The Effect of Profile Choice and Profile Gathering Methods on Profile-Driven Optimization Systems

Geoff Langdale

October 2003

CMU-CS-03-195

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee
Thomas Gross, Chair
Peter Lee
Todd Mowry
Robert Cohn, Intel Research

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

This research was sponsored by the US Air Force Research Laboratory (AFRL) under grant F306029610287 and by a series of grants from the Intel Corporation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Intel, the AFRL, the U.S. government or any other entity.

Keywords: Compilers, Optimization, Performance of systems, Modeling techniques

Abstract

Profile-driven optimization can produce substantial improvements in the quality of code produced by a compiler or link-time optimizer. In this work, we analyze several important aspects of profile-driven optimization. We examine the effectiveness of profile-driven optimization in two commercial-quality optimizers (Digital's GEM compiler and the link-time optimizer 'alto'). We perform analyses to determine how much variability in profile-driven optimization performance results from choosing different training profiles, and to determine how much optimization benefit results from choosing more 'accurate' profiles (that is, profiles that better predict the way that a program is actually run). We examine low-overhead profiling methods such as static estimation (estimating profiles using static heuristics) and statistical sampling (gathering profiles by sampling only a small number of basic block executions). We analyze some profile-driven optimization results in great detail, and show a methodology for accounting for the profile-driven optimization effects of profile data associated with individual functions.

Our results show that profile-driven optimization is effective on average, but unreliable when considering any individual benchmark. Using more accurate profiles is only weakly connected to improved profile-driven optimization performance for most benchmarks. However, low-overhead profiling techniques result in substantial degradations in the reliability and average performance of profile-driven optimization, often to the point of rendering the entire profile-driven optimization process useless. Our analysis also shows that the effects of profile-driven optimization are highly concentrated in the profile data associated with a few functions. Whether profile data improves or worsens the performance of optimized code, it is often possible to attribute the vast majority of this effect to the profile data associated with just a few functions.

Acknowledgements

I'd like to thank Professor Thomas Gross for his unfailing support during this long and arduous process.

I'd like to thank my wife, Dr. Alice Crawford, for her patience and fortitude over the last few months. Her support was absolutely invaluable and all the more impressive given that she earned her own doctorate and gave birth to our son, Marcus Joseph Langdale, all in the last six months.

I'd like to thank my parents, John and Cynthia Langdale, for instilling in me the intellectual curiosity that led to this degree and all their love and support over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Profile-driven optimization is a familiar technique in compiler and link-time optimization. A large proportion of new optimization techniques developed make use of profile-driven optimization; some techniques go so far as to require some form of execution counts to operate.

This thesis does not propose new profile-driven optimizations or new ways to gather profiles. Instead, we analyze the effects of profile-driven optimization and answer (at least for two different optimizers and a handful of benchmarks) some important questions about how profile-driven optimization works in practice.

The findings of this work are that profile-driven optimization is a far less straightforward process than most people would have expected. This lack of straightforwardness manifests itself in several ways:

- Substantial variation in the relative performance of programs that have been optimized using profile-driven optimization (as opposed to optimizing without profile-driven optimization).

- Not-infrequent worsening of performance as a result of using profile-driven optimizations.

- For a majority of benchmarks, a weak or non-existent connection between the accuracy of profiles (where accuracy is defined as ‘a good prediction of future program behavior’) and the profile-driven optimization improvement gained by using the profiles.

- Related to the previous point, surprisingly weak relative performance of profiles that represent ’perfect information’

- A strong locality effect in profile-directed optimization - the bulk of the improvement (or worsening) of performance due to profile-driven optimization can be localized to the profile-driven optimization effects of a few functions' profile data.

We develop a methodology that allows us to discover and quantify these complex effects. While one or more of the above phenomenon have been observed anecdotally and informally in other work, no systematic methodology exists for the measurement and evaluation of this kind of profile-driven optimization performance.

These results are important in a number of ways.

First, these results suggest that profile-driven optimization is not a technique that can be used carelessly - the extra effort involved in choosing training runs and gathering profiles does not guarantee speed-ups, and may accomplish quite the opposite. The use of profile-driven optimization is much like the use of high levels of optimization flags ('-O4' or '-O5'): while it may produce a substantial speed-up, programmer analysis of the outcome of the optimization is necessary.

Second, these results show that the evaluation of new profiling techniques in terms of accuracy alone (that is, without testing the results of using the profiling techniques to optimize real programs) is flawed. If a profiling technique is promoted as being useful for profile-driven optimization, it must be tested in this context.

Third, low-overhead techniques such as sampling and static estimation are shown, on average, to negate much if not all the benefits of profile-driven optimization. Further, statistical sampling adds another source of variability to an already uncertain process.

Finally, the strong locality effects of profile usefulness suggest that future techniques for profiling and profile-driven optimization may be more easily developed and analyzed in the context of fairly small amounts of code. Only a few functions showed significant profile-driven optimization effects in our experiments in Chapter 5.

1.1 Experimentation Framework

1.1.1 Definitions

In the process of profile-driven optimization, a given *run* (deterministic execution of a benchmark program with a certain input) produces a profile that is associated with that run. This profile is then used as input to a profile driven optimizer, and is thus called a *training profile*. The resulting binary can be evaluated with an *evaluation* run. The latter type of run will also have a profile associated with it, the *evaluation profile*, which is the basic block profile that would have resulted from profiling the binary with the evaluation run.

We draw our benchmarks from the SPEC95 and SPEC2000 benchmarks (if a benchmark exists in both benchmark sets, we use the SPEC2000 version and append "2000" to the benchmark name). The SPEC benchmarks define three standard runs, called `ref`, `test` and `train` (each of which can be combinations of multiple program runs). The profile-driven optimizations allowed in the context of SPEC benchmarks involve using `train` as the training run and `ref` as the evaluation run (`test` may only be used for a relatively short-running test of the correctness of a given benchmarking setup). In our work, we use all of the available runs as training and evaluation runs, in all combinations. Where the SPEC benchmarks call for aggregating multiple runs into a single evaluation or training run, we consider each run individually. Thus, instead of testing a single training profile and evaluation run for profile-driven optimization, in our chapter on exact profiling (Chapter 2) we may gather information on as many as 100 possible combinations of training profiles and evaluation runs (we use 10 different evaluation runs and 10 different training profiles for the SPEC2000 benchmark `perl` resulting in 100 possible combinations). More commonly, we have only the three standard SPEC runs available to us and thus gather information on 9 such combinations.

When presenting the names of non-standard SPEC runs (that is, runs that are not simply the SPEC training, testing or reference runs), we will indicate the source of the run as needed. There are two `perl2000` benchmark runs that involve calculation of "perfect" numbers. We refer to the run that is one of the multiple runs in the SPEC reference benchmark as `ref/perfect` and the run that was part of the SPEC training benchmark as `train/perfect`. Generally the names of these runs are not significant and are included only for reference.

We define *profile usefulness* in terms of an evaluation run. That is, it is meaningless to say that profile p_1 is more useful than p_2; only that p_1 is more useful than p_2 with respect to some evaluation run.

Profile accuracy, as measured by one of our profile comparison metrics, measures how well the behavior associated with a training profile predicts the behavior associated with an evaluation profile, strictly in terms of the contents of the two profiles. Once again, accuracy is defined in terms of an evaluation profile. The accuracy of profile p_1 (given a comparison metric) is calculated strictly by comparing the profile data associated with p_1 with the profile data associated with the evaluation run.

1.1.2 Profile-Driven Optimization Platform

We have implemented a system for evaluating profile usefulness and accuracy. This system consists of a set of profile gathering tools, a profile manipulation tool, and two optimization platforms (the `alto`[10]) system and the standard Digital Unix C Compiler[4]) that use the profiles that we gather. All of these tools target the Alpha architecture (the EV5 variant, specifically). The following steps outline the operation of our system.

First, we produce "base" binaries using the Digital Unix C compiler (DEC C V5.6, subsequently referred to as `cc`). The base binaries are unoptimized for the `alto` system (as we want to measure `alto`'s profile-driven optimizations, not the optimizations in the base binary). For the `cc` optimization path, we must compile the base binaries with the same optimization settings that we use for the profile-driven optimization step[1] Thus, the base binaries are more heavily optimized for the `cc` optimization path.

Second, we use `alto` to gather profile information and build a Control Flow Graph (CFG). The base binaries are instrumented by `alto` and used to gather profile information for the various runs of the benchmark.

Third, these profiles and the benchmark's control-flow graph are passed to the profile manipulation tool, which may apply transformations to real profiles or generate new profiles from scratch. The profile optimization tool can generate profiles in `alto` format or in the standard `pixie` format. At this stage we also gather data on profile characteristics and comparisons between profiles.

Fourth, these new profiles are used as inputs to the profile-driven optimization process. These profiles are used with either `alto` (with full optimizations switched on) or the Digital C compiler (see [4] for details of the optimizations performed) to produce an optimized binary. The profile-driven optimizations that provide the most substantial improvements are similar in both optimzers: code placement optimizations, procedure inlining, and super-block formation (profile-driven optimization steps in super-block formation also affect many subsequent optimizations that are not themselves profile-driven). We also produce binaries

[1] Specifically, the `cc` binary is produced with options `"-g1 -O3 -inline speed -ifo -assume whole_program -om"`. Both base binaries use `"-Wl,-r -Wl,-d -Wl,-z -non_shared"` to produce base binaries that `alto` can process; the `alto` optimization path however does not switch on any optimizations.

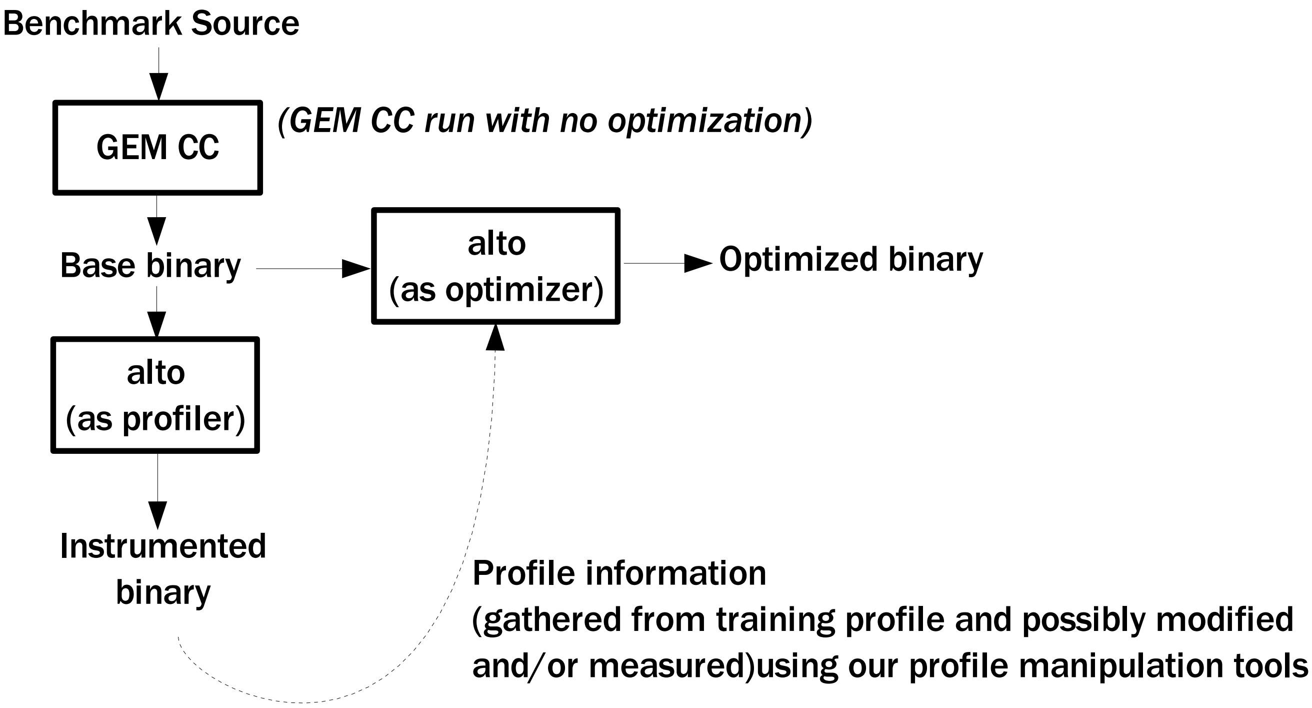

Figure 1.1: Block Diagram of System for Alto Optimizer

with the same set of optimization flags but without using profile information, for purposes of comparison.

Finally, the optimized binary is run [2] . We can compute cycle counts (using the EV5 performance counters) for all our evaluation runs at this time. We are often measuring only subtly different binaries, with very small variations in run-time. We run our benchmarks on a 333Mhz EV5 21164 machine with 1GB of memory (running Digital UNIX V4.0). The machine, while old, has highly accurate performance counters and mature and well-tuned optimizers.

Figure 1.1 and Figure 1.2 summarize the overall process of profile-driven optimization in our two optimizers.

---

[2]Currently, we have some missing data points due to bugs in one or the other of the optimizers, including a large number of the baseline "non-profile-directed optimization case" results. We are also missing some entire benchmarks in the `cc` optimization context. Our results are not significantly altered by restricting the benchmark sets to only those benchmarks that worked across both optimization environments, so we have opted to present more information (the benchmarks that worked only under the `alto` environment) rather than less.

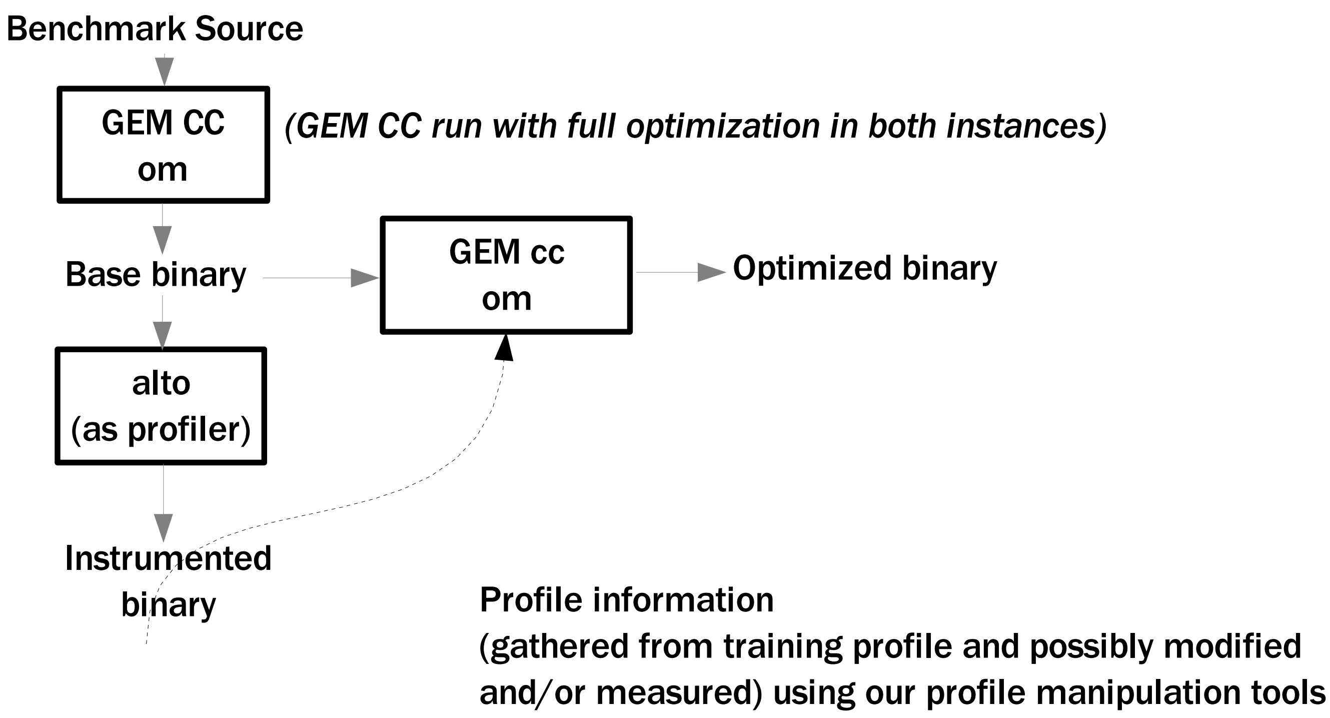

Figure 1.2: Block Diagram of System for CC Optimizer

Our work is not focused on producing peak optimization performance. Our focus is on studying the effects of profile-driven optimization and methods for evaluating its effectiveness, not implementing the fastest possible optimizations. In general, the optimization performance of our system (through either the `alto` path or the `cc` path) is good.

We use the technique of using the evaluation profile as a training profile, a case that we call, after Savari and Young [11], *resubstitution*. While not valid as a practical technique (why run the exact same program execution twice?), resubstitution frequently generates interesting results, allowing us insight into how much benefit results from having "perfect" information. We do not use resubsitution cases when reporting average benefits from using profile-directed optimization.

Our goal is to investigate the usefulness and accuracy of profiles, not to generate superior SPEC results. Our use of non-standard SPEC training profiles and evaluation runs means that our results cannot be considered to be valid SPEC results. This does not render the results invalid in a research sense. As stated above, even the (highly questionable in a benchmarking sense) use of resubstitution can generate interesting data. We carry out analyses to determine whether our observed performance effects from shorter-running evaluation runs than the SPEC "ref" benchmarks represent real effects or whether the effects are simply due to experimental error; the former is true for nearly all combinations of optimizer, benchmark and evaluation run.

1.2 Common Difficulties in Analysis of Profile-Driven Optimization

1.2.1 A Heterogenous, Artificial Population of Benchmarks and Runs

Our analyses of profile-driven optimization are comparatively difficult, in a statistical sense. There is a fundamental problem in attempting to discover consistent rules for the behavior of profile-driven optimization:

There is no 'natural' population of benchmarks, nor is there a 'natural' population of executions of those benchmarks or profiles for those benchmarks.

That is, it is meaningless to talk about taking a random or representative sample of benchmarks or benchmark runs. All benchmarks are man-made creations; these programs have not been built at random or by natural processes. Thus, the task of analyzing the behavior of benchmarks is complicated by the fact that the results we see is heavily influenced by benchmark choice.

We have chosen to deal with this problem in two ways:

- First, we use a fairly large number of benchmarks (taken from both SPEC95 and SPEC2000)[3]. While our results will still be determined by our benchmark choice, we are less likely to have our results biased by the behavior of a single benchmark.
- Second, we attempt to remain aware of the 'contingency' (as opposed to 'necessity') of our results. We carefully qualify our conclusions based on the limitations of our experimental framework.
- Finally, while not every SPEC benchmark runs on both optimizers used in our framework, we used all of the benchmarks which did. The results that we present are not over a 'hand-picked' set of SPEC benchmarks.

1.2.2 Experimental Error and Non-Determinism

Most of our measurements in this work are of program's running time (measured with a cycle counter). Unlike measurements of program static characteristics (such as program size), measurements of deterministic events (number of loads executed in a program) or measurements of a simulation, our measurements are subject to experimental error.

Our benchmarks and runs are all deterministic in the sense that they do not have different behavior at the instruction set architecture level on each execution. That is, every single instruction in repeated runs of the same benchmark run reads and writes the same data across all repeated runs.

However, we cannot guarantee that the instruction cache behavior of the machine is identical, nor can we entirely prevent non-determinism due to other processes or operating system behavior on the machine. We make an effort to keep

[3] We do not use 'extra' benchmark runs - that is, benchmark runs that were not in the set of available SPEC benchmark runs. While it would have been feasible to have created distinct new runs for a small subset of the benchmarks, many of the SPEC benchmarks simply don't have that many distinct ways that they can be run. Worse still, the SPEC benchmarks that would allow easy creation of new distinct runs tend to fall into a single category - that is, interpreters of one kind or another (`perl`, `xlisp`, `m88ksim`), so the process of adding multiple inputs for some benchmarks could systematically bias our results. In addition, the runs we report are standard and familiar to other systems researchers.

this interference to a minimum, but it is not practical to reduce this behavior to zero.

The impact of experimental error on our results is reduced in the usual fashion. We run multiple experiments and eliminate implausible outliers (which in our case seem to nearly always be large slowdowns due to other processes on the machine - we did not observe outliers where the benchmark ran faster). We frequently employ statistical tests to quantify the level of effect that experimental error has had on the results that we have seen.

We make much use of the concept of 'statistical significance'. The reader is reminded that a statistically significant effect is not necessarily a large one; simply an effect that is very unlikely to have occurred as a result of chance.

1.2.3 Limits on Total Experimental Time

Further complicating this work is that we must deal a limited resource - machine time for running experiments. Every separate evaluation of some profile results in another run of one of our optimizers (`cc` or `alto`) for the benchmark in question. For our largest benchmarks, this could take up to 20 minutes.

After a binary has been produced, further experimental time is consumed by the necessity of running each benchmark enough times to reduce experimental error to acceptable levels. There is a tradeoff between total experiment time and minimizing the impact of experimental error. We do not have a single, fixed strategy for making this tradeoff - for some experiments, we are more concerned with quickly exploring a large number of data points, and for others, we are concerned with producing results that are as accurate as possible.

1.3 Outline of Thesis

The remainder of this thesis is organized as follows. In Chapter 2 we evaluate the practice of using 'Exact Profiles' (that is, profiles obtained by gathering accurate basic block profiles of actual executions) for profile-driven optimization.

The next chapters are concerned with techniques that reduce or eliminate the overhead of gathering basic block profiles. In Chapter 3 we discuss generating synthetic profiles through simple static estimation techniques. Chapter 4 analyzes the practice of gathering profiles by statistical sampling.

Chapter 5 details the results of experiments involving systematic manipulation of profile data (on a function-by-function basis). In this chapter we discover that much of the effects of profile-driven optimization can be localized to a small subset of functions.

Chapter 6 summarizes related work, and our conclusions and discussions of future work are found in Chapter 7.

Chapter 2

Exact Profiles

2.1 Introduction

In this chapter, we attempt to discover:

- the effects of using "exact" profiles
- whether the differences we observe between different training profiles are significant
- whether the accuracy of these training profiles is connected to better profile-driven optimization outcomes.

"Exact profiles" are profiles that result from using exact basic block profiling on some execution of the run and are distinct from profiles that are approximations to exact profiles (for example, sampled profiles) or profiles that are wholly synthetic (for example, statically-estimated profile data). Profile-directed optimization (PDO) depends on the assumption that having more accurate predictions of future behavior will result in better optimization performance. Both the assumption of effectiveness of profile-directed optimization, and the assumption of a connection between more accurate profiles and better profile-driven optimization performance, require quantification. In this chapter, we attempt to evaluate both assumptions.

We must ensure that the measurement of differences between the "usefulness" (average cycle counts) of different profiles for the same reference run actually reflects real differences and not experimental error. We summarize the effects at different levels of generality (over individual benchmark runs, over all results for a given benchmark or for an entire optimization context).

We evaluate metrics that attempt to measure how accurately a profile predicts a given future program execution. Suppose we have two training profiles p_1 and p_2 and an evaluation run (which when profiled, produces a profile p_E - all three profiles p_1, p_2 and p_E applying to the original 'base binary'). Suppose also that the binary produced by using p_1 for profile-driven optimization performs better on that evaluation run than the binary produced by using p_2. A "good" profile comparison metric in this case would be one that shows that p_1 is a more accurate prediction of p_E than is p_2; that is, a metric that correlates strongly with profile usefulness.

We must immediately clarify the scope of this chapter. We cannot derive results that apply to all profile-gathering techniques, benchmarks and optimizers. In this chapter we work within the context of profiles that have been directly gathered by basic block profiling (as opposed to approximate methods such as statistical sampling, or profiles that are entirely synthetic, such as those generated by static estimation - we deal with these results in a subsequent chapter). We use two optimizers and a wide range of benchmarks but cannot generalize from these optimizers and benchmark programs to a hypothetical "universe" of benchmarks and optimizers.

We present results for our two optimizers and our benchmarks as well as a methodology for evaluating the effectiveness of profile-directed optimization, for determining the significance of variability in profile-directed optimization performance and for measuring the strength of the connection of profile accuracy and profile "usefulness". We derive somewhat cautionary results concerning the commonly-held assumptions about profile-directed optimization.

Our methodology for analyzing profile-driven optimization performance and its relationship to accuracy is applicable to other optimizers, architectures, benchmark sets, and profiling methods. Applying our methodology to the domain of exact basic-block profiles is the logical starting point for analysis of the relationship between profile usefulness and profile accuracy. Regardless of which profiling methods are available and which are chosen, a training profile must always be selected.

2.2 Usefulness of Profile-Directed Optimization

We gathered cycle counts for each combination of optimizer, benchmark, training profile and evaluation run. We repeated each evaluation run 11 times, discarding the first cycle count score due to significant differences in the first run (no doubt due to virtual memory issues) and calculated average cycle counts from the other 10. We present these average cycle counts normalized by the average cycle counts of the comparison binaries; that is, the optimized binaries that did not use profile-directed optimization. Thus, for a given evaluation run, a binary produced by profile-directed optimization that runs 5% faster than the binary produced by non-profile directed optimization is assigned a score of 0.95 in Table 2.1.

In Table 2.1, we present results showing the relative performance of profile-directed optimization for our different benchmarks as compared to the same benchmarks optimized without profile directed optimization. As each benchmark has multiple evaluation and training runs, we present the average profile-driven optimization performance for all of the combinations of evaluation and training runs, excluding the "resubstitution" case. We do not show "resubstitution" in this table as it could be claimed to represent an unrealistically good case (later in this chapter, we will evaluate the extent to which resubstitution is actually effective).

We provide limited information about the spread of these values (maximum and minimum only), as it is difficult to reasonably compare `ijpeg` (which has only 6 different non-resubstitution training profiles and evaluation run pairs) with `perl2000` (which has 90). To some extent, there is a connection between the range between the maximum and minimum values in Table 2.1 and the total number of non-resubstitution training profiles and evaluation run pairs, as would be expected - as the number of observations of a variable increase, we expect to see the range of the observations increase (at least up to a certain point). However, the issue of causality is not clear: part of the reason that there are in fact more benchmark runs for `perl2000` than `ijpeg` is because there were more reasonable and distinct choices for benchmark runs when the designers of the SPEC benchmarks were selecting benchmark runs.

Overall, profile-directed optimization is an effective technique (an average improvement of 3%), but the results are sharply variable: there are several benchmarks where all training profiles either fail to show any effect or make the program slower for each evaluation run. A majority of benchmarks for both optimizers have at least one combination of training profile and evaluation run where profile-directed optimization performs badly.

These aggregate numbers conceal a great deal of variation - two of the bench-

Optimizer	Benchmark	Number of runs	Normalized execution time		
			Minimum	Maximum	Mean
`alto`	`ammp`	3	0.97	0.98	0.98
	`bzip2`	5	0.87	1.01	0.93
	`compress`	3	0.94	1.06	0.99
	`crafty`	3	0.89	0.93	0.91
	`gap`	3	0.95	0.97	0.95
	`go`	5	0.96	1.06	0.99
	`gzip`	7	1.00	1.14	1.06
	`ijpeg`	3	0.96	0.98	0.97
	`li`	3	0.97	0.99	0.98
	`m88ksim`	3	0.83	1.00	0.89
	`mcf`	3	1.00	1.02	1.01
	`parser`	3	1.00	1.02	1.01
	`perl2000`	10	0.83	1.08	0.96
	`twolf`	3	0.93	1.01	0.97
	`vortex2000`	5	0.86	0.91	0.89
	ALL CASES		0.83	1.14	0.97
`cc`	`ammp`	3	0.99	1.04	1.02
	`bzip2`	5	0.91	1.06	0.96
	`compress`	3	0.92	1.02	0.99
	`crafty`	3	0.94	0.98	0.96
	`equake`	3	0.95	1.01	0.99
	`gap`	3	0.92	0.99	0.95
	`go`	5	0.99	1.14	1.06
	`ijpeg`	3	0.94	0.98	0.96
	`li`	3	0.84	0.92	0.87
	`m88ksim`	3	0.88	1.07	0.96
	`mcf`	3	0.99	1.00	1.00
	`perl2000`	10	0.86	1.13	1.00
	`twolf`	3	0.93	0.98	0.95
	`vortex2000`	5	0.90	0.99	0.94
	ALL CASES		0.84	1.14	0.97

Table 2.1: Execution time of PDO binaries over all evaluation runs and training profiles (each set of evaluation run results normalized such that the non-profile-directed optimization case is equal to 1.0 for each evaluation run).

Optimizer	Benchmark	Evaluation run	Fastest	Slowest	Mean	Std. Deviation
alto	perl2000	train/diffmail	0.90	1.05	0.96	0.0457
alto	perl2000	ref/diffmail	0.90	1.06	0.96	0.0457
alto	perl2000	ref/perfect	0.80	0.96	0.89	0.0446
cc	perl2000	train/scrabble	0.82	1.00	0.93	0.0433
cc	go	ref2	1.00	1.12	1.08	0.0427
cc	go	train	1.01	1.14	1.08	0.0426
cc	go	test	1.00	1.12	1.08	0.0412
alto	perl2000	ref/makerand	0.78	0.93	0.87	0.0408
...	...	...	...	...	...	...
alto	gzip	program	1.13	1.14	1.13	0.0021
alto	parser	ref	1.00	1.00	1.00	0.0020
alto	ijpeg	train	0.98	0.98	0.98	0.0013
alto	ammp	train	0.98	0.98	0.98	0.0010
cc	mcf	ref	1.00	1.00	1.00	0.0009
alto	parser	train	1.01	1.01	1.01	0.0008
alto	ammp	ref	0.98	0.98	0.98	0.0006

Table 2.2: Evaluation runs with highest and lowest variability due to profile-directed optimization profile choice; units are normalized as for Table 2.1.

marks (`vortex2000` and `m88ksim`) receive a 10% or more speedup under `alto` and a 4%-6% speedup under `cc`, while one benchmark (`go`), on average, is slowed by 6% by the use of profile information. This result indicates that while profile-driven optimization is a valuable technique overall, it should not be used blindly. Also interesting is the fact that the "pessimization" cases (under `alto`, `gzip`, `mcf` and `parser`, under `cc`, `ammp` and `go`) were generally bad for nearly all training profile / reference run pairs. Thus, the fact that a benchmark does not benefit much from PDO is usually apparent from a small sample of runs.

Examining the individual benchmark runs, we observe a wide range of performance variability. Table 2.2 presents the top and bottom benchmark runs by profile-driven optimization variability. There is a huge range of variability among evaluation runs.

2.3 The Role of Measurement Error

The cycle counts that we gather for each measurement of profile-driven optimization results are not precise. There is some indeterminacy in the process of using the cycle counters, in addition to variations in machine behavior caused by factors outside the experimenter's control. To minimize the impact of experimental error, we take the mean of several repeated measurements. We must pay rather more attention to measurement error than is usual in systems research.

First, this is because we are attempting to measure small differences in average cycle counts, as discussed above. Second, because we are interested in modelling the behavior of programs over a wide range of profiles and runs, we must often execute the benchmarks with comparatively brief benchmark runs. The usual practice in systems research is to measure the standard SPEC reference runs, which, depending on hardware, can often take as long as 10 minutes. In such a context, a error in cycle count that corresponds to a half second is trivial. However, if such an error were to be introduced into our cycle count measurements for shorter runs, our ability to distinguish small differences in profile-driven optimization performance from different training profiles would be greatly reduced.

We gather multiple cycle counts (for each combination of training profile and evaluation run) and calculate the average cycle counts. The average cycle counts that result from using different training profiles on the same evaluation run will differ. We need to calculate whether those differences are significant or whether they result from experimental error. This issue is somewhat more pressing for this work than it is for more conventional profile-driven optimization research, as some of the runs which we were using as evaluation runs were comparatively brief (as compared to the standard SPEC "`ref`" runs).

2.3.1 One-way ANOVA Results: Are There Usefulness Differences Between Training Profiles?

The one-way ANOVA procedure ("one-way" because we vary only a single variable; "ANOVA" is short for "ANalysis Of VAriance") attempts to determine, given a set of experimental results gathered at different 'levels' (in this case, using different training profiles), whether there are statistically significant differences among the results for different levels. That is, we attempt to disprove the null hypothesis that the average cycle counts for a given evaluation run are the same regardless of which training profile was used. If the probability that this could be the case is sufficiently low, we can reject this null hypothesis and conclude that in fact there are statistically significant differences between the profile-driven optimization effects of different training profiles.

The use of ANOVA depends somewhat on several assumptions:

- Independence of observations

  This assumption is met. The results of running a binary produced with one training profile are not affected by the results of running a binary produced with another training profile. We attempt to eliminate any paging effects by discarding the first measurement for each binary and evaluation run; there is no other way that one binaries' results could affect another's.

- Normality of data

  This assumption is not met. However, the failure to meet this assumption is not considered critical for the use of ANOVA as long as the size of the groups being compared is equal and that the data does not contain any extreme observations. Both of these conditions are met.

- Equality of variance

  This assumption is not met either. Again, this assumption is not considered critical as long as the group sizes are equal and the "no extreme observations" condition is met.

Table 2.3 shows only the significance value for each context, benchmark and reference run. This value is the probability, given no relationship between a training profile and the resulting cycle counts of a binary executing the reference run, that we could have observed the ratio of variability attributed to experimental error versus to the variability attributed to using different training profiles. Thus, a low significance value in this table indicates that there is a low probability that there are no real differences between the usefulness of the profiles for different benchmarks. It does not, however, mean that all training profiles have significant usefulness differences - merely that significant usefulness differences exist somewhere among our set of training profiles.

We were able to reject the null hypothesis of "no significant difference exists between the effect of training profiles" at a significance level of 0.05 (that is, we found that it is no more than 5% likely that, given no effect at all from training profiles, we would have seen the pattern of variability that we did) for

Optimizer	Benchmark	Reference run	ANOVA Significance Value
alto	ammp	ref	0.006
		test	0.000
		train	0.000
	art	ref1	0.223
		ref2	0.811
		test	0.000
		train	0.000
	bzip	ALL	0.000
	compress	ref	0.000
		test	0.056
		train	0.000
	crafty	ALL	0.000
	equake	ref	0.000
		test	0.000
		train	0.039
	gap	ALL	0.000
	go	ALL	0.000
	gzip	ALL	0.000
	ijpeg	ref	0.034
		test	0.000
		train	0.000
	li	ALL	0.000
	m88ksim	ALL	0.000
	mcf	ALL	0.000
	parser	ref	0.000
		test	0.000
		train	0.282
	perl2000	ALL	0.000
	twolf	ALL	0.000
	vortex2000	ALL	0.000
	vpr	ALL	0.000
cc	compress	ref	0.000
		test	0.134
		train	0.000
	ALL others	ALL	0.000

Table 2.3: ANOVA Results Summary

all but 5 evaluation runs (4 under `alto` - two runs in the `art` SPEC2000 benchmark and one run each for `compress` and `parser`, 1 under `cc` - one run under `compress`). For the vast majority of our benchmark runs, the probability that we would have observed the variability that we did due to factors other than the training profile is negligible (under 0.001).

All the values in Table 2.3 reported at 0.000 are no greater than 0.0005 (and are thus rounded to 0.000); these values mean that the chance that we would falsely assume that there were differences between training profiles when in fact the differences we observed were due to experimental error is less than 1 in 2000.

Thus, for nearly all combinations of benchmarks, training profiles and evaluation runs, we can show that using different training runs has a real effect that cannot be explained strictly in terms of experimental error. The only benchmark which has a large number of evaluation runs where we cannot discern much variation in performance due to training profile choice is `art` using `alto` as an optimizer. Two training runs (both of the different `ref` profiles provided) have extremely high significance values; thus the pattern of variation that we observe is almost certainly due to random chance in this case.

2.3.2 Post-hoc ANOVA Results: Which Training Profiles Are Different?

The results in the previous subsection are quite limited. All that doing a simple ANOVA for each evaluation run tells us is that there is some significant difference in average cycle count due to profile choice. If we have 10 different training profiles, we do not know whether:

- there are two groups of 5 profiles, any pair of profiles from different groups being significantly different in optimization usefulness but any same-group pair of profiles not being significantly different, or

- there is 1 profile that has a significantly different effect to the other 9 (which all have no significant differences from each other), or

- all profiles in this group have significantly different effects from each other, or,

- any number of other possibilities.

All we know from ANOVA is that at least one significant difference exists between a pair of profiles. Obviously, we want to know more than this. Specifically, we would like some idea of how many of the training profiles have effects indistinguishable from other training profiles and what sort of "structure" of differences exist between the training profiles. To discover this, we use a "post-hoc analysis". A post-hoc analysis is a technique that is done after ANOVA, and is used to evaluate the significance of differences between the results at different factor levels. It is called a "post-hoc" analysis to distinguish it from a "planned contrast" (a more powerful technique where the experimenter carries out a experiment with a given hypothesis in mind, as opposed to looking for patterns in the data after the experiment has already been carried out). We cannot simply do pair-wise comparisons of means at each factor level - if we had 10 different training profiles, that would mean 45 pair-wise comparisons. Done at a significance level of 0.05, we would be likely to make 2-3 Type I errors in these comparisons (reporting a significant difference where none existed).

Many different post-hoc techniques exist. Some are relatively simple, and involve little more than carrying out multiple t-tests between each pair of factor levels (adjusting the significance level to avoid the above-mentioned problem of proliferating Type I errors). We use a post-hoc technique called Games-Howell, which is considered fairly powerful (that is, less likely to make Type II errors, where a significant difference is overlooked) and more importantly, is robust under circumstances where the variances at each factor level are not equal and/or the distributions in question are not normal.

Games-Howell gives us pairwise estimates of differences between two different factors (in this case, training profiles); and tells us whether or not the range of possible differences (at a given significance level) includes zero. As an example, we will show the output from a single example of Games-Howell, for the benchmark `ammp`, running its reference evaluation run (using `alto` as the optimizer). The output from a Games-Howell post hoc analysis for this case is shown in Table 2.4. Note that the ANOVA significance value for this case was 0.006 - so we are very sure that some significant difference exists between the different profiles.

As can be seen from Table 2.4, a single post-hoc test generates a large quantity of data. The above table shows the mean differences between the performance of the binaries produced by using the differing training profiles, as well as a significance value[1] . This significance value summarizes the probability that, if the true

[1]Note, that in the results of the Games-Howell analysis, the mean differences between two training profiles is an anti-symmetric relation and the significance values are symmetric.

Training Profile 1	Training Profile 2	Mean Difference ($train_1$ - $train_2$) in optimization usefulness (Gcycles)	Significance
`ref`	`test`	2.13	0.218
	`train`	6.56	0.024
`test`	`ref`	-2.13	0.218
	`train`	4.43	0.155
`train`	`ref`	-6.56	0.024
	`test`	-4.43	0.155

Table 2.4: Example Games-Howell Results: `alto`, `ammp`, Evaluation run `ref`

mean difference between the factors (training profiles) was zero (that is, no difference), that we would see the degree of difference between the groups that we did. Thus a low significance value, once again, indicates that it is very unlikely that such a difference would arise from sheer experimental error.

In this particular case, interpretation of the results is fairly simple. At the 0.05 level of significance, there is only one significantly different pair of profiles - the `ref` and `train` profiles. There exist differences between the other pairs of training profiles, but they are not significant at the 0.05 level. The `test` profile is intermediate in usefulness from the other two but its performance cannot be distinguished from either at the 0.05 level. However there is only a 15% chance that if in fact were there no difference in the underlying mean usefulness of `test` and `train`, we would see a performance difference of this magnitude.

Rather than repeating Table 2.4 once for every combination of benchmark, run and optimizer (and remembering that the equivalent table for benchmarks like `perl2000` has 90 lines - one for each ordered pair of training profiles given 10 training profiles), we will merely report the total proportion of distinct training profile pairs - that is, in the above case, the number reported would be 33% (only 2 pairs out of 6 ordered pairs or 1 out of 3 unordered pairs).

We have already found, through ANOVA, that most benchmarks and evaluation runs have significant performance variation due to profile-driven optimization.

The post-hoc analyses summarized in Tables 2.5, 2.6, 2.7 and 2.8 show us that the vast majority of benchmarks and evaluation runs yield significant profile-driven optimization differences between a fair proportion of their profiling input pairs; particularly among those benchmarks and evaluation runs that the original

Optimizer	Benchmark	Evaluation run	Proportion of distinct (unordered) profile pairs
alto	ammp	ref	1/3
		test	2/3
		train	2/3
	art	ref1	2/6
		ref2	0/6
		test	4/6
		train	5/6
	bzip	ref1	8/10
		ref2	7/10
		ref3	8/10
		test	9/10
		train	8/10
	compress	ref	3/3
		test	1/3
		train	2/3
	crafty	ref	3/3
		test	3/3
		train	3/3
	equake	ref	2/3
		test	2/3
		train	1/3
	gap	ref	3/3
		test	3/3
		train	3/3
	go	ref1	4/10
		ref2	8/10
		ref3	8/10
		test	9/10
		train	10/10
	gzip	graphic	20/21
		log	19/21
		program	15/21
		random	19/21
		source	15/21
		test	18/21
		train	20/21
	ijpeg	ref	1/3
		test	2/3
		train	3/3
	li	ref	3/3
		test	2/3
		train	3/3

Table 2.5: Games-Howell Results: Proportion of significantly distinct profile pairs, alto, Part 1

Optimizer	Benchmark	Evaluation run	Proportion of distinct (unordered) profile pairs
alto	m88ksim	ref	3/3
		test	3/3
		train	2/3
	mcf	ref	2/3
		test	2/3
		train	2/3
	parser	ref	2/3
		test	3/3
		train	0/3
	perl2000	ref_diffmail	44/45
		ref_makerand	44/45
		ref_perfect	44/45
		ref_splitmail1	43/45
		ref_splitmail2	44/45
		ref_splitmail3	45/45
		ref_splitmail4	43/45
		train_diffmail	45/45
		train_scrabble	45/45
	twolf	ref	2/3
		test	3/3
		train	3/3
	vortex2000	ref1	9/10
		ref2	9/10
		ref3	7/10
		test	9/10
		train	9/10
	vpr	ref_place	15/15
		ref_route	8/15
		test_place	13/15
		test_route	14/15
		train_place	15/15
		train_route	12/15

Table 2.6: Games-Howell Results: Proportion of significantly distinct profile pairs, alto, Part 2

Optimizer	Benchmark	Evaluation run	Proportion of distinct (unordered) profile pairs
cc	ammp	ref	2/3
		test	3/3
		train	3/3
	bzip	ref1	8/10
		ref2	10/10
		ref3	10/10
		test	10/10
		train	10/10
	compress	ref	3/3
		test	1/3
		train	2/3
	crafty	ref	2/3
		test	3/3
		train	3/3
	equake	ref	2/3
		test	3/3
		train	2/3
	gap	ref	3/3
		test	3/3
		train	3/3
	go	ref1	7/10
		ref2	10/10
		ref3	10/10
		test	9/10
		train	10/10
	ijpeg	ref	3/3
		test	3/3
		train	3/3
	li	ref	3/3
		test	3/3
		train	3/3

Table 2.7: Games-Howell Results: Proportion of significantly distinct profile pairs, cc, Part 1

Optimizer	Benchmark	Evaluation run	Proportion of distinct (unordered) profile pairs
cc	m88ksim	ref	3/3
		test	3/3
		train	3/3
	mcf	ref	2/3
		test	3/3
		train	3/3
	perl2000	ref_diffmail	45/45
		ref_makerand	44/45
		ref_perfect	44/45
		ref_splitmail1	45/45
		ref_splitmail2	43/45
		ref_splitmail3	38/45
		ref_splitmail4	32/45
		train_diffmail	45/45
		train_perfect	43/45
		train_scrabble	42/45
	twolf	ref	3/3
		test	3/3
		train	2/3
	vortex2000	ref1	9/10
		ref2	7/10
		ref3	10/10
		test	10/10
		train	10/10

Table 2.8: Games-Howell Results: Proportion of significantly distinct profile pairs, cc, Part 2

ANOVA analysis found to have some significant profile usefulness difference[2]. In general, these results allow us to proceed with analysis of these average cycle counts without much reason to suspect that the differences between the average cycle counts are merely experimental error.

Note that this claim is quite independent of any claims as to the causes of profile-driven optimization usefulness. All that this section has so painstakingly established is the existence of significant differences due to profile-driven optimization - it does not, for example, make any assertions that profile-driven optimization is a good thing, or connected to how accurate the profiles are (covered in the next section), etc. To use a ridiculous example: an "profile-driven optimizer" that simply hashed the name of the training profile to an enormous integer i, and inserted a delay loop that took i iterations would result in significant profile-driven optimization performance differences as we define it in this section. More realistically, significant improvements in optimization performance are not necessarily the result of well-designed optimizations successfully making use of good information, as we shall establish later in this chapter.

If this seems like a meager conclusion after so much analysis, it is worth remembering that in the absence of this analysis, we would have no reason to believe that the differences in profile usefulness that we will subsequently attempt to connect to profile accuracy are anything more than experimental error. While this is a banal conclusion if one is only looking at large differences in optimization performance for long-running SPEC "reference" evaluation runs, it is quite significant for subsequent sections.

In theory, this analysis should be repeated for each and every subsequent experiment on profile-directed optimization performance in future chapters. To avoid tedium, we have decided to present it only once, to raise the level of faith that the differences that one observes from profile-directed optimization are usually the results of actual performance differences, not experimental error.

[2]Technically, we should probably not proceed with a post-hoc analysis at all if the finding of the original ANOVA was that there was no significant difference. In this case, nothing that we found from the post-hoc analysis disagrees with the original ANOVA: the benchmark `art`'s reference runs `ref1` and `ref2` are found to have little significant performance difference among profiling runs by both ANOVA and the post-hoc analysis, and are probably poorly suited to further analysis of profile-driven optimization performance

2.4 The Connection of Usefulness and Accuracy

Profile-driven optimization implicitly assumes a connection between better predictions of the future and better optimization. After all, the whole point of profiling is to gain a better idea of what will happen in future executions than could be predicted by static estimation. In this chapter, we evaluate this connection only in the context of "exact" profiles - in our experiment, benchmark runs that were available with the SPEC95 or SPEC2000 benchmark set. We set forth the Usefulness/Accuracy Hypothesis as follows: The usefulness of a profile is determined to a large extent by the degree to which it predicts future behavior. Profiles that are more accurate predictors of future behavior on a given execution run will result in better profile-driven optimization performance (for that run) than less accurate profiles.

This hypothesis needs to be qualified in several ways. Firstly, it applies only to a single optimizer at a time; it cannot be expected to predict performance across optimizers (and perhaps not across different benchmarks, either). It can only be evaluated in terms of a population of profiles and benchmark runs. It also implies the existence of a single metric for determining profile accuracy; we will evaluate and present the results for several different profile accuracy metrics.

We must also define the cases for which we are trying to correlate the two variables of profile accuracy and profile usefulness. Should we make comparisons between two different benchmarks - for example, suppose the accuracy metric STATICCOVERAGE (explained in detail later) shows that the "train" benchmark run for gcc covers only 75% of the "ref" benchmark run's basic blocks, while the "train" benchmark for vortex covers 99% of the "ref" benchmark run's basic blocks. Are we trying to establish that the "train" benchmark for gcc is somehow relatively less useful for gcc "ref" benchmark run PDO than the "train" benchmark for vortex is among potential training runs for vortex? Establishing such a connection would be helpful, as if it was true, we could immediately assume that, even for a benchmark that we had never seen before, a run with poor static coverage scores was going to be a bad choice for profile-driven optimization. At the other extreme, it is possible that the only legitimate connections between profile-driven optimization performance and profile-accuracy occur strictly on a per-benchmark and per-run basis. That is, knowing the static coverage score for a given training profile and reference run pair is only useful in predicting profile-driven optimization performance relative to other training profiles on that run. These two cases are illustrative but not exhaustive. An intermediate possibility is the case where per-benchmark scores can be compared across runs, but where scores cannot be used to predict cross-benchmark performance.

It is important to emphasize that this chapter shows the results of evaluating profile-driven optimization performance on a relative handful of benchmark runs. Given that many of the programs that we studied have a small number of potential runs - and in some cases, that all the potential runs are extremely similar - it is important not to make sweeping generalizations about the behavior of all potential profiles for these benchmarks. Especially infrequent in the profiles in this chapters analysis are profiles that generally score very poorly in terms of accuracy when compared to other profiles for the benchmark. Overall, most of our benchmark runs are very similar to each other; we have a dearth of “bad” profiles. This is not a bad thing for many of our benchmarks; after all, they themselves are not capable of being run in radically different ways. However, we must be aware that there is a wider spectrum of potential profiles available, particularly by means such as static estimation and statistical sampling, which generally produce much less accurate profiles - even for programs that have almost no variation between the behaviors of actual runs. Our results in this chapter do not necessarily predict results in these wider domains.

In this section, we again use the measurements gathered for each benchmark, exact profile and evaluation run in the previous sections in this chapter, as well as gathering profile accuracy metrics for each of the training profiles used.

2.4.1 The Usefulness of Perfect Information

The first indication that profile accuracy is not necessarily strongly determinant of profile-driven optimization performance is immediately apparent from our scaled average scores. We use the mean cycle counts for resubstitution - that is, the resulting cycle counts from using the reference runs profile as a PDO training profile - as our baseline for normalizing the different cycle counts produced for each benchmark and benchmark run. Thus, for a given run and benchmark, a profile that produces a binary that performs just as well as the reference runs profile gets a score of 1.0. Using the data we have already gathered, we can easily measure each profile’s usefulness relative to perfect information.

Examining Figure 2.9 shows us that the connection between accuracy and usefulness is at least problematic. These figures show histograms of the distribution of scaled averages for each of our two optimization contexts (taken over all possible training and evaluation pairs for all of our benchmarks). A substantial number of profiles have scores that are under 1.0 - in fact, 35% of alto training profile/evaluation run pairs score below 1.0 (i.e. better than perfect information)

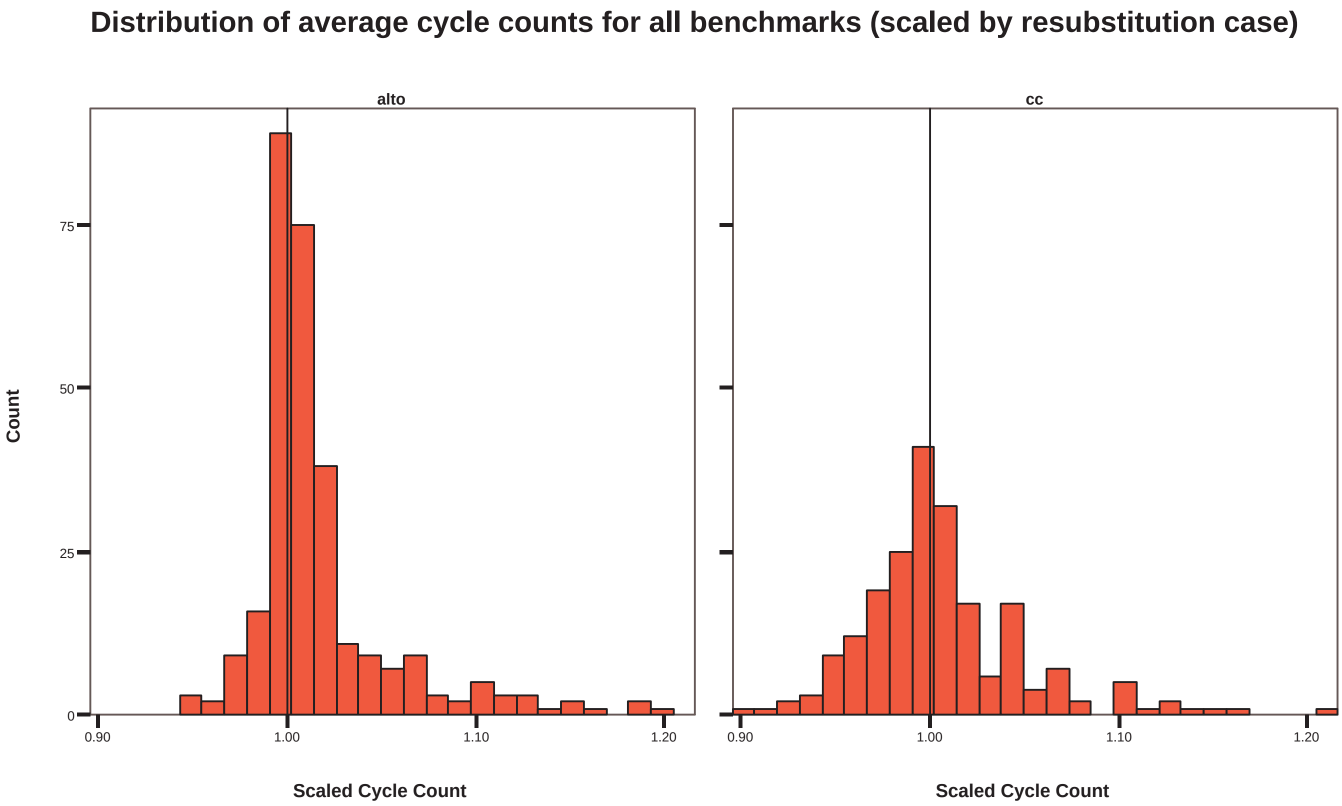

Figure 2.1: Histogram of average cycle counts for each benchmark, training profile and evaluation run (resubstitution excluded), scaled so that the resubstitution case for that benchmark and run is 1.0

and almost half of the cc pairs score below 1.0.

This is an obvious sign that the correlation between profile accuracy and profile usefulness is not strong. If the correlation between profile accuracy and profile usefulness was perfect, no profile could be more useful than the profile that constitutes perfect information for that run. The fact that so many profiles outperform perfect information, some by substantial amounts, is a sure indicator that the correlation is not perfect.

Table 2.9 shows the proportion of the non-resubstitution cases that outperform resubstitution. Note that for some benchmarks (`ammp`, `parser` and `ijpeg` under `alto`, `equake` and `go` under `cc`), the number of benchmark run / profile pairs that perform better than resubsitution is in excess of 75%.

This is a quite strong result, as it is independent of our choice of profile accuracy metric.

2.4.2 Profile Accuracy Metrics

All of our comparison metrics compare a list of basic block counts in a training profile with a list of basic block counts in an evaluation profile. They return a single number, a score that indicates how well the basic block counts in the training profile predict the basic block counts in the evaluation profile. Thus, a more accurate training profile better predicts the CFG-level behavior of the evaluation run. Most of these metrics are asymmetric.

A profile comparison metric consists of a comparison type and a way of applying it over the program. The comparison types we use in this paper are key-matching, static coverage and relative entropy.

Key-matching is introduced in [14]. It uses a parameter that determines how many blocks are selected for key-matching. For example, if a function has 50 blocks, and the matching level is 40% (or 0.4), then we perform key-matching on the top 20 blocks as follows: the key-match score is the number of blocks in the top 20 blocks in the training profile that are also in the top 20 of the evaluation profile. Key-matching metrics are denoted by KM(level) - “level” is always 0.1 in this paper.

Static coverage (denoted “STCOV”) measures what proportion of the blocks executed (“covered”) in the evaluation profile are also executed in the training profile.

Relative entropy (denoted “ENT”) as a method of comparing profiles was introduced by Savari and Young [11] and is fully described there. Relative entropy treats the profiles being compared as distributions of random variables and uses

Optimizer	Benchmark	Number of non-resubstitution runs	Number of non-resubstitution runs better than resubstitution
alto	ammp	6	5
	art	12	4
	bzip2	20	10
	compress	6	0
	crafty	6	0
	equake	6	2
	gap	6	1
	go	20	10
	gzip	25	17
	ijpeg	6	5
	li	6	0
	m88ksim	6	1
	mcf	6	2
	parser	6	5
	perl2000	81	19
	twolf	6	3
	vortex2000	20	5
	vpr	30	9
	ALL CASES	291	98
cc	ammp	6	2
	bzip2	20	8
	compress	6	2
	crafty	6	2
	equake	6	5
	gap	6	3
	go	20	15
	ijpeg	6	4
	li	6	3
	m88ksim	6	3
	mcf	6	3
	perl2000	90	40
	twolf	6	3
	vortex2000	20	9
	ALL CASES	200	102

Table 2.9: Number of training profile / evaluation run pairs (not counting resubstituion) and number of such pairs that perform better than perfect information
.

an information-theoretic approach to measure the difference between the two distributions.

We use two methods for applying these comparisons to our programs. Firstly, we can apply the comparisions to the whole program's set of basic block counts directly. This is the default method. Secondly, we can apply them only to the entry counts of functions, ignoring all other basic block data (denoted by prefixing "FE-" to the comparison name in our results).

2.4.3 Evaluating the Connection Between Comparison Metrics and Usefulness

To measure the association between profile usefulness and a given profile comparison metric, we use the Spearman Rank Correlation Coefficient [15], r_s. The coefficient r_s can be calculated by assigning ranks to the values being compared (scoring ties as the average rank values - so if there is a tie between the top two values, they both are assigned the rank of 1.5) and calculating the more familiar Pearson correlation coefficient [15] over those ranks. Thus, calculations of r_s discard the magnitude of the differences between data points. This makes r_s weaker (more likely to miss a real effect) than Pearson's correlation coefficient but much more robust in the presence of non-linear relationships, outliers and (more generally) data that does not hold to a bi-variate normal distribution.

When analyzing the correlation between profile accuracy and usefulness, we must be aware that there is no "natural" population of profiles for a given benchmark. For most benchmarks, we have a limited number of runs available to us, and they have been chosen artificially. If we include other profile types besides profiles derived directly from real runs, we are introducing further artificial biases into our population. Admittedly, the choice of benchmark runs from the SPEC benchmark sets are artificial also, but they are not the artificial choices of the authors of this paper - that is, they are not hand-picked in order to advance our favored hypotheses.

Evaluating a single benchmark and single evaluation run, we show an example of how we summarize the connection between profile usefulness and accuracy with a single correlation scores. We then show how we combine multiple scores (one for each individual run) to yield an aggregate score for the benchmark, and then present scores for all of our benchmarks and optimizers.

Training run name	Average cycle count (Gcycles)	Relative entropy
`ref/diffmail`	45.819	8.05
`ref/makerand`	47.581	22.57
`ref/perfect`	40.774	0
`splitmail1`	46.495	8.52
`splitmail2`	45.640	8.20
`splitmail3`	47.176	8.35
`splitmail4`	45.281	8.29
`train/diffmail`	45.615	8.06
`train/perfect`	42.515	2.45
`train/scrabble`	48.923	20.44

Table 2.10: Example 1: `perl2000` scaled cycle counts and accuracy metrics for a single evaluation run (`ref/perfect`)

Gathering Usefulness/Accuracy Correlation Results for a Single Evaluation Run

Firstly, we present the average cycle count scores and usefulness scores for the benchmark `perl2000` and the `ref/perfect` benchmark evaluation run. For each training profile, we have an average cycle count (reflecting how many cycles the binary that was produced by profile-driven optimization using that profile took to run the evaluation run) and an accuracy score (reflecting how close the training profile was to the profile produced by the evaluation run). For this example, we will use the accuracy scores provided by relative entropy[3].

Table 2.10 shows the cycle counts and relative entropy scores for a list of training runs (the names refer to the different benchmark runs available for `perl2000` and are not of any interest aside from the fact that they label cases). Figure 2.3 shows the same information in graphical form. To calculate a score for how closely relative entropy predicts scaled cycle counts, we take the r_s value of two variables (cycle count and relative entropy) over the list of cases (training profiles). This amounts to calculating the r_s values of the two value columns in Table 2.10, which turns out to be $r_s = 0.87$. This value is statistically significant at the 0.01 level; that is, if there was no association whatsoever between two variables, we'd expect to see a r_s value this high less than 1 in 100 times. The proportion of scaled cycle count variation explained by relative entropy is $r_s^2 = 0.75$ - that is, 75% of

[3]More accurate profiles produce lower relative entropy scores; zero represents a perfect match.

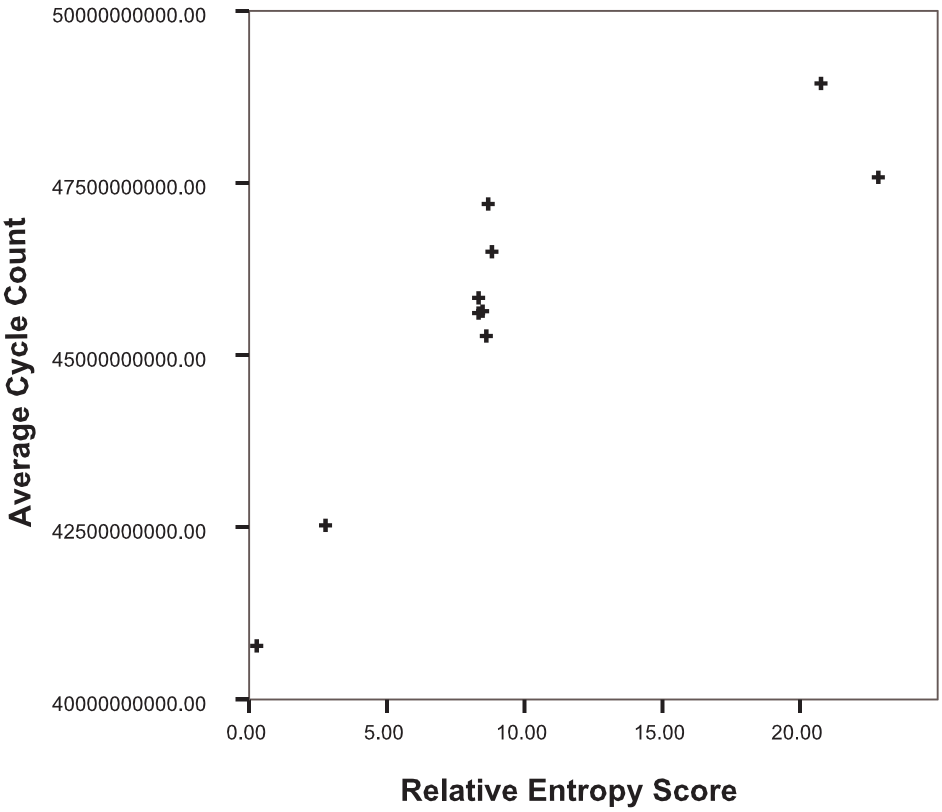

Figure 2.2: Example 2 (continued): `perl2000` evaluation run `ref/perfect`: Scatter-plot of average cycle count versus relative entropy score

Evaluation run name	r_s score
ref/diffmail	0.382
ref/makerand	0.778**
ref/perfect	0.867**
splitmail1	0.697*
splitmail2	0.612*
splitmail3	0.685*
splitmail4	0.612*
train/diffmail	0.394
train/scrabble	0.285

Table 2.11: Example 2: All perl2000 evaluation runs with the rank-correlation values of cycle counts and relative entropy calculated over each training run ('**' results significant at 0.01 level, '*' results significant at 0.05 level).

the variation in average profile-driven optimization performance in this particular case can be explained in terms of relative entropy.

Note that this benchmark has a quite large number of possible training profiles (10). Many of our benchmarks have only 3 or 4 runs available, so we are often in the situation of calculating correlations over a tiny set of cases. In this circumstance, it is possible to have apparently strong correlations that are in fact statistically meaningless *considered singly* . Only when they occur as a pattern across multiple evaluation runs and/or benchmarks can we attach any weight to these results.

Gathering Usefulness/Accuracy Correlation Results for Multiple Evaluation Runs

Table 2.11 shows this analysis repeated for all of our evaluation runs in perl2000. We see a larger set of results - now, we have a table with r_s numbers for each evaluation run. Not all of the correlations are significant at a 0.01 level (those that are are marked with a "**") or even at a 0.05 level (marked with a "*"). For example, the value $r_s = 0.382$, seen for the evaluation run ref/diffmail is fairly low: there is a 14% chance that two unconnected variables might show a rank correlation equal to or greater than this value (3 evaluation runs fall into the category of not being significant at the 0.05 level). However, even considering only these three values in isolation, it is unlikely that we would see three such correlations

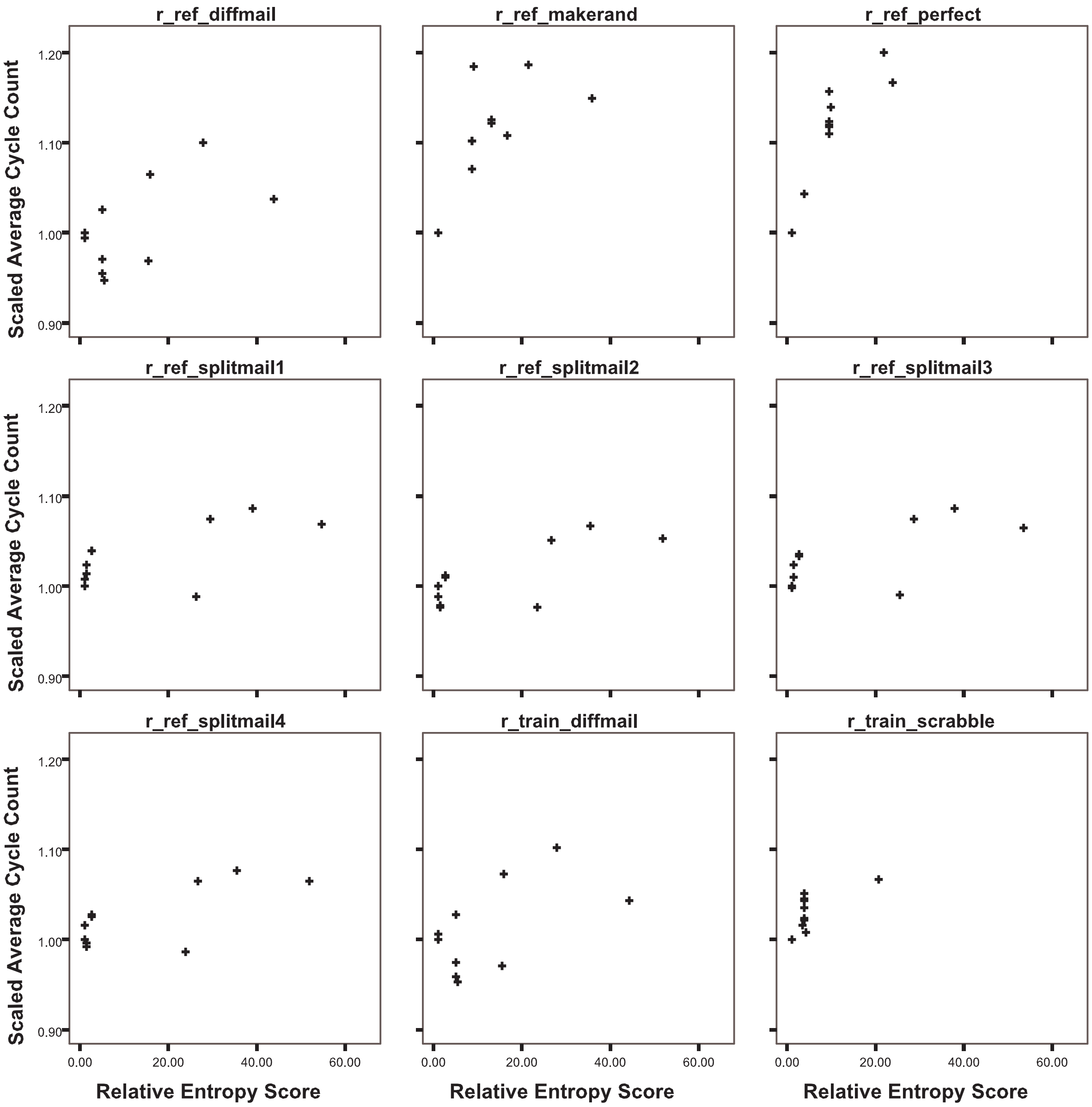

Figure 2.3: All `perl2000` evaluation runs: Scatter-plot of scaled average cycle count (scaled by resubstitution case) versus relative entropy score

(that is, positive and in the range $0.285 < r_s < 0.394$) between relative entropy and average cycle count if overall, there was no connection between relative entropy and average cycle count for any of these runs. In fact, the chance that such three correlations this strong or stronger would have arisen by chance given no connection between relative entropy and cycle count is about 1%.

Note that it is quite possible to have negative r_s scores; in this case, more accurate profiles actually result in worse profile-driven optimization performance. In addition, it is worth noting that profile accuracy metrics that return a constant value for the set of profiles considered will result in a zero r_s score regardless of the usefulness scores. Such a metric is not a very good predictor, of course, but an accuracy metric that fails to differentiate between profiles at all is still more closely connected to usefulness than one that is negatively correlated with usefulness! For example, `ijpeg` optimized with `alto` has a strong inverse relationship between usefulness and accuracy. The 'relative entropy' metric predicts about 66% of the variation in usefulness - unfortunately, in this case less accurate profiles consistently produce better optimization effects than more accurate ones.

Combining Usefulness/Accuracy Correlation Results from Multiple Runs to Yield a Single Aggregate Score

We can compute a summary value for the overall connection of usefulness and accuracy over a benchmark by simply averaging the r_s values for each evaluation run, yielding an aggregate correlation of $mean(r_s) = 0.59$ for the example of the `perl2000` benchmark.

This is not generally good practice; an approach that is more statistically rigorous is to transform each r_s value to a z-score (normal score), take the average over these z-scores and transform back into the range of r_s. However, this procedure is complex and results in average r_s scores in most cases that are little different from those that we derive from simple averaging. Further, the transformation to a z-score is problematic for extreme cases (perfect correlation r_s = 1 or perfect negative correlation r_s = -1); the formulas to transform correlation scores to z-scores produce essentially meaningless results in these cases.

To calculate whether aggregate r_s scores are significant, we must calculate what the expected value of our aggregate r_s score would be if the null hypothesis was true for each correlation being aggregated. So, for example, the distribution on the null hypothesis of the aggregate r_s score that sums up the association of some profile metric and the average cycle counts on all cases of `perl2000` is equal to the distribution of the average of seven random variables chosen from seven identical r_s distributions. This highly heterogeneous method of calculating significance (given that we have results that average 3 r_s distributions over 3 items all the way up to results that average 10 r_s distributions over 10 items) turned out to yield complicated and difficult to interpret data. Generally, we will assume that individual benchmark scores for benchmarks with low numbers of runs (e.g. not `vortex`, `go`, `perl2000`, texttvpr and `gzip`) are of questionable significance when considered single; particularly if the magnitude of the average r_s score is small (under 0.4). Aggregates over entire benchmark sets are nearly all significant due to the much larger number of data points involved.

Aggregate Usefulness/Accuracy Correlation Results for all Benchmarks

Using such a procedure to gather aggregate numbers for each benchmark, this time over a range of comparison metrics, we derive Table 2.12. This table shows the aggregate r_s scores for each comparison, benchmark and optimizer, as well as overall mean scores for r_s comparison and optimizer. It is clear that the `perl2000` benchmark, presented above, and particularly the `perfect` evaluation run, represent a quite favorable case - note the large number of benchmarks in this table for which the aggregate r_s scores are either very low (i.e. no correlation) or actually negative (i.e. more accurate profiles have worse profile-driven optimization performance). Particularly, the results for the `cc` optimizer show no overall pattern of a connection between profile usefulness and profile accuracy.

In the `alto` case, all of the profile comparison metrics yielded small but significant correlations between profile accuracy scores and profile usefulness scores. Key-matching performed slightly worse than the other two profile accuracy metrics, entropy and static coverage. The "function-entry" versions of these latter accuracy metrics performed slightly better than the versions that considered all of the basic blocks in the program, although such a small difference is not likely to be significant.

On the other hand, the correlation results for `cc` suggest no significant and systematic (that is, consistent across different benchmarks and profile accuracy metrics) connection between profile usefulness and profile accuracy at all.

There was a substantial amount of variability among the aggregate r_s scores for each benchmark. Some of this variability is simply random; the aggregate r_s scores for the benchmarks with a small number of runs are subject to a great deal of randomness as they involve comparisons among only 9 or 16 values. However, some benchmarks clearly have far stronger associations between usefulness and accuracy than others. Recall that the correlation coefficients in Table 2.12 rank how well profile usefulness correlates with profile accuracy; they have nothing to say about how well profile-directed optimization works overall.

A major weakness of the above approach to evaluating the connection of profile-directed optimization performance and profile accuracy is that, due to the use of non-parametric methods and averaging across different benchmarks, small variations in one benchmark are weighted as heavily as huge variations in another. There is no simple way to avoid this problem without recourse to parametric correlation methods. However, we can derive results that are more useful by restricting our above analyses to only those evaluation runs with greater variability due to profile-directed optimization. The overall (per-optimizer) results from restricting our analysis to the top half of evaluation runs with the highest level of profile-directed optimization variability are shown in Table 2.13.

The failure of our profile accuracy metrics to explain `cc` profile-directed optimization performance turns out to be unconnected to profile-directed optimization variability. Even considering only benchmarks and benchmark runs that had large variations in profile-directed optimization performance did not improve the connection between profile accuracy and profile usefulness when using `cc`. However, our `alto` results become substantially stronger when we eliminate benchmark runs with small variations in profile usefulness. Entropy-based methods, in particular, improve markedly. The "FE-ENT" accuracy metric predicts 34% of the variation in our profile-directed optimization results under `alto` - a modest result, but the strongest one so far.

We found no similar improvements from restricting our analysis to smaller (e.g. top quarter by PDO variability) subgroups of our evaluation runs. Not surprisingly, the bottom half of evaluation runs by PDO variability showed no significant correlation (under `alto` or `cc`) between profile accuracy and profile usefulness.

2.4.4 Discussion

There is no reason to suppose that any reliable connection between accuracy and usefulness existed in the `cc` optimization context whatsoever. We conjecture that the much more extensive and high-level optimizations present in `cc` sufficiently transform the control-flow-graph to the point where the relatively subtle differences between training profiles are irrelevant. This does not mean that profile-driven optimization does not work in `cc`, nor does it mean that arbitrarily inaccurate profiles will produce profile-driven optimization performance indistinguishable from good ones. What it does mean is that, within the fairly narrow range

Optimizer	Benchmark	$mean(r_s)$					
		ENT	STC	KM(0.1)	FE-ENT	FE-STC	FE-KM(0.1)
`alto`	`ammp`	-0.67	-0.79	-0.50	-0.50	-0.58	-0.67
	`art`	0.20	0.23	0.35	0.25	0.07	0.23
	`bzip2`	-0.12	-0.06	0.09	0.00	0.14	0.16
	`compress`	1.00	0.91	0.83	1.00	0.29	0.58
	`crafty`	0.83	0.67	0.67	0.67	0.17	0.50
	`equake`	0.17	0.17	0.00	0.17	0.00	0.58
	`gap`	0.83	0.83	0.50	0.83	0.79	0.50
	`go`	0.26	0.13	0.20	0.34	0.30	0.23
	`gzip`	-0.16	-0.24	-0.28	-0.26	0.12	-0.24
	`ijpeg`	-0.83	-0.67	-0.67	-0.83	0.00	-0.79
	`li`	0.83	0.96	0.50	0.83	0.96	0.50
	`m88ksim`	0.67	0.67	0.67	0.67	0.79	0.67
	`mcf`	0.50	0.62	0.67	0.50	0.58	0.87
	`parser`	-0.50	-0.83	-0.83	-0.50	-0.29	-0.67
	`perl2000`	0.59	0.60	0.45	0.52	0.60	0.50
	`twolf`	0.33	0.33	0.17	0.33	0.58	0.29
	`vortex2000`	0.38	0.36	0.38	0.64	0.17	0.48
	`vpr`	0.52	0.59	0.57	0.55	0.51	0.54
	`alto` MEAN	0.27	0.25	0.21	0.29	0.29	0.24
`cc`	`ammp`	0.00	-0.04	0.33	0.33	-0.58	0.00
	`bzip2`	0.16	0.04	0.15	0.06	-0.07	-0.09
	`compress`	0.33	0.17	0.17	0.33	0.29	0.29
	`crafty`	0.67	0.33	0.00	0.33	-0.46	0.00
	`equake`	-0.83	-0.83	-0.50	-0.50	0.00	0.00
	`gap`	-0.33	-0.33	-0.33	-0.33	-0.46	-0.17
	`go`	-0.72	-0.76	-0.74	-0.72	-0.60	-0.65
	`ijpeg`	0.33	0.00	0.00	-0.33	0.00	-0.33
	`li`	-0.17	-0.46	0.33	-0.17	-0.46	0.33
	`m88ksim`	-0.17	-0.17	-0.17	-0.33	-0.12	-0.17
	`mcf`	0.33	0.33	0.17	0.33	0.58	0.29
	`perl2000`	-0.18	-0.23	-0.17	-0.19	-0.21	-0.14
	`twolf`	0.33	0.33	0.17	0.33	0.58	0.33
	`vortex2000`	0.04	-0.01	-0.02	0.06	0.02	0.06
	`cc` MEAN	-0.06	-0.12	-0.04	-0.06	-0.11	-0.02

Table 2.12: The connection of usefulness and accuracy: aggregated r_s scores over optimizers, benchmarks and different comparison metrics

Optimizer	ENT	STC	KM(0.1)	FE-ENT	FE-STC	FE-KM(0.1)
alto mean	0.57	0.52	0.48	0.58	0.45	0.45
cc mean	-0.15	-0.22	-0.14	-0.17	-0.03	-0.03

Table 2.13: Aggregated r_s scores over optimizers, considering only the top half of evaluation runs by PDO variability

of profiles and benchmarks we tested, accuracy could not be shown to have any connection to usefulness. We evaluated many other profile comparison metrics than (carrying out key- and weight- matching at multiple levels, using dynamic coverage) presented here and found that none of them performed any better than the comparison metrics presented.

Our results for the `alto` optimization context were more encouraging, but still relatively weak. Even when restricting our analysis to benchmarks with large profile-directed optimization variability, we could explain no more than a third of the variation in average cycle counts by some accuracy metric.

One of the most startling results was the fact that the accuracy metric "FE-STC" performed as well as it did despite the fact that it ignores nearly all of the information in the block profile. This extremely simple metric can be calculated by determining the number of functions entered in the training profile and the evaluation run divided by the total number of functions entered in the evaluation run.

The effectiveness of this metric (and similarly restricted metrics) could result from there being little variation in within-function behavior from run to run (that is, when profiles produced from benchmark runs differ, it is because they cover a different set of functions, not because they have radically different behavior within those functions). An alternate possibility is that the optimizations in `alto` really only effectively worked at a per-function level and thus made little use of the within-block information. These possibilities are not easily separated, although the fact that our "function-entry only" comparison metrics are strongly correlated ($r_s > 0.9$) with their whole-program counterparts for nearly all benchmarks is suggestive that the former possibility is true (across both optimizers, `bzip2` and `gzip` were the only exceptions).

2.5 Conclusion

Profile-directed optimization is a worthwhile technique, on average, in both of the optimizers evaluated. On average, we saw an improvement over non-profile-directed optimizations of about 3.5% on `alto` and 5% on cc; these aggregate numbers concealed substantial variations (the best case for either optimizer was approximately 17% better than non-profile-directed optimization and the worst case for either was approximately 14% worse).

Nearly all of the benchmark runs showed significant variation in profile-directed optimization performance. In only 1% of our evaluation runs were we unable to detect significant variation among profile-directed optimization performance (that is, no variation due to profile-directed optimization existed or it was so small that we were unable to separate this variation from experimental error). Again, large differences existed between the evaluation runs with the largest amount of profile-directed optimization variability and those with the smallest - the standard deviations in speed-up over the non-profile-directed-optimization case ranged from effectively zero to nearly 5%.

Profile accuracy is only weakly associated with profile usefulness in one of our optimizers (`alto`) and not connected at all with profile usefulness in another (`cc`), for our set of benchmarks and benchmark runs. While considering only benchmarks or runs with higher variability in profile-driven optimization performance improved the connection on `alto`, the connection between usefulness and accuracy still accounted for only 34% of the observed variation in profile-driven optimization performance. While the comparatively weak (non-parametric) correlation methods that we had to use may have caused us to be overly conservative, it seems unlikely that any accuracy metric whatsoever would explain in excess of 50% of the variation. Of the variation in profile usefulness explainable by profile accuracy metrics, much of it was explainable by fairly simple profile accuracy metrics, most notably static coverage of function entries ("FE-STC").

These results (negative for `cc` and weak for `alto`) are not entirely surprising. Much of the variation in our training profiles does not necessarily cause different optimization outcomes. That which does cause different outcomes does not necessarily help. Not every optimization "decision" produces better performance, regardless of whether it is based on good information - few compiler optimizations are truly "optimizations", particularly when interacting with many other optimizations.

Note that these results (in addition to being in the context of a certain sets of benchmarks and optimizers) are in the context of profiling with exact basic block profiles. Our range of profiles was intentionally limited to the profiles that result from the standard benchmark runs that are provided with the SPEC benchmarks; adding a few inaccurate synthetic profiles to our analysis may have made our results stronger but would introduce problems with statistical rigor - adding such heterogeneous items to our data sets would essentially amount to “rigging” our results.

Given all of these caveats, what have we shown? Aside from the methodological contributions, our results show that there exists at least one optimizer for which usefulness and accuracy are not correlated (in our experimental context) and one in which this correlation exists but fails to explain the bulk of profile-directed optimization performance. Thus any claims about profile-directed optimization techniques or more accurate profiling techniques (or the necessity of obtaining more accurate precise basic block profiles - dynamically or otherwise) should be evaluated *experimentally*, not in terms of profile accuracy. We have shown that there are a range of cases where little or no connection between profile accuracy exists. Thus, it is incumbent on designers of profile-directed optimization systems to demonstrate that the profile-directed optimization in their systems is actually effective over a wide range of benchmarks, rather than merely showing that the profiles gathered are of high accuracy.

Further chapters will address the issue of the profile “usefulness” of profiles generated by imprecise profiling techniques such as statistical sampling and static estimation. Our results in this chapter make no predictions about the performance of such techniques; generally, the far more profiles generated by such methods are far more inaccurate than the profiles examined in this chapter.

Chapter 3

Static Estimation of Block Profiles

3.1 Introduction

The instrumentation overhead from gathering exact profiles is high and the task of instrumenting a arbitrary program in an efficient way is not always straightforward (particularly in the presence of dynamic linking or dynamic code generation, or in object files where finding all code is difficult due to object file or instruction format). One solution to these problems is to simply statically estimate profiles based on the structure of the program. Another is to employ a non-intrusive (the program code is not changed) technique called sampling, discussed in Chapter 4.

Static estimation of program execution profiles takes advantage of the fact that control-flow behavior is fairly predictable from program to program. Simple heuristics can allow us to discover that branches that exit loops, for example, are usually taken very infrequently. From these heuristics, we can reconstruct a basic block profile which we use for profile-driven optimization.

3.2 Static Estimation

In this work, we evaluate only a fairly simple (or naive) static estimator. While evaluating a more sophisticated static estimator would be interesting, we are necessarily limited in the scope of this work. Further, building a good static estimator in our optimization context would be of only limited research interest - after all, the control flow graphs that we process have already been heavily transformed by the compiler. Thus, when we look at branch behavior, we are looking at the output of a complex set of algorithms which are already making heuristic guesses about the same branch behavior. Thus, any branch model we built for our benchmarks would have as much to do with the `cc` compiler's branch prediction heuristics and code transformations as it would with the original program's behavior. We would need to deal with a control flow graph much earlier in the optimization process if we wanted to avoid these problems.

Thus, our naive static estimator has only a limited set of heuristics, based on overall program structure. A branch that exits a loop is assumed to be not-taken with a branch frequency of 0.1. Similarly, a conditional branch that continues a loop's execution is assumed to be taken with frequency of 0.9. All other branches are assumed to be evenly balanced with a frequency of 0.5.

Like most static estimators, our static estimator makes no attempt to estimate conditional branches as either "always-taken" or "never-taken". This is a fundamental problem with static estimation: a large number of conditional branches actually fall into this category. Unfortunately, most static estimators represent their level of uncertainty about whether a branch is taken within the branch estimate. Thus, a conditional branch in a loop that is heuristically estimated to be very unlikely (for example, it leads to a `exit()` call) is given a very small - but non-zero - branch frequency as a way of expressing the uncertainty about the static estimation process.

The net effect of these kind of estimates is that most static estimators, including our own, perform very poorly at estimating which program blocks are never executed[1]. Our statically estimated profiles perform very well on coveraged-based profile accuracy metrics, but not for the right reasons. Even a profile that simply consists of a constant value for each block will be considered a perfect estimate of any other profile if our accuracy metrics are static or dynamic coverage.

We use a two-stage process to turn our branch estimates (from our simple heuristics) into block profiles. In both stages, we represent the block counts (or function entry counts) as a linear equation and solve the linear equation using the IML library[1]. First, we estimate "fractional counts" for each block in each function: the estimated number of executions for each block if the entry block of the function was executed 1.0 times. Then, we calculate the frequency of function calls within each function to other functions and, setting the entry function of the program to a count of 1.0, we estimate the frequency that every function in the program is entered. Finally, we use these entry counts in conjunction with the

[1] The use of a threshold (that is, if the estimated count for a block is below a certain value, consider it zero) fails to improve matters: if the threshold is made high enough to estimate a large number of basic blocks as 'not executed', then the threshold usually applies to far too many basic blocks.

"fractional counts" to establish basic block profiles for every function.

3.2.1 Combining Real Profile Information with Static Estimation

As we will see, static estimation of this naive type does not perform very well for optimization. In an effort to discover what is wrong with static estimation, we shall artificially insert two different types of dynamically-obtained information into our static profiles. If either of these types of information substantially improves the usefulness of the statically estimated profiles, we will know that the lack of that type of dynamic information was a significant deficiency in the static profile.

- Entry Count information: entry count information consists of basic block profiles for every function entry block in the program. Having entry counts allows us to use our static estimation process only for individual functions. Since one of the major problems for static estimation is an accumulation of errors for deeply nested block counts, having entry count information might help to bound the total amount of error for basic block count estimates.

- Coverage information: coverage information involves a single bit for each block, indicating whether the block is executed or not executed. Having such information allows us to get a picture of how much more useful it is to know when branches are "never-taken", although some situations exist where coverage information does not help. For example, the fact that a simple 'if' construct (with no 'else' clause) is always taken will not be detected by coverage profiling.

There is some overlap between these two sources of information. Entry count information conveys coverage information for whole functions: if a function has an entry count of zero, then the function can be treated as "not-covered" as a whole.

Note that both of these types of information have comparatively simple implementations as low-overhead profiling techniques. Entry count information can be gathered in a similar way to basic block profile gathering, but with instrumentation code added only to functions. The same general class of optimizations that apply to gathering of basic block profiles apply also to entry count information; for example, if function is always called exactly once per execution of some other set of functions, we can derive its entry count from the entry counts of those functions.

Coverage information is even simpler to derive. The Plan 9 debugger, `acid` [17] implemented such a coverage profiler by placing a breakpoint at the beginning of each basic block and eliminating the breakpoint after the first execution of a such a basic block. The overhead of this technique is negligible; it depends only on the total static number of blocks executed within a program execution, while most profiling techniques impose costs that are proportional to the number of dynamic events that take place in a program execution. This overhead can be further reduced using control-flow graph based optimizations. For example, if a group of basic blocks consists of basic blocks such that for any pair of basic blocks b_i and b_j, b_i dominates b_j and b_j post-dominates b_i, or vice versa, then only one block in that group needs to be monitored during "coverage profiling".

Results from Static Estimation

In our evaluation, we compare statically-estimated profiles with dynamic profiles. We use the following profiles in our evaluation:

- Statically-estimated profiles, using naive heuristics only
- A single profile from the "reference" and "train" profiles sets for that benchmark
- For each profile p of our "reference" and "train" profiles,
  - Statically-estimated profile using entry counts from p
  - Statically-estimated profile using coverage information from p, and
  - Statically-estimated profile using entry counts and coverage information from p.

We first compare the results of using the simple naive static estimator with the results of using the exact "reference" and "training profiles" in Table 3.1. The use of statically estimated profiles (in this table, scaled against the "resubstitution" case and averaged over each different benchmark run) caused significant worsening of performance for most benchmarks (ranging up to more than 10% for `crafty` and `m88ksim` using `alto`). Our results for `cc`, by comparison, showed only a slight worsening of performance as a result of using static estimation instead of exact profiles.

Next, we analyze whether or not adding coverage information and/or entry counts from real profiles improves the naive static estimation profile usefulness. First, we look at the impact of adding coverage information.

Optimizer	Bench	Exact Profile train	Exact Profile ref	Static Profile
alto	art	1.00	1.00	1.00
	bzip2	0.99	1.00	1.03
	compress	1.00	1.00	1.00
	crafty	0.99	1.00	1.10
	gap	1.00	1.00	1.05
	go	1.00	1.00	1.07
	gzip	1.00	1.00	0.97
	ijpeg	1.00	1.00	1.01
	li	1.01	1.00	1.05
	m88ksim	1.01	1.00	1.11
	mcf	1.00	1.00	1.00
	twolf	0.99	1.00	1.08
	vpr	0.98	1.00	1.01
	Alto Mean	**1.00**	**1.00**	**1.04**
	Common Mean[2]	**1.00**	**1.00**	**1.04**
cc	bzip2	1.01	1.00	1.00
	compress	1.01	1.00	1.01
	crafty	1.02	1.00	0.96
	go	0.97	1.00	0.97
	ijpeg	1.00	1.00	1.04
	li	0.96	1.00	1.04
	m88ksim	0.94	1.00	1.05
	mcf	1.00	1.00	1.01
	vpr	1.01	1.00	1.03
	CC Mean	**0.99**	**1.00**	**1.01**

Table 3.1: Dynamic versus Static Profile Performance (geometric mean across all runs for benchmark and scaled to be relative to resubstitution case). "Common Mean" is a mean over only the `alto` benchmarks that were also measured with `cc`

Optimizer	Bench	Static Profile	Coverage	Entry Counts	Both
alto	art	1.00	1.00	1.00	1.00
	bzip2	1.03	1.00	1.02	0.97
	compress	1.00	1.01	1.01	1.02
	crafty	1.10	1.09	1.08	1.10
	gap	1.05	1.07	1.06	1.06
	go	1.07	1.05	1.07	1.06
	gzip	0.97	0.97	1.01	1.00
	ijpeg	1.01	0.98	1.00	1.00
	li	1.05	1.06	1.04	1.03
	m88ksim	1.11	1.13	1.03	1.07
	mcf	1.00	1.01	1.01	1.00
	twolf	1.08	1.09	1.01	1.03
	vpr	1.01	0.99	0.99	0.99
	Alto Mean	**1.04**	**1.03**	**1.02**	**1.02**
	Common Mean	**1.04**	**1.04**	**1.03**	**1.03**
cc	bzip2	1.00	1.03	1.00	1.02
	compress	1.01	1.02	0.99	0.99
	crafty	0.96	1.00	1.00	1.04
	go	0.97	0.99	1.02	1.02
	ijpeg	1.04	1.05	1.03	1.03
	li	1.04	1.03	0.99	0.99
	m88ksim	1.05	1.04	0.97	1.09
	mcf	1.01	1.01	1.01	1.01
	vpr	1.03	1.02	1.02	1.03
	CC Mean	**1.01**	**1.02**	**1.00**	**1.02**

Table 3.2: Static Profile Performance (geometric mean across all runs for benchmark and scaled to be relative to resubstitution case) Using Information From A Single SPEC "Reference" Profile)

Optimizer	Bench	Static Profile	Coverage	Entry Counts	Both
alto	art	1.00	1.00	1.00	1.00
	bzip2	1.03	1.06	0.96	0.96
	compress	1.00	1.04	1.03	1.05
	crafty	1.10	1.09	1.06	1.07
	gap	1.05	1.09	1.06	1.07
	go	1.07	1.06	1.04	1.04
	gzip	0.97	0.97	1.02	1.00
	ijpeg	1.01	1.00	1.00	1.02
	li	1.05	1.11	1.02	1.05
	m88ksim	1.11	1.08	1.03	1.12
	mcf	1.00	1.00	1.00	1.00
	twolf	1.08	1.08	1.02	1.03
	vpr	1.01	0.99	0.98	0.99
	Alto Mean	**1.04**	**1.04**	**1.02**	**1.03**
	Common Mean	**1.04**	**1.05**	**1.01**	**1.03**
cc	bzip2	1.00	1.02	0.99	1.02
	compress	1.01	1.02	1.00	1.02
	crafty	0.96	1.00	1.02	1.03
	go	0.97	0.96	1.03	1.01
	ijpeg	1.04	1.05	1.02	1.02
	li	1.04	1.05	1.03	1.08
	m88ksim	1.05	1.04	1.12	0.96
	mcf	1.01	1.01	1.01	1.01
	vpr	1.03	1.04	1.03	1.02
	CC Mean	**1.01**	**1.02**	**1.03**	**1.02**

Table 3.3: Static Profile Performance (geometric mean across all runs for benchmark and scaled to be relative to resubstitution case) Using Information From A Single SPEC "Training" Profile)

As Tables 3.3 and 3.2 show, there is very little consistent improvement of optimization performance as a result of adding coverage information to static profiles. In fact, in a number of benchmarks (e.g. for `alto`, `gap` and `twolf`)), the addition of coverage information actually makes the static profiles less useful. We do not have a good explanation for this and must instead appeal to the principle already discovered that increased profile accuracy does not necessarily result in better performance.

Focussing on the addition of entry count information instead shows somewhat more positive results. Table 3.3 and Table 3.2 show there is a significant improvement in performance under `alto` as a result of using entry count information.

For the `cc` optimizer, as we note about, we saw only a comparatively small worsening of performance as a result of using statically estimated profiles rather than exact ones. As a consequence, it is not surprising that we did not observe any real improvement in performance as a result of using some information from exact profiles in our statically estimated profiles.

We do not include a formal analysis of the 2-way interaction between entry count information and coverage information; coverage information did not result in a significant improvement in profile-driven optimization performance whether or not entry count information was provided.

Nor do we include accuracy metrics for the different statically estimated profiles or attempt to correlate accuracy and usefulness in this chapter, as many of our accuracy metrics break down or yield identical results for all profiles of a given type. For example, it is obvious that coverage metrics and metrics that only take into account function entry counts will be overly effected by the profiles generated in this chapter that explicitly make use of coverage or function entry information.

3.2.2 Static Estimation Conclusions

It is obvious that static estimation does not perform very well, generally worsening performance by around 4% on `alto` and over 1% on `cc`, as compared to the use of a "good profile" (one from a SPEC reference run). For many of our benchmarks, we would be better off turning off profile-driven optimization entirely as opposed to using statically-estimated profiles. This is due to a number of reasons.

First, the statically estimated profiles were not produced through very sophisticated heuristics. It is possible that a more sophisticated static estimator would produce much better results.

Second, the profile-driven optimizers were not given any information about the static nature of the profiles that they had been given. In normal usage, many optimizers will turn off - or make less aggressive - their more risky optimizations if there is no dynamic information available. For example, `alto` reverts to only the safest procedure inlining behavior if no dynamic profile is available - even though, in normal, non-profile-driven optimization use, it generates a simple statically estimated profile (for other optimizations). To anthropomorphize, the net effect of this, is to give the profile-driven optimization routines a sense of over-confidence. They were designed for to use hopefully more reliable dynamic information and do not perform well with the estimates that result from static estimation.

We attempted to quantify the impact of adding two types of dynamic profile information to otherwise static profiles. We found that adding coverage information to the static estimator did not improve the performance of statically estimated profiles, despite the improvement in accuracy that this can provide. Entry count information reduced the penalty associated with the use of static profiling significantly under `alto` (approximately halving the slowdown due to static estimation), but showed no significant effect on `cc`.

Our evidence suggests that the technique of reducing profiling overhead by using only static estimation in combination with coverage profiling and/or profiling of entry counts (as opposed to gathering profiles for the whole program) is not particularly effective. The effort required to gather even these limited profiles and recompile is too large, and the potential benefits are too small.

Given our results from Section 2.4, it is not surprising that steps to improve the accuracy of static estimation generally had limited success - particularly in the `cc` optimization context. While there are many more sophisticated mechanisms for producing more accurate statically estimated profiles, we feel that their effect may be limited by the failure of optimizers to deliver improvements in performance as a result of more accurate profiles.

Chapter 4

Sampled Profiles

4.1 Introduction

The principle of sampling is to periodically interrupt the program (either at timed intervals or when a certain number of events have occurred), and record some aspect of the program state (usually the program counter) at the time of the interrupt. From our observations of the program state, we can reconstruct an estimate of the original program behavior.

The simplest use of an event-based sampling system is to interrupt the program after every N instructions have executed and to record the program counters at that point. If the distribution of the program counters over the basic blocks in the program is scaled up to reflect the fact that we have recorded only 1 in N instruction executions, we will now have a basic block profile that approximates the actual basic block profile.

Statistical sampling techniques can be used to gather a wide range of profile data, not just basic block executions. The range of possibilities include gathering a samples of program path behavior, calculating the distribution of the delays caused by various architectural events such as instruction cache misses, and calculating the distribution of real time spent in routines (as opposed to number of instructions or cycles spent in routines). Again, we concentrate on basic block profiles because this is the information that a wide variety of optimizations in both our optimizers make use of.

4.2 Sampling and Simulated Sampling

Simulated sampling involves a transformation of a exact profile (which we call the *base profile*) into a profile that attempts to introduce "sampling error". This transformation simulates the performance of a perfect event-count based sampling system that samples based on successful instruction completion (as opposed to a cycle-based event counter). A basic block is assigned a probability based on the sampling interval and the number of instructions in the block, a randomly generated sample count is generated for the block to simulate the sampling process.

After gathering these simulated sample counts for each block, we then use the sample counts to estimate the original basic block counts and thus get a new, less accurate basic block profile derived from the base profile. Optionally, we can use control-flow graph information to "patch" our sampled profiles to reduce the amount of error introduced by sampling.

The resulting sampled profile is thus a function of:

- the *base profile* used to generate it,
- the sampling interval
- whether or not we employed our various (control-flow-based) methods to improve sampled profile accuracy, and
- sheer random chance.

The random chance is introduced by the fact that we must generate a randomly distributed value that approximates the number of times that a given basic block's execution would be sampled. To make sure our experiments are repeatable, the sampling results are deterministic based on the seed passed to the random number generator at the beginning of the production of a given profile. Thus, our simulated sampling only simulates random behavior rather than actually performing randomly.

Justification of Simulated Sampling

An obvious question is raised by our use of simulated sampling: do the results derived from using simulated sampling have any relevance to the performance of actual sampled profiles? There are several sources of error using real sampling systems that simulated sampling does not model:

- Constant or variable skew between when a sample is triggered and when it is recorded - it may be difficult for the sampling interrupt handler to locate exactly which instruction triggered the sampling interrupt. Even knowing precisely how many instructions might occur between the event that triggered the interrupt and the location it was recorded will not always allow us to pinpoint the exact instruction that triggered the interrupt; the situation worsens considerably if our knowledge is imprecise. This source of error is comparatively unimportant, as modern architectures implement mechanisms to avoid such effects (a description of how this is avoided in DCPI can be found in [5]).

- Additional sampling inaccuracies resulting from samples being aggregated spatially - that is, a sampling system such as the one used by DCPI aggregates all samples taken over a 16-byte (or four Alpha instruction) "bucket". It is possible that such a group of instructions may include instructions from different basic blocks (although for performance reasons, basic blocks are usually not so short and critical basic blocks are usually aligned on boundaries similar to those used by the sampling code for aggregation).

- Time-based sampling systems can be used to derive block profiles but are greatly affected by variability in the number of cycles that a given basic block takes to execute. More modern, event-based sampling systems (that can count, for example, number of instructions) are not subject to this source of error.

Our simulated sampling system does not simulate any of these sources of error. The only error modelled by our system is the fundamental error introduced by sampling: that is, not recording every event. We feel that all of the other sources of error are both less important than sampling error and that their behavior is far more implementation dependent than sampling error. Errors in pinpointing the exact position of the sampling interval can be minimized by good architectural design, and the degree to which samples are spatially aggregated across important basic blocks can also be controlled. However, nothing can eliminate the fundamental error that looking at only a subset of events introduces.

Our simulated sampling system also allowed us to generate hundreds of different sampled profiles without re-running the benchmark hundreds of time. This provided a considerable speed-up in our experimental process. This speed-up was not enormous, however, as for our sampled profiles we still had to run multiple evaluation runs for each sampled profile to accurately evaluate the profile-driven optimization effect of the profile.

Another advantage to our use of simulated sampling was that there were no extraneous and unrepeatable perturbations of our sampling process; our sampled profiles are entirely deterministic given the random number seed used to initialize the random number generator used in simulated sampling.

Statistical Properties of Simulated Sampling Counts

Our sample counts for each basic block and the resulting estimates we for the basic block counts are random variables. We need to understand more about the expected values and distributions of these random values.

A basic block is assigned a probability p based on the sampling interval I (which corresponds to the number of instructions that execute between sampling interrupts in a real sampling system) and the number of instructions in the block b_{insns} (so $p = \frac{b_{insns}}{I}$), and a sampled value is chosen for that block as if we made n trials at probability p, where n is the number of times the block was executed in the run we were "sampling". This value is the binomial distribution $b(x; n, p)$.

The binomial distribution is quite computationally expensive to calculate (particularly for large n) and approaches the normal distribution with $\mu = np$ and $\sigma^2 = np(1-p)$ when n is is large or when p is close to 0.5 whether n is large or not. A typical rule of thumb is that the normal approximation to the binomial is reasonable when $np > 5$; we follow this rule of thumb when randomly generating binomially distributed values (using the more expensive binomial random number generator only when necessary).

The binomial distribution $b(x; n, p)$ has mean $\mu = np$ and $\sigma^2 = np(1-p)$ (exactly like its normal approximation). This means that the expected sample counts b_{sample} for a block b are distributed as follows:

$$\mu = b_{count} \frac{b_{insns}}{I}$$

$$\sigma^2 = b_{count} \frac{b_{insns}}{I} (1 - \frac{b_{insns}}{I})$$

or when the probability of any block being sampled is very small (true of the sampling intervals we use in this chapter where $I \geq 1000$), something which we will assume from now on,

$$\sigma^2 \approx b_{count} \frac{b_{insns}}{I}$$

In order to re-estimate the basic block count from the sample count we scale up the basic block count by the reciprocal of the probability that is was sampled, or I/b_{insns}. If a random variable X has mean μ_X and variance σ_X^2, the random value aX (the scaling of X by a constant a) has mean $\mu_{aX} = a\mu_X$ and variance $\sigma_{aX}^2 = a^2\sigma_X^2$. Note that flow properties (that the sum of the counts entering and leaving a block should be the same as the count for that block) are not preserved by this process.

Thus, the sampled estimate $b_{estimate}$ has the properties:

$$\mu = b_{count}\frac{b_{insns}}{I}\frac{I}{b_{insns}} = b_{count}$$

and

$$\sigma^2 \approx b_{count}\frac{b_{insns}}{I}(\frac{I}{b_{insns}})^2 = b_{count}\frac{I}{b_{insns}}$$

The fact that the $\mu_{b_{estimate}} = b_{count}$ is obvious and exactly what we want. The variance of the estimate values is not necessarily a very useful metric, as it does not allow us to compare the variability of blocks with different basic block counts. More useful is the *coefficient of variation*, or cv, which is defined as

$$cv = \frac{\sigma}{\mu}$$

and functions a standard deviation metric that allows us to compare the variability of estimated block counts with different means (a "relative standard deviation"). The coefficient of variability for our estimated basic block counts is:

$$\begin{aligned} cv &\approx \frac{\sqrt{b_{count}\frac{I}{b_{insns}}}}{b_{count}} && (4.1) \\ &\approx \frac{\sqrt{I}}{\sqrt{b_{count}}\sqrt{b_{insns}}} && (4.2) \end{aligned}$$

Thus, the coefficient of variationof a sample estimate

- increases in proportion to the square root of the sampling interval,
- decreases in proportion to the square root of the original, and block count
- decreases in proportion to the square root of the number of the instructions in the block.

The sample estimates for different basic blocks are independently distributed random variables, at least in the sense of sampling error. Obviously the original basic block counts are not statistically independent of each other. However, the outcome of the sampling process for a given basic block is entirely independent of the outcome of the sampling process for a different basic block.

A implementation of sampling would have to deal with a awkward detail of event-based sampling that violates the assumption of independence; that is, the possibility that the events generated by the program might have a periodic nature and occur in multiples of the sampling interval. For example, if a loop executes exactly 128 instructions per iteration and our sampling interval is some multiple of 128, then the sampling system will sample the same point in the loop on each iteration, leading to a non-representative profile. This problem is fairly easily dealt with[1] and does not affect simulated sampling in any case.

Re-estimation of Basic Block Counts

We have already discussed how we re-estimate basic block counts from our sample values (simply individually scaling each block count up by the reciprocal of the probability that it was sampled). This process does not preserve the expected properties of basic block counts in a control flow graph. The most obvious improvement to this is to consider basic blocks that are "equivalent" in a control flow graph sense; to look at basic blocks whose basic block counts will necessarily always be the same by properties of the flow graph. That is, two blocks b_1 and b_2 are "equivalent" if and only if:

- b_1 dominates b_2 and b_2 post-dominates b_1 or vice versa, and
- b_1 and b_2 are immediately contained the same loop OR neither b_1 and b_2 are in any loop.

This relation partitions our set of basic blocks into sets of equivalent basic blocks. Such a group of basic blocks will always have the same basic block count (except in unusual circumstances, such as exceptions that do not return or if the program calls functions such as `setjmp` and `longjmp`; the cases are rare and not especially amenable to optimization in any case). For example, in Figure 4.1, blocks A, D and F are all equivalent in a control-flow sense.

[1]Existing sampling systems generally use intervals that vary by small amounts; instead of an interval of 65536, they might use a range of sampling intervals from 65516 to 65556.

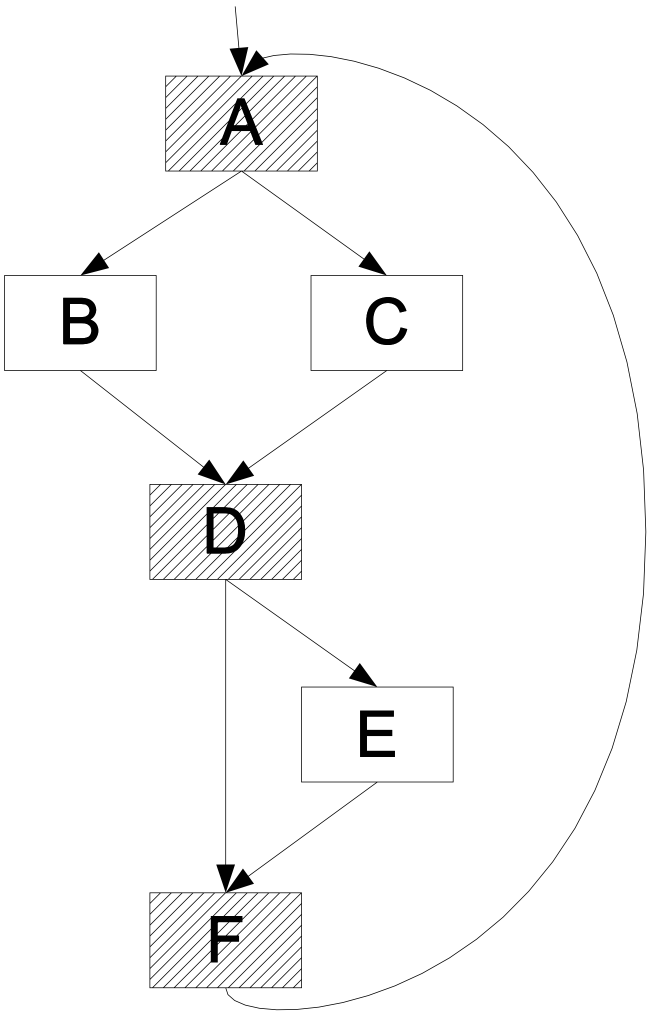

Figure 4.1: Equivalent Basic Blocks

We have already shown that the coefficient of variation of our estimated basic block counts decreases in proportion to the square of the number of instructions in a block. Considering all the blocks in a "equivalent" set of basic blocks as a single basic block and summing both the probability that one of these blocks was executed and the sample counts that each block actually had will result in a more accurate estimation of all of the blocks. In Section 4.2 we established that as the number of instructions in a block increased, the coefficient of variation of the estimated count for the block diminished in proportion to the square root of the number of instructions in the block. For example, suppose in Figure 4.1 block A has 20 instructions, D has 10 instructions and F has 10 instructions. If we consider blocks A, D and F as a single block with 40 instructions rather than 3 different blocks with 10 or 20 instructions each, the resulting estimate for the block counts will have a coefficient of variation that is half ($sqrt(10/40) = 1/2$) the size of the coefficients of variation for the smaller block counts (for D and F) and $\frac{1}{\sqrt{2}}$ of the coefficient of variation for the estimate of block A alone.

Another improvement that can be made is to take control-flow properties into account. At many blocks, we find that the blocks have a set of successors that they dominate (i.e. there is no other way to execute the successors without executing a block) or predecessors that they post-dominate, or both. We show a simple flow graph with these properties in Figure 4.2. In the example in Figure 4.2 we have one estimate for the basic block count of block D and can form another estimate for the basic block count of D by adding blocks B and C. Such a block's basic block count must be equal to the sum of its successors or predecessors (depending on structure). We can adjust the estimated block counts for the blocks accordingly.

The adjustment for the basic blocks is not an obvious task. Even in the simplest case, where a single basic block count is known to be the sum of two other basic blocks, there are three different block counts that may be adjusted to satisfy flow properties. Which block count or counts should we adjust, and by how much?

We proceed by using the insight that the blocks whose re-estimated basic block counts have the lowest expected variance are the blocks with the most reliable sampled basic block counts. We will produce a more reliable estimate of the final basic block counts by using some weighted combination of the different basic block counts.

If we have two different estimates of basic block counts b^i_{est} and b^j_{est} with variances $\sigma_{b^i_{est}}$ and $\sigma_{b^j_{est}}$, a weighted combination of the estimate using weighting s is:

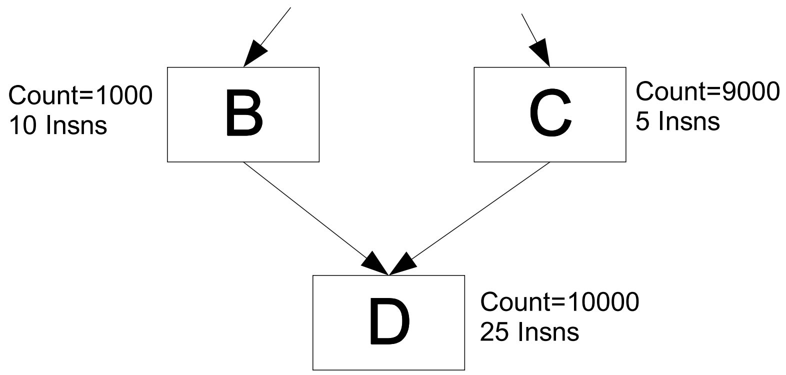

Figure 4.2: Patching Example

$$b_{final} = sb_{est}^{i} + (1-s)b_{est}^{j} \tag{4.3}$$

We derive a value of s that minimizes the variance of b_{final}. Presuming the two estimates of the basic block counts are independent, the variance of a linear combination of random values can be found by

$$\begin{aligned}\sigma_{b_{final}} &= s^2\sigma_{b_{est}^{i}} + (1-s)^2\sigma_{b_{est}^{j}} & (4.4)\\ &= \sigma_{b_{est}^{j}} - 2s\sigma_{b_{est}^{j}} + s^2(\sigma_{b_{est}^{i}} + \sigma_{b_{est}^{j}}) & (4.5)\end{aligned}$$

Differentiating in this equation over s, we find that the first and second derivatives of this equation are:

$$\begin{aligned}\frac{d(\sigma_{b_{final}})}{ds} &= -2\sigma_{b_{est}^{j}} + 2s(\sigma_{b_{est}^{i}} + \sigma_{b_{est}^{j}}) & (4.6)\\ \frac{d^2(\sigma_{b_{final}})}{ds} &= 2(\sigma_{b_{est}^{i}} + \sigma_{b_{est}^{j}}) & (4.7)\end{aligned}$$

Note that variances are always non-negative, and in this application are always non-zero; this means that the second derivative of $\sigma_{b_{final}}$ in terms of s is also always greater than zero. Thus if we solve $\frac{d(\sigma_{b_{final}})}{ds} = 0$, we get:

$$s = \frac{\sigma_{b^j_{est}}}{\sigma_{b^i_{est}} + \sigma_{b^j_{est}}} \tag{4.8}$$

which we know to be a minimum (and seeing as it is a unique solution, this value of s is a global minimum). We can also calculate the variance of the resulting basic block count $\sigma_{b_{final}}$ by substituting s back into Equation 4.3:

$$\begin{aligned} \sigma_{b_{final}} &= (\frac{\sigma_{b^j_{est}}}{\sigma_{b^i_{est}} + \sigma_{b^j_{est}}})^2 \sigma_{b^i_{est}} + (1 - (\frac{\sigma_{b^j_{est}}}{\sigma_{b^i_{est}} + \sigma_{b^j_{est}}}))^2 \sigma_{b^j_{est}} && (4.9) \\ &= \frac{\sigma_{b^i_{est}} \sigma_{b^j_{est}}}{\sigma_{b^i_{est}} + \sigma_{b^j_{est}}} && (4.10) \end{aligned}$$

Thus, we can repeat the process of combination as we have a new estimate of variance for the resulting estimate. This is useful for combining the estimate of block counts derived from a block and its predecessors with the block count estimates from its successors.

The procedure described above is carried out simultaneously, in both directions, for every block in the program that has neighbors that satisfy the preconditions for these analyses. An analogous procedure (not detailed here) is used to calculate basic block counts that are based on a difference between two different basic block counts (that is, in our running example shown in Figure 4.2, calculating a good estimate of the counts of B and C). These procedures are only used in a local fashion; we re-estimate each block's basic block count based on its predecessors and successors (if possible) but do not iteratively carry out this re-estimation procedure globally. Attempting to propagate better profile information through the control flow graph is theoretically possible but we found that the quality of the profile information was rapidly degraded by what is, after all, an approximate technique. Furthermore, the theoretical justification for using this re-estimation technique is that the estimates of in the original block counts are statistically independent[2] from block to block; Equation 4.4 depends on this assumption. Fortunately, the technique of pooling the block counts for control-flow equivalent blocks does a good job of propagating higher-quality information through the control-flow graph.

Another improvement of sampled block count estimates can be made under circumstances where, while we cannot express the basic block count of a block in

[2]In the sense of the errors introduced by simulated sampling, that is

terms of a precise sum of one of its predecessors or successors, there is nonetheless a useful inequality relation between the two. Looking back to Figure 4.1, we see that the count for basic block E will always be no greater than the count for basic block D. We can use this to improve the accuracy of basic block counts if our current re-estimation of block counts fails to satisfy the necessary inequality between the blocks.

We refer to sampled profiles that have been improved by using all of these techniques as "patched" and those that have not been altered as "unpatched". By far the biggest contribution to profile accuracy is made by taking profile equivalence into account. All of these techniques improve accuracy but do not guarantee better results. It is possible to worsen individual estimates as a result of using these techniques.

An important point to remember is that these patching techniques are nearly useless in re-estimating profile information for functions where there the total number of events is too small relative to the sampling interval. Thus, if a function has an average dynamic instruction count of 100 per invocation and is invoked 25 times in the course of an execution, we will not expect to have good information on this function if our sampling interval is 10000 - in fact, it is unlikely that we would see even one sample falling within this function. In the presence of so little information on function execution, no patching technique will provide a reasonable picture of the function's behavior. The one piece of information that we *can* obtain is that the function is probably not executed very frequently.

Unfortunately, such infrequently executed functions are often indistinguishable (when using sampled profiles) from functions that did not execute at all. We have chosen to avoid any techniques that cause block sample counts with zero estimates to be estimated as non-zero unless there is strong evidence otherwise (for example, a block with a non-zero sample count that dominates the block with the zero sample count). Otherwise we are put into the position of having to assign some small non-zero profile value to all the "cold" and "dead" blocks in the program.

4.3 Results

4.3.1 Our Sampling Experiments

We carried out our sampling experiments over the same set of benchmarks that was used in the previous chapter[3]. We evaluated 80 sampled profiles, using all combinations of:

- Two base profiles (SPEC "train" and SPEC "ref" profiles for benchmarks that has single profiles of this kind, or a pair of distinct profiles that appeared in the "train" and "ref" runs individually). To compare our results with the results from our work on exact profiles, we continue with our practice of not aggregating "train" and "ref" profiles and evaluation runs.
- Unpatched and patched re-estimation techniques
- Sampling intervals from once every 100, 1000, 10000 and 100000 instructions.
- Five different random number seeds (thus producing five entirely distinct sampled profiles for each combination of the above factors)

There is no special significance to the use of powers of 10 for the sampling intervals. A sampling interval of 100 instructions is too low to be practically implemented in a real sampling system, but we were interested in observing what happened to sampling behavior at very high rates of sampling. At the upper limit of a sampling interval of one in 100000 instructions, further overhead savings in terms of sampling are marginal. For example, pessimistically assuming that there are 500 instructions in the code that records machine state at each interrupt (and assuming that these 500 instructions are, on average, no more or less expensive than the instructions in the program being monitored), the overhead of sampling at the rate of one in 100000 is 0.5%. Reducing overhead beyond this point is nearly pointless.

We gathered some results for unreasonably small sampling intervals (every 10 dynamic instructions). These sampling intervals might in theory provide some

[3]Unfortunately, the huge range of profiles tweaked some bugs in the `cc` optimizer; if a benchmark failed to optimize with any of the sampled profiles we were forced to remove it from consideration. Otherwise we run the risk that there is some systematic class of profiles that causes these compiler failures and that the remaining profiles might be distributed in a far from random fashion.

information about a continuum of sampling usefulness ranging from exact profiles - which can be considered as a sampling interval of 1 - all the way to very large sampling intervals. However, we are inclined to limit our discussion of such sampling intervals for a number of reasons. First, such intervals have no practical implementation - sampling at such a high rate is would impose astonishingly high overhead (probably higher than simply instrumenting the program). Second, the analytic work we have done depends on the sample interval being large relative to individual block sizes. Third, our re-estimation procedures were not designed with such low sampling intervals in mind and may behave in less than ideal fashion.

For example, if a block has 20 instructions in it, and the sampling interval is 10 dynamic instructions, we will know for a fact that the sample count for the block is strictly greater than the actual number of times the block executed. If the sample count is zero, we will know that the block was not executed at all; a situation that we can only guess at if we use more reasonably sized sampling intervals. Therefore, there are a number of ways that such highly accurate sampling information could be made use of that are useless for higher sampling intervals. By not implementing these optimizations, our system may understate the theoretical accuracy possible with sampled profiles produced with extremely low sampling rates.

Given the huge number of profiles to evaluate we use a smaller number of repeats (3) of each evaluation run and scale back the number of evaluation runs for the extremely long-running SPEC 2000 "ref" runs to a single run. This introduces a slightly larger experimental error, but the overall effects of the choice of sampled profiles for these runs are much larger than the levels of experimental error (for these runs) that we established in the chapter on exact profiling. Generally speaking, it was the shorter evaluation runs for which experimental error tended to obscure any observed effects.

We evaluate our profile-driven optimizations on all of the evaluation runs available for each benchmark. For results describing the overall effect of sampling, we will scale the average cycle count on any given run by the average cycle count that results from using the *exact* profile for that run. Thus, an adjusted score of 1.02 for a given sampled profile and evaluation run indicates that the binary produced by using that training profile was 2% slower (when executing the evaluation run) than the binary produced by the exact profile.

Optimizer	Relative execution time			
	Minimum	Maximum	Geo. mean	Std. Deviation
`alto`	0.86	1.17	1.016	0.023
`cc`	0.93	1.29	1.016	0.041

Table 4.1: Overall Sampled Profile Performance and Variability (relative to non-sampled base profile)

4.3.2 Aggregate Sampling Results

Overall Effect of Sampling

The first issue that we will consider is the overall effect of profile-directed optimization using our different sampled profiles. In the work on exact profiles, we found that certain benchmarks had little to no interesting profile-directed optimization performance variation. However, in that context it was impossible to determine whether this was an inherent property of the benchmark or of the limited selection of benchmark runs. We can look at our benchmarks again in the context of sampled profiling to see the magnitude of the variability of profile-directed optimization and whether the variability of profile-directed optimization for sampled profiles is connected to the variability of profile-directed optimization for exact profiles.

Our metric of profile-driven optimization variability is somewhat easier to calculate in this context, as we have the same number of profiles per benchmark and a quite large number at that. We present the mean, maximum, minimum and standard deviation of the adjusted scores (ratio of the sampled profile's performance and the base profile's performance) across our combinations of optimizers, benchmarks, sampled profiles - considering all our combinations of sampled profiles for now - and evaluation runs in Table 4.1 and Table 4.2.

Table 4.1 shows a fairly similar penalty for the use of sampling overall. There is an average slowdown of 1.6% over all of our different sampling types, which is identical in both optimization contexts. The variability of the populations is significantly different, with the standard deviation of the slowdowns over `alto` almost half that of `cc`.

While the mean loss of profile usefulness associated with sampling is not large (a mere 1.6%), is it worth remembering that the mean speed-up associated with use of profile-driven optimization over the non-profile-driven equivalent is quite small (around 3%, as shown in Section 2.2), so that the mean effect of using sampling is quite substantial relative to the total expected benefits of profile-driven optimization. Of course, we cannot make the simplistic claim that sampling reduces our mean profile-driven optimization improvement by half (this constitutes a "fallacy of composition"); further, there is a great deal of per-benchmark variability in a) how useful exact profiles are relative to sampled profiles and b) how useful exact profiles are relative to not using profiles at all.

Table 4.2 shows that benchmarks can have high profile-driven optimization variability using sampled profiles. On the whole, sampling will worsen average performance - only a handful of benchmarks (`alto/bzip`, `cc/crafty`, `cc/parser`) show a benefit from the use of sampled profiles, and in these cases the speed-up is trivial (relative performance is only 1% better than exact profiles). On the other hand, a relative slowdown of 3-4% (again, as opposed to exact profiling) is quite common among our benchmarks.

Some benchmarks which showed very little profile-driven optimization variability in the exact profiling case show substantial variability when optimized with sampled profiles. Profile-driven optimization variability is quite dependent on the class of profiles under consideration; it is not an inherent quality of the benchmark. Of course, some benchmarks show little PDO variability whether we use sampled profiles or exact profiles. It is reasonable to suspect that some benchmarks simply do not spend much execution time in functions that can be improved further by our profile-driven optimizers.

Usefulness Behavior at Different Profiling Intervals

The next task is to discover how our different sampling intervals affect optimization performance. Intuitively, one would expect performance to become steadily worse as the sampling interval increases and the accuracy of the sample profiles decreases. However, we have already shown that the connection of usefulness and accuracy is a problematic one in at least one other context (that of exact profiles). We will examine the profile-driven optimization performance of patched profiles (we will examine the usefulness contribution of patching profiles in the next section).

Table 4.3 summarizes the average profile-directed optimization performance for the different sampling levels over all our benchmarks and evaluation runs. For `alto`, higher sampling intervals cause both worse overall performance and larger variability. No such systematic effects for `cc` can be observed at all, despite the slight decline in standard deviation of profile-directed optimization perfomance as

Optimizer	Benchmark	Relative (to non-sampled profile) execution time			
		Minimum	Maximum	Geo. Mean	Std. Deviation
alto	art	0.99	1.01	1.00	0.003
	bzip2	0.93	1.17	0.99	0.028
	compress	0.96	1.04	1.00	0.012
	crafty	0.99	1.08	1.02	0.012
	gap	1.00	1.09	1.03	0.010
	go	0.86	1.13	1.04	0.033
	gzip	0.99	1.04	1.01	0.010
	ijpeg	0.98	1.07	1.01	0.009
	li	1.00	1.09	1.02	0.014
	m88ksim	0.99	1.16	1.04	0.028
	mcf	0.99	1.04	1.00	0.005
	twolf	0.98	1.05	1.02	0.011
	vortex2000	1.00	1.08	1.04	0.018
	vpr	0.94	1.08	1.00	0.021
cc	ammp	0.94	1.07	0.99	0.016
	bzip2	0.98	1.18	1.02	0.019
	compress	0.97	1.06	1.02	0.016
	crafty	0.94	1.04	0.99	0.025
	equake	0.99	1.03	1.00	0.009
	ijpeg	1.01	1.08	1.03	0.011
	li	0.96	1.16	1.04	0.047
	m88ksim	0.95	1.29	1.08	0.080
	mcf	1.00	1.04	1.01	0.007
	parser	0.93	1.05	0.99	0.019
	vortex2000	0.94	1.13	1.02	0.031

Table 4.2: Overall Sampled Profile Performance and Variability By Benchmark (relative to base, non-sampled profile)

Optimizer	Interval	Relative (to non-sampled profile) execution time			
		Minimum	Maximum	Geo. Mean	Std. Deviation
alto	1000	0.86	1.08	1.012	0.022
	10000	0.89	1.13	1.015	0.024
	100000	0.91	1.17	1.020	0.030
cc	1000	0.94	1.24	1.016	0.041
	10000	0.95	1.27	1.018	0.040
	100000	0.94	1.23	1.018	0.036

Table 4.3: Sampled Profile Performance and Variability By Sampling interval (relative to base, non-sampled profile)

the sampling interval increases. [4]

Effects of Sampled Profile Patching

To evaluate the effects of patching sampled profiles, we will carry out what is known as a paired comparison. That is, given a single optimizer, benchmark, evaluation run and set of sampling settings (random number seed, sampling interval, base profile) we have two different sampled profiles - patched and unpatched. We will proceed by using a statistical technique (the "Paired-Sample T-Test") that evaluates the mean differences of pairs of values. We will examine both usefulness and accuracy values with this technique. Once again, the usefulness values that we evaluate will be scaled so that the average profile-driven optimization performance of the binary produced by an non-sampled version of the profile is equal to 1.0 on each evaluation run.

The "Paired-Sample T-Test" allows us to estimate the average difference of two paired variables and the likelihood that this difference is statistically significant.

First, we will evaluate whether or not our patching technique contributes to improved profile accuracy.

As Table 4.4 shows, patching almost always results in an improvement of profile accuracy scores. Only for a few combinations of metrics and benchmarks does it lead to a degradation of profile accuracy. This is not surprising in the case of the coverage metrics, as the patching process never causes us to set the estimated count of a block to zero if it was previously non-zero.

[4] Using an F-test to compare these variances, we did not see that differences between these

Optimizer	Benchmark	K(01) improvement	ENT improvement
alto	art	0.01	-1.18
	bzip2	0.02	1.92
	compress	0.00	5.70
	crafty	0.02	0.94
	gap	0.03	0.71
	go	0.03	2.54
	gzip	0.03	0.93
	ijpeg	0.02	1.17
	li	0.03	1.85
	m88ksim	0.02	1.24
	mcf	0.02	4.92
	twolf	0.03	0.33
	vortex2000	-0.04	-0.28
	vpr	0.03	1.78
cc	bzip2	0.01	1.95
	compress	0.00	2.64
	crafty	0.02	0.96
	ijpeg	0.03	1.39
	li	0.03	1.91
	m88ksim	0.02	1.41
	mcf	0.01	1.26
	vortex2000	-0.03	-0.22
	ammp	0.04	0.16
	equake	0.02	0.07
	parser	0.01	1.68

Table 4.4: Profile accuracy differences, showing differences between patched scores and unpatched scores for two accuracy metrics - key-matching at the 0.1 level and relative entropy.
All differences were significant at the 0.01 level except the relative entropy score for `equake` under `cc`. Positive scores represent improvement. For a rough comparison, the majority of relative entropy scores in the exact profile work were under 5.0, so a difference of 1-2 is substantial.

Of course, our experiences with exact profiling lead us to be skeptical about whether these accuracy improvements will lead to better profile-driven optimization performance. What is the overall effect on average performance? We evaluate the improvement due to patching across our range of benchmarks in Table 4.5.

Table 4.5 shows that the overall usefulness improvement due to patching is not impressive. On most benchmarks the effect is either trivial and/or statistically insignificant. Worse still, for several benchmarks the average effect is negative (that is, patching produces worse performance). Only a few benchmarks do we see a definite improvement on profile-directed optimization performance as a result of patching; these cases generally show larger magnitude of change than the negative improvement cases.

We found that patching produced a slightly larger positive effect when only higher sampling levels were considered, but the overall picture remained the same even at our highest sampling intervals. It is impossible to conclude that patching is worth doing from these equivocal results.

Effect of Different Base Training Profiles

Another important issue in the use of sampled profiles is the choice of the base from which the sampled profiles are derived and how well they match the run-time behavior of the evaluation run. We attempt to evaluate whether, if we know the behavior resulting from sampling for a given benchmark, base profile and evaluation run, we can predict the effects of using different base profiles.

The results that we gathered in this section up to this point intentionally avoid these questions. By scaling our usefulness scores to be relative to the usefulness scores of the base profile we have ignored the first-order effects of the base profile choice. An example will clarify this.

Suppose, for a single evaluation run and benchmark B^1, the average cycle count of a binary optimized with the `ref` base profile is 1.0 Gcycles and that the average cycle count of a binary optimized with the `train` base profile is 1.1 Gcycles. Suppose further that sampling has a consistent effect and generally produces a normally-distributed set of usefulness scores clustered around 1.02 times the exact profile with a standard deviation of 0.01. If the effects of sampling are similar when using either `ref` or `train` as base profiles, then we will see a set of scores around 1.02 Gcycles corresponding to sampled profiles based on `ref` and a set of scores around 1.12 Gcycles for sampled profiles based on `train`. If we are mea-

variances were statistically significant at the 0.05 level

Optimizer	Benchmark	Mean diff.	t-value	Sig. value.	Summary of difference
alto	art	0.0002	0.506	0.615	Not significant
	bzip2	0.0201	8.119	0.000	Patched performs worse
	compress	-0.0104	-9.322	0.000	
	crafty	-0.0100	-7.376	0.000	
	gap	-0.0031	-2.045	0.045	
	go	-0.0017	-0.786	0.435	Not significant
	gzip	0.0009	1.413	0.163	Not significant
	ijpeg	-0.0016	-0.993	0.325	Not significant
	li	0.0055	3.751	0.000	Patched performs worse
	m88ksim	0.0016	1.003	0.320	Not significant
	mcf	0.0007	0.690	0.493	Not significant
	twolf	0.0008	0.613	0.542	Not significant
	vortex2000	-0.0075	-3.794	0.000	
	vpr	-0.0045	-1.637	0.107	Not significant
cc	bzip2	-0.0092	-4.623	0.000	
	compress	0.0012	0.727	0.470	Not significant
	crafty	0.0064	2.769	0.008	
	ijpeg	-0.0060	-7.585	0.000	
	li	-0.0186	-4.439	0.000	
	m88ksim	0.0156	2.391	0.020	Patched performs worse
	mcf	-0.0048	-9.468	0.000	
	vortex2000	-0.0139	-5.598	0.000	
	ammp	0.0013	0.624	0.535	Not significant
	equake	-0.0001	-0.132	0.896	Not significant
	parser	0.0028	2.016	0.048	Patched performs worse

Table 4.5: Profile usefulness differences (scores are normalized relative to non-sampled profile), showing difference between patched scores and unpatched scores (positive values indicate that mean value of unpatched score was lower i.e. better); significance tested at 0.05 level

suring the effects of sampling alone, we obviously do not want to pool these two sets of scores together and conclude that the range of outcomes produced from sampling runs from around 1.0 Gcycles all the way to 1.12 Gcycles. Thus, we want to scale by the exact profile's usefulness score before evaluating sampling.

However, this throws away any distinction between the above case, and one for (say) benchmark B^2, where the cycle count of a binary optimized with the `ref` base profile is 1.0 Gcycles and that the average cycle count of a binary optimized with the `train` base profile is 1.01 Gcycles, while the relative behavior of sampling is the same as for benchmark B^1. For such a benchmark, the variability due to choice of base profile will actually be dominated by the effects of sampling.

Which case is more typical - benchmark B^1 or benchmark B^2? Table 4.6 shows the relative size of contributions to overall variability in our sampling experiments due to profile choice as opposed to the contributions due to sampling settings (whether sampling interval size, the use of patching, or the randomness associated with sampling).

From Table 4.6, it is clear that both types of behavior exist in our benchmark sets. For example, profile choice is much less important in the `cc` context for the `mcf` benchmark; its contribution to the variability in our experiment is comparatively small, whether we consider only patched profiles or the entire population of profiles. On the other hand, profile choice in the `crafty` benchmark dwarfs the effects of the other sampling factors.

4.3.3 The Connection Of Profile Usefulness and Accuracy with Sampled Profiles

We show the correlation between profile usefulness and profile accuracy for our sampled profiles in Table 4.7, using patched profiles only (the results are very similar for unpatched profiles, or when considering both types of profiles pooled together). As in our work on exact profiles, we find substantial variation from benchmark to benchmark. Entropy-based profile accuracy metrics perform fairly well, although even the best profile accuracy metrics work well on only a handful of profiles.

Overall, the connection between usefulness and accuracy in the context of sampled profiles is more easily discovered to be significant[5] (given the much

[5]Nearly all of the Spearman rank-correlation coefficients presented are statistically significant; the use of sampled profiles gave us far more data points to work with thus greatly increasing our chance of discovering significant correlations. However, a statistically significant correlation

Optimizer	Benchmark	Variability due to profile choice / variability due to sampling setting choice)	Same ratio, patched profiles only
alto	art	0.90	0.39
	bzip2	0.28	0.71
	compress	7.24	3.10
	crafty	19.71	12.70
	gap	25.35	17.33
	go	11.02	4.37
	gzip	0.53	0.17
	ijpeg	0.00	1.15
	li	0.06	0.49
	m88ksim	7.95	2.21
	mcf	0.00	2.84
	twolf	0.69	3.70
	vortex2000	3.48	1.19
	vpr	19.72	8.82
cc	bzip2	6.03	8.18
	compress	45.23	29.33
	crafty	23.56	27.56
	ijpeg	31.48	17.20
	li	120.94	57.37
	m88ksim	27.30	23.18
	mcf	0.24	0.12
	vortex2000	9.86	5.70
	ammp	0.44	0.15
	equake	15.56	45.26
	parser	0.51	0.03

Table 4.6: Ratio between variability introduced by profile choice and variability introduced from sampling settings

CONTEXT	BENCH	ENT	STC	K01	FEENT	FESTC	FEK01
`alto`	art	0.18	-0.14	-0.11	-0.05	-0.12	-0.12
	bzip2	-0.11	-0.03	-0.13	0.06	0.01	-0.07
	compress	0.64	0.00	-0.06	-0.19	-0.07	-0.09
	crafty	0.43	-0.10	0.64	0.48	-0.15	0.07
	gap	0.63	0.03	0.51	0.40	0.01	0.54
	go	0.55	0.64	0.54	0.59	0.67	0.51
	gzip	0.71	0.07	-0.01	0.18	0.10	0.10
	ijpeg	-0.25	-0.15	-0.22	-0.08	-0.17	-0.17
	li	0.05	0.14	0.26	0.38	0.14	0.38
	m88ksim	-0.36	0.25	0.13	-0.12	0.24	0.13
	mcf	-0.20	0.16	0.09	0.19	0.18	0.12
	twolf	0.20	-0.04	-0.10	0.14	-0.04	-0.08
	vortex2000	-0.31	0.27	0.08	-0.23	0.26	0.01
	vpr	-0.11	-0.02	0.37	-0.09	-0.13	0.20
	Mean	0.19	0.09	0.16	0.14	0.08	0.12
`cc`	bzip2	0.56	0.38	0.17	0.27	0.32	0.39
	compress	0.56	0.32	0.19	0.04	0.33	0.26
	crafty	-0.25	-0.05	-0.12	-0.09	-0.06	-0.10
	ijpeg	0.48	0.62	0.60	0.57	0.60	0.54
	li	0.54	0.44	0.51	0.29	0.42	0.24
	m88ksim	0.19	-0.06	0.00	0.36	-0.09	0.04
	mcf	0.91	0.15	0.24	-0.68	0.11	0.10
	vortex2000	-0.35	0.16	0.09	-0.27	0.12	-0.06
	ammp	-0.08	-0.34	-0.42	0.01	-0.34	-0.23
	equake	-0.18	-0.08	-0.10	0.08	-0.08	0.15
	parser	0.04	0.07	0.07	0.09	0.06	0.05
	Mean	0.22	0.14	0.11	0.06	0.13	0.13

Table 4.7: The connection of usefulness and accuracy: aggregated r_s scores over optimizers, benchmarks and different comparison metrics for sampled profiles

larger number of training profiles involved) but its magnitude is not, as a rule, substantially greater than the connection of usefulness and accuracy for exact profiles. In both cases, only a minority of combinations of benchmark and optimizer yielded strong and significant connections between usefulness and accuracy while most did not.

Once again, the results presented are only a subset of the profile accuracy metrics that we evaluated. Overall, we evaluated 48 accuracy metrics, using 16 different accuracy primitives (6 levels each of key- and weight-matching, static and dynamic coverage, rank correlation of profile scores and relative entropy) and 3 different ways of using these primitives (over every block in the program, over only function entry count data and a weighted combination of the results of these metrics when applied to every function in the program individually). The results presented in this section perform substantially better than the alternatives, and yet their predictive power as regards profile-driven optimization usefulness is very low.

The discovery that profile-driven optimization performance is very weakly linked to profile accuracy in a second domain (the first being the domain of exact profiles) raises serious doubts that such metrics have any strong connection to accuracy in either of our optimizers.

4.3.4 Explanatory Hypotheses

We have established that the connection between usefulness and accuracy is weak on non-existent for many of our benchmarks. This raises a disturbing problem - if there is little connection between usefulness and accuracy, why is sampling generally worse than exact profiling? We suggest two hypotheses that might explain these results in this section. Both rely on the idea that profile usefulness is strongly localized and affected by statistical sampling in a way that is not captured by our accuracy metrics.

"Fragile Function" Hypothesis

The hypothesis that we call the "Fragile Function" hypothesis is as follows:

Per-function behavior is quite similar from benchmark run to benchmark run, but the weight of function execution differs. Thus a function that is executed heav-

---

does not guarantee a strong connection (e.g. `alto/li`'s relative entropy correlation of 0.05) or a connection that runs in the appropriate direction (e.g. `cc/crafty`, for which no accuracy metric positively correlated with improved performance).

ily in a reference run may be only executed fairly lightly in a training run. If we use sampling to gather a profile of the training run, the profile of that critical training run that we gather may be too inaccurate to be very useful to improve profile-driven optimization performance on the evaluation run.

The implications of the "Fragile Function" hypothesis are interesting.

Firstly, if the "Fragile Function" hypothesis is true, evaluating sampling systems using the "resubstitution" profile is a dangerous waste of time. For a given evaluation run, the functions for which it is important to have a high degree of accuracy are usually heavily executed. If we use the "resubstitution" case as the basis for our sampled training profiles, these heavily executed functions will have quite accurate profiles in the sampling case.

Secondly, the "Fragile Function" hypothesis implies that the increasing the sampling interval will not cause a gradual decrease in sample usefulness, and that even quite small sampling intervals may cause a sharp drop-off in profile usefulness as compared to the exact profile. If an acyclic region with 20 instructions that is important in the reference run is only executed 1000 times in the training run that we sample, and our sampling interval is on the order of or greater than 20000, we will expect that the sampled profile for that region will be extremely inaccurate.

In Chapter 5 we in fact show that profile usefulness is localized in a comparatively small number of functions, which lends some support to this hypothesis.

"Fragile Branch" Hypothesis

A similar hypothesis to the "Fragile Function" hypothesis is the "Fragile Branch" hypothesis. The distribution of branch taken frequency is not a uniform distribution from 0.0 to 1.0; in fact, it is a comparatively idiosyncratic (in statistical terms) distribution with two large bulges at 0.0 and 1.0 and a substantial bulge around 0.5. That is, most conditional branches are either always-taken, never-taken, or taken exactly 50% of the time (often plus or minus a very small number of iterations).

An implication of this is that many of our branches are very close to perfectly even. The sampling process, however, is almost certain to upset this balance. If optimizations attempt to favor the more-executed path, even a small perturbation from sampling can result in poor optimization performance. Such a small perturbation can occur even when the sampling interval is very low and the sampled profile is, overall, relatively accurate.

A reason that perturbations of these branches might have disproportionate effects on profile-driven optimization performance is that it is exactly these branches that require profile feedback. Often, branches that are nearly always taken or never taken are very easy to statically estimate - error exits from loops and loop closing branches are two examples. Even in systems that do not explicitly generate a static profile, many optimizations are written in such a fashion as to have implicit models of the behavior of such branches.

Most loop optimizations rest on the assumption that loop-closing branches are frequently taken whether they use profiling information or not. Even if sampling causes substantial perturbations in the estimated loop count, it is unlikely to cause different behavior in loop optimizations (an optimizer will usually optimize a loop with estimated iteration count of 800 the same way that it will optimize a loop with estimated iteration count of 1000). Thus, perturbations in the estimated counts of blocks around loop branches are comparatively unlikely to cause bad profile-directed optimization behavior.

Thus, a sampled profile seems quite accurate because it "gets loop behavior right" may not actually perform very well for profile-driven optimization because it has gotten some amount of conditional branch behavior wrong.

For example, if the blocks B and C in Figure 4.1 are both executed 50 times in the base profile, and each block has a 10% chance of each of its executions being recorded by the sampling process (an artificially low sampling interval), then we will expect that the estimated counts of B and C will differ in the final profile (the chance of an unpatched estimation process assigning the same estimate to both blocks is equal to the probability that two values taken from the binomial distribution $b(50, 0.1)$ are equal - below 15%).

More generally, the difference between blocks B and C after sampling approximates the difference between two normally distributed values. If the original block count was 1000000 and the sampling interval was 10000, the probability that the two estimates will be within 5% of each other is a mere 25%, and the probability that they will be within 10% is about 50%. The variation of this difference increases in direct proportion to the sampling interval and in direct proportion to the reciprocal of the block count.

To summarize, while sampling can yield fairly accurate profiles, it is unlikely to preserve accurate branch predictions between branches that are balanced nearly 50/50. If optimizers are very dependent on getting accurate predictions at this level, the usefulness of such profiles will be greatly reduced even if the overall accuracy of the profiles remain high.

4.4 Conclusion

Our results suggest that gathering profiles via sampling can cause substantial degradations in profile-driven optimization performance, often degrading profile-driven optimization performance to the point of rendering PDO worthless.

We hypothesize that this is due to inherent characteristics of the sampling process - the fragile branch and fragile function hypotheses suggest reasons why even statistically minor disruptions of profile data can cause substantial degradations in profile-driven optimization performance.

It may be possible to reduce the effect of these inaccuracies by modification of our optimization systems. Both of our optimizers were given no information that the sampled profiles were in any way less accurate that the exact profiles we have used in other work. It is possible that optimizations could be made more "robust" in the presence of unreliable information. If an optimization pass was given information that a given conditional branch was not actually observed to be taken 60% of the time, but rather that this branch frequency was an estimate and that the true branch frequency has a substantial possibility of being under 50%, the optimization might be able to behave more conservatively - not carrying out optimizations that would cause a disproportionate penalty if indeed the branch frequency was 49% and not 60%.

Profile-driven optimization using basic-block profiles is not the only application for profiling systems. There are a number of situations when exact profiling is simply not possible and/or not desirable, and in these situations using sampling techniques may become necessary:

- when no instrumentation system is implemented,
- when events of interest in question are not measurable by instrumentation code,
- when instrumentation is too disruptive of program behavior,
- when the overhead of instrumentation too high, or
- when only a approximate knowledge of an event is required (for example, when we are gathering feedback for a human rather than a profile-directed optimizer).

However, if carrying out automatic profile-driven optimization using basic block counts, our results suggest that sampling-based systems may provide poor performance and that the tradeoff of profiling overhead versus improved performance will turn out to be not worth making. This point is especially telling when we consider that, if the overhead of an exact profiling method is 25% and that of a sampling method is 0.25%, the relative difference between these overheads overhead is not $25/0.25$ - a factor of 100, but instead $125/100.25$ or around a factor of 1.25. The fact that the underlying program task of the training profile must be run regardless of the profiling method reduces the significance of differences among profiling methods. This is particularly true for the normal usage of profile-directed optimization (where the requirement is that the gathering of training profiles does not take an unreasonable length of time - a factor of 10 might be prohibitive), as opposed to more specialized circumstances (such as a "continuous profiling" system where profiling is always occurring and thus the overhead should be imperceptible and certainly on average smaller than the likely benefits of optimization).

The position on the connection of usefulness and accuracy that we developed from our observation of exact profiles remains unchanged from our observation of sampled profiles. As before, the connection of profile accuracy with profile usefulness is:

- Inconsistent across benchmarks - some benchmarks show a stronger connection between usefulness and accuracy but most show a comparatively weak connection
- Statistically significant for some benchmarks
- Of moderate strength for the benchmarks where there is a connection; again, it is rare that our r^2 values (corresponding to percentage of the variation explained by accuracy) are greater than 0.5.

We found certain large-scale effects - that generally, sampling both degraded accuracy and profile usefulness, and that patching reliably improved accuracy but only sometimes improved usefulness, but these sort of results were not sufficient to generate assertions about the connection of usefulness and accuracy. When we evaluated usefulness and accuracy within the population of an entirely patched or entirely unpatched set of profiles, using the same base profile and different sampling intervals and random number seeds, we found little to no reliable association with any whole-program metric of usefulness and accuracy.

Chapter 5

Systematic Variation of Profiles

5.1 Introduction

In this chapter we present a method to localize the effects of profile-driven optimization at a per-function level. We use statistical methods to efficiently determine whether profile data associated with individual functions has any effect on overall profile-driven optimization outcomes and if so, how much.

This material is of key importance in explaining the results that have come before. We have conjectured that individual functions might have a disproportionate effect on profile-driven optimization outcomes, thus rendering our whole-program accuracy metrics ineffective.

We present results on localization of usefulness "after the fact" (that is, localizing usefulness effects for a given profile and evaluation run pair after we ran this pair as an experiment), which we were able to do with a high level of success. We also present results that show our attempt to predict these 'more useful' functions "before the fact" - an attempt that was completely unsuccessful.

5.1.1 Experimental Design: Factorial Experiments

In this chapter, we are interested in looking at experimental results may be influenced by many different factors. It is computationally expensive to attempt to evaluate each factor on its own, while holding all the other factors constant. Worse, it is ineffective some of these factors may interact with other factors.

For example, optimizations may overlap in such a way that they either reduce the possibilities for the other optimization to work, or they may actually create more possibilities for the other optimization to work. An example of the former case is the interaction between some cases of procedure inlining and instruction cache optimizations. Inlining small functions may reduce i-cache misses. However, the same benefits might be achieved by instruction cache placement optimizations. Suppose optimization o_{inline} is procedure inlining and o_{icache} is whole-program cacheplacement. While, on their own, these optimizations might each yield a speedup of 10%, the combined effect of using these optimizations together may yield a speedup of only 17%. Thus we might write that the interaction term $o_{inline} \times o_{icache}$ is actually -3% to account for the "dysergy" between optimizations (dysergy being the reverse of synergy).

In other cases, optimizations may actually be more effective in combination. There is usually a positive interaction between register allocation and other optimizations that increase register pressure, such as global code motion.

Thus, the simple-minded "One Factor At-A-Time" methodology, where a single factor is experimented on while all other are held constant, is not only computationally inefficient, it misses potentially important interactions. We use a comparatively simple methodology known as factorial designs in this chapter to conduct and analyze our experiments[1].

As a rule, the design and analysis of experiments using only 2-level factors is far simpler than that of experiments involving 3 or more levels per factor, so we confine ourselves to using 2-level factors in this chapter. This is a very common statistical practice, and 2-level factors fit very well with most of our uses of factorial analysis in this chapter.

Table 5.1 shows the 2^3 factorial design over 3 factors, A, B, and C (conventionally I is used to represent the overall mean). There are 8 parameters to estimate for a full factorial model of the experimental results:

- 1 main effect (I, or the overall mean for the whole experiment),
- 3 single-factor effects (A, B, C),
- 3 two-factor interactions ($A \times B$, $A \times C$, $B \times C$), and
- 1 three-factor interaction ($A \times B \times C$)

[1]"Design of Experiments" is a thriving research area, and factorial designs are a very basic method from this area. They are comparatively easy to implement and analyze, but in many ways inflexible and inefficient. An introduction to experimental design is contained in [15], though there are dozens of works in this area

Experiment	A setting	B setting	C setting	AB interaction term
e_1	0	0	0	1
e_2	0	0	1	1
e_3	0	1	0	0
e_4	0	1	1	0
e_5	1	0	0	0
e_6	1	0	1	0
e_7	1	1	0	1
e_8	1	1	1	1

Table 5.1: 2^3 Full Factorial Experiment Runs (with example interaction term)

In this work, we use factor levels '0' and '1' and the 'logical equivalence' operation to represent interactions; this is equivalent to using factor levels '-1' and '1' and using straight multiplication to represent interactions (a somewhat more common representation). The difference is purely notational. However, it should be noted that given this notation, the interaction term for $A \times B$ will be 1 if A and B are different and 0 otherwise.

The 2^3 factorial design requires 8 experiments. To estimate the effect of a factor or interaction, we calculate the difference between the experiments where the factor is high and the experiments where the factor is low. Thus, the factor and interactions can be calculated as follows:

$$I = 1/8(e_1 + e_2 + e_3 + e_4 + e_5 + e_6 + e_7 + e_8) \quad (5.1)$$
$$A = 1/4((e_5 + e_6 + e_7 + e_8) - (e_1 + e_2 + e_3 + e_4)) \quad (5.2)$$
$$B = 1/4((e_3 + e_4 + e_7 + e_8) - (e_1 + e_2 + e_5 + e_6)) \quad (5.3)$$
$$C = 1/4((e_2 + e_4 + e_6 + e_8) - (e_1 + e_3 + e_5 + e_7)) \quad (5.4)$$
$$AB = 1/4((e_1 + e_2 + e_7 + e_8) - (e_3 + e_4 + e_5 + e_6)) \quad (5.5)$$
$$AC = 1/4((e_1 + e_3 + e_6 + e_8) - (e_2 + e_4 + e_5 + e_7)) \quad (5.6)$$
$$BC = 1/4((e_1 + e_4 + e_5 + e_8) - (e_2 + e_3 + e_6 + e_7)) \quad (5.7)$$
$$ABC = 1/4((e_2 + e_3 + e_5 + e_8) - (e_1 + e_4 + e_6 + e_7)) \quad (5.8)$$

Obviously, the approach of testing every factor level against every other will be very expensive as the number of factors and/or factor levels grows. Later in this chapter, we evaluate a model that has 11 2-level (on or off) factors. Such a model would require 2048 (2^{11}) experiments to perform a full factorial exper-

Experiment	A setting	B setting	C setting
e_1	0	0	1
e_2	0	1	0
e_3	1	0	0
e_4	1	1	1

Table 5.2: 2^{3-1} Factorial Experiment Runs (one alternative)

iment; at about 20 minutes of computer time per experiment per benchmark, it would take almost a month to evaluate a single benchmark. If we are willing to make some simplifying assumptions about our model, we can make these types of experiments more tractable.

The 2^{3-1} experiment requires only 4 experiments (as opposed to 8 for the full factorial 2^3 experiment. Table 5.2 gives one such design.

This raises a question - how can we estimate 8 parameters using only 4 experiments? The simple answer is, we cannot. We must "sacrifice" certain parameters. In this case, we design our experiment so that certain factors are "confounded" with each other. That is, we will organize our experiment so that the factor A is grouped with BC, B with AC, C with AB and I with ABC (note that in Table 5.2 C is in fact equal to AB). This means that we will not be able to tell whether an effect we observed is due to A acting along or the interaction of BC - these effects will be pooled together into a single term. The three-factor effect ABC in this case is completely 'sacrificed', as it's effect is indistinguishable from the group mean.

$$I = 1/4(e_1 + e_2 + e_3 + e_4) \tag{5.9}$$
$$A = BC = 1/2((e_3 + e_4) - (e_1 + e_2) \tag{5.10}$$
$$B = AC = 1/2((e_2 + e_4) - (e_1 + e_3)) \tag{5.11}$$
$$C = AB = 1/2((e_1 + e_4) - (e_2 + e_3) \tag{5.12}$$

This is a reasonable way to structure our model if we think that the likelihood of a 2-factor interaction is very small. It is important to note that none of the main effects are confounded with each other.

The 2^{3-1} experiment is only one of an endless variety of possible fractional factorial experiments, generally expressed as 2^{n-k} experiments (where n is the number of factors and k is the 'fraction', thus a 2^{n-2} is referred to as a 'quarter fraction of a 2^n experiment. These experiments are usually designed so as not to confound main effects with two factor interactions at the very least, or to provide even higher levels of resolution. The trade-off is between resolution and the number of experiments that must be conducted in the model.

5.1.2 Choosing Interesting Benchmarks And Runs

We choose to carry out the experiments in this chapter for only a small subset of the benchmarks and evaluation runs used in previous chapters. Our basic reason for this reduction in scope was simple: one cannot derive a detailed accounting of effects that are small or non-existent. Attempting to establish multiple sources for the 1% difference between a good profile for `ammp` and a bad one is not only preposterously difficult statistically, it is pointless in the engineering sense.

For each evaluation run and benchmark that we included, we used the following criteria:

- Tractability: some of the SPEC2000 reference runs were simply too large to run them hundreds of times, as required by some of the experiments in this section. Thus, we eliminated from consideration runs that were too large.
- Stability: some of the benchmark runs were too short. Measurement error was too large a source of variation, concealing the effects that we sought to measure. While it is interesting, and in many cases valid, to use these shorter runs to evaluate gross differences between training profiles, it was not desirable to use such small runs to attempt to do a fine-grained accounting of optimization behavior.
- Interesting behavior: this is divided into one or more of:
  - Substantial change in profile-directed optimization performance over a non-profile-directed optimization baseline. Note that the criteria is “change”, not improvement - we are as interested in the failures of profile-directed optimization as we are in its successes.
  - Substantial variability in profile-directed optimization performance between different profiles, particularly exact profiles. This criteria is not the same as the previous one - some programs that respond very well (or very poorly) to profile-directed optimization perform very similarly for all training profiles, while programs with a relatively large range of performance differences due to PDO are not necessarily the

Optimizer	Benchmark	Reference Run	Training Profile	Scaled Performance
alto	bzip2	test	ref1	0.87
	go	test	test	0.97
	m88ksim	test	ref	0.83
	perl2000	ref_makerand	ref_splitmail	0.87
		ref_perfect	ref_perfect	0.80
		train_diffmail	ref_splitmail	0.93
	vortex2000	test	ref1	0.86
		train	ref1	0.86
cc	bzip2	test	r_ref1	0.98
	go	test	test	1.08
	m88ksim	test	ref	0.94
	perl2000	ref_makerand	ref_splitmail	0.91
		ref_perfect	ref_perfect	0.94
		train_diffmail	ref_splitmail	1.05
	vortex2000	test	ref1	0.90
		train	ref1	0.92

Table 5.3: "Interesting" benchmarks and runs chosen for this chapter, along with scaled performance (relative to non-profile-driven optimization case).

ones that have the best or worst performance relative to the non-PDO case.

Our choice was further limited by the fact that each of these analyses presented in this chapter requires a large degree of individual attention and a certain amount of space for presenting the results.

We summarize the benchmarks and runs chosen for this chapter in Table 5.4.

5.2 Selective Inclusion of Profile Data

Intuitively, it seems obvious that not all the information in a basic block profile is equally important. Some parts of the program are not executed heavily in the evaluation runs of interest; some are not executed at all. Further, some parts of the program that are heavily executed are not necessarily particularly affected by profile-driven optimization - they are either not amenable to optimization at all, or non-profile-driven optimization already does a perfectly adequate job with the parts of the program.

In this section, we attempt to determine whether there is locality of profile "usefulness" in profile-directed optimization, and if so, we attempt to measure the distribution of this locality through several programs and profiles.

5.2.1 Methodology

We choose to work at the spatial level of functions (as opposed to smaller-sized regions or even individual conditional branches). Functions are an intuitively understandable way of breaking up a program and it is relatively straightforward to combine profile data at the per-function level from two different profiles. By contrast, how to combine profile data from different profiles over arbitrary regions is far less straightforward.

Our methods involve establishing a "background"; that is, a profile that is used as a baseline for performance. We then substitute per-function data from an experimental profile into that background profile to establish whether or not the the profile data for that function had an effect. We order our functions in order of total dynamic instruction count in the evaluation run and consider only the top 16 functions as possibilities.

The three choices of "background" and experimental profile that we made are:

- Zero profile (all block counts are artificially set to zero) as background with an exact profile serving as an experimental profile
- Statically-estimated profile as background with an exact profile serving as an experimental profile
- Exact profile as background with a statically-estimated profile as the experimental profile.

We would expect that program performance would improve from the addition of profile optimization performance (as far as profile information can be expected to improve optimization performance, that is) in the first two cases, as a "zero profile" or a statically-estimated profile provides very little information and inclusion of any profile data from an exact profile should be expected to improve matters. The reverse is true for the third setup, where including functions from the statically-estimated profile makes the profile less accurate.

Obviously, we are limited in how many functions can be included in our experiments. The more functions that we consider as possibly interesting, the more different combinations of function inclusion and exclusion we have to measure.

Optimizer	Benchmark	Reference Run	$N = 4$	$N = 8$	$N = 16$	$N = 24$	$N = 32$
alto	bzip2	test	1.01	1.00	1.00	1.00	1.00
	go	test	1.06	1.03	1.00	1.01	1.01
	m88ksim	test	1.11	1.09	1.03	1.03	1.01
	perl2000	ref_makerand	1.15	1.10	1.10	0.95	0.98
		ref_perfrect	1.22	1.21	1.17	1.18	1.14
		train_diffmail	1.04	1.05	1.05	1.02	1.03
	vortex2000	test	1.12	1.12	1.12	1.09	1.07
		train	1.15	1.14	1.11	1.09	1.07
cc	bzip2	test	1.01	1.01	1.01	1.00	1.01
	go	test	0.95	0.97	1.00	1.00	1.01
	m88ksim	test	0.98	1.11	0.93	0.97	0.93
	perl2000	ref_makerand	0.93	0.99	1.04	0.95	0.97
		ref_perfrect	0.98	1.02	1.02	0.99	0.98
		train_diffmail	1.01	1.02	1.02	1.03	1.01
	vortex2000	test	0.93	0.93	0.92	0.98	0.97
		train	0.91	0.92	0.93	0.93	0.97

Table 5.4: Relative Execution Time of Including N Top Functions from Training Profile in Zero-Based Background Profile (execution time of binary produced using training profile normalized to 1.0)

We chose the methods where individual function's profiles are included in a background "zero profile", for the sake of simplicity. All three methods produced plausible effects, but the advantage of the zero-based method is that we are not seeing interactions with the behavior of our static estimator, which itself can result in interesting optimization behavior.

Therefore, we will conduct some preliminary experiments designed to establish how much of the profile usefulness is localized in all of the top N functions (where $N = 4, 8, 16, 24$ and 32 functions). This may allow us to raise our level of confidence that subsequent measurements designed to localize usefulness in profiles are in fact working with a set of functions that are reasonable - for example, if the usefulness of a profile with only the top 8 profiles by dynamic instruction count included was the same as the usefulness of the full profile from which it was derived, we would conclude that most of the profile usefulness was localized in these top 8 functions.

For most of our benchmarks, the preliminary experiments showed that much of the profile-driven optimization behavior of the training profile was captured by profiles that included only the top 8 or top 16 functions by dynamic instruction count. Generally, if this was not the case, then adding more functions (even up to the top 32 functions) failed to improve matters substantially. Table 5.4 shows the results of these preliminary experiments for the "zero profile" background case. Note that even using the top 32 profiles fails to give similar results to the base profile on some benchmarks. On a number of the cases (`cc/vortex`, particularly), the profile behavior remains very different from the base profilebehavior until we use 24 or 32 functions.

The cases `alto/m88ksim` is the best example of the profile-driven optimization behavior clearly being modelled somewhere in the top 8-16 functions. Note how in the `alto/m88ksim` case the profile behavior converges on the base profile behavior (improving, in this case) at somewhere between 8 and 16 functions. Not all benchmarks are so well-behaved. Some appear to converge quickly on the behavior of the base profile even with only 4 functions included (e.g. `bzip` with both optimizers), while others show suprisingly non-monotonic behavior (e.g. `cc/m88ksim` for the `test` run, with a single odd behavior at $N = 8$). In one case (`alto/perl2000` for the `ref_perfect` run), even including the top 32 functions does not produce similar effects to the base profile - there is still a 14% gap in performance. Overall, however, it is clear that the profile-driven optimization behavior of most profiles converges on the base profile as we increase the value of N.

5.2.2 Experimental Design

From our above results, we have discovered that profile usefulness seems to extend well beyond the top 4 or top 8 functions (by dynamic instruction count). Carrying out factorial experiments with huge numbers of interacting factors is extremely expensive, even using fractional factorial designs. Further, we are not entirely sure that second-order interactions are unimportant, so we are inclined to avoid potential aliasing between main effects and second-order interactions. This means that we must use what is called a "Resolution IV Design"; that is, a fractional factorial experiment that is designed so that no main effect is aliased with any two-factor interaction term[2]

[2]The "IV" (4) in "Resolution IV Design" refers to the shortest possible aliasing identity ; i.e. a aliasing between 2 two-factor interaction terms. A "Resolution III Design" requires far fewer experiments to be run but, as the name suggests, aliases two-factor interaction terms with main effects - the use of such designs is considered extremely risky and they are generally used only when domain experts conclude that it is unlikely that there are any significant two-factor

To balance the competing concerns of experimental tractability, the need to include as many functions as possible, and the need to avoid introducing dangerous aliases, we have chosen a Resolution IV design over 11 functions that calls for 32 experiments. Our preliminary results showed that a range of 8-16 functions captured a substantial proportion of the profile-driven optimization effect of our training profiles. The choice of 11 functions is the maximum that we could reasonably evaluate within a Resolution IV design. While a model with more functions (for example, 24 or 32 functions) would capture more profile-driven optimization behaviors, it would be extremely difficult to evaluate models with so many factors. Thus we expect (of necessity) to not be able to capture all the behaviors modelled in Table 5.4.

A Resolution IV design does not allow us to accurately estimate two-factor interactions - many of our two-factor interactions are aliased with other two-factor interactions. Thus, using this design, we will only attempt to estimate main effects over our top 11 functions. We are somewhat interested in discovering whether significant two-factor interactions exist among at least some of our functions; unfortunately, a design that allows accurate discovery of all two-factor interactions among such a large set of functions could involve hundreds of experiments.

Nonetheless, we are interested in finding whether there are at least some two-factor interactions among the function profile data that we deem more likely (prior to the experiment) to influence the outcome of the experiments. Using similar logic as we used to select the top 11 functions, we have decided (somewhat arbitrarily) that the top 4 functions (again, by dynamic instruction count in the reference run) are the most likely to be important and have significant interaction effects.

Thus, we carry out a supplementary full factorial experiment over the top 4 functions (with all other function data absent from the profile). This yields another 16 experiments or "design points" in the nomenclature of experimental design. Thus, given the amount of detail that this supplementary experiment yields, we can estimate 2-factor (and higher level) interactions among the "hottest" functions in the evaluation run.

A full factorial experiment allows estimation of all higher-order effects, including 3-factor and 4-factor effects in this case. However, our tests of 3-factor and 4-factor interactions found almost no significant interactions at this level that could not be explained by simpler interactions. We decided to eliminate these high order interactions from consideration for a simpler model. Choosing a sim-

---

interactions, a statement to which we cannot easily commit.

pler models also needs more degrees of freedom for estimation of experimental error.

Admittedly, the model we have built does not account for all of the profile-driven optimization variability observed in our benchmarks. We will estimate how much variability observed in the benchmarks is captured within this analysis.

An interesting quirk of our design is that we perform fractional factorial experiment with repeated observations; that is, while we do not carry out experiments at all 2^{11} possible factor levels, we do repeat our experiments (3 times in this case, as well as an initial run to avoid paging effects). This is an unusual practice in most experimental designs - after all, if we are running $(32 + 16) \times 3 = 144$ experiments per benchmark, why not evaluate 144 design points, instead?

The justification for the use of repetition lies in the costs (in terms of machine time) in carrying out our experiments. First, we need to run our optimizer (`cc` or `alto`) once *per design point*. Further, for any given run of the optimizer, we must execute the resulting binary at least once on the evaluation run and discard the results. Thus, there is a certain amount of overhead per design point; as long as we have reached this point, it is cost-effective for us to run some repeats of the evaluation run to help characterize how much of the variability we observe is due to experimental error rather than profile-driven optimization error.

5.2.3 Results

Summary of Our Model

So, as previously discussed, the model of the system that we designed is capable of discovering the following effects:

- The effect of including each individual function out of our top 11, which we will denote by $f_1 ... f_{11}$.
- The two-way interaction terms between each of the top 4 functions, denoted by $f_1 \times f_2, f_1 \times f_3, ... f_3 \times f_4$.

Our model will only account for the proportion of the variance of profile-driven optimization performance due to the inclusion or exclusion of individual function data. Variation due to the inclusion of other functions, or interactions that we cannot measure, will not be accounted for in our model. Further, some of the variation exhibited by our benchmarks will be due to experimental error. We will use our exploratory results to give a somewhat informal guess at how much of the profile-driven optimization variability that our model could capture, and use the extra degrees of freedom provided by our repetitions and our avoidance of high-level interactions to estimate experimental error.

Proportion of Variation Explained By Model

First, we found that, for many benchmarks, a great deal of the performance effects produced by the "interesting" training profile selected as a base could be explained in terms of a relatively small number of profiles. In fact, the range of effects produced in our 11-factor experiment generally encompassed the effect of using the base "training" profile with all functions intact.

That is, our model would seem more successful if the range of profile-driven optimization outcomes in our factorial experiment included - or came close to - the profile-driven results of using the base profile on its own, and showed a substantial variation as well.

If the "base profile" PDO results are not included in the range of results produced by the factorial experiment, and the range of profile-driven optimization performance variation due to our 11 factors is small, it suggests whatever effect that the profile had has not been adequately captured by the model.

Of course, just because the experiments that we carry out to build our model show a substantial variation does not automatically mean that the model actually explains this variation. But this question is something that analysis of the model itself can answer. In this subsection, we merely concern ourselves with the question of determining whether or not our model appears to be producing an adequate range of effects.

We will examine a number of histograms plotting the distribution of the profile-driven optimization performance of the binaries produced within the factorial experiments, relative to the optimization performance of the base profiles used. These histograms provide a rough idea of how much variation takes place in the experiments[3].

At one extreme, depicted in Figure 5.1, the benchmark `m88ksim` using the `cc` optimizer, the `test` evaluation run and the `ref` base training profile, shows huge variability in our factorial experiment, ranging between 0.9 and 1.4 times the profile-driven optimization performance of the binary produced with `ref` on the same run, depending on which functions' profiles are included.

[3]There are different values of N for the two histograms shown, as the individual benchmark runs are run for different numbers of repeats between the two different benchmarks. This does not affect the models developed for each benchmark.

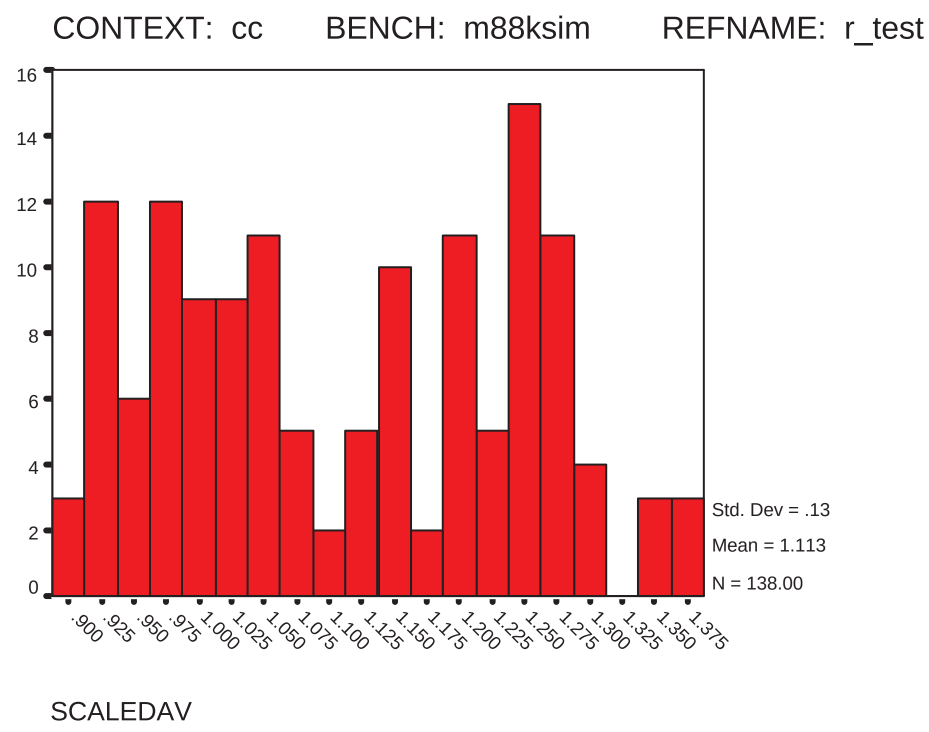

Figure 5.1: Distribution of Performance in Factorial Experiments (`m88ksim` using `cc`) - relative to base profile

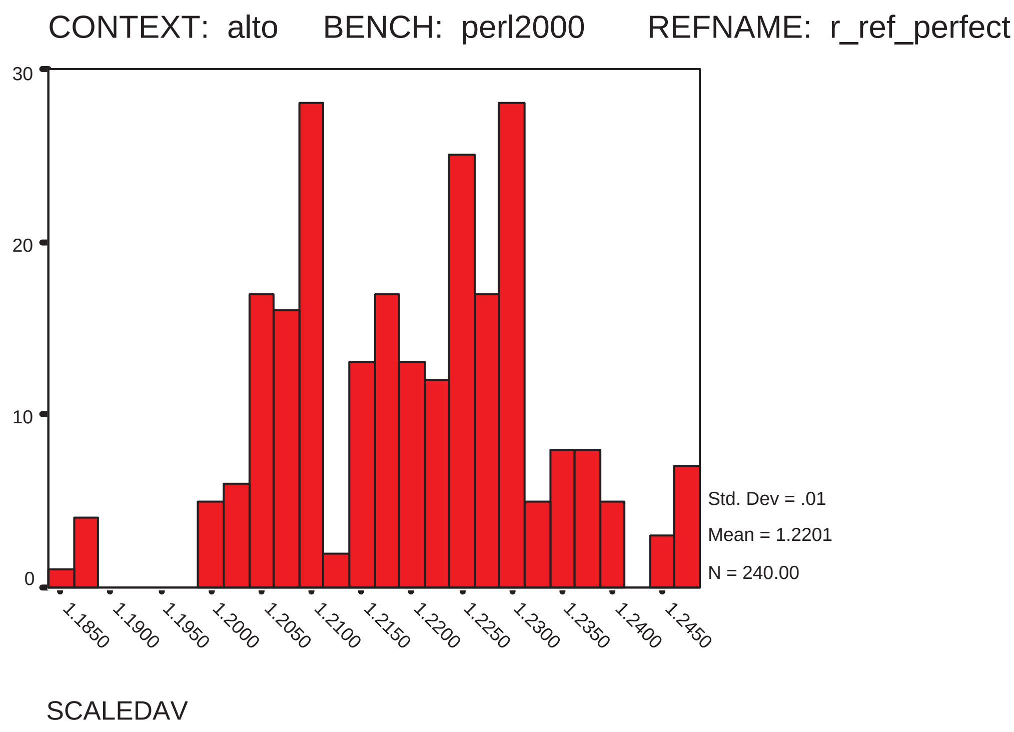

Figure 5.2: Distribution of Performance in Factorial Experiments (`perl2000` using `alto`) - relative to base profile

At the other extreme, depicted in Figure 5.2, the benchmark `perl2000` using the `alto` optimizer, the `ref_perfect` evaluation run and the `ref_perfect` base training profile, shows a variability in the equivalent situation ranging from 1.18 to 1.25.

Informally, it seems like the model in the `m88ksim` case has captured more interesting information about what makes the `ref` training profile useful or not useful, as compared to the model in the `perl2000` case. Which case is more typical of our benchmarks? Table 5.5 summarizes the cases across our "interesting" benchmark set.

Table 5.5 gives an idea of the range of results across for each factorial experiment (over the range of benchmarks and runs used in this chapter). We do not present a formal accounting of the variance here, as it is due to a synthetic set of profiles. However, we present the range, minimum and maximum perfor-

Optimizer	Benchmark	Reference Run	Range	Minimum	Maximum
alto	bzip2	test	0.26	0.90	1.16
	go	test	0.06	1.01	1.08
	m88ksim	test	0.24	1.05	1.28
	perl2000	ref_makerand	0.14	1.06	1.20
		ref_perfect	0.06	1.19	1.25
		train_diffmail	0.11	0.99	1.10
	vortex2000	test	0.11	1.07	1.18
		train	0.12	1.11	1.23
cc	bzip2	test	0.07	1.00	1.07
	go	test	0.09	0.91	1.00
	m88ksim	test	0.48	0.90	1.38
	perl2000	ref_makerand	0.12	0.92	1.04
		ref_perfect	0.08	0.95	1.03
		train_diffmail	0.12	0.93	1.05
	vortex2000	test	0.09	0.92	1.01
		train	0.09	0.91	0.99

Table 5.5: Summary of Distribution of Performance in Factorial Experiments - relative to base profile

mance (relative to the base profile from which the synthetic profiles have been constructed) so as to give an idea of how much variability is captured by our experiments, and in what direction.

It is clear that the range of performance that results from our factorial experiments is quite large, and that the bulk of our experiments come fairly close to the performance of the "base" profile from which the profile experiments were made (in the table, a scaled value of 1.00). The two most glaring departures from this come with `alto` on two different benchmark and reference run / training profile combinations: `perl2000` with `ref_perfect` and `vortex` with `train`.

For these benchmarks, it is clear that whatever makes the base profiles effective, it is not fully captured by our factorial experiment. Perhaps it is the 12th function by dynamic instruction count - or the 20th - or an interaction that we did not explore. We did, however, capture substantial variability even in these cases - we continue to construct our model, while remembering that the model does not fully describe the performance effects observed from the use of the base profile.

We note that many of the results using the `cc` optimizer generally show better results using a factorial subset of the important functions than they do with the full "base" profile. One reason for this is that more of the selected benchmarks and benchmark runs actually turned out to be bad profiles under `cc` (the benchmark and run choice was generally driven by the intent to use benchmarks, profiles and runs that showed substantial effects with `alto`).

To cite the most extreme case, the use of the profile generated by the `test` run for `go` under `cc` actually worsened performance by 7% as compared to the non-profile driven optimization case. Thus, it is not surprising that a procedure that involves including only some functions from a 'bad' base profile produces better performance than that base profile. We still consider these cases interesting and the building of a performance model for them to be useful; optimization failure might well be as localized as optimization success.

Overall, it looks like nearly all of the benchmarks show substantial profile-driven optimization variation during our factorial experiments, and it seems likely that the effects observed have something to do with the performance effects of the base profiles.

Model Results: Main Effects and Interactions

Our model is an additive one and consists of a mean value and 17 coefficients - 11 coefficients representing the effects of including an individual function, and 6 coefficients representing the 2-way interactions between our top 4 functions.

Fortunately for our presentation, not all of these 17 effects are significant (in a statistical sense or in an engineering sense). We remove from consideration any factor that is not significant at the 0.01 level[4]. Further, we will remove from our presentation factors that, while statistically significant, caused less than a 0.5% change in performance.

Not all our factors operate in the same direction. Some of our factors will *worsen* performance when "switched on" (that is, when the function corresponding to the factor is included). In a profile-directed optimization system where usefulness perfectly correlated to accuracy, inclusion of accurate function data should always improve performance. However, we must consider several reasons why our functions do not improve performance.

- First, many of the combinations of profile and evaluation run are not "resubstitution" cases, and thus any given function may not be an accurate reflection of program behavior on our evaluation run.
- Second, and probably more significant, we have extensively demonstrated that the use of accurate profile data does not always improve performance
- Finally, some of the profiles that we have evaluated are interesting because they worsen performance - thus we would expect to see the selective inclusion of some of their parts would also worsen performance.

Our interaction terms show the effect of having both factors "switched on" that is *independent* of the individual effects of the factors. This may indicate a synergy between two profiles (added benefits from having both functions in the profile) if positive or a dysergy (where including both profiles in the function causes worse performance than you would expect from their individual effects). In some situations this may represent a situation of diminishing returns; while the inclusion of functions foo and functions bar might both yield a 5% performance improvement, the sum of their effects might be smaller than a 10% performance improvement.

Initially we will show the overall success of the model described above in accounting for the experimental results that we observed in our factorial experiments. Table 5.6 shows the r^2 values for our model for each experiment; as in the case of correlation, this value indicates the proportion of variability explained by

[4] Why so strict? We hold that, given the large number of factors involved (17 for each benchmark), using a 0.05 significance level would result in around a 60% chance ($1 - 0.95^{17} = 0.58$) chance of claiming that a factor was significant when in fact it was not.

Optimizer	Benchmark	Reference Run	Model r^2
alto	bzip2	test	0.82
	go	test	0.44
	m88ksim	test	0.62
	perl2000	ref_makerand	0.65
		ref_perfect	0.42
		train_diffmail	0.65
	vortex2000	test	0.66
		train	0.68
cc	bzip2	test	0.93
	go	test	0.89
	m88ksim	test	0.97
	perl2000	ref_makerand	0.93
		ref_perfect	0.91
		train_diffmail	0.85
	vortex2000	test	0.73
		train	0.78

Table 5.6: r^2 Values for each Factorial Experiment - Proportion of Variability in Each Experiment Explained By Our Model

our model as opposed to the variability that results from either systematic effects that our model does not capture or the variability that results from experimental error.

The r^2 values are generally quite good; only `go` and one case in `perl2000` show poor (less that 50% of variance explained) results. Note that even a perfect model will not explain 100% of the variability in these experiments, as there is a certain amount of experimental error in the measurement of the performance of the optimized binaries. Unfortunately, we cannot separate this error from errors due to the inadequacies of our model.

We now present the models themselves, in Table 5.7 and Table 5.8. We will not give a full account of all 17 factors for every benchmark and profiling run, but instead present the model factors that meet our criteria for statistical and engineering significance (that is, factors that meet the criteria of at least a 0.5% effect on final performance).

Each table is grouped by optimizer, benchmark and run (we use only a single training profile for each reference run, as described in Table 5.4). Each grouping contains the description of a single model that is designed to predict the profile-driven optimization usefulness of a training profile that includes one or more functions from the training profile that we are using as a baseline. The model performance is scaled so that 1.0 is the profile-driven optimization performance of the unmodified training profile.

The functions are numbered from f_1 to f_{11} based on their dynamic instruction count in the reference run. This means that for different reference runs of the same benchmark , f_1 may refer to a different function.

The entries for each combination of benchmark, optimizer and run contain the following information:

- The "Base" term indicates the model's predicted relative performance without any of the functions included. For example, if a "Base" term was 1.054, this would mean that an all-zero "background" profile would run 5.4% slower than the original training profile.

- An entry for each function or two-function interaction term. We use the first per-function entry in Table 5.7 (optimizer `alto`, benchmark `bzip`, run `train` and function f_2 as an example. Each entry comprises:

  - The function or interaction identified by its dynamic instruction count rank (for example, f_2 refers to the function with the 2nd highest dynamic instruction count)[5].

  - The coefficient of the term associated with that factor in our model - that is, if the coefficient associated with f_2 is -0.114, that means that average execution time (relative to the performance of the original binary optimized with the unmodified training profile) drops by 11.4% if the data from function f_2 is included.

  - The significance ("Sig.") of the factor. This is the probability that if there was no real difference between the profile-directed optimization usefulness of the profile whether the function was included or not, we would have seen as much variation associated with that particular function. In the example we chose, this value is so low that it is rounded to 0.000. In this case, there is a negligible chance that we would have seen such a large difference in the effect of function

[5]We were fortunate enough not to have to consider ties within the top 11 functions considered for these benchmarks and runs

profiles that included f_2 and those that did not. We filter significance levels at 0.01.

- A metric of how much variation the factor accounted for, called "Partial Eta Squared". This metric gives a value between 0 and 1.0 that approximates the ratio between the amount of variability explained with and without that individual factor (or interaction term). Each factor is considered in isolation in the calculation of partial eta squared, so the values do not add up to 1.0.

- The actual name of the function or functions. In a few cases, we are unable to find the function names, as this information has been removed from functions that came from certain object files, particularly libraries.

Table 5.7 and Table 5.8 show that the number of factors that are a significant effect of profile-driven optimization performance can be quite small, and that the magnitude of the effects of some of the top factors can be quite large, often much larger than the effects of the other factors put together . Profile information for the function f_2 in the `alto` /`bzip` example produces a 11.4% average speedup!

These factors are not strictly uni-directional. It should be obvious by now that not every application of profile-driven optimization results in speedups. This pattern continues on the per-function level; note the group of effects in the `cc` /`perl2000`/`train_diffmail` case (associated with f_1, f_4 and f_6) that each produce over 4% slowdown each! Thus, our factorial model does a reasonably good job of localizing profile-direction pessimization as well as optimization - note that the use of the chosen training profile for this particular benchmark and run causes a 5% slowdown, not a speedup.

Whether the effects of individual function's profile data are harmful or beneficial, it remains true that the usefulness of the profile data is quite heavily localized to individual functions, and that not every function that accounts for a large share of dynamic instruction count is heavily represented in the model.

We evaluated an alternate model, where we included 3-factor and 4-factor interactions among our various benchmarks. This model showed almost no improvement in the proportion of variability explained over the simpler model. This raises the question - how important, overall, were our 2-factor interactions?

The effects of 2-factor interactions were generally smaller than the single-factor effects. There were occasional 2-factor interactions of reasonably large magnitude (for example, the $f_3 \times f_4$ term in `alto/m88ksim` result). However,

Optimizer	Bench	Run	Factor	Coefficient	Sig.	Partial Eta Squared	Function Name
alto	bzip2	test	Base	1.085			
			f_2	-0.114	0.000	0.75	generateMTFValues
			f_6	-0.035	0.000	0.20	spec_putc
			f_8	0.017	0.000	0.05	getRLEpair
			f_{10}	-0.034	0.000	0.18	spec_ungetc
			f_{11}	0.035	0.000	0.19	bsR
			f_1 * f_4	0.040	0.000	0.10	getAndMoveToFrontDecode * spec_getc
			f_5	0.016	0.001	0.05	sortIt
			f_1 * f_3	0.027	0.001	0.05	getAndMoveToFrontDecode * sendMTFValues
			f_2 * f_3	0.021	0.010	0.03	generateMTFValues * sendMTFValues
	go	test	Base	1.047			
			f_2	-0.010	0.000	0.23	getefflibs
			f_6	0.010	0.000	0.16	ldndate
			f_1 * f_3	-0.008	0.002	0.04	mrglist * addlist
	m88ksim	test	Base	1.250			
			f_1	-0.050	0.000	0.27	killtime
			f_2	-0.059	0.000	0.22	ckbrkpts
			f_6	-0.022	0.000	0.06	Pc
			f_9	-0.058	0.000	0.30	rdwr
			f_3 * f_4	-0.034	0.001	0.05	Data_path * execute
			f_1 * f_3	-0.027	0.007	0.03	killtime * Data_path
	perl2000	ref_makerand	Base	1.197			
			f_2	-0.037	0.000	0.28	Perl_pp_nextstate
			f_3	-0.044	0.000	0.28	Perl_pp_rand
			f_9	-0.027	0.000	0.22	Perl_pp_next
			f_2 * f_3	0.036	0.000	0.15	Perl_pp_nextstate * Perl_pp_rand
			f_1 * f_2	-0.017	0.004	0.04	Perl_pp_padsv * Perl_pp_nextstate
			f_3 * f_4	-0.017	0.004	0.04	Perl_pp_rand * Perl_pp_modulo
			f_1 * f_3	0.016	0.005	0.03	Perl_pp_padsv * Perl_pp_rand
			f_6	0.009	0.010	0.03	Perl_pp_const
		ref_perfect	Base	1.228			
			f_5	-0.011	0.000	0.19	Perl_pp_modulo
			f_7	0.008	0.000	0.12	Perl_sv_setnv
			f_{10}	0.007	0.000	0.10	Perl_pp_divide
			f_1 * f_4	0.016	0.000	0.15	Perl_pp_gvsv * Perl_pp_nextstate
		train_diffmail	Base	1.042			
			f_1	-0.023	0.000	0.48	++undef++
			f_2	0.019	0.000	0.21	Perl_my_bcopy
			f_4	-0.020	0.000	0.07	++undef++
			f_{10}	0.007	0.000	0.05	Perl_leave_scope
			f_2 * f_3	-0.021	0.000	0.15	Perl_my_bcopy * undef
			f_2 * f_4	0.016	0.000	0.09	Perl_my_bcopy * undef
			f_3 * f_4	0.014	0.000	0.08	undef * undef
			f_5	-0.006	0.002	0.04	Perl_pp_padsv
	vortex2000	test	Base	1.163			
			f_1	-0.019	0.000	0.12	Chunk_ChkGetChunk
			f_3	-0.027	0.000	0.24	TmFetchCoreDb
			f_4	-0.020	0.000	0.28	Mem_GetAddr
			f_5	0.011	0.000	0.11	OaGet
			f_6	-0.008	0.000	0.06	Mem_GetBit
			f_7	-0.013	0.000	0.15	OaGetObject
			f_9	-0.010	0.000	0.09	Hm_FetchDbObject
			f_2 * f_3	0.025	0.000	0.16	Mem_GetWord * TmFetchCoreDb
			f_8	-0.007	0.001	0.05	TmGetObject
			f_1 * f_4	0.010	0.007	0.03	Chunk_ChkGetChunk * Mem_GetAddr
			f_1 * f_2	0.010	0.008	0.03	Chunk_ChkGetChunk * Mem_GetWord
		train	Base	1.171			
			f_3	-0.032	0.000	0.57	TmFetchCoreDb
			f_4	-0.026	0.000	0.22	Mem_GetAddr
			f_5	0.008	0.000	0.06	OaGet
			f_7	-0.011	0.000	0.11	OaGetObject
			f_{10}	-0.010	0.000	0.09	Hm_FetchDbObject
			f_2 * f_4	0.018	0.000	0.09	Mem_GetWord * Mem_GetAddr
			f_3 * f_4	0.010	0.007	0.03	TmFetchCoreDb * Mem_GetAddr

Table 5.7: Model Description for `alto`

Optimizer	Bench	Run	Factor	Coefficient	Sig.	Partial Eta Squared	Function Name
cc	bzip2	test	Base	1.054			
			f_1	-0.036	0.000	0.91	getAndMoveToFrontDecode
			f_5	0.005	0.000	0.12	sortIt
			f_{10}	0.006	0.000	0.17	bsR
			f_1 * f_3	-0.006	0.006	0.06	getAndMoveToFrontDecode * sendMTFValues
	go	test	Base	0.932			
			f_2	-0.010	0.001	0.09	getefflibs
			f_3	0.024	0.000	0.77	addlist
			f_5	0.013	0.000	0.43	dellist
			f_6	0.007	0.000	0.15	ldndate
			f_8	0.006	0.000	0.13	killist
			f_9	0.013	0.000	0.41	iscaptured
			f_{10}	0.005	0.001	0.08	radiatepiece
			f_{11}	0.005	0.002	0.08	livesordies
			f_2 * f_4	0.007	0.009	0.06	getefflibs * lupdate
	m88ksim	test	Base	1.160			
			f_1	-0.209	0.000	0.96	killtime
			f_2	-0.060	0.000	0.60	ckbrkpts
			f_3	0.057	0.000	0.43	Data_path
			f_4	0.081	0.000	0.70	execute
			f_5	0.037	0.000	0.34	test_issue
			f_9	0.032	0.000	0.27	checklmt
			f_8	0.015	0.002	0.08	check_scoreboard
			f_{10}	-0.013	0.007	0.06	getmemptr
			f_{11}	0.012	0.009	0.05	Statistics
	perl2000	ref_makerand	Base	0.949			
			f_1	-0.015	0.000	0.72	Perl_pp_padsv
			f_3	0.022	0.000	0.31	Perl_pp_rand
			f_7	0.046	0.000	0.85	Perl_sv_setsv
			f_9	0.010	0.000	0.21	Perl_pp_next
			f_{11}	0.007	0.000	0.13	Perl_pp_gt
			f_1 * f_2	-0.012	0.000	0.12	Perl_pp_padsv * Perl_pp_nextstate
			f_1 * f_4	-0.014	0.000	0.14	Perl_pp_padsv * Perl_pp_modulo
			f_2 * f_3	-0.013	0.000	0.12	Perl_pp_nextstate * Perl_pp_rand
			f_6	-0.006	0.001	0.09	Perl_pp_const
			f_3 * f_4	-0.008	0.009	0.06	Perl_pp_rand * Perl_pp_modulo
		ref_perfect	Base	0.972			
			f_5	-0.006	0.000	0.17	Perl_pp_modulo
			f_7	0.008	0.000	0.26	Perl_sv_setsv
			f_8	0.040	0.000	0.89	Perl_pp_add
			f_{10}	-0.005	0.001	0.09	Perl_sv_upgrade
		train_diffmail	Base	0.954			
			f_1	0.043	0.000	0.30	regmatch
			f_4	0.046	0.000	0.54	regtry
			f_6	0.041	0.000	0.72	Perl_sv_setsv
			f_7	0.010	0.000	0.14	Perl_regexec_flags
			f_1 * f_4	-0.048	0.000	0.53	regmatch * regtry
			f_2 * f_3	-0.018	0.000	0.13	Perl_my_bcopy * regrepeat
	vortex2000	test	Base	0.974			
			f_1	-0.025	0.000	0.56	Chunk_ChkGetChunk
			f_8	-0.008	0.000	0.10	TmGetObject
			f_9	0.009	0.000	0.11	Hm_FetchDbObject
			f_{11}	0.013	0.000	0.21	Part_Delete
			f_7	0.007	0.002	0.07	OaGetObject
		train	Base	0.956			
			f_1	-0.013	0.000	0.67	Chunk_ChkGetChunk
			f_6	0.010	0.000	0.21	OaGet
			f_7	-0.013	0.000	0.30	Mem_GetBit
			f_3	-0.012	0.000	0.36	TmFetchCoreDb
			f_1 * f_4	-0.011	0.001	0.09	Chunk_ChkGetChunk * Mem_GetAddr

Table 5.8: Model Description for `cc`

these 2-factor interactions usually involved at least one factor that was identified as important in itself (the `alto/m88ksim` term mentioned above is one of the few exceptions).

This suggests that a more effective way to proceed in future work of this kind is to conduct a two-stage experiment. First, a low-resolution screening experiment should be carried out to determine which single factors are important.

Second, these factors can be analyzed exhaustively for interactions. It may be possible to use our knowledge of the static or dynamic behavior of the program to estimate which factors are likely to have substantial interactions. For example, if we have two pairs of functions A, B and C, D such that:

- A calls B extensively
- C and D are distant from each other in the call graph and are almost certain never to be active at the same time

then we would expect that the potential for 2-factor interaction terms corresponding to the pair of functions C and D is comparatively small as compared to those corresponding to A and B. A carefully constructed experimental design might confound the interaction terms $C \times D$ and $A \times B$, and we would analyze the design with the assumption that the effect of $C \times D$ was negligible.

We should not entirely dismiss the importance of 2-factor interactions - particularly two-factor interactions that occur between factors that are not significant in themselves. The above two-stage design runs the risk of entirely missing a pair of factors that are only effective in combination. It is an open question as to whether a more detailed accounting of factor interactions (for example, all 2-factor interactions among our top 11 functions) is more important than including more functions (for example, extending our calculation of main effects from the 11th to the 22nd function).

5.2.4 Characteristics of Important Functions

Our models showed that only a handful of functions had any significant effect on performance. Given that our models generally (but not always) explained a large proportion of the effects of profile-driven optimization for our chosen benchmarks, training profiles, and reference runs, it is interesting to look at the characteristics of the functions deemed particularly "influential" by our model. Does inspection of these functions reveal common characteristics that would allow us to predict *in advance* which functions might be particularly important for profile-driven optimization? It is worth remembering that all the analysis in this chapter is “after the fact”: our models explain what happened with an set of optimized binaries already executed on a given reference run, but offer no predictions as to what would happen with a different profiles and/or a different benchmark.

We examined various static and dynamic characteristics of two sets of functions out of our set of the top 11 functions in each combination of optimizer, benchmark and reference run that we measured. We formed the sets of functions by separating the functions whose profiles had a large effect on profile-driven optimization from those that had little effect on profile-driven optimization. We found very little difference between these two sets of functions. Unsurprisingly, there was an association between the property of a function being “influential” and a function having a large dynamic instruction count.

We did not find any significant difference between various metrics of static and/or dynamic function complexity and the influence of the function’s profile on profile driven optimization outcomes. Among the metrics that we examined were:

- Function call graph behavior (number or dynamic count of in edges, number or dynamic count of out edges)
- Number or nesting depth of loop nests in the function
- Number or dynamic count of conditional branches
- Number or dynamic count of conditional branches in only one direction
- Static instruction count

The only metrics that seemed to suggest a function’s potential influence were either dynamic instruction count itself or various other metrics that tracked dynamic instruction count fairly closely. This negative result (no association) was stable regardless of whether we attempted to predict whether functions were influential or whether functions were both influential and the results of using the function’s profile were good.

5.3 Conclusion

Our results show that usefulness, for a large proportion of our “interesting” benchmarks and runs, can definitely be localized in a small proportion of functions.

Usually, a few functions - not necessarily the functions with the highest dynamic instruction count in the program - account for a very large proportion of the variance due to profile-directed optimization. We built models using the top 11 functions by dynamic instruction count that accounted (informally, at least) for a large proportion of the profile-driven optimization effects of our training profiles, at least for most of our benchmarks. Within these models, we were able (this time more formally) to show that most of the performance effect was due to a relatively small number of factors within the model. At most half of the 17 single factor and 2-factor interactions that we modelled had a substantial effect, and generally the bulk of the variation was accounted for by far fewer factors.

While 2-factor interactions between functions were never among the most important parts of our model, we did find significant 2-factor interactions. This validated our used of methods from the "Design of Experiments" methodology; a experimental procedure that considered function profile data only in isolation would not have identified these 2-factor interactions - or worse, would have mistakenly attributed them to individual functions.

We have identified a procedure by which a large proportion of the variance due to profile-directed optimization can be assigned to the effects of a comparatively small number of functions and the interactions between them. More work is needed to make this procedure more robust, complete and efficient.

Ultimately, the methodology developed in this chapter could help with a number of problems in the area of profile-directed optimization. We must be cautious about any claims that these techniques could be used to improve overall profile-directed optimization effectiveness per se. While, theoretically, our models could be used to generate a profile that is better than the training profile used as a base (by excluding functions' profiles whose effect was to worsen optimization performance), the overhead of building our model is very high and our model is specific to a known reference run. There is no guarantee that our model will help with doing a better job of optimizing a program for a unknown future execution.

However, the ability to identify small parts of the profile as particularly important and effective may prove useful when we have a very good idea of future program executions. First, as outlined above, it may be possible to generate more useful profiles as a result of our model. Second, being able to reduce a profile to a few functions may make the task of maintaining profiles for a program that is constantly being modified easier. For example, we might flag the functions that have the most influence on profile-driven optimization and only re-profile the program if these functions are modified.

Chapter 6

Related Work

6.1 Profile Accuracy and Static Estimation

Wall [14] makes the first systematic attempt to evaluate profile accuracy. Wall compares real profiles and static estimates for accuracy, introducing the key- and weight-matching comparison metrics. His comparisons use key- and weight-matching at both fixed levels (top k) and, similar to our work, at levels proportional to the total number of blocks (top $N\%$). He shows strong improvements in accuracy from using real profiles over static estimates.

The static estimation methods evaluated in the Wall paper are fairly primitive, using only loop nesting levels and counts of static call sites.

Wall analyzes two purely theoretical profile-driven optimization algorithms; global register allocation and a hypothetical "intensive" optimization (that cuts the execution time of a procedure in half but is so expensive that it can only be applied to 5% of the procedures). The results from this analysis are of necessity limited; the global register allocation optimization did not provide much speed-up and the "intensive" optimization example is extremely unrealistic. However, his call for caution about the promised benefits of profile driven optimization proved prescient given our results.

Larus and Ball introduce a series of heuristics for compile-time branch prediction in [3], applying the heuristics in a fixed order. Wu and Larus extend this work in [18], by using Dempster-Shafer theory to combine branch prediction heuristics (rather than a fixed ordering) and introducing an algorithm to construct basic block profiles from branch predictions. Their approach is similar to, but more efficient than, the linear algebra-based approach in our work (although our linear algebra approach proved to be acceptable in performance in almost all cases - its inefficiencies are a bigger problem in theory than in practice). The earlier work uses a simple metric (”instructions per mispredicted branch”), while key- and weight-matching (at 10%, 20%, 30%, 40% and 50% - they thus present 10 different accuracy metrics) are used to evaluate the accuracy of the static estimation methods.

Wagner et. al do a similar analysis in [13], using Markov models to model control flow on an abstract syntax tree-based representation of programs. They compare estimated profiles to exact profiles using weight-matching.

None of the above works attempts to establish any connection between profile accuracy and real profile-driven optimization performance. Our work diverges from all of these works by connecting accuracy metrics to actual profile-driven optimization performance in two mature optimizers. The work contained in this thesis suggests that claims of greater or lesser accuracy from one static estimation method or another may have very little actual consequence for profile-driven optimization; of these papers, the only paper to deal with this issue in any way is Wall [14].

This issue is particularly important in the context of static estimation. In a profiling system that is designed for both profile-driven optimization and the gathering of profiles for the enlightenment of the user, such as DCPI [2, 5], a more accurate profile may be helpful even if it does not improve profile-driven optimization performance. Such a claim cannot be made for static estimators; few programmers would look to a static estimator for insight about the execution frequencies of their programs.

An extensive treatment of information-theoretic methods for comparing and combining profiles, including the relative entropy comparison used in this work, appears in Savari and Young [11]. Our work validates the use of relative entropy as a profile comparison metric in many domains. However, the comparative effectiveness of simple, more intuitively understandable metrics such as static coverage of profiles does raise some doubts about the wisdom of using relative entropy in some circumstances.

6.2 Characteristics of Dynamic Profiles

In a paper that also deals with static branch prediction, Fisher and Freudenberger report that profile data gathered from previous runs yields good branch predictions [8]. Like Ball and Larus [3], this work uses the “instructions per mispredicted branch" metric. Unlike most of the works on profile accuracy, this work deals with the accuracy of "exact profiles", rather than static estimation.

Fisher and Freudenberger mention the possibility that the differences in real benchmark runs might be related to the benchmark's coverage of the program as opposed to differences in behavior in code that is covered by both runs. To quote from the "Informal Observations" section of their paper: "We felt that when a dataset predictor did poorly, it was usually because it emphasized a different part of the program than the target dataset, rather than that the branches changed direction". This is an interesting observation, which unfortunately they were not able to quantify. Our results suggest that this intuition was correct (at least in terms of what information `alto` was able to use effectively); the comparatively strong predictive value of the accuracy metric "FE-STC" (function entry static coverage) supports this. Further, for our exact profiles, we measured the correlation between the "FE-STC" metric and two whole-program accuracy metrics (relative entropy and key-matching at the 10% level). We show the `alto` results in 6.1[1]; the `cc` results are almost identical, as profile accuracy metrics alone behave very similarly across both optimizers.

Calculating these correlations across all benchmarks for a given optimization context also yields a strong (r_s close to 0.7 for both metrics and both optimization contexts) connection between static coverage and other metrics. All these relatively strong correlations indicate that static coverage of functions is very strongly connected with program behavior in general. The correlations do not help determine why this connection exists, but they lend some support to the Fisher and Freudenberger hypothesis.

Eeckhout et al. [7] use statistical data analysis techniques to cluster similar "program-input pairs" (in our terms, pairs consisting of a benchmark and an evaluation run). They concentrate on overall benchmark characteristics as opposed to profile accuracy and/or profile usefulness.For our analyses in this paper, we have little need to reduce the number of "program-input pairs" to cover a hopefully representative set of benchmarks, training profiles, and evaluation runs, as our analyses benefit from more data points rather than fewer. This is true even if some of the training profiles and evaluation runs produce very similar effects.

[1]Some benchmarks are missing from this table, as the FE-STC accuracy metric may be identical for all profile pairs, rendering correlation impossible

CONTEXT	BENCH	ENT	K01
alto	ammp	0.737*	0.32
	art	0.534*	0.49
	bzip2	0.562**	0.15
	compress	0.42	0.53
	crafty	0.50	0.37
	gap	0.744*	0.50
	go	0.520**	0.402*
	gzip	0.388**	0.313*
	li	0.850**	0.52
	m88ksim	0.691*	0.767*
	mcf	0.737*	0.53
	parser	0.54	0.44
	perl2000	0.860**	0.669**
	twolf	0.737*	0.53
	vortex2000	0.38	0.37
	vpr	0.956**	.838**

Table 6.1: Correlation between static coverage of functions and two selected accuracy metrics, over all training profile and evaluation run pairs for each benchmark. '*' represents a value significant at the 0.05 level. '**' represents a value significant at the 0.01 level.

6.3 Usefulness of Profile-Driven Optimization

Profile-directed optimization in our two optimizers is discussed in [10] for `alto` and [4] for `cc`.

Cohn and Lowney compare the differences in usefulness between profile-driven optimization and static estimation on the Compaq Alpha in [4]. They report a substantial speedup (17%) on the SPEC 95 integer benchmarks from using feedback directed optimization.

Cohn and Lowney report a larger effect from profile-driven optimization than this paper. There are a number of reasons why their results are much stronger than ours for `cc` profile-driven optimization:

- They use more aggressive optimizations on a more recent iteration of the Alpha architecture.
- Another difference between their work and ours is that we use a wider variety of benchmarks (including SPEC2000 and floating point benchmarks) and benchmark runs than they do. This may also contribute to the performance gap between this paper and their work, particularly given that the optimizers we use pre-date SPEC2000 and have not been tuned for these applications at all.
- Our results include many short runs (as opposed to long-running SPEC “reference” runs) that generally show less optimization effect as they spend a much larger proportion of their time running hard-to-optimize start-up code and spend less time in intensive inner loops.

The Morph System is discussed in [19, 9]. Morph allows automatic collection of profile information by sampling and automatic profile-directed optimization. They claim a sampling overhead of 1%. They use the χ^2 goodness-of-fit test as a profile comparison metric[2], as well as key-matching, and report good results.

In a analysis of the Morph system in [9], the authors show accuracy and usefulness results for multiple sampling runs (up to 32 ”sampled runs of our test workloads”), combined into a single profile, as well as each of the 32 individual sampling runs and exact profiles. The overall results show that the combined sampling runs do almost as well (sometimes better) in usefulness terms on the

[2] We avoided implementing this profile comparison metric, as the χ^2 goodness-of-fit test produces questionable results when values in the distributions being compared are smaller than 5. This is the case for a large number of basic block counts in nearly all programs.

benchmarks as exact profiles. However, the average performance of the 32 different ”sample runs”, taken singly, was substantially worse, with extremely high variability.

The use of these combined profiles is extremely suspect. If a profiling technique with an alleged overhead of 1% must be run 32 times to produce good profile usefulness numbers, surely the actual profile-directed optimization overhead is 1% plus $31 \times 101\%$ or 3132%? It is clear that the authors discovered the substantial unreliability of sampled profiles but chose to focus instead on the rather questionable numbers produced by aggregating many sampled profiles together.

Wang and Rubin discuss the gathering of “user-specific” profiles in [16]. They carry out statistical analysis of the differences between different user profiles across a range of real-world application benchmarks. Their equivalent to an accuracy metric was the extent to which different procedures appeared in common in user-specific profiles. Note this is not quite the same thing as our “function-entry static coverage” metric - our metric considers the number of functions common to training profile and evaluation run as a percentage of those functions appearing in the evaluation run, and unlike their metric, is not symmetric. They found that (for different benchmarks) while 61% to 83.1% of procedures appeared in all users' profiles if they appeared in any, 1% to 9% of functions appeared only in the profile gathered by a single user.

Wang and Rubin go on to analyze the differing effects of these profiles on profile-driven optimization and translation using Digital's FX!32 profiling system, on the Microsoft Office applications “Microsoft PowerPoint” and “Microsoft Word”. While the different user-specific profiles they used caused little performance variation for PowerPoint, the variation was pronounced (9% difference between best and worst profiles) for Microsoft Word[3]. To use our terms, they observe that profile accuracy and profile usefulness have a “rough” association with each other but that the relationship is “not monotonic” (an observation that is supported in great detail by the results described in Section 2.4). They conclude that differences in program behavior from user to user is a important factor to consider for profile-directed optimization.

[3]It should be noted that the much of this variation is caused by a single profile (under the user pseudonym “Dopey”) that is described as a “small individual profile” included to “examine the issue of under-training”

6.4 Robustness under Uncertain Profile Data

Very little formal work on designing profile-driven optimization algorithms that are robust to variations in profile data exists. A paper by Deitrich and Hwu [6] introduces a heuristic (in the area of path-based global instruction scheduling) that reduces the profile driven optimization variability due to profile variation substantially. By using this heuristic as opposed to more conventional heuristics (speculative yield and dependence height), they report anywhere from a two- to five-fold reduction in profile-driven optimization variability due to this optimization.

Chapter 7

Conclusion

The results established in this thesis are, to some extent, artifacts of the benchmarks and optimizers analyzed. However, we feel that our methodology allows similar analyses of other benchmarks and optimizers to be carried out fairly easily.

Our major conclusions strongly suggest that profile-driven optimization remains an art, and not a science. The extremely local nature of profile usefulness in many benchmarks, combined with the inherently unreliable performance of even accurate profiles, makes profile-driven optimization a procedure that requires careful testing as opposed to something that can be used routinely.

The major contributions of the thesis are as follows:

- Quantification of the effects of profile-driven optimization under circumstances of exact profiling, sampling and static estimation
- A methodology for estimating the extent to which profile-driven optimization is significant, even with relatively small evaluation runs
- A methodology for estimating the connection between profile-driven optimization performance and profile accuracy
- The discovery that the connection between profile-driven optimization performance and accuracy is extremely limited; a property that is likely to be stable over a large range of profile accuracy metrics
- A methodology for localizing the effects of smaller units of profile information
- The discovery that profile-driven optimization performance is localized to a comparatively small set of functions in many benchmarks

The limited association between profile usefulness and profile accuracy suggests that more accurate profile information is not always used especially effectively, and that a major influence on the profile-directed optimization outcomes is a nearly-random perturbation of optimization outcome as opposed to a systematic improvement resulting from better optimization information. More positively, it is possible that a weak connection between usefulness and accuracy might exist in a profile-driven optimization system that was especially robust in dealing with 'bad profiles'; however, the wide range of results that we observed (including many cases where profile-driven optimization caused slowdowns) does not support this optimistic view.

Our work strongly suggests that any innovations in profiling (at least those that are primarily directed at profiling for automatic program optimization as opposed to those that are intended for providing source-level information to the programmer) be evaluated for actual impact of optimization performance as opposed to their profile accuracy as compared to the corresponding exact profiles.

As might be expected given the limited connection between usefulness and accuracy, sampled profiles are slightly but not severely worse than exact profiles. However, the small performance loss associated with sampled profiling - as well as the added indeterminacy associated with this kind of imperfect information gathering system, may make sampling a poor choice for traditional applications of profile-directed optimization. We feel that in such contexts - where a program's profiling execution is run strictly for the purpose of profiling - many of the arguments for sampled profiling are not applicable. While exact profiling may impose an order of magnitude more profiling overhead over sampling, the overall impact of this overhead is small.

We can only speculate about the context of run-time optimizations, not having studied any such optimizations in this work. The reduced overhead of sampling is highly significant in this context. However, if the connection between profile accuracy and profile-driven optimization performance was as slender in this context as it has been in our optimizers, then, at least for many benchmarks, there would be comparatively little point waiting until run-time to gather this kind of highly accurate profile. Again, applicability of such systems may turn out to be highly localized and again, much more of an art than a science.

It seems premature to declare, as Smith does in [12], that run-time profile-directed optimization methods enjoy a clear superiority over allegedly "ineffective" traditional methods. Smith conflates the issue of whole-program optimizations versus single unit optimizations with the issue of when to perform profile-directed optimization. Certainly, until a study similar to this thesis is conducted in the context of dynamic optimizations, it will be unclear whether the additional information that can be gained at any given run-time is that much more useful than that derived from earlier executions of a program.

If run-time optimizations behaved in a similar fashion to the compile-time profile-driven optimizations we evaluated, we might encounter a major problem: much of the benefit of profile-driven optimization in practice seems to come from manual enabling and evaluation of profile-driven optimization success. After all, the average improvement due to profile-driven optimization on our various benchmarks was quite small and included a number of cases of profile-driven pessimization. At run-time, no such selectivity is possible. This problem may not occur in practice, as it may be the case that dynamic optimizations have better connections between the gathering of accurate information and actual profile-driven optimization success.

Static estimation seems like a promising way of deriving profiles with low overhead. However, in keeping with current optimization practice, we have established that statically estimated profiles cannot be used interchangeably with exact profiles and typically result in performance that is no better, and often worse, than not using profile-directed optimization at all. If static estimation is to be used in profiling systems, following the example of `alto` and making profile-directed optimizations very conservative when using statically-estimated profiles seems like the ideal approach.

The addition of other forms of profile information to static estimation resulted in some improvement. However, coverage information was generally unhelpful. Statically-estimated profiles augmented with function entry count information outperformed regular static profiles. Unfortunately, the overhead involved in gathering even the limited amount of data required for entry counts might render this improvement academic; as long as we require the programmer to instrument their program and execute training runs, it seems as if gathering an exact profile is worthwhile. A similar argument applies to the use of sampled profiles (although sampled profiles are still useful in contexts where exact profiles cannot be gathered).

Our methodology for localization of profile-driven optimization effects will be valuable mainly as an analytic tool. When developing future profile-driven optimizations and evaluating their effects, this methodology will allow the specific functions (or interactions between functions) that have major profile-driven optimization effects to be noted and examined. Unfortunately, the profiling implications of our localization discoveries are few - we found ourselves unable to find any indicators *before the fact* that profile information for a given function might have a disproportionately large effect on profile-driven optimization performance.

7.1 Future Directions

Our results should be repeated using a more modern architecture and a more recent optimization system. Profile-directed optimization is often cited as an increasingly important factor in compiler performance for more modern architectures, as both potential instruction-level parallelism and the depth of the memory hierarchy increase.

The work can and should be generalized for other forms of profile data, especially the use of trace- and path-based profile directed approaches. A detailed analysis of the potential improvements due to path profiling - as well as the sensitivity of path-based approaches to imprecise information would help establish the value of path profiling in practice. Similarly, there are many optimizations in the realm of object-oriented programming that need to be analyzed in detail.

A wider choice of benchmarks is called for. Of particular importance is the the inclusion of larger applications that are more typical of user applications.

In addition, our factorial experiments to determine which functions were most important in the profile-driven optimization process need to be expanded in a number of directions. First, more powerful statistical methods from the area of screening experiments should be used to reduce the experiments down to a more manageable size. Second, the results that use whole functions are less illuminating than breaking up the profile data into smaller units: into regions, loops or even individual branches. Of course, the finer-grain the factors are, the more factors that must be screened for significant effects.

Our results suggest that optimization design should be made more robust in response to profile data. It is clear that in a number of benchmarks, profile-directed optimization causes substantial performance degradation. This indicates one of two things. First, the optimization could be inherently unreliable and prone to producing bad performance with any profile - including "perfect" information about future program behavior. Second, the optimization could produce startlingly bad performance only with poor profile data.

The first case can be fixed by comparatively simple methods. The second case, however, creates more of a dilemma. Attempting to make an optimization more robust in the presence of unreliable profile data may reduce profile-directed improvements that come from taking a more aggressive response. However, we contend that the current unreliability of profile-directed optimization is a major factor in its neglect by most programmers.

Bibliography

[1] Iml++ (iterative methods library). http://math.nist.gov/iml++/.

[2] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? Technical Report 1997-016, Digital Equipment Corporation Systems Research Center, 1997.

[3] T. Ball and J. R. Larus. Branch prediction for free. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313, 1993.

[4] R. Cohn and P. Lowney. Feedback directed optimization in Compaq's compilation tools for Alpha. In *In Proc. 2nd Workshop on Feedback Directed Optimization*, pages 1–10, 1999.

[5] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

[6] B. L. Deitrich and W. Hwu. Speculative hedge: Regulating compile-time speculation against profile variations. In *International Symposium on Microarchitecture*, pages 70–79, 1996.

[7] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Workload design: Selecting representative program-input pairs. In *The Eleventh International Conference on Parallel Architectures and Compilation Techniques (PACT-2002)*, pages 83–94, 2002.

[8] J. Fisher and S. Freudenberger. Predicting conditional branches from previous runs of a program. *Proceedings Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–95, 1992.

[9] N. Gloy, Z. Wang, C. Zhang, B. Chen, and M. Smith. Profile -based optimization with statistical profiles. Technical Report 02-97, Havard, 1997.

[10] R. Muth, S. Debray, S. Watterson, and K. de Bosschere. alto: A link-time optimizer for the DEC Alpha. Technical Report TR98-14, Department of Computer Science, The University of Arizona, 1998.

[11] S. Savari and C. Young. Comparing and combining profiles. In *Proc. Second Workshop on Feedback-Directed Optimization (FDO)*, pages 50–62, 1999.

[12] M. Smith. Overcoming the challenges to feedback-directed optimization. In *Proc. ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, 2000.

[13] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1994.

[14] D. W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 59–70, Toronto, Ontario, Canada, June 1991.

[15] R. Walpole, R. Myers, and S. Myers. *Probability and Statistics for Engineers and Scientists*. Prentice Hall, 1998.

[16] Z. Wang. A statistical analysis of user-specific profiles. Technical Report 09-98, Havard, September 1998.

[17] P. Winterbottom. Acid: A debugger built from a language. Technical report, Murray Hill, NJ, USA, 2000.

[18] Y. Wu and J. Larus. Static branch frequency and program profile analysis. In *In 27th International Symposium on Microarchitecture*, pages 1–11, 1994.

[19] C. X. Zhang, Z. Wang, N. C. Gloy, J. B. Chen, and M. D. Smith. System support for automated profiling and optimization. In *Symposium on Operating Systems Principles*, pages 15–26, 1997.