inliners, and none of the pairwise evaluations reported in Table 7.11 show

statistically-significant performance differences.

The workload performance results for gobmk in Figure 7.5(a) display the only statistically-significant workload-performance differences between inliners in this study. Benefit is worse than Static and several of the FDI inliners, while Static improves performance compared to most of the FDI inliners. The only significant result between the FDI inliners indicates that Mean is worse than QRangeQ=25, 75. These results are explained by Figure 7.5(b). On a per-input basis, Static is strictly better than the FDI inliners, while Benefit is worse than the FDI inliners for all inputs except score2. The performance of the FDI inliners on each input is quite similar, and does not vary significantly between inputs. However, Figure 7.5(a) shows that amount of variation in performance across the workload is not consistent between inliners. For instance, the performance of QRangeQ=50, 100 is very consistent between inputs, while the performance of QPLinearQ=5, 95 varies to a larger degree.

The rank analysis and pairwise evaluation results confirm that Mean is the worst inliner for gobmk. The average inliner rankings for gobmk, shown in Table 7.7, span a much larger range than the rankings for the other programs. However, the most significant result of the rankings is the 12.5 average rank for Mean. Across $\mathcal{W}$, Mean is never ranked better than 11^{th}. Conversely, QPSqrtQ=50, 75 is ranked 2^{nd} on over half the inputs in $\mathcal{W}$, but falls to 9^{th} for connect. Nonetheless, QPSqrtQ=50, 75 is a good inliner for gobmk. These ranking results are supported by the pairwise workload evaluations in Table 7.10. The differences in workload performance between inliners are necessarily small, given the slight impact of inlining on gobmk observed in Figure 7.5(b). QPSqrtQ=50, 75 produces better results, by a statistically-significant margin, than all but three of the other FDI inliners. On the other hand, Mean produces worse results than all but three of the other FDI inliners. These results demonstrate, for the first time, that simply taking the average of multiple profiles may not be an effective profile-combination methodology.

Reward	mean	max	QPointQ=25	QPointQ=50	QPointQ=75	QPLinearQ=50,75	QPSqrtQ=50,75	QPLinearQ=5,95	QPSqrtQ=5,95	QRangeQ=50,100	QRangeQ=25,75	QRangeQ=5,95	QRLinearQ=0,25,75,100	QRSqrtQ=0,25,75,100
mean	–	0.0±0.5	0.2±1.1	0.4±1.6	0.0±0.7	0.9±2.5	-0.0±0.4	1.0±1.9	0.2±0.8	-0.0±0.6	0.6±1.9	0.2±1.3	0.4±1.3	0.3±0.8
max	-0.0±0.5	–	0.2±1.3	0.4±1.5	0.0±0.4	0.9±2.6	-0.0±0.4	1.0±2.1	0.2±0.4	-0.1±0.4	0.6±2.1	0.2±1.3	0.4±1.5	0.3±0.7
QPointQ=25	-0.2±1.2	-0.2±1.3	–	0.2±1.9	-0.2±1.3	0.7±2.8	-0.3±1.2	0.8±2.5	-0.0±1.4	-0.3±1.3	0.4±2.5	-0.1±1.9	0.2±1.9	0.1±1.3
QPointQ=50	-0.5±1.7	-0.5±1.5	-0.2±2.0	–	-0.4±1.8	0.5±3.2	-0.5±1.6	0.5±2.8	-0.2±1.7	-0.5±1.9	0.2±2.8	-0.3±2.2	-0.0±2.2	-0.1±1.9
QPointQ=75	-0.0±0.7	-0.0±0.4	0.2±1.3	0.4±1.7	–	0.9±2.6	-0.1±0.5	1.0±2.0	0.2±0.4	-0.1±0.5	0.6±2.1	0.1±1.3	0.4±1.6	0.3±0.8
QPLinearQ=50,75	-1.0±2.7	-1.0±2.8	-0.8±3.0	-0.6±3.3	-1.0±2.8	–	-1.0±2.8	-0.0±3.6	-0.8±2.8	-1.1±2.7	-0.4±3.5	-0.8±3.1	-0.6±3.2	-0.7±2.9
QPSqrtQ=50,75	0.0±0.4	0.0±0.4	0.2±1.2	0.5±1.5	0.1±0.5	1.0±2.6	–	1.0±2.0	0.2±0.6	-0.0±0.6	0.6±2.1	0.2±1.3	0.4±1.5	0.4±0.7
QPLinearQ=5,95	-1.1±2.0	-1.0±2.2	-0.8±2.6	-0.6±2.9	-1.0±2.2	-0.1±3.5	-1.1±2.1	–	-0.8±2.1	-1.1±2.1	-0.4±1.6	-0.9±1.8	-0.6±1.5	-0.7±2.2
QPSqrtQ=5,95	-0.2±0.8	-0.2±0.4	0.0±1.4	0.2±1.6	-0.2±0.4	0.7±2.6	-0.2±0.6	0.8±2.0	–	-0.3±0.4	0.4±2.2	-0.0±1.2	0.2±1.7	0.1±0.8
QRangeQ=50,100	0.0±0.6	0.1±0.4	0.3±1.3	0.5±1.7	0.1±0.5	1.0±2.5	0.0±0.6	1.1±2.0	0.3±0.4	–	0.7±2.1	0.2±1.2	0.5±1.6	0.4±0.7
QRangeQ=25,75	-0.7±2.1	-0.7±2.3	-0.4±2.6	-0.2±2.9	-0.6±2.3	0.3±3.5	-0.7±2.3	0.4±1.5	-0.5±2.4	-0.7±2.3	–	-0.5±2.7	-0.2±0.8	-0.3±2.2
QRangeQ=5,95	-0.2±1.3	-0.2±1.4	0.0±1.9	0.2±2.1	-0.2±1.3	0.8±2.9	-0.2±1.3	0.8±1.7	0.0±1.2	-0.2±1.2	0.4±2.6	–	0.2±2.1	0.1±1.6
QRLinearQ=0,25,75,100	-0.5±1.4	-0.4±1.6	-0.2±2.0	-0.0±2.2	-0.4±1.7	0.5±3.1	-0.5±1.5	0.6±1.4	-0.2±1.7	-0.5±1.7	0.2±0.8	-0.3±2.1	–	-0.1±1.6
QRSqrtQ=0,25,75,100	-0.3±0.8	-0.3±0.7	-0.1±1.3	0.1±1.8	-0.3±0.8	0.6±2.7	-0.4±0.7	0.7±2.1	-0.1±0.8	-0.4±0.7	0.3±2.1	-0.2±1.6	0.1±1.6	–

Table 7.8: Pairwise matched-pairs workload performance comparison for `bzip2`

Reward	mean	max	QPointQ=25	QPointQ=50	QPointQ=75	QPLinearQ=50,75	QPSqrtQ=50,75	QPLinearQ=5,95	QPSqrtQ=5,95	QRangeQ=50,100	QRangeQ=25,75	QRangeQ=5,95	QRLinearQ=0,25,75,100	QRSqrtQ=0,25,75,100
mean	–	0.4±2.3	1.3±3.5	-0.2±1.6	1.7±4.1	0.7±2.8	0.5±2.1	0.8±2.8	0.6±1.9	0.0±1.7	0.9±3.6	-0.2±1.6	0.0±1.5	0.1±2.4
max	-0.4±2.3	–	0.8±3.9	-0.6±2.3	1.3±3.3	0.3±1.0	0.0±2.4	0.5±0.8	0.2±2.6	-0.4±2.5	0.5±3.8	-0.6±2.3	-0.4±2.0	-0.3±3.3
QPointQ=25	-1.4±3.8	-1.0±4.1	–	-1.6±3.5	0.3±5.5	-0.7±4.6	-0.9±4.0	-0.6±4.5	-0.8±3.9	-1.4±3.6	-0.5±4.9	-1.6±3.5	-1.4±3.3	-1.3±3.8
QPointQ=50	0.2±1.5	0.6±2.2	1.5±3.2	–	1.8±4.0	0.9±2.7	0.6±1.6	1.0±2.7	0.8±1.5	0.2±1.4	1.1±3.2	0.0±0.8	0.2±0.6	0.3±1.9
QPointQ=75	-1.9±4.5	-1.4±3.8	-0.6±5.7	-2.0±4.4	–	-1.1±3.8	-1.4±4.5	-0.9±3.7	-1.3±4.7	-1.8±4.4	-0.8±2.2	-2.0±4.4	-1.8±4.2	-1.7±4.9
QPLinearQ=50,75	-0.8±3.0	-0.4±1.0	0.5±4.5	-1.0±2.9	0.9±3.3	–	-0.3±2.7	0.1±0.6	-0.2±3.1	-0.8±3.0	0.1±4.1	-1.0±2.8	-0.8±2.5	-0.7±3.7
QPSqrtQ=50,75	-0.5±2.1	-0.1±2.3	0.8±3.7	-0.7±1.7	1.2±4.0	0.3±2.6	–	0.4±2.7	0.1±1.1	-0.5±1.8	0.5±3.5	-0.6±1.4	-0.4±1.6	-0.4±1.7
QPLinearQ=5,95	-0.9±3.0	-0.5±0.8	0.4±4.4	-1.1±2.9	0.8±3.3	-0.1±0.6	-0.4±2.8	–	-0.3±3.2	-0.9±3.1	0.0±4.1	-1.1±2.9	-0.9±2.6	-0.8±3.8
QPSqrtQ=5,95	-0.6±1.9	-0.2±2.5	0.7±3.6	-0.8±1.5	1.1±4.2	0.1±3.0	-0.1±1.1	0.2±3.1	–	-0.6±1.3	0.3±3.5	-0.8±1.6	-0.6±1.4	-0.5±1.1
QRangeQ=50,100	-0.0±1.7	0.4±2.4	1.3±3.3	-0.2±1.4	1.6±4.0	0.7±2.7	0.4±1.7	0.8±2.9	0.6±1.3	–	0.9±3.2	-0.2±1.2	0.0±1.1	0.1±1.6
QRangeQ=25,75	-1.1±3.9	-0.7±4.1	0.2±5.0	-1.2±3.6	0.7±2.1	-0.3±4.4	-0.6±3.9	-0.2±4.4	-0.5±3.9	-1.0±3.7	–	-1.2±3.6	-1.0±3.6	-0.9±4.0
QRangeQ=5,95	0.1±1.5	0.6±2.2	1.4±3.2	-0.0±0.8	1.8±3.9	0.9±2.6	0.6±1.4	1.0±2.7	0.7±1.5	0.2±1.2	1.1±3.1	–	0.2±0.8	0.3±1.9
QRLinearQ=0,25,75,100	-0.1±1.5	0.4±1.9	1.2±3.0	-0.2±0.6	1.6±3.7	0.7±2.4	0.4±1.5	0.8±2.5	0.6±1.3	-0.0±1.1	0.9±3.1	-0.2±0.8	–	0.1±1.7
QRSqrtQ=0,25,75,100	-0.2±2.4	0.2±3.2	1.1±3.5	-0.3±1.9	1.5±4.4	0.6±3.4	0.3±1.6	0.7±3.6	0.5±1.1	-0.1±1.6	0.8±3.6	-0.3±1.9	-0.1±1.8	–

Table 7.9: Pairwise matched-pairs workload performance comparison for `gzip`

Reward	mean	max	QPointQ=25	QPointQ=50	QPointQ=75	QPLinearQ=50,75	QPSqrtQ=50,75	QPLinearQ=5,95	QPSqrtQ=5,95	QRangeQ=50,100	QRangeQ=25,75	QRangeQ=5,95	QRLinearQ=0,25,75,100	QRSqrtQ=0,25,75,100
mean	–	0.0±0.8	-0.0±0.8	-0.3±0.7	0.2±0.8	-0.0±0.6	0.3±1.1	0.3±0.7	-0.2±0.5	-0.1±0.7	-0.3±0.8	0.2±0.9	-0.2±0.8	0.4±0.9
max	-0.1±0.8	–	-0.1±0.7	-0.3±1.0	0.2±0.7	-0.1±0.9	0.2±1.3	0.3±0.6	-0.2±0.5	-0.1±1.0	-0.3±1.0	0.1±1.0	-0.2±0.9	0.3±0.8
QPointQ=25	0.0±0.8	0.1±0.7	–	-0.3±0.9	0.2±0.6	0.0±0.6	0.3±1.0	0.4±0.8	-0.2±0.7	-0.0±1.0	-0.2±1.0	0.2±1.1	-0.2±1.0	0.4±1.2
QPointQ=50	0.3±0.6	0.3±1.0	0.2±0.8	–	0.5±0.8	0.2±0.7	0.5±1.2	0.6±0.8	0.1±0.9	0.2±0.5	-0.0±0.7	0.4±1.2	0.1±0.5	0.6±1.2
QPointQ=75	-0.2±0.8	-0.2±0.7	-0.3±0.6	-0.5±0.8	–	-0.3±0.8	0.0±1.2	0.1±0.6	-0.4±0.8	-0.3±0.9	-0.5±1.0	-0.1±1.2	-0.4±1.0	0.1±1.1
QPLinearQ=50,75	0.0±0.6	0.1±0.9	-0.0±0.6	-0.3±0.7	0.2±0.8	–	0.3±0.8	0.4±1.0	-0.2±0.7	-0.0±0.8	-0.2±0.8	0.2±1.0	-0.2±0.9	0.4±1.0
QPSqrtQ=50,75	-0.3±1.1	-0.2±1.3	-0.3±1.1	-0.5±1.2	-0.0±1.2	-0.3±0.8	–	0.1±1.4	-0.5±1.1	-0.3±1.3	-0.5±1.1	-0.1±1.2	-0.4±1.2	0.1±1.3
QPLinearQ=5,95	-0.4±0.8	-0.3±0.6	-0.4±0.8	-0.6±0.8	-0.1±0.6	-0.4±1.0	-0.1±1.4	–	-0.5±0.8	-0.4±0.9	-0.6±1.1	-0.2±1.2	-0.5±0.9	0.0±1.0
QPSqrtQ=5,95	0.2±0.5	0.2±0.5	0.2±0.7	-0.1±0.9	0.4±0.8	0.2±0.6	0.4±1.1	0.5±0.8	–	0.1±0.8	-0.1±0.9	0.3±0.9	0.0±0.8	0.6±0.8
QRangeQ=50,100	0.1±0.7	0.1±0.9	0.0±1.0	-0.2±0.5	0.3±0.9	0.0±0.8	0.3±1.2	0.4±0.9	-0.1±0.8	–	-0.2±0.6	0.2±1.1	-0.1±0.6	0.4±0.9
QRangeQ=25,75	0.3±0.8	0.3±1.0	0.2±1.0	-0.0±0.7	0.5±1.0	0.2±0.7	0.5±1.1	0.6±1.1	0.1±0.9	0.2±0.6	–	0.4±1.0	0.1±0.7	0.6±0.9
QRangeQ=5,95	-0.2±0.9	-0.1±1.0	-0.2±1.2	-0.4±1.2	0.1±1.2	-0.2±1.0	0.1±1.2	0.2±1.2	-0.4±1.0	-0.2±1.1	-0.4±1.0	–	-0.3±0.9	0.2±0.8
QRLinearQ=0,25,75,100	0.2±0.8	0.2±0.9	0.2±0.9	-0.1±0.5	0.4±1.0	0.2±0.8	0.4±1.2	0.5±0.9	-0.0±0.8	0.1±0.6	-0.1±0.7	0.3±0.9	–	0.6±1.0
QRSqrtQ=0,25,75,100	-0.4±1.0	-0.3±0.8	-0.4±1.2	-0.6±1.2	-0.1±1.1	-0.4±1.0	-0.1±1.3	-0.0±1.0	-0.6±0.8	-0.4±0.9	-0.6±0.9	-0.2±0.8	-0.6±1.0	–

Table 7.10: Pairwise matched-pairs workload performance comparison for `gcc`

Reward	mean	max	QPointQ=25	QPointQ=50	QPointQ=75	QPLinearQ=50,75	QPSqrtQ=50,75	QPLinearQ=5,95	QPSqrtQ=5,95	QRangeQ=50,100	QRangeQ=25,75	QRangeQ=5,95	QRLinearQ=0,25,75,100	QRSqrtQ=0,25,75,100
mean	–	-0.3±0.8	**-0.5±0.5**	**-0.5±0.3**	-0.2±0.4	**-0.6±0.6**	**-1.2±0.5**	-0.3±0.8	**-1.3±0.7**	**-0.5±0.3**	**-1.0±0.3**	**-0.5±0.4**	**-0.4±0.4**	**-0.9±0.4**
max	0.3±0.8	–	-0.2±0.9	-0.1±0.8	0.1±1.0	-0.3±0.4	**-0.9±0.7**	-0.0±0.2	**-1.0±0.9**	-0.2±0.7	-0.6±0.8	-0.2±0.9	-0.1±1.1	-0.6±0.7
QPointQ=25	**0.5±0.5**	0.2±0.9	–	0.0±0.5	0.3±0.7	-0.1±0.8	-0.8±0.8	0.1±1.0	-0.9±1.1	-0.1±0.3	**-0.5±0.2**	-0.1±0.3	0.0±0.6	-0.4±0.6
QPointQ=50	**0.5±0.3**	0.1±0.8	-0.0±0.5	–	0.3±0.6	-0.2±0.6	**-0.8±0.5**	0.1±0.9	**-0.9±0.7**	-0.1±0.3	**-0.5±0.3**	-0.1±0.3	0.0±0.4	**-0.4±0.3**
QPointQ=75	0.2±0.4	-0.1±1.0	-0.3±0.7	-0.3±0.6	–	-0.4±0.7	**-1.0±0.6**	-0.2±1.0	**-1.1±0.7**	-0.3±0.6	**-0.8±0.6**	-0.3±0.7	-0.2±0.4	**-0.7±0.6**
QPLinearQ=50,75	**0.6±0.6**	0.3±0.4	0.1±0.8	0.1±0.6	0.4±0.7	–	**-0.6±0.5**	0.3±0.4	**-0.7±0.6**	0.1±0.6	-0.4±0.6	0.1±0.8	0.2±0.8	-0.3±0.5
QPSqrtQ=50,75	**1.2±0.5**	**0.9±0.7**	0.7±0.8	**0.8±0.5**	**1.0±0.6**	**0.6±0.5**	–	**0.9±0.8**	-0.1±0.3	**0.7±0.6**	0.3±0.6	**0.7±0.6**	**0.8±0.6**	**0.4±0.3**
QPLinearQ=5,95	0.3±0.8	0.0±0.2	-0.1±1.0	-0.1±0.9	0.1±1.0	-0.3±0.4	**-0.9±0.8**	–	**-1.0±0.9**	-0.2±0.8	-0.6±0.8	-0.2±1.0	-0.1±1.1	-0.5±0.8
QPSqrtQ=5,95	**1.3±0.7**	**1.0±0.9**	0.8±1.1	**0.9±0.7**	**1.1±0.7**	**0.7±0.6**	0.1±0.3	**1.0±0.9**	–	0.8±0.9	0.4±0.9	0.8±0.9	**0.9±0.7**	0.4±0.5
QRangeQ=50,100	**0.5±0.3**	0.2±0.7	0.1±0.3	0.1±0.3	0.3±0.6	-0.1±0.6	**-0.7±0.6**	0.2±0.8	-0.8±0.9	–	**-0.4±0.1**	0.0±0.3	0.1±0.5	-0.3±0.4
QRangeQ=25,75	**1.0±0.3**	0.6±0.7	**0.5±0.2**	**0.5±0.3**	**0.8±0.6**	0.3±0.6	-0.3±0.6	0.6±0.8	-0.4±0.9	**0.4±0.1**	–	**0.4±0.3**	0.5±0.5	0.1±0.4
QRangeQ=5,95	**0.5±0.4**	0.2±0.9	0.1±0.3	0.1±0.3	0.3±0.7	-0.1±0.8	**-0.7±0.6**	0.2±1.0	-0.8±0.9	-0.0±0.3	**-0.4±0.3**	–	0.1±0.5	-0.3±0.4
QRLinearQ=0,25,75,100	**0.4±0.4**	0.1±1.1	-0.0±0.6	-0.0±0.4	0.2±0.4	-0.2±0.9	**-0.8±0.6**	0.1±1.2	**-0.9±0.7**	-0.1±0.5	-0.5±0.5	-0.1±0.5	–	-0.4±0.5
QRSqrtQ=0,25,75,100	**0.9±0.3**	0.6±0.7	0.4±0.6	**0.4±0.3**	**0.7±0.6**	0.3±0.5	**-0.4±0.3**	0.5±0.8	-0.4±0.5	0.3±0.4	-0.1±0.4	0.3±0.4	0.4±0.5	–

Table 7.11: Pairwise matched-pairs workload performance comparison for `gobmk` (statistically-significant differences in bold)

Variation and Performance Evaluation

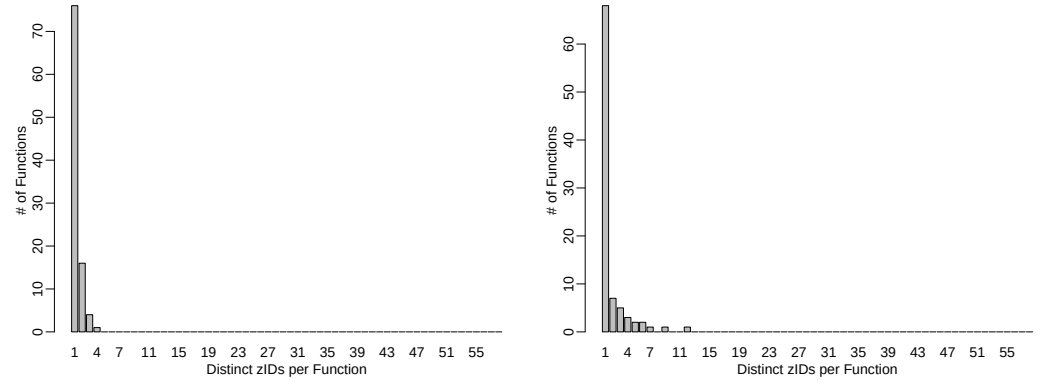
The evaluation results presented in this section are based on a thorough evaluation of each inliner across a moderately-size workload of inputs. Most compiler evaluations in the literature, for both FDO and static code transformations, use a SPEC-style methodology where a single input is used to evaluate performance. However, the results presented here show large performance variations for both an individual inliner across the inputs in $\mathcal{W}$, and between different inliners for the individual inputs, corroborating the findings of previous studies [15, 16] . Large variations are present even for Benefit and Static, which do not use any profile information. Consider if Figure 7.3(a) used only the evaluation of each inliner on avernum. Each FDI inliner would report a single performance value between 0.85 and 0.98 that together imply a total order on the relative quality of those inliners. Benefit would be judged to be more effective than the best FDI inliners, producing more than a 15% improvement. A similar analysis using only proteins-1 for evaluation suggests that all FDI inliners are effectively the same as the default inliner; using sherlock-mp3 suggests that Benefit causes a large performance degradation that is far worse than the impact of FDI. In reality, the performance relationships between inliners and inputs, even for static inliners, is complex and not adequately characterized by any single-input evaluation. Any such analysis is likely to be specific to the evaluation input and not generalizable to other runs.

7.3.3 Equivalent Inlining Outcomes

In total, 59 non-default versions of each program are created for a workload of 15 inputs (each fold of cross-validation creates a different version). If the global hash values of these versions are compared to determine the number of different final inlining outcomes, there are 10 versions of `bzip2`, 45 versions of `gzip`, and 59 versions of `gcc` and `gobmk`. For `bzip2` and `gzip`, many of the single-profile inlinings produce identical decisions. These single-profile groupings contain 2, 5, and 8 of the 15 versions of `bzip2`; and 2, 2, and 8 of the 15 versions of `gzip`. For `bzip2`, 34 of the 14×3=42 versions of the FDI inliners make identical inlining decisions.

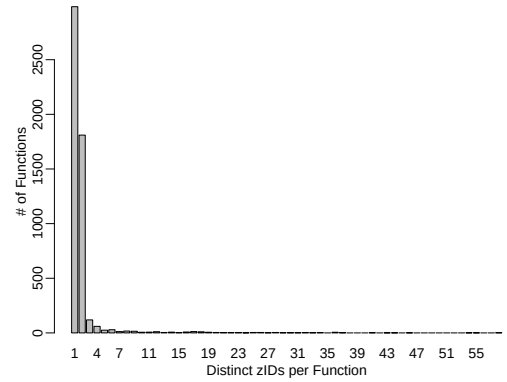
In order to investigate performance differences between FDI inliners, the zID hash values can be compared on a function-by-function basis to eliminate any identical inlining outcomes between two (or more) inliners. Investigating the alternative sets of inlining decisions that produce different versions of a function can inform a compiler designer about the origins of performance differences. However, if many versions of many functions must be investigated, such an approach may be infeasible.

Figure 7.6 presents histograms describing the distribution of the number of distinct versions of each function created by the inliners examined in this section. Bins range from 1 (every inliner produces the same version of the function), to 59 (every inliner produces a different version of the function). The weight in each bin corre-

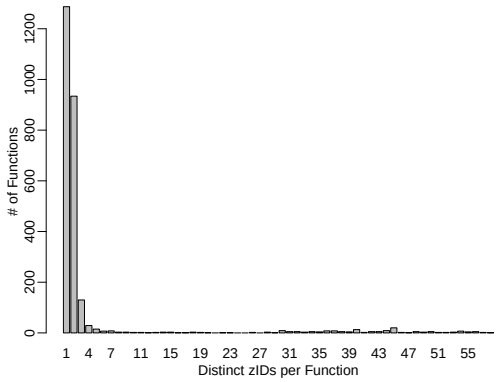


(a) Real `bzip2`: 97 functions

(b) Real `gzip`: 90 functions



(c) SPEC 2006 `gcc`: 5,205 functions



(d) SPEC 2006 `gobmk`: 2,597 functions

Figure 7.6: Distribution of the number of unique, non-zero, zIDs per function across all inliners

sponds to the number of functions in the program for which that number of versions exist across all inliners. The most significant result from the histograms is that the majority of the functions in all programs have a single distinct version after inlining, regardless of the inliner used. When different versions of a function are produced by different inliners, only a few versions are created for most functions. Therefore, even for large programs like `gcc` and `gobmk`, the zID hash values can help a compiler designer identify the differences between alternative inliners by separating the small number of inlining decisions that produce different final code from those that are equivalent across the alternative inliners.

7.4 Conclusion

This chapter reports a thorough evaluation of inlining across a workload of input for four example programs. While results on individual inputs range from nearly a 25% performance degradation to more than a 15% performance improvement, there are no statistically-significant differences in workload performance between any inlining approach for `bzip`, `gzip`, or `gcc`.

However, the Benefit inliner performs particularly poorly for many inputs. This result suggests that the currently-implemented heuristics used to predict the utility of inlining a given call site are ineffective. Inlining is mostly an enabling transformation, but the code-size-focused Benefit(CS) estimation uses only an abstracted and simplified assessment of constant-propagation opportunities. If the estimation of potential inlining benefit were enhanced by considering a wider scope of impact, for instance, potential improvements to alias analysis, by using a more precise analysis on inlining impact, as discussed in Chapter 6.3.5, and/or by changing additional transformations to also use profile information, both FDI and static inlining might be able to reliably improve program performance. Furthermore, an early inliner, perhaps similar to Never, should be included in the transformation sequence discussed in Section 7.1.3. The addition of an early inliner may improve the interaction between inlining and other transformations, while allowing FDI to focus on execution frequency instead of code size.

Therefore, while the results presented here show no statistically-significant performance improvements from inlining, there remain many aspects of inlining in `LLVM` that can likely be substantially improved, and many other transformations that might benefit from the use of combined profiling.

Chapter 8

Related Work

This chapter presents work related to the workload-reduction methodology presented in Chapter 4 and the profile-combination technique presented in Chapter 5. In general, existing approaches to workload reduction use low-level behaviors that are not informative to an FDO compiler, while attempts to combine profiles often combine single-run offline profile with the online profile collected by a JIT, or otherwise fail to capture inter-run behavior variation in a way that is meaningful for an ahead-of-time (AOT) FDO compiler.

8.1 Program Workloads

Input characterization and workload reduction are not new problems. However, the similarity metrics used for clustering in Chapter 4 are unique in their applicability to workload reduction for an FDO compiler. Most input similarity and clustering work is done in the area of computer architecture, where research is largely simulation-based, thus necessitating small workloads of representative programs using minimally-sized inputs. The architectural metrics of benchmark programs are repeatedly scrutinized for redundancy, while smaller inputs are compared with large inputs. Alternatively, some work bypasses program behavior and examines the inputs directly.

Shen and Mao propose the XICL language to allow programmers to formally describe how to extract the important properties of an input directly [88]. A feature selection process removes correlated features, and basic-block frequency counts are predicted using an input-behavior model constructed by regression. However, the programmer must understand the code, the input, and the compiler, in order to hypothesize important features, and then determine a procedure to automatically extract those features from an arbitral input. Furthermore, the system only predicts basic-block execution frequencies. While critical to many current transformations, these frequencies do not inform any value specialization transformations.

Maxiaguine *et al.* examine the variability of input streams for the system-level design of multimedia system-on-chip devices [73]. They reduce the input set to

corner cases and the best-case and worst-case scenarios, the critical concerns of real-time data processing systems. However, input characterization that does not take the run-time characteristics of the program into account does not provide much useful information in the context of FDO.

Most input characterization and workload reduction research aims to reduce the time required for detailed architectural simulation, without compromising the applicability of simulation results. Similarity metrics in this area are based on architecture-level program characteristics (instructions per branch, cache miss rates, *etc.*). Most techniques use clustering to choose a representative subset of the full workload.

KleinOsiwski and Lilja create the MinneSPEC benchmark suite from the SPEC CPU 2000 suite by reducing the sizes of the input data [61]. The reference inputs are truncated, sampled, replaced with the test or train input, or run with a modified command line. However, the authors warn that these reduced inputs do not always conserve all the program characteristics of the original inputs, and should be used with caution.

Vandierendonck *et al.* cluster SPEC CPU 2000 benchmarks based on performance numbers reported on the SPEC web-site [96]. Rank analysis validates the clustering results, but simply predicting the relative performance of machines does not provide enough information to determine the reasons for these performance differences, nor to illuminate opportunities to improve performance in the future.

A prevailing methodology to select representative program-input pairs uses Principal Component Analysis (PCA) to reduce the dimensionality of program characteristics, and then clusters the data in the resulting space. Eeckhout *et al.* employ this methodology with the SPEC CPU 95 and TCP benchmarks [38]. They find that while there is significant redundancy between the program-input pairs, the behavior of some programs is significantly impacted by the choice of input. Phansalkar *et al.* find that the SPEC CPU 2006 benchmark suite [80] is more varied than earlier versions of the suite, though some redundancy still exists.

Alternately, Hoste *et al.* use a correlation reduction technique with a genetic algorithm on microarchitecturally-independent program characteristics [55]. They find that this technique provides superior results to PCA and clustering for emerging benchmarks, while the results are more easily interpreted because the dimensions of the similarity space are the measured characteristics. In particular, two apparently similar programs according to micro-architecturally dependent characteristics may be significantly different, as many different program behaviors can produce similar performance counter values. While this observation is to be expected, its implications are paramount to the study of FDO compilers. Calculated input similarity is dependent on the level where similarity is measured, which must match its intended use. Consequently, it is essential for FDO that input similarity be determined using compiler-level program representations such as CFGs and CGs.

Sherwood *et al.* propose SimPoint, a tool to that identifies representative program phases that can be sampled to make predictions about a full simulation [90].

Phases are detected using Basic-Block Vectors (BBV) containing basic block execution counts for 100 million instruction intervals. Random projection reduces the dimensionality of the BBVs. The Manhattan or Euclidean distance is the similarity metric for K-means clustering. However, sampling does not reduce the number of inputs in a workload.

The methodology for the selection of workload inputs presented in Chapter 4 differs from previous work in several ways. In the preceding works, dimensionality-reducing techniques, such as PCA, are applied before the similarity metric is calculated. The primary motivation for reducing dimensionality is to minimize the correlations between dimensions, and thus reduce bias toward redundant characteristics in the clustering. The methodology presented in Chapter 4 assumes that the number of measured characteristics, *i.e.*, code transformations, is relatively small. Detecting correlations between transformations is one goal of the study. Thus, the correlations between transformations are investigated directly.

In most studies, only the (small number of) inputs provided with the benchmark are considered: The emphasis is the reduction of simulation time for the benchmark suite, and thus the reduction of the number of program-input pairs that must be simulated. The use of program-input pairs not withstanding, the focus of workload reduction for architectural simulation is to select the *programs* that are representative of the suite. The motivation for the workload reduction technique presented in Chapter 4 is finding representative inputs from a large workload *for a particular program*.

Finally, this work takes a uniquely compiler-oriented perspective. The primary point of interest is how training inputs interact with a compiler; how different training inputs result in different code transformations by a profile-directed compiler. While variations in code transformations may change architecture-level program characteristics, these metrics may be too far removed from the compiler to quickly assist designers in their effort to improve the compiler.

8.2 Combining Profile Information Across Runs

Chapter 5 proposes *combined profiling*, a data representation for multi-run profiles based on histograms that store execution frequencies normalized according to program structure. Most compilers take a single-run approach to FDO: a single training run generates a profile, which is used to guide compiler transformations. Some profile file formats support the storage of multiple profiles (*e.g.*, `LLVM`), but when such a file is provided to a compiler, either all profiles except the first are ignored, or a simple sum or average is taken across the frequencies in the collected profiles.

An early attempt to combine profiles is due to Fisher and Freudenberger. They measure instructions per break in control flow and sum profiles to provide better branch prediction [40]. Such summations produce similar results to summing normalized frequencies. While better than single-run profiles, they still yield poor behavior modeling in the presence of multiple program use cases and poor training

input selection.

Krintz and Calder annotate Java bytecode with the optimization decisions made in previous program runs so that the JVM can exploit the benefits of those decisions immediately in subsequent runs [65]. However, this approach largely negates the inherent input-sensitivity of dynamic compilation. Furthermore, the profile information from previous runs is lost, preventing the system from detecting or considering the impacts of cross-run behavior variations. Sandya guides dynamic compilation with an off-line profile, but requires the user to specify a confidence level for the accuracy of that profile [84]. The N hottest methods in the off-line profile are candidates for dynamic compilation, and are compiled when a hotness threshold is exceeded. With a high confidence level, the frequency stored in the off-line profile is used to initialize each method's hotness. As the confidence level decreases, a reduced proportion of that frequency is used. On-line profiling contributes to each method's hotness during execution until a threshold is exceeded and the method is compiled. Thus, the input-sensitivity of the JIT can be maintained by setting a low confidence level for the off-line profile, but this approach largely negates the utility of supplying the off-line profile. Furthermore, this approach does not solve the problem that the off-line profile is taken from a single runs and thus cannot inform the JIT regarding the variability of behaviors across different inputs.

Arnold *et al.*. use histograms to combine the profile information collected by a Java JIT system over multiple program runs [9]. The online profiler detects hot methods by periodically sampling the currently-executing method. After each run of a program, histograms for the hot methods stored in a profile repository are updated. The histogram bins represent the number of time the method is sampled during the execution of the program, and thus represent the length of time spend executing that method in each run. While wall-clock execution time is important for a JIT system in order to amortize time spent compiling code, this concern is not relevant to an AOT compiler. Furthermore, variations in execution time may simply indicate scaled, rather than varying, program behaviors. For instance, the probability that a particular branch is taken is likely to have little correlation with the number of times that the branch is executed during a run. The root of these issues is the use of raw profile information in the histograms, a problem addressed by the hierarchical normalization performed when constructing combined profiles.

Salverda *et al.* model the critical paths of a program by generating synthetic program traces from a histogram of profiled branch outcomes [82]. To better cover the program's footprint, they do an ad-hoc combination of profiles from SPEC training and reference inputs. In contrast, combined profiling and hierarchical normalization provide a systematic method to combine profile information for multiple runs.

Savari and Young build a branch and decision model for branch data [85]. Their model assumes that the next branch and its outcome are independent of previous branches, an assumption that is violated by computer programs (*e.g.*, correlated branches). One distribution is used to represent *all events* from a run; distributions from multiple runs are combined using relative entropy — a sophisticated way to

find the weights for a weighted geometric average across runs. Thus, the model describes the average branch probability of all branches in the program, and how this average varies across inputs. The model cannot provide specific information about a particular branch, which is exactly the information needed by FDO. However, this information is provided by combined profiles because each event is represented separately.

Shen *et al.*. investigate the sensitivity of Java garbage collectors to the inputs given to programs [89]. They find that varying program inputs can dramatically change the impact of garbage collection on program performance, and that the best garbage collector for a program is not consistent across inputs. Furthermore, runs on many inputs are required in order to adequately assess the performance of the individual garbage collectors in order to select the collector that best meets the desired performance goals. These results suggest that a JIT that attempts to select the best garbage collector for the executing program at or near the beginning of execution could greatly benefit from the behavior-variation information collected in a combined profile over many runs.

8.3 Input-Conscious Dynamic Compilation

An alternative to profiling application is to use machine learning techniques to identify input-dependent patters of behavior and to learn effective compilation strategies for these patterns.

Mao and Shen use classification trees to select the optimization level at which methods are compiled by the Jikes RVM [72]. The user must use XICL (see Section 8.1) to specify how to automatically extract important input features. This feature set is automatically pruned to the set of informative features, which is used to build one classification tree for each method. The input-feature vector is computed for each run. Rather than storing profile information, the number of times each method is sampled during a run is used to decide the best optimization level for that method, given the observed input. The feature vector for the input of that run is used with the selected optimization level to update each method's tree, as well as a confidence measure based on accuracy of past predictions. In subsequent runs, the input feature vector is computed near the beginning of execution; if this vector matches a high-confidence decision in a method's tree, the predicted decision (classification) is immediately used to compile the method. For all benchmarks, the worst-case performance is a degradation compared to the default optimization selection mechanisms. For about half the programs, the median speedup is nearly 1.0, *i.e.*, there is no impact. All but 2 programs exhibit performance differences greater than 10% between the best and worst cases, *e.g.*, speedups ranging from 0.85 to 1.8 for Mtrt.

Jiang *et al.* use *seminal behaviors*, program behaviors that are highly correlated with subsequent behaviors, to predict behaviors at runtime [59]. Unsurprisingly, the bounds on loop trip counts calculated before entering a loop are excellent predic-

tors of the frequency of other behaviors inside the loops. Tian *et al.* exploit seminal behaviors in C programs to select between function versions at runtime. Each version is created using a standard single-input FDO compilation [95]. While many runs achieve impressive speedups, worst-case performance suffers by over 5% on average. Traditional FDO produces similar results. These results highlight the fact that FDO performance is sensitive to program inputs and can vary widely across the workload. Furthermore, optimization based on one input frequently reduces performance for some other inputs.

8.4 Conclusion

Existing work in workload reduction is targeted at architectural simulation, and is thus poorly matched to the task of selecting a reduced training workload for an FDO compiler. The clustering technique presented in Chapter 4 is specifically designed for this purpose, and thus measures input similarity using the compiler's own heuristics. On the other hand, combining profile information from multiple runs is often investigated in the context of dynamic FDO. In this context, the profiling done in previous runs is leveraged to quickly focus the actions of the JIT compiler at the start of execution, before run-time profiling is available or reliable. However, no previous work captures inter-run behavior variability in a manner suitable for use by static FDO. Combined profiling provides a cross-run characterization of program behaviors that is appropriate for static FDO.

Chapter 9

Conclusion

This thesis presents an end-to-end investigation of feedback-directed optimization in ahead-of-time compilers, with respect to the issues surrounding input-dependent program behavior. Performance evaluation is sensitive to the data input used in the evaluation, but standard practices typically employ only one testing input. Similarly, the choice of training input(s) used with FDO change the transformation decisions made by a compiler, and consequently also impacts the results of performance evaluation. Furthermore, overlap between the testing and training inputs can reward over-fitting by an FDO compiler, and consequently produce unrealistically-positive performance improvements. These issues are solved in Chapter 3 by employing 3-fold cross-validation in the evaluation of FDO compilers.

Selecting the workload of inputs to use in cross-validated evaluations and to drive compiler development is a challenging problem. Results from Chapter 4 suggest that inputs intuitively judged to be similar by humans can be significantly different in the way they interact with FDO. Therefore, an automated clustering technique, based on the code transformation decisions made by the FDO compiler when informed by an input's profile, is used to select reduced workloads. These reduced workloads enable accurate performance evaluation without the expense of training and testing on the full workload. Furthermore, clustering can also prevent very similar or duplicated inputs in the full workload from rewarding over-fitting and artificially inflating evaluation results.

Cross-validation and reduced workloads allow a rigorous evaluation of FDO where the performance impact of cross-input behavior variation can be assessed. While useful for analysis, post-hoc detection of performance variations across a workload does not provide any mechanism by which the compiler can be made aware of these variations, in order to pro-actively account for input-dependent behavior when making code transformation decisions. Chapter 5 presents combined profiling, a bounded-size profile representation that enables multi-run profile information to be collected from an arbitrary number of training runs. In addition, CP preserves the cross-input distributions of input-varying behaviors. Unlike the single-point values of current profiles, an FDO compiler can query a combined profile to assess the impact of code transformations across the range of observed

behaviors. The FDI inlining framework presented in Chapter 6 demonstrates how these queries can be used. FDI provides reward functions parameterized by either quantile points or quantile ranges. These reward functions are used to estimate the run-time benefits of inlining candidate call sites.

Chapter 7 demonstrates the use of combined profiling with FDI and rigorous cross-validated performance evaluation for inlining in LLVM. The surprising result of this study is that for three of the four case-study applications (bzip2, gzip, and gcc), none of the inliners, including the default static inliner, have any statistically-significant impact on workload performance. In the fourth application (gobmk), only the default inliner provides a small statistically-significant performance improvement, while Benefit, the static estimate of inlining benefit underlying FDI, produces a small statistically-significant performance degradation. Furthermore, for gobmk, rank and pairwise evaluation of the FDI inliners demonstrates that Mean produces worse performance results than any other FDI inliner. For bzip2 and gzip, all inliners display large input-dependent variations in performance, while performance measured using a single input varies considerably from inliner to inliner in many cases. These results suggest two conclusions. First, it is likely that the evaluations of inliners (both FDO and static) in the literature using the standard single-evaluation-input methodology also produce significant performance variations when that input is changed. In fact, any transformations evaluated using single program inputs may potentially be subject to similarly large variations in performance if evaluated on a workload of inputs. Additionally, the frequently abysmal performance of Benefit suggests that there is significant room for improvement in the heuristics used to estimate the impact of inlining a call site in LLVM.

Collectively, the work presented in this thesis represents a significant step forward in the statistical and scientific soundness of the implementation and evaluation of FDO compilers. However, the use of these techniques will continue to pose a challenge to researchers until benchmark suites routinely provide sufficiently-large and diverse workloads of input for their included programs, and refrain from artificially combining or unnecessarily removing program use cases. The variations in behavior and performance between realistic program inputs have been repeatedly demonstrated in this and other work. Failure to include the program-input dimensions in performance analysis demonstrates a blindness to the complexity of rigorous evaluation. Continued acceptance of this practice threatens to delay progress in compiler design by encouraging the reporting of non-generalizable performance results.

Bibliography

[1] Optimzation options – using the GNU compiler collection (GCC). http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[2] Zobrist hashing. http://en.wikipedia.org/wiki/Zobrist_hashing. accessed July 16, 2012.

[3] 11^{th} Computer Olympiad. http://www.cs.unimaas.nl/Olympiad2006/, June 2006.

[4] 14^{th} World Computer-Chess Championship. http://www.cs.unimaas.nl/wccc2006/, June 2006.

[5] Swiss-prot protein knowledgbase. http://www.expasy.org/sprot/, November 2006.

[6] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 72–84, Montreal, Canada, 1998.

[7] Taweesup Apiwattanapong and Mary Jean Harrold. Selective path profiling. In *Program Analysis for Software Tools and Engineering (PASTE)*, pages 35–42, Charleston, South Carolina, 2002.

[8] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo)*, pages 52–64, Boston, Massachusetts, January 2000.

[9] Matthew Arnold, Adam Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 297–311, San Diego, California, October 2005.

[10] Thomas Ball and James R. Larus. Branch prediction for free. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 300–313, Albuquerque, New Mexico, June 1993.

[11] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, July 1994.

[12] Thomas Ball and James R. Larus. Efficient path profiling. In *Intern. Symposium on Microarchitecture (MICRO)*, pages 46–57, Paris, France, December 1996.

[13] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: The showdown. In *Symposium on Programming Languages (POPL)*, pages 134–148, San Diego, California, January 1998.

[14] Paul Berube. Additional FDO inputs. http://www.cs.ualberta.ca/∼berube/compiler/fdo/inputs.shtml.

[15] Paul Berube. *Aestimo*: A feedback-directed optimization evaluation tool. Master's thesis, University of Alberta, October 2005.

[16] Paul Berube and José Nelson Amaral. *Aestimo*: A feedback-directed optimization evaluation tool. In *Intern. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 251 – 260, Austin, Texas, March 2006.

[17] Paul Berube and José Nelson Amaral. Benchmark design for robust profile-directed optimization. In *Standard Performance Evaluation Corporation (SPEC) Workshop*, Austin, Texas, January 2007.

[18] Paul Berube and José Nelson Amaral. Combined profiling: A methodology to capture varied program behavior across multiple inputs. In *Intern. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 210–220, New Brunswick, New Jersey, April 2012.

[19] Paul Berube, José Nelson Amaral, Rayson Ho, and Raul Silvera. Workload reduction for multi-input feedback-directed optimization. In *Code Generation and Optimization (CGO)*, pages 59–69, Seattle, WA, 2009.

[20] Paul Berube, Adam Preuss, and José Nelson Amaral. Combined profiling: Practical collection of feedback information for code optimization. In *Intern. Conf. on Performance Engineering (ICPE)*, pages 493–498, Karlsruhe, Germany, 2011. Work-In-Progress Session.

[21] Paul Berube, Adam Preuss, and José Nelson Amaral. Extended description of the combined profiling methodology. Technical report, University of Alberta, Edmonton, AB, Canada, March 2011.

[22] Christian Bienia and Kai Li. Scaling of the PARSEC benchmark inputs. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 561–562, Vienna, Austria, September 2010.

[23] Rastislav Bodik. *Path-sensitive, value-flow optimizations of programs (program analysis)*. PhD thesis, Pittsburgh, PA, USA, 1999. Chair-Rajiv Gupta and Chair-Mary Lou Soffa.

[24] Rastislav Bodík and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 159–170, Las Vegas, Nevada, May 1997.

[25] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Intern. Symposium on Microarchitecture (MICRO)*, pages 259–269, Research Triangle Park, North Carolina, December 1997.

[26] David R. Chakrabarti and Shin-Ming Liu. Inline analysis: Beyond selection heuristics. In *Code Generation and Optimization (CGO)*, pages 221–232, New York, New York, 2006.

[27] Dhruva R. Chakrabarti, Luis A. Lozano, Xinliang D. Li, Robert Hundt, and Shin-Ming Liu. Scalable high performance cross-module inlining. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 165–176, Antibes Juan-les-Pins, France, October 2004.

[28] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, and Harini Srinivasan. Dependence analysis for java. In *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, pages 35–52, La Jolla, California, August 1999.

[29] Tony F. Chan, Gene H. Golub, and Randall J. LeVeque. Updating formulae and a pairwise algorithm for computing sample variances. Technical Report STAN-CS-79-773, Stanford University, November 1979.

[30] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling C programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 246–257, Portland, Oregon, 1989.

[31] 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, 1992.

[32] Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321, 1991.

[33] Chandra Chekuri, Richard Johnson, Rajeev Motwani, Balas Natarajan, Bob R. Rau, and Mike Schlansker. Profile-driven instruction level parallel scheduling with application to super blocks. In *Intern. Symposium on Microarchitecture (MICRO)*, pages 58–67, Paris, France, December 1996.

[34] Peng-Sheng Chen, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Interprocedural probabilistic pointer analysis. *Transactions on Parallel and Distributed Systems*, 15(10):893–907, 2004.

[35] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 273–286, Las Vegas, Nevada, May 1997.

[36] Robert Cohn and P. Geoffrey Lowney. Hot cold optimization of large Windows/NT applications. In *Intern. Symposium on Microarchitecture (MICRO)*, pages 80–89, Paris, France, 1996.

[37] Standard Performance Evaluation Corporation. SPEC: The standard performance evaluation corporation. http://www.spec.org/.

[38] Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Workload design: Selecting representative program-input pairs. In *Parallel Architectures and Compilation Techniques (PACT)*, page 83, Charlottesville, Virginia, September 2002.

[39] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–95, Boston, Massachusetts, 1992.

[40] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–95, Boston, Massachusetts, October 1992.

[41] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, March 1986.

[42] Ira R. Forman. On the time overhead of counters and traversal markers. In *International Conference on Software Engineering (ICSE)*, pages 164–169, San Diego, California, March 1981.

[43] Grigori Fursin, John Cavazos Michael OBoyle, and Olivier Temam. MiDataSets: creating the conditions for a more realistic evaluation of iterative optimization. In *High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 245–260, Ghent, Belgium, January 2007.

[44] Andrew G. Glen, Lawrence M. Leemis, and John H. Drew. Computing the distribution of the product of two continuous random variables. *Computational Statistics and Data Analysis*, 44(3):451 – 464, 2004.

[45] Nikolas Gloy and Michael D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):977–1027, September 1999.

[46] Darryl Gove and Lawrence Spracklen. Evaluating the correspondence between training and reference workloads in SPEC CPU2006. *Computer Architecture News*, 35(1):122–129, 2007.

[47] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. gprof: A call graph execution profiler. In *Compiler Construction (CC)*, pages 120–126, Boston, Massachusetts, 1982.

[48] SPEC Open Systems Group. SPEC CPU2006 run and reporting rules. http://www.spec.org/cpu2006/docs/runrules.html. accessed June 25, 2012.

[49] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 108–123, Austin, Texas, 1995.

[50] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path profile guided partial dead code elimination using predication. In *Parallel Architectures and Compilation Techniques (PACT)*, page 102, San Francisco, California, October 1997.

[51] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path profile guided partial redundancy elimination using speculation. In *Intern. Conf. on Computer Languages (ICCL)*, pages 230–239, Chicago, Illinois, May 1998.

[52] Richard E. Hank, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Region-based compilation: An introduction and motivation. In *Intern. Symposium on Microarchitecture (MICRO)*, pages 158–168, Ann Arbor, Michigan, 1995.

[53] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*, chapter 7, pages 214–221. Springer Series in Statistics. Springer, 2003.

[54] John M. Hoenig and Dennis M. Heisey. The abuse of power. *The American Statistician*, 55(1):19–24, 2001.

[55] Kenneth Hoste and Lieven Eeckhout. Comparing benchmarks using key microarchitecture-independent characteristics. In *Intern. Symp. on Workload Characterization (IISWC)*, pages 83–92, San Jose, CA, October 2006.

[56] 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: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, 1993.

[57] Intel Corporation. Intel C++ compiler options. ftp://download.intel.com/support/performancetools/c/linux/v9/copts_cls.pdf, 2006.

[58] International Business Machines. Detailed descriptions of the XL Fortran compiler options. http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.xlf101a.doc/xlfcr/opts-details.htm.

[59] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, Feng Mao, Malcom Gethers, Xipeng Shen, and Yaoqing Gao. Exploiting statistical correlations for proactive prediction of program behaviors. In *Code Generation and Optimization (CGO)*, pages 248–256, Toronto, Canada, April 2010.

[60] Hyesoon Kim, José A. Joao, Onur Mutlu, and Yale N. Patt. Profile-assisted compiler support for dynamic predication in diverge-merge processors. In *Code Generation and Optimization (CGO)*, pages 367–378, San Jose, California, March 2007.

[61] AJ KleinOsowski and David J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[62] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 147–158, Orlando, Florida, 1994.

[63] Donald E. Knuth. An empirical study of Fortran programs. In *Software: Practice and Experience*, volume 1, pages 105–133, April/June 1971.

[64] Chandra Krintz. Coupling on-line and off-line profile information to improve program performance. In *Code Generation and Optimization (CGO)*, pages 69 – 78, San Francisco, California, march 2003.

[65] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 156–167, Snowbird, Utah, 2001.

[66] Arvind Krishnaswamy and Rajiv Gupta. Profile guided selection of ARM and Thumb instructions. In *Joint Conf. on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES)*, pages 56–64, Berlin, Germany, June 2002.

[67] Asif Lakhany and Helmut Mausser. Estimating the parameters of the generalized lambda distribution. *Algo Research Quarterly*, 3(3):47–58, December 2000.

[68] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)*, Palo Alto, California, March 2004.

[69] Paul Lokuciejewsk, Fatih Gedikli, Peter Marwedel, Katharina Morik, and TU Dortmund. Automatic WCET reduction by machine learning based heuristics for function inlining. In *Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART)*, 2009.

[70] Grigorios Magklis, Greg Semeraro, David H. Albonesi, Steven G. Dropsho, Sandhya Dwarkadas, and Michael L. Scott. Dynamic frequency and voltage scaling for a multiple-clock-domain microprocessor. *IEEE Micro*, 23(6):62 – 68, November 2003.

[71] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *International Symposium on Microarchitecture (MICRO)*, pages 45–54, Portland, Oregon, 1992.

[72] Feng Mao and Xipeng Shen. Cross-input learning and discriminative prediction in evolvable virtual machines. In *Code Generation and Optimization (CGO)*, pages 92–101, Seatle, Washington, March 2009.

[73] Alexander Maxiaguine, Yanhong Liu, Samarjit Chakraborty, and Wei Tsang Ooi. Identifying "representative" workloads in designing MpSoC platforms for media processing. In *Embedded Systems for Real-Time Multimedia (ESTImedia)*, pages 41– 46, September 2004.

[74] Eduard Mehofer and Bernhard Scholz. Probabilistic data flow system with two-edge profiling. In *Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo)*, pages 65–72, New York, New York, 2000.

[75] David Gordon Melski. *Interprocedural path profiling and the interprocedural express-lane transformation*. PhD thesis, University of Wisconsin, 2002.

[76] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 305–315, Toronto, Canada, June 2010.

[77] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[78] Erez Perelman, Trishul Chilimbi, and Brad Calder. Variational path profiling. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 7–16, Saint Louis, Missouri, 2005.

[79] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Programming Language Design and Implementation (PLDI)*, pages 16–27, White Plains, New York, 1990.

[80] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *SIGARCH Computer Architecture News*, volume 35, pages 412–423, New York, New York, 2007.

[81] Vinodha Ramasamy, Paul Yuan, and Dehao Chen. Feedback-directed optimizations in GCC with estimated edge profiles from hardware event sampling. In *GCC Developers' Summit*, pages 87–101, Ottawa, Canada, June 2008.

[82] Pierre Salverda, Charles Tuker, and Craig Zilles. Accurate critical path prediction via random trace construction. In *Code Generation and Optimization (CGO)*, pages 64–73, Boston, Massachusetts, April 2008.

[83] Alan Dain Samples. *Profile-driven compilation*. PhD thesis, Berkeley, CA, USA, 1992.

[84] S. M. Sandya. Jazzing up JVMs with off-line profile data: Does it pay? *SIGPLAN Notices*, 39(8):72–80, August 2004.

[85] Serap Savari and Cliff Young. Comparing and combining profiles. *Journal of Instruction-Level Parallelism*, 2, May 2000.

[86] Bernhard Scholz, Nigel Horspool, and Jens Knoop. Optimizing for space and time usage with speculative partial redundancy elimination. In *Joint Conf. on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES)*, pages 221–230, Washington, DC, June 2004.

[87] Andreas Sewe, Jannik Jochem, and Mira Mezini. Next in line, please!: Exploiting the indirect benefits of inlining by accurately predicting further inlining. In *Workshop on virtual machines and intermediate languages (VMIL)*, SPLASH '11 Workshops, pages 317–328, Tuscon, Arizona, 2011.

[88] Xipeng Shen and Feng Mao. Modeling relations between inputs and dynamic behavior for general programs. In *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, Urbana, Illinois, October 2007.

[89] Xipeng Shen, Feng Mao, Kai Tian, and Eddy Zheng Zhang. The study and handling of program inputs in the selection of garbage collectors. *SIGOPS Opererating Systems Review*, 43(3):48–61, July 2009.

[90] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, San Jose, California, 2002.

[91] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 22, pages 888–905, August 2000.

[92] Jeff Da Silva and J. Gregory Steffan. A probabilistic pointer analysis for speculative optimizations. In *Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 416–425, San Jose, California, 2006.

[93] Richard L. Sites. Programming tools: Statement counts and procedure timings. *SIGPLAN Notices*, 13(12):98–101, 1978.

[94] Standard Performance Evaluation Corporation. SPEC CPU2006. http://www.spec.org/cpu2006/, August 2006.

[95] Kai Tian, Yunlian Jiang, Eddy Z. Zhang, and Xipeng Shen. An input-centric paradigm for program dynamic optimizations. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 125–139, Reno/Tahoe, Nevada, 2010.

[96] Hahs Vandierendonck and Koen De Bosschere. Experiments with subsetting benchmark suites. In *Workshop on Workload Characterization (WWC)*, pages 55–62, October 2004.

[97] Rajeshwar Vanka and James Tuck. Efficient and accurate data dependence profiling using software signatures. In *Code Generation and Optimization (CGO)*, pages 186–195, San Jose, California, 2012.

[98] David W. Wall. Global register allocation at link time. In *Compiler Construction (CC)*, pages 264–275, Palo Alto, California, 1986.

[99] David W. Wall. Predicting program behavior using real or estimated profiles. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 59–70, Toronto, Canada, June 1991.

[100] Larry Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 2003.

[101] Reinhold Weicker and Kaivalya Dixit. (osgcpu-10955) re: Your question to SPEC about input data selection for benchmarks. Personal email correspondences, July 2004.

[102] H. Yuanjie Y. Chen, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating iterative optimization across 1000 data sets. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 448–459, Toronto, Canada, June 2010.

[103] Reginald Clifford Young. *Path-based compilation*. PhD thesis, Cambridge, MA, USA, 1998. Adviser-Michael D. Smith and Adviser-Paul C. Martin.

[104] Peng Zhao and José Nelson Amaral. To inline or not to inline? Enhanced inlining decisions. In *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, College Station, Texas, October 2003.

[105] Peng Zhao and José Nelson Amaral. Feedback-directed switch-case statement optimization. In *Intern. Conf. on Parallel Processing (ICPP)*, pages 295 – 302, Oslo, Norway, June 2005.

[106] Peng Zhao and José Nelson Amaral. Function outlining and partial inlining. In *Intern. Symp. on Computer Architecture and High Performance Computing (SBAC)*, pages 101 – 108, October 2005.

Appendix A

General Background

Most of the work done by a compiler during the compilation process involves transformations on an internal representation of a program. These transformations are designed to reduce the number of instructions that must be executed to generate correct output. The abstraction of high-level programming languages, convenient language features, and idiosyncratic programming styles provide compilers with tremendous opportunities to improve program efficiency. However, these factors also present challenging analysis problems to identify and exploit those opportunities.

Many code transformations are based on the observation that programs tend to have typical, or expected, dynamic behavior. While unexpected behaviors must always execute correctly, if they occur infrequently there is no need for them to execute quickly. Profiling provides a summary of past program behavior to inform those code transformations that try to predict future program behavior and optimize accordingly.

A.1 Compiler Terminology

Compiler transformations and program analysis require precise terminology to describe the computation performed by a program.

Program analysis often determines which facts hold at each *point* in a program. A point exists:

1. between each instruction in a basic block.
2. at the entry to a basic block, before the first instruction.
3. at the end of a basic block, after the last instruction.

Figure A.1 contains the two BBs from the body of the loop in Figure 2.1. The points in these blocks are identified by the labels *p1* though *p10*.

The definition (*def*) of a variable v occurs when v is the left hand side of a statement. A *use* of v occurs when v appears on the right hand side of a statement. For example, in Figure A.1 statement `S1` is a def of a and a use of both $\&A$ and i. A def of v *kills* any previous def of v that reaches the point preceding the new def v. A def of v at point p_d *reaches* a point p if there exists a directed path from p_d to p that does not contain a def of v. The def of r in `S3` is alive at, and thus reaches, $p4$ and $p5$, but is killed at `S4`, and is dead at $p6$. Points are particularly useful in this situation as they allow for a distinction between the r that is alive at $p5$ and used in

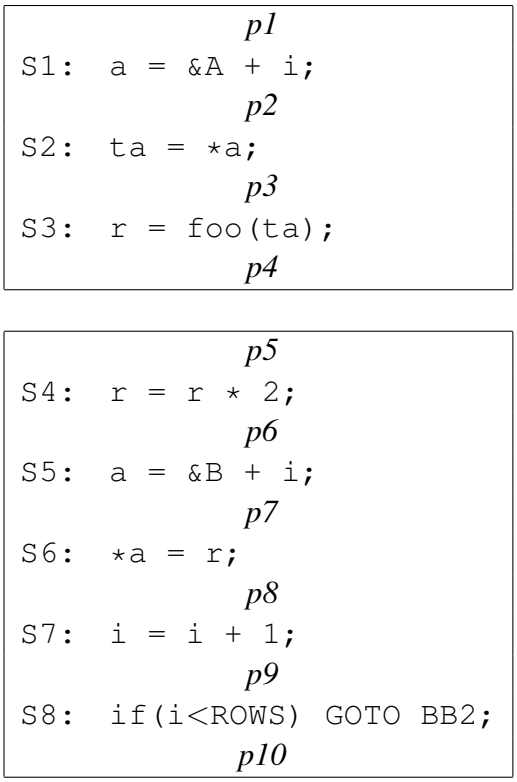


Figure A.1: Points for the loop body from Figure 2.1 (`BB2` and `BB3`)

`S4`, and the r that is defined in `S4` and alive at $p6$. A variable is *dead* at a point p if it has no use on any path in the CFG starting at p. That is, every path from p either contains a def that kills v, or v goes out of scope, before there is a use of v. v is *live* starting at the point after its def until the point where it is dead. The region of the program where v is live is referred to as the *live range* of v. Thus, the live range of the a defined in `S12` is the set of points $\{p2, p3, p4, p5, p6\}$.

An expression is the right-hand side of a statement. The definitions of use and dead for variables also apply to expressions. Furthermore, an expression e is *available* at point p if and only if two conditions hold: First, e is computed on every path from the entry node of the CFG to p. Furthermore, the variables used to compute e must not have changed between the computation of e and p. An expression is *redundant* at point p if it is available on all paths leading to p.

A.2 Computer Architecture

Many code transformations are designed to make better use of processor features. This section provides a brief overview of elements of computer architecture that are significant for code transformations. There is a large degree of diversity among processors, however several basic components of the execution pipeline are common to most processors:

Fetch: The next instruction(s) is (are) fetched from memory. Fetch proceeds sequentially through memory unless the fetch location is modified by a taken branch, procedure call, or procedure return.

Decode: The instruction(s) is (are) interpreted by the processor to determine what type of instruction is being executed (*e.g.*, branch, arithmetic, memory access), and the source registers are read. The condition of a branch may be computed in decode.

Issue: Instructions and operands are routed to the appropriate functional units.

Functional Units: Usually called the execute phase of the pipeline, instructions are processed by functional units. Different functional units usually take a different number of cycles to compute a result. Moreover, the number of cycles required by a functional unit need not be constant (*e.g.*, a load from the memory unit will take longer if the required data is not cached).

Commit: The result from a functional unit is written to the instruction's destination register.

Main memory is slow compared to modern processors and direct access to memory requires many cycles. Therefore smaller, faster, caches are placed between the processor and main memory. Caches exploit both *temporal locality*, repeatedly accessing the same data over a short amount of time; and *spatial locality*, accessing nearby data with greater probability than distant memory locations. Often, there are several layers of caches that become larger and slower over the progression between the processor and main memory. When a memory access occurs at a location that is not stored in a cache a *cache miss* occurs. On a miss, the cache fetches a block of memory containing the requested location (not just the requested location) and replaces a previously obtained block by this new one. A *cache hit* occurs when requested data is already in a cache. One goal of an optimizing compiler is to minimize memory access time by making effective use of these caches; *i.e.*, by maximizing the temporal and spatial locality of memory accesses.

An *instruction cache* (icache) sits between the processor's fetch unit and main memory. Usually, the fetch unit accesses sequential instructions. If an icache miss occurs, the block fetched by the cache will contain not only the requested instruction, but also the next several instructions. In the ideal case for a loop, the entire loop body is fetched into the icache during the first iteration. Thus, subsequent loop iterations do not incur any cache misses.

A *data cache* (dcache) sits between the processors memory access unit and main memory. A dcache performs the same function for data that the icache performs for instructions: A dcache speeds up memory access to sequential memory locations, for example, when processing an array.

Data my be brought into cache before it is requested, or *prefetched*, by means of hardware prediction of memory access patterns or by special prefetch instructions inserted by the compiler. A prefetch may eliminate the cache miss for the first access to a new block of memory. However, prefetching too often or too early can cause the prefetched data to replace cached data that is still being used.

Instruction-level parallelism (IPL) exists in an instruction sequence when instructions are both data and control-flow independent. ILP allows instructions to be executed in an arbitrary order and still maintain program correctness. *Superscalar* architectures are processors designed to exploit ILP by issuing multiple instructions each cycle. They frequently have multiple copies of each functional unit. In order to utilize these functional units, superscalar processors fetch multiple instructions each cycle. Each instruction is individually issued to functional units when a) the appropriate functional unit is available, and b) the processor has determined that the operands of the instruction have been computed. Therefore, compilers for Superscalar architectures must maximize the instruction-level parallelism (data and

control-independent instructions) in the instruction sequence to enable effective use of processor resources.

In contrast to superscalar processors, very large instruction word (VLIW) architectures process instructions in *bundles*. Several instructions are grouped together into a bundle that is fetched, decoded, and issued as a unit. The compiler must ensure that all the instructions in a bundle are independent of each other and that the processor has enough functional units to execute all the instructions at once. As with superscalar processors, a compiler for VLIW processors must find ILP in order to fill bundles with as many useful operations as possible.

Compilers for superscalar and VLIW processors analyze the dependencies between instructions in order to group independent instructions together. However, there is frequently insufficient IPL within a BB to fully utilize the available processor resources. Therefore, many of the code transformations discussed below are designed to increase ILP, either by moving instructions beyond BB boundaries, or by combining BBs into larger regions.

Some processors support *predicated execution*. Predicates are single-bit registers that are set by special test instructions. Every instruction includes a predicate-register argument. When an instruction is executed the indicated predicate register is checked. If the predicate is false the instruction will be *squashed*. That is, the result of the instruction will not be committed. However, the location in the pipeline where the instruction is squashed, and thus the resources consumed by the squashed instruction, are implementation-specific. Regardless of implementation, every instruction must be fetched and decoded before it is possible to check the predicate. In VLIW processors, different instructions in the same bundle may use different predicates. Two instructions may have the same destination register if they use mutually exclusive predicates, as only one of them will commit.

A.3 Code Transformations

Many code transformations benefit from profile information that accurately identifies the hot elements of a program.

A.3.1 Code Placement

Code placement is the problem of finding a linear ordering BBs that can be transformed into a sequence of instructions that can be stored in memory. Pettis and Hansen use edge profiles to place BBs in memory such that the most probable outcome of a branch causes execution to continue at the next memory location rather than jumping to some other address [79]. Consequently, the fetch unit is more likely to continue requesting instructions sequentially, thus reducing the probability of an icache miss.

Procedure placement is the same problem as code placement, but operates at a coarser granularity by creating a linear ordering of the procedures in a program. Gloy and Smith use profiling to measure both the frequency of call graph edges, and also the temporal locality of those calls [45]. Code placement attempts to improves instruction cache efficiency, and thus the code placed adjacent in memory must not only be frequently accessed together, but multiple accesses in succession are required to exploit the cache contents before other code replaces the cached instructions.

A.3.2 Data Flow Analysis and Instruction Scheduling

```
S1:   n=n+1;          S1:   n=n+1;
S2:   m=m+2;          S3:   a=a*2;
S3:   a=a*2;          S5:   b=b+a;
S4:   c=c*2;          S7:   y=b/n;
S5:   b=b+a;          S2:   m=m+2;
S6:   d=d+c;          S4:   c=c*2;
S7:   y=b/n;          S6:   d=d+c;
S8:   x=d/m;          S8:   x=d/m;
S9:   *q=y;           S9:   *q=y;
S10:  *p=x;           S10:  *p=x;
```

(a) Original sequence (b) Scheduled code

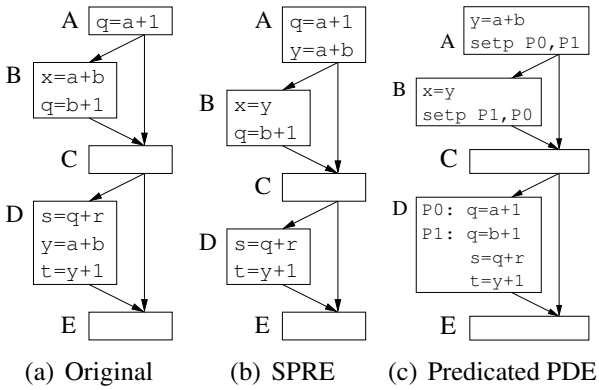
Figure A.2: Local instruction scheduling

Data flow analysis uses the defs and uses in a procedure to determine which defs reach which uses (the flow of values), what the live ranges for each def are, and when a value is dead. These data flow facts define constraints on the ordering of instructions. For example, a def of a value must occur before any uses of the value. These constraints are used by the instruction scheduler to ensure correct program execution and by the register allocator to know which values need to be in a register at each point in the program.

Instruction scheduling is the problem of determining the order to list instructions in the compiled program. Many instructions may be independent of each other. Thus, the compiler is free to arrange them in any order to take advantage of ILP. Local instruction scheduling orders the instructions within a BB, while global instructions scheduling considers larger regions of a program and can move instructions across BB boundaries.

One goal of instruction scheduling is to ensure that the resources required to execute an instruction are free in the processor when the instruction enters the issue phase. If the required resources are not available, the instruction cannot be issued, and may cause the processor to stall until the resources become available. For example, the code in Figure A.2 contains two divisions. Placing two divide instructions is succession, as in Figure A.2(a), on a processor with a single, multi-cycle, functional unit for division guarantees that the second division cannot be issued for several cycles. Thus, other (non-division) instructions should be placed between the two divisions if possible. In the scheduled code in Figure A.2(b), `S2`, `S4`, and `S6` have been moved down between `S7` and `S8`. This change gives `S7` 4 cycles to complete before `S8` requires the division unit.

A second goal instruction scheduling is to ensure that the operands of an instruction are available when the instruction is due to be issued. In Figure A.2(a), `S9` could be moved up above `S8` to further separate the two divisions. However, `S9` requires the value produced by `S7`. While `S8` must wait on `S7` to use the division unit, `S9` must wait the same amount of time for `S7` to calculated the value of y. Thus, moving `S9` up only trades one cause for delay with another. Moreover, this change would reduce the distance between `S8` and `S10`; it is better to leave `S9` in place so it can overlap one cycle of the division in `S8`. Typically, moving long-latency instructions such as memory loads and divisions, in order for them to execute sooner, is one of the greatest priorities for a scheduler.



(a) Original (b) SPRE (c) Predicated PDE

Figure A.3: Speculative PRE and PDE using predication

Speculation is a technique that allows a global instruction scheduler to move instructions above branches that they depend on. By doing so, an instruction that was originally executed conditionally will be executed unconditionally. Instructions that cannot cause exceptions can always be moved above a branch. In this case, the expression calculated by the instruction becomes available on one or more paths that do not use it. Consequently, these path execute more instructions than necessary. However, program execution remains correct. Instructions that can cause exceptions, such as memory accesses or division, require hardware support for speculation. The processor must suppress any exception until it is determined that the excepting instruction would have definitely executed without speculation. More aggressive speculation may move instructions such that additional *compensation code* is needed on other execution paths to ensure program correctness. For example, speculation may make an expression unavailable, requiring paths with uses of that expression to recompute it. Profile information is important for speculation to ensure that instructions scheduled speculatively are from hot paths and that useless instructions and compensation code are only inserted into cold paths. If the profile information is not representative of all important program-execution paths, or an edge profile fails to correctly predict the hot path, speculation may degrade program execution time rather than improving it.

The computation of an expression is *partially redundant* if it is redundant on some paths of execution but not on other. *Partial redundancy elimination* (PRE) is a data-flow analysis and code transformation that *hoists* the computation of an expression higher in the CFG such that it is computed once on all paths. However, an expression will not be moved above a branch if this would cause the expression to become available on any paths where it was not previously available. Doing so would introduce extra instructions on the paths that do not originally calculate the expression. Speculative PRE (SPRE) extends PRE by moving expressions above such branches when the extra instructions are expected to only impact cold paths.

Similarly, an expression is *partially dead* if it has no uses on some path of execution [62]. *Partially dead elimination* is a data-flow analysis and code transformation where the compiler *sinks* partially dead expressions down the CFG until they are only computed on paths where they are used. A *merge point* for an expression occurs at the first point in a BB with multiple incoming edges where the expression is available, but may have different values, on different incoming edges. An expression cannot sink past a merge point as this change would cause incorrect program execution along some paths that would receive the wrong value for the expression. Predication can be used to sink partially dead expressions below merge points.

Figure A.3 shows an example of PRE and PDE. The expression $a + b$ is com-

puted in both B and D. Thus, $a + b$ is partially redundant in B. Because $a + b$ is not fully redundant, PRE cannot hoist the expression upward from D. However, using speculation, $a + b$ can be hoisted into A. Then, the calculation in D can be removed and the calculation of $a + b$ in B can be replaced with a copy. This copy will likely be removed by other code simplification transformations.

The def of q in A is partially dead – the second def of q in B kills it before it is used. However, the def of q in A is not dead along the ACDE path. Predication allows the correct q to be calculated in D, when it is needed. Both definitions of q are removed and replaced by a predicate calculation that encodes the path of execution. setp is an inexpensive instruction to set predicates; the first argument is set to true and the second argument is set to false. In block D, predicated execution calculates the value for q.

SPRE reduces the number of instructions executed on the paths ABCDE and ACDE but increases the number of instructions executed on the ACE path and possibly on the ABCE path. Predicated PDE reduces the cost of instructions along paths ABCDE and ABCE but increase number of instructions along the ACDE path. Furthermore, predicated PDE increases resource utilization in block D. Consequently, even in this simple example, the benefit of these transformations is highly dependent on the frequency with which each path is executed. If ABCDE is the dominant path, then both transformations will reduce execution time. However, if ACDE is the dominant path, SPRE has no impact while PDE is detrimental. Finally, if ACE is the dominant path, the two transformations counteract each other, with the net result of one extra predicate calculation. The code size explosion observed by Scholz *et al.*. attests to the large number of opportunities to apply SPRE. However, every additional instruction represents the potential for increased execution time if the hot path is not predicted correctly. Furthermore, in a case where behavior diversity affects the hot path, the program might run faster on some inputs but significantly slower on others. Consequently, compiler heuristics tend to be conservative and only optimize for "very hot" paths.

Mehofer and Scholz propose probabilistic data-flow analysis, where profile information is used to weight the importance of data-flow facts, such as available expressions for PRE and PDE [74]. They use 2-edge profiles as an approximation to path profiles to calculate the probability with which data flow facts hold at each point in the program. They compare the 2-edge solution to a traditional 1-edge approach and the theoretically optimal solution. While the 2-edge approximation is significantly better than the 1-edge approximation, there are occasions where even the 2-edge approach deviates significantly from optimal solution.

Gupta *et al.*. use path profiles to inform an aggressive PDE algorithm [50]. Predication is used to allow expressions to sink below a merge point in the CFG, similarly speculation allows them to be hoisted above a branch. The predicate allows the expression to be computed only if control flow has taken a path where the expression is not available. Thus, an existing (correct) value is not overwritten. Gupta *et al.*. extend this idea to PRE, using path profiles to inform speculation [51]. In both cases, a cost-benefit framework using path frequencies is used to determine when predication or speculation is profitable. In neither case are experimental results presented. Consequently, the impact of these transformations on execution time is unknown.

Scholz *et al.*. use edge profiles to inform speculative PRE (SPRE) [86]. They use a network flow formulation for cost-benefit analysis that can use any linear combination of factors, including execution speed, code size, or power consumption. Using integer programs from the SPEC 95 benchmark suite they find that optimizing only for speed often results in a code size explosion, up to 56 times larger than non-speculative PRE for the GCC benchmark. However, by including a small factor for space and a large factor for speed in the objective function, space

is reduced nearly as much as optimizing for space alone, while the execution time improvement is the same as when only optimizing for speed.

A.3.3 Register Allocation

Due to the relatively slow speed of memory, most architectures require that all instruction operands reside in registers and that the result of each instruction be stored in a register. *Register allocation* assigns program variables to machine registers. Two variables with overlapping live ranges cannot be stored in the same register and are therefore said to *conflict*. Many register allocators construct a graph to represent conflict relationships and then apply graph coloring to the graph.[1] However, there are frequently more live values than the processor has registers. Consequently, the compiler must *spill* some variables to memory after a def or use, then restore the registers from memory before the next use. Spills are costly because they introduce memory accesses. Therefore, a register allocator attempts to minimize the dynamic number of spills.

Wall proposes a global, inter-procedurally aware register allocator that uses profile information [98]. Inter-procedural register allocation is concerned with reducing the spill code inserted to preserve local variable values across procedure calls or when different procedures access a global variable through different registers. Wall's allocator assumes that all variables reside in memory and then *promotes* variables to registers if they are frequently used. Using the call graph, procedures which are not simultaneously active (*i.e.*, they cannot occur on the same call chain) are identified and grouped together. As with variables whose live ranges do not overlap, these procedures may use the same registers for local variables. Blocks of registers are assigned to these groups of procedures according to the execution frequency – blocks containing the most frequently executed procedures are allocated registers first. Furthermore, the frequency of use of each global variable is used to allocate a register to a global variable instead of to a group of procedures. While these frequencies can be statically estimated, profiling provides more accurate results. For the largest of six small programs, on a machine with fast memory access, profiling improves execution time by 4%. This improvement would be larger on larger programs and on machines with more expensive memory access.

Different use-cases for a program can dramatically change the relative execution frequencies for procedures, or even cause the hot procedures for one input to never execute for a different input. For example, Gove and Spracklen find this to be the case between the training and evaluation inputs for the `wrf` benchmark from the SPEC 2006 suite [46]. In this case, Wall's allocator could assign many registers to procedures that are never called for an input, leaving few registers available to that input's frequently called procedures.

A.3.4 Enlarged Compilation Regions

Procedure calls impose overhead on program execution, but more importantly, they limit the effectiveness of many transformations. For example, in most cases an instruction scheduler cannot move instructions past a procedure call. Chang *et al.*. propose automatic *inlining*, a transformation where the compiler replaces a procedure call with the body of the callee [31]. A context-insensitive call graph profile is used to identify procedure execution frequency. Procedures are sorted by frequency

[1] There are a large number of register allocation algorithms; coloring the conflict graph is the classic technique.

and the hottest procedure is selected for inlining first. A procedure selected for inlining is inlined at all of its call sites in the program. Furthermore, a procedure is not inlined unless all of its outgoing call sites have already been inlined. Inlining stops when a code expansion budget has been consumed.

Arnold *et al.*. present an inlining strategy similar to that used in modern compilers [8]. They use a call-site sensitive call graph profile, thus allocating procedure executions frequencies to individual call sites. Using code size expansion as the cost and call site frequency as the benefit, call sites are inlined in decreasing cost/benefit order up to a code expansion limit. They find that a 1% code size expansion limit accounts for 73% of dynamic calls and reduces execution time by 9% to 57%.

A *superblock* is a region of code designed to improve global instruction scheduling and improve the amount of instruction-level parallelism available for superscalar and VLIW processors [56]. Unlike a BB, a superblock is a single-entry, but multiple-exit region. A superblock is constructed from the BBs along a hot path. Paths are processed in a hottest-first order until all BBs have been included in one superblock. Each branch out of a superblock is a *side exit*. However, it is possible that branches could enter the superblock in the middle of the path. Such *side entrances* would violate the single-entry semantic of a superblock. Therefore, *tail duplication* creates a copy of the remainder of the path from the potential side entrance for use by the cold code. Instruction scheduling for a superblock employs extensive use of speculation. Side exists are ignored and the instructions are scheduled as if the superblock were a basic block. Then, compensation code is added on the side exits in order to maintain program correctness. Due to this aggressive speculation, superblock-formation algorithms expect profile information to inform the identification of hot paths. If the hot paths are not correctly identified, the hot path may take an early side-exit from a superblock. Consequently, not only will the hot path contain the poorly-optimized code in the tail, but may also incur substantial compensation code to correct for speculation along the expected path.

Chang *et al.*. present an early implementation of a C compiler that uses profile information to inform ten classic code transformations applied to superblocks [32]. The profiler provides block profiles, branch probabilities, and a context-insensitive call graph profile. Experiments over a collection of programs show that superblock optimization alone always improves program execution time for a MIPS-R2000 processor; on average a 4% speedup compared to the commercial MIPS C compiler and a 12% speedup compared to GCC. Profile information improves upon that performance by another 15% speedup on average, illustrating the potential for profile-directed optimization to significantly improve performance.

Young uses path profiling to direct the formation of superblocks [103]. Compared to superblocks formed using edge profiles, superblocks formed using path profiles reduce both the cycle count for program execution and the icache miss rate. These benefits are the result of improved instruction scheduling and more useful speculation: The hottest paths were identified more accurately; consequently speculation was correct more often.

Hank *et al.*. propose *region-based compilation* to provide the compiler with a larger scope of instructions than a superblock and thus enable more aggressive code transformation [52]. Region formation assumes that profile information is available to accurately determine the execution frequencies of BBs. Aggressive inlining is first applied to expand the size of procedures. Compilation complexity is then managed by using each region as a compilation unit, rather than each procedure (since only a few large procedures remain). A region is formed by selecting the most frequently executed BB in a CFG that is not yet in a region. Blocks are then selectively added to the region. The region is grown down by selecting the most likely successor of the last block in the region. Likewise, the region is grown upward by selecting the most likely predecessor of the first block in the region.

However, an execution frequency threshold ensures that the region only contains hot blocks. Once the region has been thus grown as much as possible, all sequences of hot blocks leading out of the region are also added to the region. Region-based compilation is effective at increasing the scope available to the compiler, while still breaking compilation into manageable units. However, no performance evaluation is presented. Thus, the execution-time benefits of these larger regions is unknown.

If-conversion is a transformation that converts a branch into a predicate calculation. If the branch condition is calculated as the predicate P, the instructions on the "true" path of the branch are predicated with P while instructions on the "false" path are predicated with the complement of P, $\overline{P}$. The instructions from the true path and false path then are merged into a single path containing all the instructions from both sides of the branch. With the branch removed, the CFG can be recomputed to create a new BB from the old BB containing the branch and its children.

A *hyperblock* is a single-entry, multiple-exit alternative to a superblock designed for regions that are not dominated by a single hot path [71]. Hyperblock formation starts by selecting all the BBs along the hottest path in the region (the region is usually the body of an inner loop but may be other sections of code). Then, additional BBs from side exits are evaluated for inclusion in the hyperblock. The BB inclusion heuristic looks for frequently executed BBs, that are not too large, and that do not contain instructions that hinder optimization (*e.g.*, procedure calls or indirect memory accesses). Cold block are not selected, since merging cold code with hot code will only slow down the hot code. Large BBs have many instructions and use many processor resources. Therefore, they should be avoided because they may impede optimization of the hyperblock. When a BB is selected for inclusion in the hyperblock, if-conversion is applied to remove the side exit and merge the block into the hyperblock. Tail duplication is used to prevent side entrances in the case that the selected block has multiple incoming edges. As with superblocks, profile information is critical for the effective formation of hyperblocks.

Cohn and Lowney state that "the scope of a superblock is too small for effective optimization, even when using techniques that expand their scope such as predication" (*i.e.*, hyperblocks) [36]. Therefore, they propose hot-cold optimization (HCO) to expand a compilation region. HCO duplicates a procedure to creates a hot and a cold version. Code is pruned from the hot version until only the hot BBs remain. All calls to the procedure call this hot version. Any side exits from the code remaining in the hot version jump into the cold version. As with side exits from other enlarged regions, compensation code is added in the cold version to ensure correct execution. Subsequent optimization of the hot version reduces the overall path length by up to 11% for large desktop applications, mostly due to PDE and more efficient register use. However, the results are unclear as to the effect of HCO on the overall execution time of applications.

Ammons and Larus create *hot path graphs* (HPG) to improve data flow optimizations on the hot paths in a CFG [6]. Path profiling is used to identify the hot paths in a procedure. Each hot path is duplicated in the CFG, eliminating all side entrances and exits, to create an HPG. Data flow analysis and code transformations proceed on the HPG using a finite automaton that recognized hot paths. A subsequent pass eliminates duplicated paths that did not benefit from separate optimization. Experiments with SPEC 95 benchmark programs report mixed results. While the dynamic number of instructions with constant operands is always increased, execution time ranges from 4.4% slower to 9.8% faster.

```
main() {
  foo =
&realFoo();
  if(
RARE_COND_1 )
    foo =
&bar();
  if(
RARE_COND_2 )
    foo =
&baz();
}

proc() {
  (*foo)();
}
```

(a) Original code

```
main() {
  foo = &realFoo();
  if( RARE_COND_1 )
    foo = &bar();
  if( RARE_COND_2 )
    foo = &baz();
}

proc() {
  if( foo ==
&realFoo() )
    x = realFoo();
  else
    x = (*foo)();
}
```

(b) Specialized code

Figure A.4: Specialization of an indirect procedure call

A.3.5 Specialization

A specialization transformation replaces a general/unknown quantity with one or more specific/known quantities guarded by checks. The general case is left as a failsafe for instances when the runtime value does not match one of the specialized cases. Value profiling is used to determine the candidates for specialization. Behavior diversity has the potential to be particularly problematic for value specialization, since many programs contain variables that are effectively constants that are set by the input. For example, the size of a data array in a scientific computing application is often directly determined by the size of the input data. This size will remain constant over the execution of any individual input. However, each input may have a different data size.

Figure A.4 shows an example of specialization for an indirect procedure call. `main` sets the function pointer `foo` to one of three procedures. Later, `proc` makes an indirect call through `foo`. However, profiling could show that in most cases `foo` points to `realFoo()`. In that case, the indirect call could be specialized as shown in Figure A.4(b). Making the direct call to `realFoo()` instead of the direct call will have limited immediate benefit on execution time. However, once the target of the procedure call is constant, other transformations such as inlining can be applied.

In object-oriented languages, polymorphism can lead to method calls where the class of the receiver object cannot be determined at compilation time. This situation requires a virtual dispatch mechanism to invoke the procedure in the correct class at runtime. In other languages, indirect calls can similarly prevent the callee from being determined statically. Grove *et al.*propose receiver class specialization [49]. They find that the distribution of possible receivers for virtually-dispatched methods is strongly peaked and stable across different program inputs. Receiver class specialization results in an average speedup of 2 on a collection of C++ and Cecil programs.

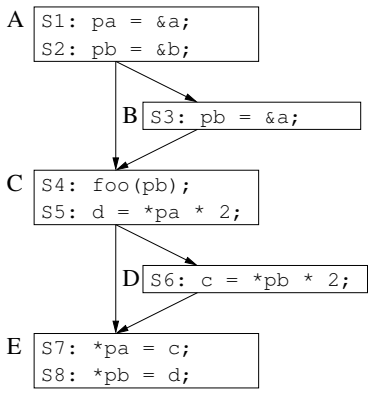


Figure A.5: Pointers and aliases

A.3.6 Alias Analysis

Pointers are variables that contain the address of a memory location and thus facilitate indirect memory access. However, this indirection allows a single location in memory to be accessed by multiple names in the program. Names that refer to the same memory location are said to be *aliases*. Aliases hinder many code transformations, particularly those that rely on data-flow analysis (such as PRE and PDE). *Alias analysis*, or pointer analysis, is a program analysis that determines if pointers definitely alias, definitely do not alias, or may alias. Pointers that definitely do or do not alias do not pose significant problems: Definitely aliasing pointers are simply multiple names for the same memory location; definitely non-aliasing pointers never refer to the same memory location and can be treated independently. However, may-alias relationships prevent the compiler from applying code transformations because it cannot prove that the transformations are safe.

Figure A.5 illustrates some of the complications introduced by pointers and aliases. In block `A`, $*pa$ is an alias for a and $*b$ is an alias for b. Statement `S3` is executed conditionally; at the entry point to block `C`, $*pb$ might be an alias for a and $*pa$, or, it might be an alias for b. In `S4`, pb is passed as a parameter. Whatever address pb holds is said to *escape* into *foo()*. In order to determine the alias relationships that (might) hold after `S4`, the effects of *foo()* on all alias relationships must be known. For some executions, it is possible that both statements in block `E` are equivalent. Likewise, `S6` may be partially redundant with `S5`. Speculation allows a compiler to perform code transformations in the presence of potentially aliasing pointers; for example, SPRE for `S6`. However, the compiler requires additional information to evaluate the benefit of a potential speculation and whether it should speculate that the pointers do, or do not, alias. If $*pb$ is seldom an alias for $*pa$ at `S6`, applying SPRE would be counter-productive.

Therefore, Chen *et al.*. propose inter-procedural *probabilistic pointer analysis* (PPA) [34]. Standard data-flow equations used to calculate alias relationships are augmented with probabilities to calculate the probability that two pointers alias. Inter-procedural analysis is facilitated by mapping the actual parameters of a call to the formal parameters and tracing the callee. Tracing results are summarized as a transfer function, which is cached to keep the analysis tractable. They find that edge profiles improve the average accuracy of the probability predictions compared to static estimates from 76% to 95%. Note that the 24% error in probability estimates is very large, particularly since a non-negligible portion of alias relationships can be determined to definitely alias or definitely not alias, and thus have 0% error. Consequently, aliasing probabilities appear to be of limited reliability without edge

profile information, making speculation a risky proposition.

Da Silva and Steffan propose a PPA analysis using an *inter-procedural control flow graph* (ICFG) which combines the CFGs from every procedure in a program with the context-sensitive call graph. In an ICFG, an edge is added between a call site in one CFG to the entry vertex of the CFG of the caller. A return edge is similarly added from the callee's exit block back to the caller. Each edge must be weighted by execution frequency, either from static estimates or profile information. Analysis traverses the ICFG and at each point assigns a probability to each alias relationship. The aliasing effects of each statement that may impact alias relationships are encoded in a sparse matrix. Simple operations on these matrices propagate the aliasing probabilities and allow the aliasing impact of each procedure to be conveniently summarized. Experiments using the SPEC 2000 integer benchmarks show that the method scales to large programs. Furthermore, a large majority of may-alias relationships have very low probability. This observation makes speculative transformations very promising. Furthermore, while static estimates of edge weights provide quite accurate alias-probability predictions for some benchmarks, using profile information significantly improves the accuracy for several other benchmarks. Surprisingly, relaxed safety conditions in the analysis (which is acceptable for speculation) provide more accurate estimates than a safe analysis, suggesting that the conservative assumptions required to ensure safety introduce many false may-alias relationships.

A.4 Summary

The code transformations described in this chapter are significant contributors to the level of program execution efficiency achieved by modern compilers. However, the list of transformations discussed in this chapter is not exhaustive; many other transformations can benefit from more accurate predictions of program behavior in simple ways that do not warrant publication. The common theme among these transformations is the idea that program execution time can be reduced by predicting what will or will not happen at run time. Many of these transformations make significant assumptions about the stability of these predictions and must be carefully guarded by conservative heuristics. Despite the often-impressive execution time gains these techniques report on benchmarks, behavior diversity is an issue. Consider for example the motivation hyperblocks: The hyperblock is designed for instances where there is no dominant path through a procedure. This situation is a manifestation of behavior diversity.

As processors and programs continue to become more complex, occurrences of problematic behavior diversity are likely to become more common, and their impact more substantial. Thus, the heuristics controlling code transformations must start to consider the implications of behavior diversity. Simply making these heuristics more conservative is not a solution; rather, they must be enhanced to determine how to appropriately apply code transformations both in the presence, and the absence, of behavior diversity.

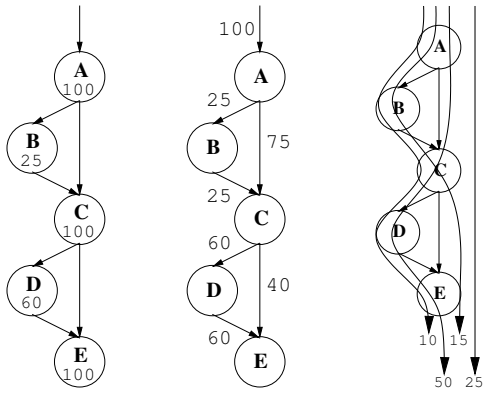
Appendix B

Profiling

B.1 Profiling Techniques

There are many different approaches to profiling, each with strengths and weaknesses. Each technique is particularly suited for certain consumers of the profiling information. A profiler may collect complete information over the run of a program, or may apply a sampling technique to capture a statistically-representative record of program behavior with much lower overhead. In dynamic optimization, overhead is a critical concern. Therefore, sampling is usually accompanied by techniques to focus profiling on the most frequently executed portions of the program. Some researchers have proposed hardware-assisted or full-hardware solutions for profiling with negligible overhead. This form of profiling is increasingly feasible due to hardware performance counters accessible in recent processors. Within this gamut of possibilities, this work only considers profiling as performed by a compiler in the context of statically-optimized code executed in a static execution environment. Thus, both sampled profiling and focused profiling for dynamic optimization, as well as hardware support for profiling, are beyond the scope of this research.

Profiling uses *monitors* to observe program behavior. Since program control flow is often critical to code transformation decisions, monitors may observe branch frequencies (equivalently, BB execution frequencies) to provide weights for the edges of a CFG. Monitors may also observe procedure calls to provide weights



(a) Block (b) Edge (c) Path

Figure B.1: Block, edge, and path profiles for a simple CFG

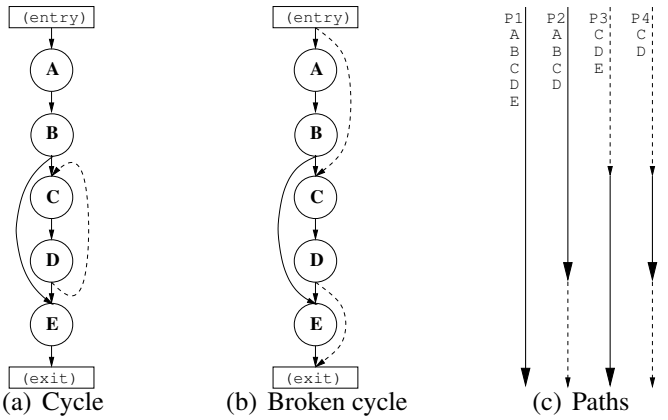


Figure B.2: Acyclic paths for Ball and Larus' path profiling

for the edges of a CG, or observe various other aspects of program behavior to inform code transformations. The simplest monitor is a counter but monitors also include more involved operations such as locating values in a hash table or traversing complex data structures.

Within the context of this research several profiling possibilities exist. This section lays out each form of profiling, while issues related to the efficient implementation of these techniques are discussed in Section B.2.

B.1.1 Vertex and Edge Profiling

Initially, profiling was concerned only with determining the execution frequencies of program instructions. Vertex, or block, profiling is an intuitive solution for this problem. By placing a monitor in each BB of a CFG, each instruction's frequency can be determined directly from the frequency of its BB. Figure B.1(a) shows weights assigned to a CFG using vertex profiling.

An alternative to vertex profiling is edge profiling. An edge profile places a monitor on each edge in the CFG instead of in each vertex. The flow through the edges incident to a vertex provide the execution frequency of the block. Figure B.1(b) shows weights assigned to a CFG using edge profiling that are equivalent to the vertex profile in Figure B.1(a).

As described, both vertex and edge profiling are very inefficient and impose a very large penalty on program execution time due to the excessive insertion of monitor code. Fortunately, as discussed in the next section, techniques exist to substantially reduce the number of monitors required and lower overhead to acceptable levels.

B.1.2 Path Profiling

Along with BB execution frequency, the sequence in which BBs execute is significant for several code transformations. Path profiling, due to Ball and Larus, is a technique to monitor the frequency with which sequences of BBs execute over a run of a program [12]. A basic path is a sequence of BBs that form an acyclic path through the CFG. Figure B.1(c) illustrates the paths through a simple CFG with the path frequencies for the corresponding vertex and edge profiles.

To maintain the acyclic property of the paths, the CFG is converted into a directed, acyclic graph (DAG), as shown in Figure B.2. In the DAG, every path starts

at the entry vertex of the CFG and ends at the exit vertex. Each back edge in the CFG (the dashed arc in Figure B.2(a)) is replaced by 2 edges; one from the entry vertex to the target of the back edge (*i.e.*, the entry point of the cycle) and another from the source of the back edge to the exit vertex (the dashed arcs in Figure B.2(b)). In the DAG, a path through an iteration of the cycle starts at the entry vertex, may optionally pass through other BBs prior to encountering the entry point of the cycle, traverses a path through the cycle to the source of the back edge, and may then optionally traverse other BBs before ending at the exit vertex. These paths are illustrated in Figure B.2(c). Thus, all paths from the CFG are preserved except those that traverse more than one iteration of the cycle. The edges added to the DAG create (all possible) paths that include only one iteration through the cycle, including the paths that are exactly one iteration of the loop. Thus, path profiling does capture all of the execution of the program.

All the possible paths through the DAG are known at compile-time. A numbering algorithm assigns unique, non-negative integers to each edge in the DAG such that accumulating the edge numbers along a path results in a unique sum for each distinct path. Furthermore, these path numbers are compact. If N is the number of paths and pn is the number of some path, all path numbers fall in the range $0 \leq pn < N - 1$. In a naive implementation monitors are placed on each edge of the DAG to accumulate the path number during execution. Additional monitors on each back edge (in the actual CFG) and in the exit vertex increment the appropriate counters in a path frequency table. In Figure B.2, a monitor on the `D-C` back-edge would record occurrences of `P2` and `P4`. A monitor in the exit vertex would record the frequency of paths `P1` and `P3`.

Young proposes an alternate form of path profiling [103]. *General paths* do not preclude back edges and can thus record information about successive iterations of a loop in the same path. Thus, general paths can detect patterns that span multiple loop iterations such as odd/even alternating paths through the loop body. Instead of numbering paths, paths have a length less than or equal to L. Each path is identified by the last L BBs executed. BB identifiers are kept in an L-entry FIFO buffer. Path frequency counts are incremented whenever execution crosses a CFG edge and the FIFO is full or when a path ends at a procedure boundary. Paths will only be less than L BBs long when the FIFO is not full when the path encounters a procedure boundary. Since most of the FIFO is unchanged at each successive edge, efficient data structures to record path frequencies exist. These data structures allow monitors with algorithmic complexity $O(1)$, the same complexity as the other profiling techniques discussed above. Nonetheless, many instructions are required to update the data structures, which introduces substantially more overhead than the additions and counters of the path numbering technique.

Due to the general acceptance of Ball and Larus's technique references to paths and path profiling in the remainder of this document refer to the Ball and Larus definition of paths and path profiling.

Path profiles are more expensive to collect than edge profiles. While Ball and Larus showed that path profiles cannot always be accurately constructed from edge profiles [12], this limitation may be acceptable in practice. Bodik develops and evaluates five algorithms to infer path profiles from edge profiles [23]. The algorithms are progressively more expensive but also progressively more accurate. Each estimated path profile is evaluated to determine an upper and lower bound on the error in the estimate compared to the true path profile. Unfortunately, error is measured with respect to a particular data flow problem[1], but nonetheless average error is only 5% for the best estimator.

In order to address the problem of measuring the accuracy of path profiles es-

[1]The data flow problem is partial redundancy elimination, which is discussed in Appendix A.

timated from edge profiles, Ball *et al.* define *definite* and *potential* path frequencies [13]. Given an edge profile P_e, the definite frequency (or flow) for path p is the minimum frequency that p *must* execute in all path profile that could induce P_e. Similarly, the potential frequency of p is the maximum frequency that p could execute in any of the path profiles that induce P_e. In experiments with the SPEC 95 benchmarks they find that 85% of the execution of FORTRAN programs and 76% of the execution of C programs is definite flow. Consequently, estimating path profiles from edge profiles is usually very effective.

B.1.3 Extended Path Profiles

Several works extend path profiling to include information about loops and procedure calls.

Melski and Reps present the idea of inter-procedural path profiling. They propose combining all the CFGs of all the procedure in a program into a *supergraph*. Ball and Larus's path profiling technique is applied to the supergraph, with a slight change to the numbering algorithm to support the huge number of paths. However, this technique does not support indirect calls and has not been implemented. Consequently, no evaluation is available. Given these limitation, this approach to inter-procedural path profiling is not a feasible solution.

Tallman *et al.* introduce *overlapping paths* (OL paths), an extension of Ball-Larus paths. An $OL-k$ path consists of a standard path with k additional BBs appended to the end. These additional BBs come from the cycle that ended the original path; that is, an $OL-k$ path is a path through a cycle plus k BBs from the next iteration of the cycle. $OL-k$ allows the standard numbering technique to efficiently identify paths while adding the ability to make inferences about program behavior across multiple loop iterations. Furthermore, $OL-k$ paths can be used to profile paths across procedure boundaries. Two forms of this inter-procedural path profiling are supported: A path in the caller plus k blocks of the callee, or a path in the callee that returns plus k blocks after the return in the caller. Experimental results using selected SPEC 2000 benchmarks show that $OL-k$ paths can effectively estimate the paths of two full loop iterations: $OL-k$ paths assign 96% definite flow when k is one third the length of the longest cycle.

Larus develops *whole program paths*, which compactly encode the full dynamic control flow for an execution of a program including loops and paths through procedure calls. Regular path profiling is used to instrument every CFG in the program, with the modification that paths end at edges entering a BB containing a procedure call. Instead of counting the frequency of each path number for a profile, the number of each detected path is emitted as a token in a trace of program execution. The tokens are generated lazily to minimize the alphabet of the trace. Additional tokens are added to the trace for procedure calls and returns. The trace is then processed by an efficient string compressor called SEQUITUR which produces a context-free grammar for the trace. This grammar dramatically reduces the size of the trace, as well as making analysis easier. The grammar is represented as a DAG which has grammar productions as the interior nodes and terminals (paths) as the leaves. A trace through the program can be generated by traversing the DAG. Weights on the nodes of the DAG allow the frequency of path sequences to be determined.

B.1.4 Call-Graph Profiling

Call-graph profiling is another important aspect of profiling. Graham *et al.* develop g-prof, a tool that determines the dynamic call graph of a program and records the edge frequencies for that graph [47]. Furthermore, execution time is assigned

to each function to inform programmers of where their program is spending the most execution time. Periodically sampling the program counter provides a low-overhead method for gprof to determine, as a statistical estimate, what proportion of execution time is spent in which procedure. Time spent in a procedure is reported both including and excluding time spent in any procedure calls. Unfortunately, gprof makes a uniform-time assumption: Total execution time for a procedure is divided evenly among every invocation of the function. Furthermore, in the case of recursion, all procedures involved in the recursion are collapsed into a single call graph node for the purpose of data collection. Execution time is proportioned evenly between the procedures in the collapsed node.

Spivey observes that the uniform time assumption can dramatically reduce the usefulness of information reported by g-prof in programs where shared procedures have different behavior when called from different callers and call sites. Creating a fully context-sensitive call graph allows execution time and frequency to be allocated to procedures based on the call chain. Full context-sensitivity is used rather than a length l context because the "layered" nature of program development often results in situations where the caller in the programmer's code is separated from the utility procedure that performs the majority of the work by a long call chain through library code.

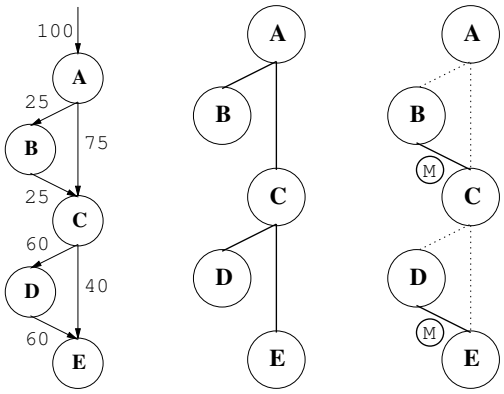
B.1.5 Value Profiling

While program control flow has traditionally been the focus of profiling, value profiling, proposed by Calder *et al.*, facilitates control-flow independent transformations when presented with *semi-invariant* variables [25]. A semi-invariant variable at the instruction level is not constant at compile time, but takes few distinct values during execution. The *Invariance-M* metric characterizes the stability of the values taken by a variable. This metric is calculated for a variable v in statement s as the proportion of the executions of s that are accounted for by the M most frequent values of v. Frequent values are determined by recording the frequency of each observed value in a table. In order to limit the space requirement, the table has a fixed size. Table entries are replaced according to the least frequently used replacement policy (LFU). However, newly observed values may compete with each other as the LFU entry. Thus, the half of the table containing the least frequent values is occasionally flushed. The unused space allows newly observed values time to accumulate frequency before they are a candidate for replacement.

B.2 Profiling Overhead

The overhead incurred to collect a profile is one of the greatest barriers to the practical application of profiling. In this work, monitors are extra instructions inserted by the compiler into the original code and thus impose execution overhead. One of the first compilers to use profiling for a large number of transformations reports that naively-instrumented versions of a programs run 25 to 35 times slower than the un-instrumented versions [32]. In order to minimize this overhead, monitors may be intelligently placed. Such a placement reduces the number of monitors needed and more frequently places those monitors in cold portions of the code. Ball reports that these efficient algorithms reduce the overhead of edge profiling to around 16% and the overhead of path profiling to about 30% [13].

Forman studies the optimal placement of counters to record CFG edge frequencies using an m-graph representation from circuit theory [42]. A foundational contribution of this work is to use existing graph theory results to intelligently place the



(a) Weighted CFG (b) Spanning Tree (c) Cotree

Figure B.3: Finding the minimum instrumentation required for edge profiling

counters. The complement of an undirected spanning tree is called a cotree. Given the cotree for a flow graph and flow values for the edges of the cotree, the law of conservation of flow allows the flow on the edges in the spanning tree to be computed. Therefore, counters need only be inserted to record the execution frequencies of the edges in the complement of the undirected spanning tree of the m-graph for a procedure. Edge frequencies are computed by placing the branch probabilities of the procedure in a Markov process. The matrix of the steady-state solution of the Markov process provides the relative frequencies of all possible edges in the graph. Forman uses these properties to determine dominating cotrees. A cotree C_A dominates cotree C_B if, for any assignment of branching probabilities, the sum of flow on the edges of C_A is less than or equal to the sum of the flow on the edges of C_B. Unfortunately, cotrees seldom dominate each other, necessitating heuristics to select a "good" cotree.

Ball and Larus implement the spanning tree technique for the CFG of a procedure in order to determine the execution frequencies of each BB [11]. They propose using the maximal spanning tree of the CFG, and consequently the minimal cotree, to insert monitors as shown in Figure B.3. For this simple CFG, monitors are only needed on the `B-C` and `D-E` edges. These monitors are indicated in the figure by a circled `M`. Edge weights are assigned to the CFG using static heuristics. Given this weighting, they show that, with the exception of some particular cases of repeat loops and break statements, inserting monitors on the edges of the minimal cotree adds minimal overhead. Furthermore, an edge profile uniquely determines a vertex profile but the reverse is not true. Thus, an optimal edge monitor solution cannot have more overhead than an optimal vertex monitor solution. Furthermore, edge profiles can be collected with similar overhead to that of vertex profiling techniques.

Samples investigates the optimality of monitor placement. He finds that the maximum spanning tree algorithms (*e.g.*, the Ball and Larus technique discussed above) are not optimal because instrumentation does not have a fixed cost [83]. Instrumenting one edge may make instrumentation of another edge more expensive (*e.g.*, by preventing monitor code from moving from the edge into an adjacent BB). Furthermore, the optimality claims of maximal spanning tree approaches hold only for the edge weights estimated when creating the tree. He provides a modified maximum spanning tree algorithm that makes better decisions but is still not optimal. Fortunately, experiments reveal that, in practice, optimality is not a major concern.

The problematic situations that lead to suboptimal solutions arise rarely and account for very little overhead. Finally, Samples reinforces the advantage of edge profiles over vertex profiles. When monitors are inserted by the compiler before code transformations are applied, vertex and edge profiles have nearly identical profile data sizes and profiling overheads. In some cases, edge profiling induces less overhead than the corresponding vertex profiling. Consequently, edge profiles are strictly superior to vertex profiles; modern compilers and recent research never use vertex profiling.

Apiwattanapong and Harrold reduce the instrumentation overhead of path profiling with *selective path profiling* (SPP) in order to enable programs to be profiled after they are deployed at client locations [7]. When the paths in a CFG are partitioned into "interesting" and "uninteresting" paths, SPP uses a modified version of the Ball and Larus path numbering algorithm to assign unique path number to the interesting paths but allows other paths to have non-unique path numbers. Consequently, any instrumentation that would be needed to profile the uninteresting paths can be eliminated. However, while the amount of instrumentation required for SPP is reduced, several new open problems arise. First, how can interesting paths be selected? For a compiler, the interesting paths are the hot paths but accurately identifying hot paths requires profiling. Once the hot paths are determined, additional profiling is unnecessary. Furthermore, if additional profiling of hot paths is desired in the field, removing the instrumentation from cold paths will have little impact on the dynamic overhead imposed by profiling.

Value profiling incurs substantial overhead for each variable profiled. Thus, Calder *et al.* propose *convergence profiling* to intelligently turn off profiling for variables who's profile has reached a steady state [25]. The Invariance-M metric is periodically calculated to test for convergence. If the invariance is increasing, then the variable is judged to be potentially semi-invariant and profiling is continued. Otherwise, the variable is unlikely to be semi-invariant and profiling is turned off. Using this approach, profiling is turned off for 98% of the time that instrumented instructions are executed. Meanwhile, the 5 most frequent values for semi-invariant variables were correctly identified 98% of the time.