Combined Profiling: Practical Collection of Feedback Information for Code Optimization

Paul Berube
University of Alberta
Dept. of Computing Science
Edmonton, Alberta, Canada
T6G 2R3
pberube@ualberta.ca

Adam Preuss
University of Alberta
Dept. of Computing Science
Edmonton, Alberta, Canada
T6G 2R3
apreuss@ualberta.ca

José Nelson Amaral
University of Alberta
Dept. of Computing Science
Edmonton, Alberta, Canada
T6G 2R3
jamaral@ualberta.ca

ABSTRACT

Feedback-directed optimization (FDO) depends on profiling information that is representative of a typical execution of a given application. For most applications of interest, multiple data inputs need to be used to characterize the typical behavior of the program. Thus, profiling information from multiple runs of the program needs to be combined. We are working on a new methodology to produce statistically sound combined profiles from multiple runs of a program. This paper presents the motivation for *combined profiling* (CP), the requirements for a practical and useful methodology to combine profiles, and introduces the principal ideas under development for the creation of this methodology. We are currently working on implementations of CP in both the LLVM compiler and the IBM XL suite of compilers.

1. INTRODUCTION

Feedback-Directed Optimization (FDO), in the context of an Ahead-of-Time (AOT) compiler, consists of collecting information about the behavior of a program from a training run and then using this information for a new compilation of the program [12]. A number of speculative optimizations are known to benefit from FDO, including speculative partial redundancy elimination [5, 8], trace-based scheduling and others [2, 4]. However, most AOTs make only very limited use of FDO. One reason is the lack of an efficient profiling infrastructure by which it is practical to combine profiling data collected from multiple runs of a program. In most cases it is not possible to characterize the expected behavior of a program at runtime from a single training run. Limited efforts in the past towards combining profiling from multiple runs were restricted to computing averages or sums of frequencies. However, often the distribution of the frequencies over multiple runs is important as well. Consider the execution of a simple `if-then-else` statement within a hot region of a procedure. Suppose that when the profile from k runs of the application with different inputs are averaged, each branch of the conditional is executed 50% of the time. Now consider two cases: (1) In each individual run the condition is true 50% of the time; (2) In half of the runs the condition is always false, while in the other half the condition is always true. Different code transformations may be implemented by a compiler designer if this distribution is known, as compared with knowing only the average execution frequency of each branch, which is identical in both cases.

Hence, there is a need for a proper and practical methodology to combine multiple profiles. One of the requirements for a practical methodology is that it must allow the *online* collection and combination of profiling data. In other words, it must be possible to update the combined profiling information without access to the profiling of each individual run of the program that contribute to the combined profiling.[1] The advantage of the online collection of the combined profile is that there is no need to store all the previous profiles that were collected. In an environment in which not all training runs happen at once, this could be a great advantage. For instance, for an important application, there could be a profiling and a normal code generated. During the normal use of the deployed application, the behavior of the application would be sampled by running a profiling run either at regular intervals or at random with a set frequency over the total runs of the application.

In such an environment, both compiler and architecture designers benefit from a much more realistic characterization of the execution of applications. More precise profiling of an application affords FDO a much better chance to succeed in commercial environments. Moreover, when presented with more nuanced information about the application's runtime behavior, designers should be able to integrate this information in their FDO algorithm.

This paper describes the development of a statistically sound and practical technique to combine the profiles from multiple runs of an application and to allow relevant queries into the combined profile. Both batch and online combination of profiles are supported. To the best of our knowledge, this work in progress is the first to address the issue of combining profiles from multiple runs of an application in a non-trivial manner. The main contributions of this work will be:

- *Combined profiling* (CP), a statistically-sound method-

ICPE'11, March 14–16, 2011, Karlsruhe, Germany.

[1] The alternative is a *batch* methodology that collects the profiling from all the runs before the computation of the combined profile.

ology to combine data from multiple FDO training runs, including a space-efficient combined representation and statistical queries to inform code transformations.

- *Hierarchical normalization* (HN), an algorithm that allows profiles to be combined while maintaining both the local and global relative frames of reference for each monitor.
- *Overlap Metrics*, which quantify the difference in dynamic code coverage captured by alternative profiles.
- An implementation of CP for both edge and path profiling in the open-source `LLVM` compiler, and an evaluation of these implementations in terms of compilation and profiling overhead, profile size, and profile overlap.

2. RELATED WORK

Savari and Young build a branch and decision model for branch data [11]. Their model assumes that the next branch and it's outcome are independent of previous branches[2]. The resulting model provides a distribution across all the events of a certain type for a single program run. Distributions from different runs are combined by using relative entropy to find a mixture that is equally (dis)similar to the original distributions. In essence, this is a sophisticated way to find the weights for a weighted geometric average across runs. Each event is still represented by a single number (expected frequency). In contrast, CP's statistically-sound distributions are for a single event across multiple program runs.

Fisher and Freudenberger measure instructions per break in control flow and sum profiles to provide better branch prediction [6]. Point summaries created by summing raw frequencies produce similar results to summing normalized frequencies. Usually, summed profiles perform better than single-run profiles, but poor prediction still occurs in the presence of multiple program use cases and poor training input selection, two issues addressed by CP.

Simple sums and averages, or in fact any point summary, cannot adequately represent workloads containing behavior variation. CP is designed for exactly this purpose.

Many code transformations have shown the utility of accurate profile information. For instance, Young and Smith use the information from a path profiler to improve global scheduling [14]. Ammons and Laurus improve the precision of data flow analysis only along hot paths [1]. Gupta *et al.* use path profiling to guide partial dead code elimination in an architecture with predication [7, 2]. Path profiling is also essential for the success of speculative partial redundancy elimination (SPE) [5, 8]. The importance of an accurate profile for these transformations foreshadows problems when program behavior varies by data input, but this information is neither captured by profiling nor taken into account by compiler heuristics.

3. COMBINED PROFILING

Significant performance gains can be achieved through the use of FDO. Nonetheless, FDO has not achieved widespread use by compiler users. A major challenge in the use of FDO is the selection of a data input to use for profiling that is representative of the execution of the program throughout its lifetime. For large and complex programs dealing with many use cases and used by a multitude of users, assembling an appropriately representative workload may be a difficult task. Picking one training run to represent such a space is far more challenging, or potentially impossible, in the presence of mutually-exclusive use cases. Moreover, user workloads are prone to change over time. Performance gains today may not be worth the risk of potentially significant performance degradation in the future.

CP eases the burden of training-workload selection while also mitigating the potential for performance degradation. First, there is no need to select a single input for training,[3] because data from any number of training runs can be merged into a combined profile. More importantly, CP preserves variations in execution behavior between inputs. The distribution of behaviors can be queried and analyzed by the compiler when making code transformation decisions. Modestly profitable transformations can be performed with confidence when they are beneficial to the entire workload. On the other hand, transformations expected to be highly beneficial on average can be suppressed when performance degradation would be incurred on some members of the workload.

Over the lifetime of a complex application, the patterns of execution of the program may change, and thus the relative importance of different execution paths will also change. However, if a large number of distinct inputs are used to generate a combined profile, it is likely that most of the patterns that will become frequent in the future are represented by the workload. Therefore, providing the FDO algorithm with the variation and the outliers in the distribution may allow for the anticipation of the impact of a code transformation in future frequent patterns of execution. A possible future use of CP is to include triggers in the generated code that either switch to an alternative version of the code, or suggest to the user that the application should be re-profiled and recompiled because of changes in execution pattern detected by some deployed sampling profiling mechanism.

3.1 Measuring Program Behavior

A methodology to combine profiles collected from multiple runs of a program is applicable to a variety of profiling techniques. The profile of a program records information about a set of *program behaviors*. A program behavior $\mathcal{B}$ is a (potentially) dynamic feature of the execution of a program. The observation of a behavior $\mathcal{B}$ at a location l of a representation of the program is denoted $\mathcal{B}_l$.[4] A behavior $\mathcal{B}$ is quantified by some metric $M(\mathcal{B})$ as a tuple of numeric values. A *monitor* $R(\mathcal{B}, l, M)$[5] is injected into a program at every location l where the behavior $\mathcal{B}$ is to be measured using metric M. At the completion of a *training run*, each monitor records the tuple $\langle l, M(\mathcal{B}_l)\rangle$ in a *raw profile*. A *raw profile* contains unmodified metric values, as opposed to other profiles that may contain processed values. We refer to the value (or distribution) of the metric of a monitor simply as the value (or distribution) of the monitor. For example, in naive edge profiling, the locations l are the edges of the

[2] This assumption of independence is never explained or justified, but is clearly violated (*e.g.*, correlated branches).

[3] We firmly believe that training must always use a workload formed by multiple inputs.

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

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

CFG, the metric M is the frequency of execution of each edge and the observation $\mathcal{B}_l$ of the behavior $\mathcal{B}$ is the traversal of the edge during program execution. In this case the raw edge profile contains a listing of $\langle \text{edgeID}, \text{count} \rangle$ pairs, with one record for each monitor/edge.

For simplicity, consider a program with a single monitor, R. When no program state is shared between executions, the raw profile from training run i provides one independent sample,[6] R_i, of the possible values of R. In other words, each R_i is an independent random variable identically distributed according to some unknown probability distribution D. If D were known, statistical inference about the values of R would be possible.

3.2 Approximating the Empirical Distribution

Section 3.1 presented the idea of constructing a distribution model of each monitor from the observed values. To facilitate the use of CP with existing FDO compilers, a CP file should be a drop-in replacement for raw profiles. In particular, a CP file created from a single raw profile should be as informative as the original raw profile. As a matter of practicality, the distribution model should have a (small) bounded size, since it must be stored on disk but will also compete with the rest of the compiler for memory during compilation. Similarly, a CP should support both incremental and batch additions of raw profiles to give the user maximum data management flexibility.

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

Assuming that a monitor is uniformly distributed within a bin, it's histogram forms a contiguous n-step probability distribution. FDO's limited precision requirements make this assumption reasonable. The probability of the value of a monitor belonging to bin b_i is the proportion of the histogram's total weight falling in bin b_i. Thus, the distribution of a monitor's value has a well-defined and piece-wise continuous cumulative distribution function (CDF) and quantile function. Likewise, individual monitor values can be seen as degenerate histograms where all the weight is contained in a single minimum-width bin. Henceforth, all monitor values are assumed to be such histograms rather than scalar values.

3.3 Building Histograms

The histogram of a combined profile may be updated in a batch, incrementally, or by a hybrid approach. The update method is unaffected by the choice of update frequency. In general, updating produces a new histogram in 3 steps:

1. Determine the range of the combined data. Create a new histogram with this range.
2. Proportionally weight the bins of the new histogram.
3. Calculate new values for the true mean and variance.

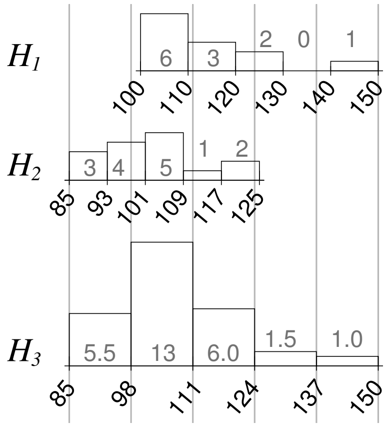


Figure 1: Combining histograms. H_1 has a bin width of 10 and a total weight of 12; H_2 has a bin width of 8 and a total weight of 15. The combined histogram H_3 has a bin width of 13 and a total weight of 27.

The combination of two histograms H_1 and H_2 into a new histogram H_3 is illustrated in Figure 1. The range of H_3 is simply the minimum encompassing range of the ranges of H_1 and H_2: $[min(100, 85), max(150, 125)] = [85, 150]$. This range is divided into the same number of bins as were present in the original histograms. The weight of a bin b_i of H_3 is given by the weights of the bins of H_1 and H_2 that overlap the range of b_i multiplied by the overlapping proportion. For example, let b_3 be the third bin of H_3 in Figure 1. In H_1 the bin width is 10, and in H_2 the bin width is 8. The weight of b_3 in H_3 is calculated as follows:

$$\begin{aligned}
W_{b_3}(H_1) &= \left(\frac{120-111}{10}\right)3 + \left(\frac{124-120}{10}\right)2 \\
&= \frac{27+8}{10} = 3.5 \\
W_{b_3}(H_2) &= \left(\frac{117-111}{8}\right)1 + \left(\frac{124-117}{8}\right)2 \\
&= \frac{6+14}{8} = 2.5 \\
W_{b_3}(H_3) &= 3.5 + 2.5 = 6.0
\end{aligned}$$

Updating the sample mean and variance of a combined profile uses the parallel algorithm due to Chan et. al.[3]. Given two data sets A and B, the inputs for this algorithm are the number of values, n, the sum of values S, and the sum of squared deviations, SS:

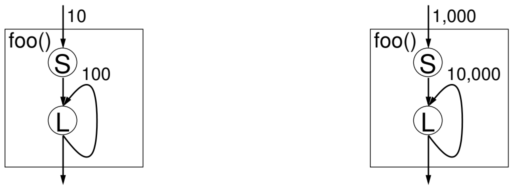
$$\begin{aligned}
S &= S_A + S_B \\
SS &= SS_A + SS_B + \frac{n_A n_B}{n_A + n_B}\left(\frac{S_A}{n_A} - \frac{S_B}{n_B}\right)^2
\end{aligned}$$

The sample mean $\bar{x}$ and the variance σ^2 of the combined profile of the data sets A and B are computed as follows: $\bar{x} = \frac{S}{n}$ and $\sigma^2 = \frac{SS}{n}$.

3.4 Why Not Use Parametric Models?

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

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



(a) `foo()` x10

(b) `foo()` x1,000

Figure 2: The loop L in `foo()` iterates 10 times.

The empirical distribution is a non-parametric model that makes no assumptions about the shape of the data. Parametric probability models assume that data comes from a family of distributions characterized by a fixed set of parameters. Building the model entails estimating the values of those parameters. For instance, a normal distribution is parameterized by the mean and standard deviation of the data. While those two parameters are easily estimated and have a small space requirement, we have no justification to assume that monitor values are distributed according to any particular distribution. More flexible parametric models can better estimate arbitrary distributions, but require a larger number of parameters. Unfortunately, accurately estimating many parameters necessitates many data samples in order to constrain each degree of freedom in the model. For example, the generalized lambda distribution can approximate a large number of well-known distributions, but is parameterized by the first four moments of the data [9]. Thus, the model may be very different from the real distribution of the data when the number of raw profiles collected is small.

4. HIERARCHICAL NORMALIZATION

Building a distribution model of a program behavior can provide a compiler with richer information to guide code transformation. However, raw profiles cannot be combined naively. Consider the case of procedure `foo()` in Figure 2, who's only control flow construct is a loop that iterates exactly 10 times. Let S be the straight-line code outside the loop code L. Two profiles are generated: in Figure 2(a), `foo()` is invoked 10 times; in Figure 2(b), 1,000 times. Combining this information naively, we learn that S executes 10 or 1,000 times, while L executes 100 or 10,000 times. However, we have lost the correlation between the frequencies of S and L. According to this combined profile, there may be a single run in which S executes 1,000 times while L executes 100 times. The problem is that the two pairs of measurements were taken under different conditions. Thus, when combining these measurements, all values recorded for a monitor must be computed relative to a common fixed reference. *Hierarchical normalization* (HN) is a process to achieve this goal by decomposing a CFG into a hierarchy of dominating regions.

HN is presented for vertex profiling. Edge profiles are treated identically, but use the line graph of the CFG instead of the CFG itself. The line graph contains one vertex for each edge in the CFG. The edges in the line graph correspond to adjacencies between the edges of the CFG. This technique may be similarly applied to a call-graph.

Decomposing a CFG into a hierarchy of dominating regions to enable HN is achieved by constructing it's dominator tree. Each non-leaf node n in the tree is the head of a region G_n, which encompasses any regions headed by descendants of n. Let node p be the immediate dominator of node n and let R_p and R_n be the monitors at these nodes. To prepare a raw profile for combination with other profiles, each R_n is normalized against the value of R_p. Normalization proceeds in a bottom-up traversal of the dominator tree, so that the head of a region is only normalized to it's immediate dominator after all of it's descendents have been normalized. The root of the dominator tree, *i.e.*, the procedure entry point, is assigned a "normalized" value of 1.

The current formulation of HN does not apply to paths. The problem lies in loops, where multiple paths overlap and no path or set of paths strictly dominates the loop body. For example, there is a set of paths starting at the procedure entry and ending at the end of the innermost loop, and a set of paths consisting of the last iteration of the loop followed by paths to the exit. None of these paths strictly dominate any of the others, nor do any of them dominate or post-dominate the loop body. Our implementation of combined path profiling in `LLVM` currently does normalization using the imprecise approximation that all paths are dominated only by the procedure entry point.

4.1 Denormalization

By design, hierarchical normalization leads to histograms built using relative execution frequencies. These histograms approximate *conditional* distributions such that the distribution of a monitor is conditioned on the execution frequency of it's immediate dominator being 1. Thus, monitor values can only be compared when their distributions share the same condition, *i.e.*, they have been normalized against the same dominator. Denormalization is the process of (statistically) reversing the effects of hierarchical normalization.

Let R_a and R_b be monitors from the same CFG, and let $dom^i(R_a)$ be the i^{th} most-immediate dominator of R_a. The least-common dominator of R_a and R_b is $R_d = dom^j(R_a) = dom^k(R_b)$. Adjusting R_a from being conditioned on $dom(R_a)$ to being conditioned on R_d is achieved incrementally by walking up the dominator tree. For example, the expected execution frequency of R_a conditioned with respect to R_d can be computed:

$$\begin{aligned}\mathbb{E}(R_a|R_d = 1) &= \mathbb{E}(R_a|dom(R_a) = 1) \times \\ &\prod_{i=2}^{j}(\mathbb{E}(dom^{i-1}(R_a)|dom^i(R_a) = 1))\end{aligned}$$

In this way, R_a and R_b can be compared via $\mathbb{E}(R_a|R_d = 1)$ and $\mathbb{E}(R_b|R_d = 1)$. Variance should be denormalized in a similar fashion to ensure a confident comparison result.

5. USING COMBINED PROFILES

Throughput and latency are two important measurements for performance evaluation. Considering throughput alone is sufficient to improve performance in a batch environment, and is the typical concern of compiler code transformations. After all, a user is usually concerned with the total execution time of a program, not the relative speed of individual control flow paths taken during the program's execution. However, when considering a workload of inputs composed of, for example, representative inputs from a software vendor's primary clients, the speed of the individual components matters. The clients do not care about overall program throughput, they care about program execution time for their particular workload. Consequently, code transformations should consider the distribution of program behaviors across the workload when presented with a combined profile.

A combined profile holds a wealth of knowledge that can be used to enhance feedback-directed code transformations. The utility of any particular statistic depends on the characteristics of the consuming transformation. A detailed treatment of non-parametric statistics and plug-in estimators is beyond the scope of this paper, but can be found in appropriate statistics texts [13]. Some examples of the types of queries that can be made to a combined profiling infrastructure are highlighted here.

Average Benefit: Code transformation heuristics that use raw execution frequencies, normalized frequencies, or sums of frequencies are attempting to maximize average benefit. However, the median can be a better representative of the common case than the mean for skewed distributions; CP allows both the skew and median of a distribution to be approximated. Of course, CP also provides the sample mean.

Worst-Case Cost: Considering latency alone is a worst-case analysis. Thus, heuristics should use the extreme values of the distribution for cost/benefit analysis. The sample maximum and minimum of a distribution are estimators for the true maximum and minimum, and are available through CP.

High-Certainty Benefit: The choice between average-case and worst-case analysis is quite coarse. However, the inverse CDF, or quantile function Q, enables finding ranges or thresholds corresponding to specified proportions of the data. Denote by R a monitor and by T a threshold value. $Q(p)$ asks: "What is the value of T such that $\mathbb{P}(R < T) = p$?" Thus, $Q(0)$, $Q(0.5)$, and $Q(1)$ are the sample minimum, median, and maximum, respectively; $Q(0.05)$ gives the threshold between the lowest 5% of the distribution and the upper 95%. A heuristic that determines that a transformation should be beneficial to 95% of the workload would likely be more reliable than one which conservatively requires that the average benefit be "large enough."

Sorting: Transformation opportunities are often evaluated in an order that requires that they first be sorted, such as a "hottest first" order. In addition to the statistics above, CP provides the sample variance to measure the spread of the distribution. Thus, a sorting function could prefer opportunities with more strongly peaked distributions. Furthermore, the variance can be used to construct a confidence interval around the mean in order to determine if a comparison result is statistically significant or the result of random chance. Overall, CP allows for the construction of nuanced sorting functions that are appropriate for the transformations for which they are used.

6. IMPLEMENTATION

We implemented both Ball-Larus path profiling and CP for edge and path profiling in the `LLVM` compiler. `LLVM` is a broadly used compiler infrastructure that has had great influence both in the development of compilers and in the delivery of commercial products to the market [10]. This implementation of path profiling was verified against an independently developed edge profiler using the SPEC CPU 2000 and the SPEC CPU 2006 benchmark suites.

Statistic	Deficiency		Exclusivity	
	#	%	#	%
Maximum	395	18.5	3	0.141
Minimum	88	4.13	0	0
Mean	210.35	9.88	0.015	7.00×10^{-4}
σ^2	90.2	0.542	0.197	9.25×10^{-3}

Table 1: Deficiency and exclusivity over 1000 inputs to `bzip2`. 2130 edges are executed at least once over the full workload.

To compare the coverage of an application behavior by CP with the coverage of profiling of a single run of the program, we defined two metrics: *deficiency* and *exclusivity*. Both metrics use a leave-one-out cross-validation methodology; Let s_i be the left-out input, and S the set of remaining inputs. Denote by $C(s_i)$ and $C(S)$ the sets of edges with non-zero frequency in the raw profile for s_i and the combined profile of S, respectively.

Deficiency counts the number of edges covered by S that are not covered by s_i, *i.e.*, information lost when s_i is used alone to estimate frequencies:

$$Def(i) = |C(S) - C(s_i)|$$

y Conversely, *exclusivity* counts the number of edges covered by s_i but not by S, *i.e.*, information lost due to omitting s_i from the combined profile:

$$Excl(i) = |C(s_i) - C(S)|$$

Table 1 shows deficiency and exclusivity metrics for edge profiles of `bzip2` (a relatively simple program). This preliminary data confirms that combined profiles capture information about parts of a program that are difficult to capture with a single input. Any one input covers at most 3 edges not covered by the combined profile, while the combined profile covers at least 88, and on average 210, more edges than a single profile.

7. ON-GOING WORK

Combined profiling is a practical approach to addressing the issue of combining profiles from multiple training runs while capturing the variations between those runs. Initial results show that behavior variation is present in even a simple program like `bzip2`, and this variation can be captured by a statistical model. We are currently working on a more extensive evaluation of CP on `LLVM`. Also, we are finalizing the design of an extension of CP to apply it inter-procedurally through the use of the program's call graph. Finally, we are examining existing FDO transformations to design new strategies for their application based on the additional information made available by CP.

8. ACKNOWLEDGMENTS

This research is supported by fellowships and grants from the Natural Sciences and Engineering Research Council of

Canada (NSERC), the Informatics Circle of Research Excellence (iCORE), and the Canadian Foundation for Innovation (CFI).

9. REFERENCES

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

[2] R. Bodík and R. Gupta. Partial dead code elimination using slicing transformations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 159–170, Las Vegas, NV, USA, 1997.

[3] T. F. Chan, G. H. Golub, and R. J. LeVeque. Updating formulae and a pairwise algorithm for computing sample variances. Technical Report STAN-CS-79-773, Stanford University, November 1979.

[4] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. R. Rau, and M. Schlansker. Profile-driven instruction level parallel scheduling with application to super blocks. In *Intern. Symposium on Microarchitecture (MICRO)*, pages 58–67, Paris, France, December 1996.

[5] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on ssa form. In *Programming language design and implementation*, pages 273–286, Las Vegas, NV, USA, 1997.

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

[7] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial dead code elimination using predication. In *Parallel Architectures and Compilation Techniques (PACT)*, page 102, San Francisco, CA, USA, 1997.

[8] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial redundancy elimination using speculation. In *Intern. Conf. on Computer Languages (ICCL)*, pages 230–239, 1998.

[9] A. Lakhany and H. Mausser. Estimating the parameters of the generalized lambda distribution. *Algo Research Quarterly*, 3(3):47–58, December 2000.

[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Intern. Symp. on Code Generation and Optimization (CGO)*, San Jose, CA, USA, March 2004.

[11] S. Savari and C. Young. Comparing and combining profiles. *Journal of Instruction-Level Parallelism*, 2, May 2000.

[12] M. D. Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1–11, 2000.

[13] L. Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 2003.

[14] C. Young and M. D. Smith. Better global scheduling using path profiles. In *Intern. Symposium on Microarchitecture (MICRO)*, pages 115–123, Dallas, TX, USA, 1998.