Combined Profiling: A Methodology to Capture Varied Program Behavior Across Multiple Inputs

Paul Berube
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
Email: pberube@ualberta.ca

José Nelson Amaral
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
Email: pberube@ualberta.ca

***Abstract*—This paper introduces *combined profiling* (CP): a new practical methodology to produce statistically sound combined profiles from multiple runs of a program. Combining profiles is often necessary to properly characterize the behavior of a program to support Feedback-Directed Optimization (FDO). CP models program behaviors over multiple runs by estimating their empirical distributions, providing the inferential power of probability distributions to code transformations. These distributions are build from traditional single-run point profiles; no new profiling infrastructure is required. The small fixed size of this data representation keeps profile sizes, and the computational costs of profile queries, independent of the number of profiles combined. However, when using even a single program run, a CP maintains the information available in the point profile, allowing CP to be used as a drop-in replacement for existing techniques. The quality of the information generated by the CP methodology is evaluated in `LLVM` using SPEC CPU 2006 benchmarks.**

I. Introduction

Current techniques to report performance based on experimental evaluation of systems lack rigor. Published evaluations reveal that many members of the system community assume that a single input is enough to evaluate the performance of a program. Moreover, many researchers fail to discuss how they address, and often do not report, variations in their measurements. Previous research found significant program behavior variations across inputs, but little research dealt with input-dependent behavior [1], [2], [3], [4].

Capturing behavior variations across inputs is important in the design of an ahead-of-time (AOT) compiler where feedback-directed optimization (FDO) consists of collecting information about the behavior of a program from training runs and then using this information for a new compilation of the program [5]. A number of speculative code transformations are known to benefit from FDO, including speculative partial redundancy elimination [6], [7], trace-based scheduling and others [8], [9]. Several open questions remain about the use of profiles collected from multiple runs of a program. How should the multiple profiles be combined? Is it sufficient to simply average the multiple measurements? Is it necessary to compute the parameters for an assumed statistical distribution of the measurements? Or is there a simple technique to combine the measurements and provide useful statistics to FDO?

This 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).

This paper addresses these questions by arguing that the behavior variations in an application due to multiple inputs should be evaluated by FDO decisions. It also argues that a full parametric estimation of a statistical distribution is not only unnecessary, but it may also mislead FDO decisions if the wrong distribution is assumed or there is insufficient data to accurately estimate the parameters. Instead, it proposes the use of a non-parametric empirical distribution that makes no assumptions about the shape of the actual distribution. The new technique presented here is called *Combined Profiling* (CP). CP is designed with the following goals: (1) It must provide FDO with information about the variability of the application behavior over multiple runs; (2) It must be computed incrementally, *i.e.* the raw data from previous runs need not be available to incorporate a new run; (3) It should be simple to compute and to understand; (4) It should capture more nuances of the program behavior than a simple average of profiles; (5) It should work for CFG edge and path profiling, for context sensitive and context-insensitive call graphs, and for value profiling. The application of CP to other situations with multiple profiling instances, such as profiling program phases individually, is not within the scope of this paper.

The main contribution of this paper are:

- *Combined profiling* (CP), a statistically sound methodology to combine data from multiple runs, including a space-efficient combined representation and statistical queries to inform code transformations (Section II).
- *Hierarchical normalization*, an algorithm to maintain the local and global frame of reference for each monitor when combining profiles (Section III).
- *Experimental Evaluation* analyzing behavior variability and CP characteristics across input workloads for SPEC CPU 2006 benchmarks (Section V).

II. Combined Profiling

A major challenge in the use of FDO is the selection of a profiling data input that is representative of the execution of the program throughout its lifetime. For large and complex programs dealing with many use cases and used by a multitude of users, assembling an appropriately representative workload

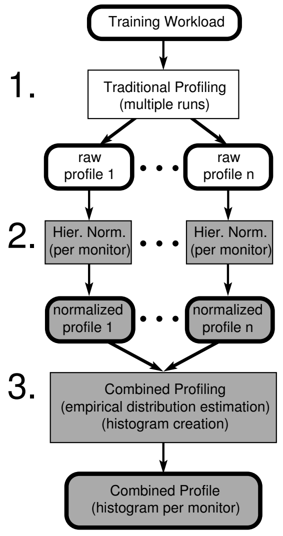

Training Workload
1.
Traditional Profiling (multiple runs)
raw profile 1
raw profile n
2.
Hier. Norm. (per monitor)
Hier. Norm. (per monitor)
normalized profile 1
normalized profile n
3.
Combined Profiling (empirical distribution estimation) (histogram creation)
Combined Profile (histogram per monitor)

Fig. 1. Combined profiling flow diagram

may be a difficult task. Picking a solitary training run to represent such a space is far more challenging, or potentially impossible if use-cases are mutually-exclusive. Moreover, user workloads are prone to change over time. Ensuring stable performance across all inputs in today's workload prevents performance degradation due to changes in the relative importance of workload components.

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

Combining profiles is a three-step process, as illustrated in Figure 1[1]: (1) Collect raw profiles via traditional profiling. (2) Apply *Hierarchical Normalization* (HN) to each raw profile. (3) Apply CP to the normalized profiles to create the combined profile.

A. Design Considerations

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

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

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

B. Measuring Program Behavior

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 $\mathcal{B}_l$.[2] A behavior $\mathcal{B}$ is quantified by some metric $M(\mathcal{B})$ as a tuple of numeric values. A *monitor* $R(\mathcal{B}, l, M)$[3] is injected into a program at every location l where the behavior $\mathcal{B}$ is to be measured using metric M. At the completion of a *training run*, each monitor records the tuple $\langle l, M(\mathcal{B}_l)\rangle$ in a *raw profile* that contains unmodified metric values. The value (or distribution) of the metric of a monitor is simply called the value (or distribution) of the monitor. For example, in naive edge profiling, the locations l are the edges of the Control Flow Graph (CFG), the metric M is the execution count of each edge, the observation $\mathcal{B}_l$ of the behavior $\mathcal{B}$ is the traversal of the edge during program execution, and the raw edge profile contains a listing of $\langle \text{edgeID}, \text{count}\rangle$ pairs.

For simplicity, consider a program with a single monitor, R_1. When no program state is shared between executions, the

[1]Shaded components of the figure represent contributions of this work.

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

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

raw profile from each training run i provides one independent sample,[4] $R_1[i]$, of the possible values of R_1. Thus, each $R_1[i]$ is an independent random variable identically distributed according to some unknown probability distribution D, which arises as the result of the interactions between a program and its inputs.

C. Queries

In an AOT compiler, profiles are used to predict program behavior. Thus, raw profiles are statistical models that use a single sample to answer exactly one question: *"What is the expected frequency of X?"* where X is an edge or path in a CFG or a Call Graph (CG). A CP is a much richer statistical model that can answer a wide range of queries about the measured program behavior. Using a set of raw profiles, a CP provides the mean monitor value, analogous to a raw profile, along with the standard deviation and minimum and maximum values. Furthermore, CP can estimate from a monitor's Cumulative Distribution Function (CDF) the probability that the monitor is within a (possibly half-bounded) range. Conversely, the inverse CDF provides estimates of the thresholds corresponding to a given quantity of probability mass. For example, the inverse CDF facilitates estimating the median value of a monitor.

This information allows an FDO heuristic to quantify the expected trade-offs between various workload-performance measurements, such as between the impact on the 5%-quantile (nearly worst-case) or the average impact on the 5%–95%-quantile range (omitting potential outliers). In some transformations, the order in which candidates are considered is important [10]. CP allows a sorting function to use, for example, both the expected frequency and the variance of the candidates in order to prioritize low-variance opportunities. Behavior variation should not, by itself, inhibit optimization. Rather, CP enables the accurate assessment of the potential performance impact of transformations based on varying monitors in a variety of ways, and with adjustable confidence in the result.

D. Approximating the Empirical Distribution

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

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. Assuming that a monitor is uniformly distributed within a bin[5], 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 monitor's distribution has a well-defined and piecewise continuous CDF and inverse CDF. Likewise, individual monitor values can be seen as degenerate histograms where all the weight is contained in a single point.

E. 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. Given and initial histogram, 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 according to their overlap with bins of the old histogram(s). Add weight for point data as usual. (3) Update the values of the true mean and variance. Berube *et al.* present an example illustrating this process [11].

III. Hierarchical Normalization

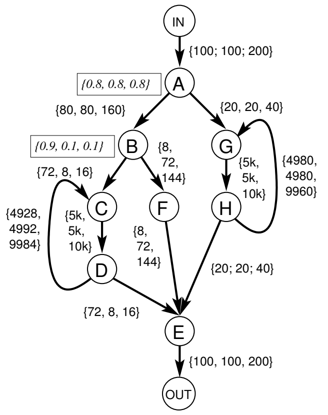
CP provides a data representation for profile information, but does not specify the semantics of the information stored in the combined profile. Raw profiles cannot be combined naively. To illustrate this point, Figure 2(a) presents a CFG and the table in Figure 2(b) provides the edge frequencies observed for three profiles: P1, P2, and P3. The numbers within the rectangles are the probabilities for edges A$\rightarrow$B and B$\rightarrow$C. First note that averaging values across profiles is misleading because it can easily characterize behavior in a way that does not correspond to any individual profile; The average branch probability at B is 0.37, hiding its strongly biased behavior. In all three profiles, the probability of entering the G-H loop from A is 0.2. The loop trip counts for P1 and P2 are identical, but the probability of entering the C$\rightarrow$D loop from B is 0.9 in P1 and 0.1 in P2. P3 is identical to P2, except that all edge counts are doubled. Therefore, P2 and P3 are essentially the same profile; if they were combined, the resulting profile should not show any variation in program behavior. However, if the two raw frequencies for an edge such as G$\rightarrow$H were combined into a histogram, the values 5,000 and 10,000 would not suggest this consistent behavior.

On the other hand, the raw frequencies for edge C$\rightarrow$D in P1 and P2 are both 5,000, but P1 enters the loop much more frequently than P2 due to the 0.9 vs 0.1 branch probability at B. Therefore, the average trip count of the loop in P1 is much lower (69.4) than in P2 (625). In this case, histogramming the raw frequencies suggests consistent behavior for the loop, which is misleading.

The problem in both cases is that the pairs of measurements were taken under different conditions. Thus, when combining

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

[5]as in a Riemann sum



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

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

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

Fig. 2. The CFG of a procedure with three possible edge profiles

these measurements, all values recorded for a monitor must be normalized relative to a common fixed reference. *Hierarchical normalization* (HN) is a profile semantic designed for use with CP that achieves this goal by decomposing a CFG into a hierarchy of dominating regions. The results of using HN for the profiles in Figure 2(b) are shown in the right portion of the table. As desired, P2 and P3 are identical, and the differences in loop trip count between P1 and P2 are identified.

HN is presented for 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. Denote the immediate proper dominator of node n by $dom(n)$. Each non-leaf node n in the tree is the head of a region G_n, which by construction encompasses any regions entered through descendants of n. To prepare a raw profile for combination with other profiles, the frequency f_n of each non-root node n is normalized against the frequency of it's immediate proper dominator, $f_{dom(n)}$. The ratio of these two frequencies is invariant when a branch probability or loop iteration count is (dynamically) constant. Along with the issues illustrated in Figure 2, this process also prevents variable behavior in an outer loop from masking consistent behaviors within the loop. Normalization proceeds in a bottom-up traversal of the dominator tree, so that the head of a region is normalized to its immediate dominator only after all 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 HN for the example is shown in the left portion of the table in Figure 2(b).

In order to understand the capabilities and limitations of a statistical model incorporating HN, and to use it correctly, the model must be precisely defined. Therefore, let $\mathcal{F}_n$ and $\mathcal{F}_{dom(n)}$ be random variables for the raw frequencies of n and $dom(n)$, respectively. Define a new random variable $Y_n = \frac{\mathcal{F}_n}{\mathcal{F}_{dom(n)}}$, which is the frequency of node n with respect to its dominator. The raw profile from run 1 of the program records $f^1_{dom(n)}$ and f^1_n, the observed frequencies of the two nodes over that run. One sample of Y_n, $y^1_n = \frac{f^1_n}{f^1_{dom(n)}}$ is calculated as the hierarchically normalized value for n. Over k runs, k samples $y^1_n, y^2_n, ..., y^k_n$ are added to the histogram of R_n. Thus, the distribution summarized by the histogram of monitor R_n is an approximation for the true distribution:

$$R^*_n(\theta) = \mathbb{P}(Y = \theta) = \mathbb{P}\left(\frac{\mathcal{F}_n}{\mathcal{F}_{dom(n)}} = \theta\right)$$

A. Denormalization

The properties of R_a can only be directly compared to those of R_b when $dom(a) = dom(b)$. However, more generalized reasoning about R_a may be needed when considering code transformations. *Denormalization* statistically reverses the effects of hierarchical normalization to lift monitors out of deeply-nested domination regions by marginalizing out the distribution of the dominators above which they are lifted. Denormalization is a heuristic method rather than an exact inference because it assumes statistical[6] independence between monitors.

Consider first the hierarchically-normalized raw profiles in Figure 2. Intuitively, the expected execution count of node F for a single execution through the graph is calculated:

$$\begin{aligned}
\text{BP}_l(A) &= \left(\frac{f_{A\leftarrow B}}{f_{A\leftarrow B} + f_{A\leftarrow G}}\right)\\
\text{BP}_r(B) &= \left(\frac{f_{B\leftarrow F}}{f_{B\leftarrow C} + f_{B\leftarrow F}}\right)\\
\mathbb{E}[f_F] &= \mathbb{E}[R_{IN\leftarrow A} \times \text{BP}_l(A) \times \text{BP}_r(B)]
\end{aligned}$$

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

$$\begin{aligned}\text{P1 :}\mathbb{E}[f_F] &= 1.0 \times 0.80 \times 0.90 = 0.72\\ \text{P2, P3 :}\mathbb{E}[f_F] &= 1.0 \times 0.80 \times 0.10 = 0.08\end{aligned}$$

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

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

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

which is the average of the expected frequencies.

The mean is a special case of marginalization; independence allows the joint distribution to be broken into the product of individual distributions, where the expectation associates over the product, simplifying the calculation to the product of means seen above. Thus, to recover an "absolute" expected execution count from an HN CP, multiply the means of each monitor up the dominator tree to the procedure entry. Then, multiply by the expected invocation frequency of the procedure (possibly using this technique over a CG CP). Denormalization is this process of multiplying monitors along a path in the dominator tree. The mean is a special case of denormalization because it does not require the distribution of monitor values. The general denormalization technique is formally presented in the remainder of this section.

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

The product of two histogram bins is a new bin whose range is the product of the two input bin ranges, and whose weight is the product of the two input bin weights. Instead of dealing with a large number of overlapping, variable-width bins, HN assumes that each histogram bin is a point distribution at the midpoint of that bin. A larger number of narrower bins reduces the modeling error induced by this assumption. Consequently, the product of two bins is a $\langle value, weight\rangle$ pair, where the *value* of the pair is the product of the bin midpoints, and the *weight* is the product of the bin weights. The point-wise product of two $b-$bin histograms, denoted $H_1 \dot{\times} H_2$, uses all pairings of bins between H_1 and H_2 to produce a set of b^2 weighted-value pairs. These pairs are used as the source data for the resulting histogram.

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

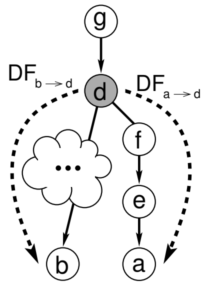


Fig. 3. Denormalization of R_a and R_b with respect to their least-common dominator R_d. Dashed lines show the path over which the marginalized histograms are computed.

Figure 3 shows a dominator tree containing the nodes a and b and their least common dominator d, which is shaded. The dashed lines illustrate the paths followed, in the dominator tree, to compute the denormalization. Nodes f and e are in the path from d to a and there might be other nodes in the path from d to b. Thus, $R_d = dom^3(R_a)$, and the histogram for $\widehat{R_a^{-3}}$ is calculated:

$$\begin{aligned}\text{pairs} &= H_a \dot{\times} H_e \dot{\times} H_f \dot{\times} H_d\\ \widehat{H_a^{-3}} &= \text{new Histogram(pairs)}\end{aligned}$$

B. Statistical Considerations

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

Hierarchical normalization models the correlations between a monitor and its immediate dominator, and assumes independence for other pairings. This assumption allows the size of the model to grow linearly with the number of monitors, and furthermore allows queries to the combined profile to be computed in constant time, or when denormalization is required, in time linear in the length of the path between the involved monitors and their least common dominator.

The inter-run independence assumptions of hierarchically normalized combined profiles are analogous to the intra-run assumption made by the underlying profiling technique. An

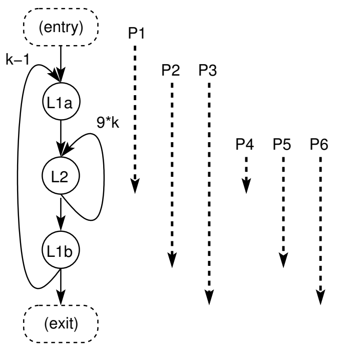


Fig. 4. Some sub-paths through a nested loop. The outer loop L_1 iterates a total of k times; the inner loop L_2 iterates 10 times per iteration of L_1.

edge profile assumes all edge probabilities are independent; a CEP assumes that edge probabilities remain independent across runs. A path profile models the (in)dependence between edges within a path, but assumes that all paths are independent within a run. A *combined path profile* (CEP) maintains the correlations between edges in a path, but extends the assumption of path independence across multiple runs.

C. Extensions to CP

The empirical-distribution methodology of CP is orthogonal to the techniques used to collect raw profiles. CP is applicable whenever multiple profile instances, including intra-run phase-based profiles, hardware performance-counter profiles and sampled profiles. The main issue when combining profiles is how normalization should be done in order to preserve program-behavior characteristics. This section discusses normalization for CFG path profiling and call-graph (CG) profiling. CP can also be used for value profiling by creating, for each frequent value v of a monitor, a histogram of the proportion of observed values accounted for by v [12].

1) CFG Paths: An algorithm that collects path profiling in a program that contains loops must break cycles. The most commonly used technique to break such cycles is due to Ball and Larus [13]. Given a simple loop, the main idea is to replace the back edge with a set of sub-paths that include (a) a path from a point outside the loop to the end of the first iteration, (b) a path from the loop entry point to a point outside the loop, and (c) a path from the entry point to the exit point in the loop. Figure 4 illustrates some of the paths inserted to replace the two back edges in a double-nested loop.

Hierarchical normalization must be adapted to work with paths because there are no dominance relationships between paths. Consider two runs of the double-nested loop of Figure 4, where the outer loop L_1 iterates a total of $k = 10,000$ times in the first run, and $k = 100$ times in the second run. In both cases the inner loop L_2 iterates 10 times per iteration of L_1. The solution is to normalize with respect to the frequency of the node that starts the path. For instance, P4 should be normalized to the frequency of L_2 to factor out k and preserve the constant nature of the inner loop.

2) Program Call-Graphs: Combined profiling can easily be extended to call-graphs (CG)[7]. Profiling a CG gathers information about the frequency of inter-procedural calls. A CG can be represented in multiple ways. For instance, a single edge may represent all calls from a procedure foo() to a procedure bar(). Alternatively, there may be a separate edge for each call-site in foo() that targets bar(). If context-sensitivity is included, there are several alternatives to keep track of the execution path that leads to a call from foo() to bar(). A common solution is to keep track of the k most recent calls on the stack when the call from foo() to bar() occurs [15]. This sequence of calls is called a *call string*.

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

IV. Methodology

Combined profiling is a data representation for profile information collected over multiple runs, and is motivated by the observation that program behavior is input dependent and varies from run to run. Therefore, the metrics in this evaluation assess the variation present in a program workload, and how this variation is stored in a CP. These metrics are used to collect the results presented in Section V and are calculated on a per-monitor basis. The presentation of the results uses probability densities to expose the distribution of values across monitors and thus allow a careful examination of the results.

While LLVM has profiling facilities for both edge and path CFG profiling, it does not currently have any transformations that make use of FDO information. Thus, the performance impact of combined profiles on compiled programs cannot yet be evaluated.

A. Behavior Variation

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

An FDO compiler uses profile information to predict future program behavior. If one behavior is very frequent, it may be sufficient for transformations to only consider this *dominating behavior*. A CP histogram is a probability distribution: the

[7]We do not attempt to extend CP to inter-procedural paths [14].

probability of the monitor taking on a value within the range of a histogram bin is equal to the proportion of the histogram's total weight found in that bin. Thus, the most likely behavior of a monitor can be estimated by finding the bin containing the most weight. The *maximum probability* of a monitor is the proportion of weight in the heaviest bin to the total weight in the histogram. The total weight only includes weight from raw profiles that cover the monitor.

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

Variation of program behaviors is practically relevant only if the variation is significant compared to typical monitor values. The *span* of a histogram is the ratio between it's range and it's maximum value. The lower-bound on the range is the smallest non-zero value in the histogram.

B. Drift

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

The final range, and thus the bin boundaries, of both histograms will be identical because the extreme values in the data are fixed. The difference in weight between corresponding bins is due to drift. Summing these differences for all pairs of bins double-counts total drift: drift is half this sum, reported as a proportion of total histogram weight. This study reports drift using the merged results of 5 different randomly-selected incremental combination orders.

V. Results

Experiments were conducted on a machine running Red Hat 4.1.2-42 on a Quad-Core AMD Opteron^{TM} Processor 2350. The experiments use C integer benchmarks from SPEC CPU 2006 with all inputs provided by SPEC. When the SPEC 'ref' run uses multiple inputs, each of these inputs is treated separately. Additional inputs for `mcf` are taken from Berube [1]. `Bzip2` uses the 1000-input workload from kDataSets [16]. Combined profiles are created using 10, 20, 30, 40, and 50 histogram bins. We present detailed results for 50 bins only due to space constraints, but briefly discuss the impact of the number of bins in Section V-D. All combined edge profiles employ HN. Experiments are performed with both edge and path profiling. The figures for path profiling are very similar to their edge profiling counterparts, and are omitted due to space constraints.

Table I summarizes the characteristics of the combined profiles for each benchmark. *Runs* indicates the number of program inputs in the workload. The *Edges* and *Paths* columns list the number of unique monitors executed at least once across all runs. All results exclude unexecuted monitors.

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

The columns labeled *Points* list the proportion of monitors that are *Point distributions* in the CP. Points arise when all non-zero values for a monitor are equal: all the probability mass occurs at a single point on the real number line. In these cases, no histogram bins are required to represent the monitor in the CP. Point histograms at 1.0 are uninteresting because they indicate that the monitor always executes the same number of times as it's dominator. For example, in Figure 2, the F$\rightarrow$E edge would have a point distribution at 1 because it is immediately dominated by, and control-flow equivalent to, B$\rightarrow$F, and thus must always have an HN frequency of 1. For edges, most of these monitors are likely statically redundant and could be removed from the CP to reduce file size with no loss of information. For paths, these point histograms identify paths that execute exactly once each time their procedure executes. In most cases, such a path is the only non-loop path executed in a function. Point distributions at values other than 1 are more interesting. These points arise is cases such as (dynamically) constant loop trip counts or branch probabilities. CP allows a compiler to evaluate potential code transformations involving these monitors with confidence that the analysis is applicable to all program runs.

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

A. Behavior Variation

Figure 5 shows the distribution of monitor coverage across the workload, excluding fully-covered monitors. The coverage value is normalized to the number of runs. For example, 670 of the 2182 executed edges in `bzip2` are not executed in every run. On average, those edges are executed by about 65% of the 1000 runs. However, the small bulge at the bottom of the plot

TABLE I
CHARACTERISTICS FOR BATCH-COMBINED PROFILES, INDEPENDENT OF THE NUMBER OF HISTOGRAM BINS.

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

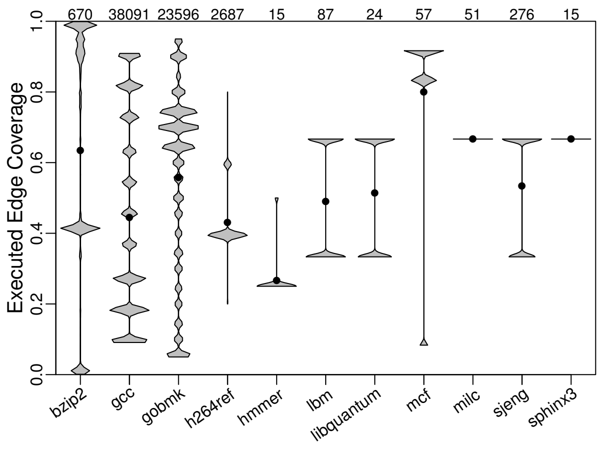


Fig. 5. Edge coverage, excluding fully-covered monitors

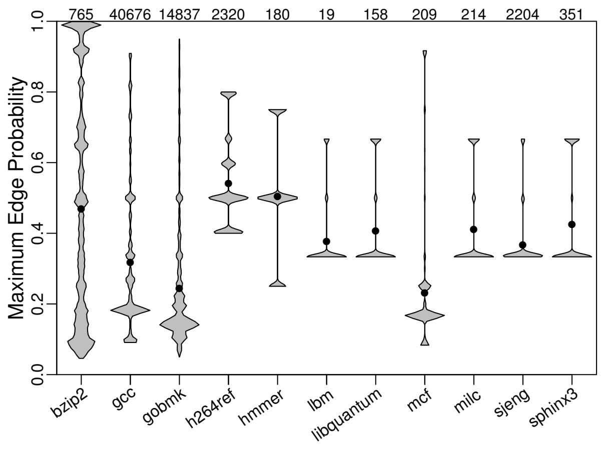


Fig. 6. Maximum edge likelihood with 50 bins (no points)

indicates that several edges are covered by very few runs; a large group of edges are covered by slightly more than 40% of the runs, and another group of edges are covered by more than 85% of the runs. An FDO compiler would be oblivious to the execution of any or all of these monitors using a single-input profile, and may consequently make suboptimal decisions from a whole-workload perspective.

`Hmmer`, `libquantum`, `milc`, `sjeng`, and `sphinx3` have more than 90% of their executed edges covered by every run, which may be due to a lack of diversity in their very small workloads. At least 30% of the edges in the other benchmarks are not executed in every run, up to 79% for `gobmk`. Furthermore, the distribution of coverage for these benchmarks shows that the number of runs that do not executed some edge is spread across the range, indicating that these differences in coverage are unlikely due to a small number of large-scale control-flow alternatives. Particularly for `gcc` and `gobmk`, the set of executed edges varies significantly from run to run. In contrast, for both `milc` and `sphinx` the 'ref' and 'train' inputs cover identical sets of edges, while the 'test' input misses a handful of edges, producing their distinctive "point violins" at 66%.

Compilers predict program behavior from profiles. In the case of point histograms, this prediction can be made correctly from any non-zero sample. The prediction is more complicated when behavior varies from run to run. However, if the behavior is consistent for most runs, then perhaps the most frequently observed behavior is a good predictor. Unfortunately, this is seldom the case. Figure 6 shows the proportion of histogram weight that occurs in the heaviest bin, *i.e.*, the probability of the most likely behavior. For the four benchmarks with more than 10 runs, this probability tends to be low: there is no dominant behavior for these monitors. No single run, and no point statistic, is a good representative of such monitors. A distribution model is needed to evaluate transformations involving these monitors, as discussed in Section II-C. Redundancy in `bzip2`'s very large workload allows for dominant behaviors in some monitors, as exemplified by the bulge near 100%.

Figure 7 presents bin occupancy: a histogram with weight in many bins increases the violin width toward the top of the figure. The maximum number of occupied bins is listed at the bottom of the figure, which is, with the exception of `bzip2`, the number of runs. The average proportion of bins used is over 50%, indicating that when variation is present, monitor values are not limited to a small number of possibilities. The maximum probabilities discussed above indicate that, in most

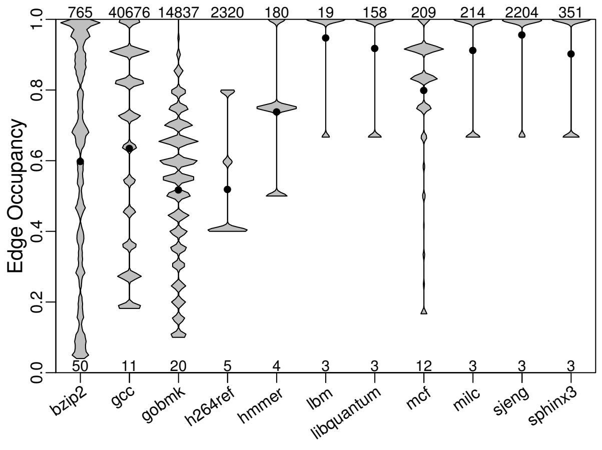


Fig. 7. 50-bin histogram occupancy for edges (no points)

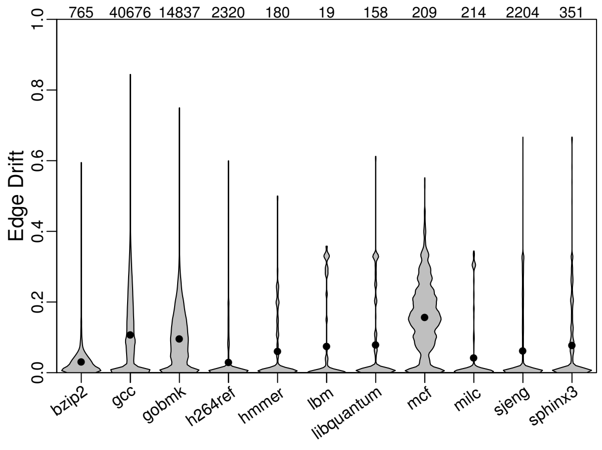


Fig. 9. Edge weight drift using 50-bin histograms (no points)

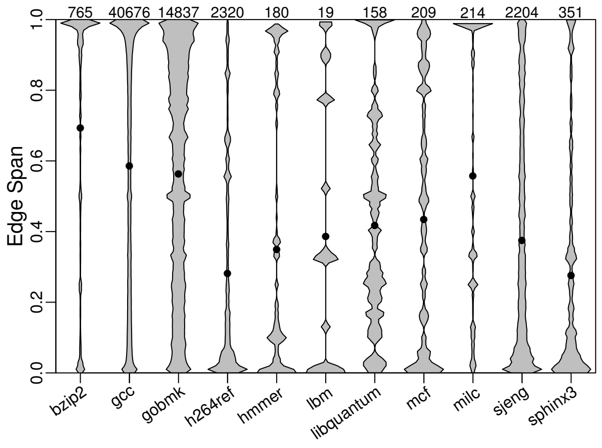


Fig. 8. Span of edge histograms (no points)

cases, none of the bins contain a majority of the histogram weight. Consequently, the weight must be distributed across many bins. Visual investigation of the individual histograms for `bzip2` reveals that no single simple parametric model (*e.g.*, uniform, normal) matches the shape of a majority of the histograms [12]. In contrast, CP's histograms match the shape of the data automatically.

Figure 8 presents histogram span, the ratio between range and maximum value. Recall that monitors use HN to keep behavior variation local to the monitor where it occurs. The distribution of monitor values in a histogram may not have practical significance if the span is small. Therefore, practically relevant behavior variation should widen the violin plot toward the top of the figure. Figure 8 suggests that all the benchmarks contain monitors that exhibit practically significant behavior variation across the workloads. An FDO compiler must take this variation into account when proposing code transformations by, for example, calculating expected benefit for the worst case or a low-quantile point as well as the average, weighting by coverage, or considering span in sorting functions.

B. Drift

Drift is due to histogram ranges growing during incremental construction. For each benchmark, five randomly-selected combination orders instruct the incremental combination of all raw profiles. Drift is maximized by combining profiles one at a time. Drift is calculated between these CPs and a batch-combined profile and presented in Figure 9. The figure merges all five comparison results. A drift of 0 indicates that the batch and incremental histograms are identical, while a value of 1 is possible only when the weight in the two histograms does not overlap at all.

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

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

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

C. Space Requirements

Edge profiles grow linearly with the number of edges in a program; path profiles grow linearly with the number of executed paths. In raw profiles, each monitor is represented

TABLE II
PROFILE FILE SIZES, IN KB, FOR 50 BINS. THE OHEAD COLUMNS GIVE THE OVERHEAD FACTOR FOR THE CP VS. THE COLLECTED RAW PROFILES.

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

by a 4-byte counter. Like combined profiles, path profiles only store executed monitors, but add a 4-byte index. In a CP, each monitor maintains the true mean and variance of all samples along with the histograms; an entry for a single monitor is 45 bytes[8], plus a 1-byte bin index and an 8-byte weight per non-empty histogram bin. Thus, even point distributions requires 11x (edge) or 5.5x (path) more space than the same monitor in a raw profile. Table II presents profile files sizes. Raw edge profiles and batch-combined profiles always have the same size. The sizes of raw path profiles and increment CPs are taken as the largest file across all runs or combination orders, respectively. Comparing batch and incremental combination, the drift observed in Figure 9 causes more bins to be non-empty, resulting in larger files. This effect is most visible for the benchmarks with several runs: `gcc`, `mcf`, `gobmk`, and `bzip2`. Likewise, these benchmarks illustrate that the file size grows slowly: as more profiles are combined, it becomes less likely that an additional profile will place weight in an empty histogram bin. `Bzip2` illustrates how CP can dramatically reduce storage requirements for profiles as the total number of profiles becomes large, *e.g.*, systems using continuous profiling.

D. Number of Bins

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

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

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

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

VI. RELATED WORK

Many attempts to model program behavior have been made. Jiang *et al.* use *seminal behaviors*, program behaviors that are highly correlated with subsequent behaviors, to predict behaviors at runtime [17]. This methodology is complementary to CP because CP's general model can be used when it is infeasible to model behavior correlations. Tian *et al.* exploit seminal behaviors in C programs to select between function versions at runtime. Each version is created using a standard single-input FDO compilation [18]. While many runs achieve impressive speedups, worst-case performance suffers by over 5% on average. Traditional FDO produces similar results. These results highlight the fact that FDO performance is sensitive to program inputs and can vary widely across the workload. Furthermore, optimization based on one input frequently reduces performance for some other inputs. Salverda *et al.* model the critical paths of a program by generating synthetic program traces from a histogram of profiled branch outcomes [19]. To better cover the program's footprint, they do an ad-hoc combination of profiles from SPEC training and reference inputs. In contrast, CP is a sound methodology to combine an arbitrary number of profiles. Savari and Young build a branch and decision model for branch data [20]. Their model assumes that the next branch and it's outcome are independent of previous branches, an assumption that is

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

violated by computer programs (*e.g.*, correlated branches). One distribution is used to represent *all events* from a run; distributions from multiple runs are combined using relative entropy — a sophisticated way to find the weights for a weighted geometric average across runs. In contrast, CP's empirical distributions are for a *single event* across multiple program runs.

Multiple inputs are used in attempts to scale input sizes (up or down). Bienia *et al.* focuses on micro-architectural features to scale the input sizes of PARSEC benchmarks [2]. When a reduced input has a large error in comparison with a reference input, they regard the reference input as "correct." In contrast CP considers input-dependent behaviors to be intrinsic to programs and captures them in a distribution over runs. Kim *et al.* compare the simulated dynamic branch prediction accuracy of FDO on the diverge-merge processor using the MinneSPEC reduced program inputs against the same benchmarks using the SPEC training inputs [21]. They find that FDO is not sensitive to program input. Their evaluation is not sound because comparing program behavior between a reference input and an input that was specifically selected by experts on the criteria that it be representative of the reference is unlikely to predict the actual variations between inputs encountered after deployment.

An early attempt to combine profiles is due to Fisher and Freudenberger. They measure instructions per break in control flow and sum profiles to provide better branch prediction [22]. Such summations produce similar results to summing normalized frequencies. While better than single-run profiles, they still yield poor behavior modeling in the presence of multiple program use cases and poor training input selection, two issues addressed by CP. A work-in-progress description of the CP methodology introduced in detail the histogram construction techniques used by CP and the types of queries CP can answer [11]. This work expands the discussion of CP and HN, introduces the monitor-level metrics of Section IV and provides the evaluation of CP in Section V.

VII. Conclusion

Combined profiling is a practical and statistically sound methodology to model behavior variation across multiple data inputs. The experimental evaluation of CP shows that behavior variation is present both in simple programs such as `bzip` and in programs with more complex control flow like `gcc` and `gobmk`. This variation can be captured and queried by the CP statistical model. The prototype of CP for both edge and path profiling in `LLVM` is a functional implementation of CP. We look forward to introducing CP-directed transformations into `LLVM`.

An extensive study of CP requires adequate workloads for benchmark programs. The publication of the CP methodology should encourage organizations that publish benchmarks to include a significant number of inputs to each benchmark, thus improving future use and evaluation of FDO code transformations.

References

[1] P. Berube and J. N. Amaral, "*Aestimo*: a feedback-directed optimization evaluation tool," in *Intern. Symp. on Performance Analysis of Systems and Software (ISPASS)*, Austin, Texas, March 2006, pp. 251 – 260.

[2] C. Bienia and K. Li, "Scaling of the PARSEC benchmark inputs," in *Parallel Architectures and Compilation Techniques (PACT)*. New York, NY, USA: ACM, 2010, pp. 561–562.

[3] D. Gove and L. Spracklen, "Evaluating the correspondence between training and reference workloads in SPEC CPU2006," *Computer Architecture News*, vol. 35, no. 1, pp. 122–129, 2007.

[4] D. W. Wall, "Predicting program behavior using real or estimated profiles," in *Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Ontario, Canada, 1991, pp. 59–70.

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

[6] 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 *Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, USA, 1997, pp. 273–286.

[7] R. Gupta, D. A. Berson, and J. Z. Fang, "Path profile guided partial redundancy elimination using speculation," in *International Conference on Computer Languages*, 1998, pp. 230–239.

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

[9] 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)*, Paris, France, December 1996, pp. 58–67.

[10] D. R. Chakrabarti and S.-M. Liu, "Inline analysis: Beyond selection heuristics," in *Code Generation and Optimization (CGO)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 221–232.

[11] P. Berube, A. Preuss, and J. N. Amaral, "Combined profiling: practical collection of feedback information for code optimization," in *Intern. Conf. on Performance Engineering (ICPE)*. New York, NY, USA: ACM, 2011, pp. 493–498, work-In-Progress Session.

[12] P. Berube, A. Preuss, and J. N. Amaral, "Extended description of the combined profiling methodology," University of Alberta, Edmonton, AB, Canada, Tech. Rep., March 2011.

[13] T. Ball and J. R. Larus, "Efficient path profiling," in *Intern. Symposium on Microarchitecture (MICRO)*, Paris, France, 1996, pp. 46–57.

[14] D. G. Melski, "Interprocedural path profiling and the interprocedural express-lane transformation," Ph.D. dissertation, University of Wisconsin, 2002.

[15] M. Might, Y. Smaragdakis, and D. Van Horn, "Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis," in *Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Ontario, Canada, 2010, pp. 305–315.

[16] H. Y. Y. Chen, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu, "Evaluating iterative optimization across 1000 data sets," in *Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Canada, June 2010, pp. 448–459.

[17] Y. Jiang, E. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao, "Exploiting statistical correlations for proactive prediction of program behaviors," in *Code Generation and Optimization (CGO)*. New York, NY, USA: ACM, 2010, pp. 248–256.

[18] K. Tian, Y. Jiang, E. Z. Zhang, and X. Shen, "An input-centric paradigm for program dynamic optimizations." New York, NY, USA: ACM, 2010, pp. 125–139.

[19] P. Salverda, C. Tuker, and C. Zilles, "Accurate critical path prediction via random trace construction," in *Code Generation and Optimization (CGO)*, Boston, MA, USA, 2008, pp. 64–73.

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

[21] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt, "Profile-assisted compiler support for dynamic predication in diverge-merge processors," in *Code Generation and Optimization (CGO)*, San Jose, CA, USA, 2007, pp. 367–378.

[22] 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)*, 1992, pp. 85–95.