Simple Profile Rectifications Go A Long Way

—Statistically Exploring and Alleviating the Effects of Sampling Errors for Program Optimizations

Bo Wu[1], Mingzhou Zhou[1], Xipeng Shen[1],
Yaoqing Gao[2], Raul Silvera[2], and Graham Yiu[2]

[1] Computer Science Department, The College of William and Mary, VA, USA
[2] IBM Toronto Lab, Toronto, Canada

Abstract. Feedback-driven program optimization (FDO) is common in modern compilers, including Just-In-Time compilers increasingly adopted for object-oriented or scripting languages. This paper describes a systematic study in understanding and alleviating the effects of sampling errors on the usefulness of the obtained profiles for FDO. Taking a statistical approach, it offers a series of counter-intuitive findings, and identifies two kinds of profile errors that affect FDO critically, namely zero-count errors and inconsistency errors. It further proposes statistical profile rectification, a simple approach to correcting profiling errors by leveraging statistical patterns in a profile. Experiments show that the simple approach enhances the effectiveness of sampled profile-based FDO dramatically, increasing the average FDO speedup from 1.16X to 1.3X, around 92% of what full profiles can yield.

1 Introduction

Feedback-Driven Optimization (FDO) is a technique modern compilers use to optimize programs based on some profiles of the program dynamic behaviors, such as the frequencies of basic blocks and function invocations. FDO has proven effective for improving program performance. It has been part of most commercial compilers (e.g., IBM XLC, Intel ICC) of traditional imperative languages. It is also essential for modern languages with a managed environment (e.g., Java, Javascript, Python.) The runtime engines of these languages nowadays commonly employ Just-In-Time (JIT) compilers to complement interpreters for producing code with a good performance. Most optimizations by the JIT compilers are FDO: They make optimization decisions based on some profiles the runtime engine collects throughout the current execution. With JIT compilation becoming popular for modern languages, the effectiveness of FDO is becoming increasingly important for the efficiency of modern computing.

The profiles used for FDO are often collected through sampling, because collecting full profiles requires some detailed instrumentations and hence incurs lots of overhead. It is especially true for JIT-based languages, as for them, profile collections happen usually on the fly.

Sampling unavoidably introduces some inaccuracy into the collected profiles, which may in turn impairs the effectiveness of FDO. There have been some studies on reducing biases in the sampling schemes used in Java runtime systems [16], and some proposals of improved sampling schemes—such as bursty sampling [5, 6, 11, 15]. Although these techniques can improve the profiles quality in a certain degree, the speedup by FDO on the sampled profiles still has a substantial gap from what it produces on full profiles, as Figure 1 shows.

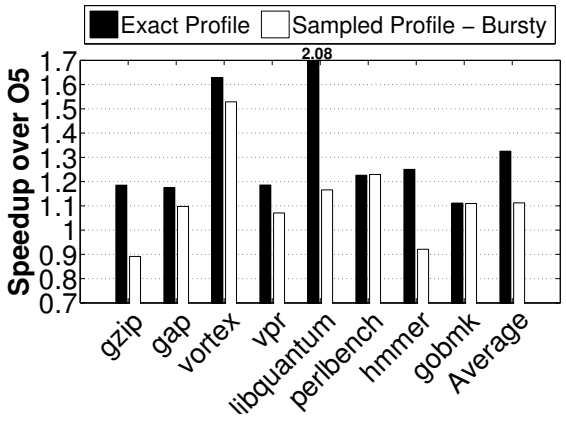

Fig. 1. The speedups (compared to static compilation with the highest-level optimizations enabled) produced by the FDO of IBM XLC (v12.1) compiler, when sampled (bursty at rate of 5%) and full profiles are used respectively.

In this work, we attack the problem from a different angle: Rather than refining sampling schemes, we try to rectify the errors in a profile after it is collected. It is called *profile rectification*.

For profile rectification to work, we must answer some fundamental questions: What are the relations between sampling rates, accuracy of the collected profile, and its usefulness for FDO? How do the errors in a profile influence the optimizations? And how to rectify the critical errors? Answers to these questions are essential for guiding the directions of profile rectification. But to the best of our knowledge, none of these questions have been systematically studied on modern compilers and programming systems.

Prompted by these open questions, we conduct two investigations in this work. First, as Section 2 will show, we design a set of systematic measurements to reveal the statistical correlations among sampling rates, profile accuracy, and the corresponding FDO benefits. To avoid biases in the analysis, we conduct 7680 runs, which cover seven most important factors in four levels, including the usage of two mature compilers, two sampling methods with six sampling rates for each, two platforms, eight SPEC benchmarks with some non-trivial FDO potential, four inputs per benchmark, and ten repetitions for each setting.

The systematic measurements reveal some counter-intuitive observations. It is commonly perceived that a higher sampling rate tends to give more accurate profiles, which in turn would help FDO produce code that has a better performance. However, the measurement suggests that even though a higher bursty sampling rate gives a more accurate profile in general, the perception does not hold for uniform sampling, which samples once in a fixed time period[3]. Moreover, the results show that for both types of sampling methods, when a more accurate profile is given to the FDO, it often fails to produce code with a better performance. In other words, in the sampling rate range, profile accuracy does *not* have an apparent correlation with the FDO speedups.

The surprising observations prompt our two-fold investigation (Section 3). We conduct a deep analysis of the influence cast on FDO by various types of sampling errors. We identify two types of errors that have some important influence. The first is *zero-count errors*, which refer to the case when a counter in sampled profile equals zero but its value in the full profile is not. The second is *inconsistency errors*, which refer to the case when two counters in the same function have different values in the sampled profile but have the same value in the full profile. Our analysis shows that although these two types of errors do not affect the overall profile accuracy much, they alter the behaviors of the FDO dramatically. Based on the findings, we propose to rectify the two types of errors through exploitation of the statistical patterns in profile counters derived from some training runs. The rectification, although being simple, turns out to boost the usefulness of the sampled profiles for FDO significantly, increasing the average speedup from 1.16X to 1.3X, around 92% of what full profiles can yield.

There are two prior studies on correcting sampling errors in a profile for FDO [10, 14]. They both apply a minimum cost circulation algorithm to the control flow graphs of a program, hence subject to the static constraints in the program (detailed in Section 4). Our rectification method is distinct in exploiting *statistical* patterns shown in *dynamic* profiles. The exploited dynamic patterns are essential for correcting the two kinds of critical errors.

In summary, this work makes three main contributions.

- **Correlations** It provides the first study in modern systems that systematically uncovers the correlations among sampling rates, profile accuracy, and the usefulness for FDO. Through a comprehensive measurement using contemporary sampling techniques and compilers, the study produces some findings contrary to common perceptions.
- **Influence of Errors** It offers a set of novel insights on sampling and its influence on FDO:
  - The zero-count and inconsistency errors in sample profiles hurt FDO substantially, despite their modest influence on the overall profile accuracy.
  - Uniform sampling not only underperforms bursty sampling, but shows a weak correlation between sampling rate and profile accuracy, which

[3] The "time" here could be wall-clock time or logical time (e.g., a number of instructions or basic blocks).

reinforces the superiority of bursty sampling over uniform sampling for FDO.
  - Commonly defined accuracy, either weighted or unweighted, fails to quantify the actual quality of a profile for FDO.
- **Statistical Profile Rectification** To our best knowledge, this is the first work showing that simple statistical profile rectification can dramatically enhance the usefulness of a profile for FDO.

The rest of the paper is organized as follows. In Section 2, we first briefly introduce the two FDO compilation systems, the two sampling methods, and some other experiment settings. Then we present the findings of the correlations. In Section 3, we present a deep analysis of the sampling errors, and describe the simple profile rectification method. We discuss some related work in Section 4 and then conclude the paper with a summary.

2 Counter-Intuitive Correlations

This section starts with an introduction to the FDO in the two compilers we use. It then presents the design of the empirical measurements and reports the findings on the correlations among sampling, profiles, and their usefulness.

2.1 Background on FDO

FDO is part of many modern compilers. The implementations of FDO in different compilers may differ in what set of optimizations they contain, but mostly follow a similar high-level design. We briefly describe the way FDO works in a mature commercial compiler, IBM XLC, as follows.

To enable FDO, two stages of compilations are necessary. For XLC, in the first stage, the compiler must be invoked with a special option ("-qpdf1"). With that option, the compiler instruments the program with some monitoring instructions. An execution of the generated executable will produce a profile of that execution. In the second stage, the compiler is invoked again with another special option ("-qpdf2"). In this round of compilation, the compiler enables FDO, which reads the profile and produces an optimized executable.

JIT follows a similar two-stage scheme. The main difference from XLC-like offline compilers is just that JIT invokes the two rounds of compilation implicitly at runtime. For instance, in a Java virtual machine JikesRVM [1], the first-time compilation of a newly loaded Java method inserts some *yielding points* into the generated code, which help to sample runtime behaviors of the method in its execution. If the Java runtime decides to recompile that method later in that run, its JIT compiler will use the sampled profile to do a FDO on that method.

As the profiles capture some runtime behaviors (e.g., the hotness of a function or basic block), they can provide the optimizers some hints that static code analysis is unable to provide. FDO heavily exploits those hints to enhance code layout, inline functions, and so on. An inaccurate profile may hence mislead FDO into making wrong optimization decisions.

2.2 Experimental Design

We design a set of experiments to empirically measure the relations among sampling rate, profile accuracy, and the influence on the effectiveness of FDO. In the design, we carefully cover seven dimensions that are closely relevant to the relations to minimize the bias in the measurement. They fall into four levels. Table 1 shows the dimensions, the number of optional values in each dimension (the "Variations" column) and a brief description of the optional values. We explain these dimensions in more details as follows.

Table 1. Dimensions covered in the experiment design

Levels	Dimensions	Variations	Description
workload	benchmarks	8	from SPEC CPU2000 and CPU2006
	inputs	4	*1 train input, 3 ref inputs*
system	compilers	2	XLC, GCC
	platforms	2	Intel Xeon & IBM POWER7
sampling	methods	2	Bursty, Uniform
	frequencies	12	6 for Bursty, 6 for Uniform
noise avoidance	repetitions	10	number of repetitive runs per setting

Benchmarks and Inputs Given that the focus of this work is on FDO, when choosing benchmarks, we concentrate on those that exhibit some non-trivial speedups when FDO is applied. Meanwhile, our current infrastructure works on C programs only. Among the programs in SPEC CPU2000 and CPU2006 [2], we find eight of them meeting both criteria, as listed in Table 2. All these benchmarks are integer programs, and have complex control flows and a large number of functions, posing challenges for static analysis and hence exhibiting good potential for FDO. As Figure 1 shows, these benchmarks show an average 1.33X speedup when FDO is applied (on the exact profiles of the execution inputs) compared to their performance through static compilations using XLC.

For each program, besides including both its *train* and *ref* inputs coming by default with the benchmark suite, when necessary, we collect or create two extra representative inputs by searching for the real usage of their original applications or reading the source code. The extra inputs are used in the experiments described in Section 3 for examining the stableness of profile value patterns across different inputs. For FDO, the profiles are collected on the train input and evaluated on the ref input.

Compilers For compilers, there are two possible choices: some mature highly polished offline compilers, or some JIT compilers. We choose the former for the

Table 2. Benchmarks and FDO speedup from exact profiles over O5 compilation

Program	Benchmark Suite	Description	FDO speedup
gzip	CPU2000	Compression	1.19X
gap	CPU2000	Group Theory, Interpreter	1.18X
vortex	CPU2000	Object-oriented Database	1.63X
vpr	CPU2000	FPGA Circuit Placement and Routing	1.19X
libquantum	CPU2006	Quantum Computing	2.09X
perlbench	CPU2006	Perl Interpreter	1.22X
hmmer	CPU2006	Search Gene Sequence	1.25X
gobmk	CPU2006	Artificial Intelligence	1.11X

following reasons. First, as Section 2.1 mentions, the FDO in JIT compilers is triggered by some decisions made by the runtime engine and is hard to control and hence experiment with. Using it would add more noise into the measurement. Second, JIT compilers are not as mature as offline compilers. The FDO in offline compilers has been developed for decades and has reached a quite stable state. Compared to JIT, the implementation in offline compilers usually tap into the potential of FDO to a much higher degree, because for the tradeoff between runtime overhead and code quality, what optimizations JIT compilers should include is still an active research topic yet to settle. For all these reasons, using current JIT compilers is hard to uncover the principled relations between profile accuracy and FDO.

We select the recent versions of XLC (v12.1) and GCC (v4.6.2) as our compilers. Both compilers have been developed for more than a decade. The former is the main commercial compiler of IBM for C and C++, and shares the core with many other IBM compilers for other languages. Its FDO is sophisticatedly polished by a large compiler team for many years, able to exploit profiles to conduct a number of advanced intra-procedure and inter-procedure optimizations. GCC is a result of the many years of efforts by the open-source community. Its performance has been shown to get close to commercial compilers in many cases. Its FDO component has also been developed for quite a while. We use both compilers for this study to examine the influence of different FDO implementations on the studied relations.

In the instrumentation stage of XLC, each procedure's control flow graph is explored to find out some straight lines of basic blocks that must have the same counts. Only the first basic block of a straight line needs to be instrumented to collect access frequency. A mapping data structure records which functions are invoked in which basic blocks based on static call graphs. During recompilation, function calling frequencies and control flow branch probabilities are inferred from basic block counters and the mapping, serving as hints for the FDO. A similar implementation scheme is shown in GCC, although differences exist in the set of optimizations they include and how those optimizations are implemented.

Platforms We run XLC-related experiments on an IBM Power7 machine, which has the AIX 7 operating system installed. We conduct the GCC-related experiments on a machine equipped with Intel Xeon W3550, running a OpenSUSE Linux, version 12.1.

Sampling Methods and Frequencies We experiment with two sampling methods. The first is uniform sampling, which is the most commonly used sampling method. It tries to get a sample after a given time interval. For example, when being applied to collect basic block frequency profiles, the runtime sampler checks which instruction is being executed and finds out which basic block that instruction belongs to after a given time interval; it then increases the counter corresponding to the basic block by one. The second sampling method is bursty sampling. In this method, there are two predefined parameters, the *execution period length* τ_e and the *profiling period length* τ_p. The runtime switches between normal execution and profiled execution periodically. During an execution, after a τ_e-long period of normal execution, the runtime switches the execution to a fully instrumented version and runs that version for a τ_p-long period of time to collect some profiles, and then switches back to normal execution. The back-and-forth switching continues throughout the program execution.

Since we use static compilers, which do not have a runtime sampling system, we simulate the two sampling methods. For uniform sampling, we assume that each instruction's execution takes equal time and thus has the same probability to be sampled. The full profiles are processed to obtain the sample profiles. For bursty sampling, we modify the instrumentation, so one execution directly produces one sample profile.

Previous studies have shown that the bursty sampling, although being more complicated to implement, can often produce a more accurate profile than the uniform sampling does at the same sampling rate [5]. Bursty sampling has been implemented in some runtime systems, such as Jikes RVM [6]. Using both sampling methods helps us examine the influence of different sampling schemes on the relations between profile accuracy and FDO.

We experiment with six sampling rates for each of the sampling methods. These rates subsume the typical range of sampling rates used in practical systems. The sampling rate of uniform sampling is determined by a single parameter, the sampling period length. Because the bursty sampling has two parameters, the execution period length τ_e and profiling period length τ_p, each of its sampling rates is represented with the ratio of a pair, $\tau_p/(\tau_e+\tau_p)$. Specifically, for uniform sampling, the used sampling rates are $100, 1000, 10,000, 50,000, 100,000, 500,000$; for bursty sampling, the rates are $1/1000, 10/1000, 50/1000, 100/1000, 200/1000$, $400/1000$.

Time Measurement In all runs, the highest optimization level is enabled. We see some minor fluctuations in the execution times of multiple runs in the same setting. But still to minimize the influence of random noise, we repeat each run for 10 times and use their average time for comparison.

2.3 Measurements and Findings

The coverage of the various factors leads to 7680 runs in total. This subsection presents the findings we have obtained from these measurements. But before that, we first explain some metrics we use to quantify profile accuracy and correlations.

Accuracy Metrics Let B_i represent the exact profile (or called full profile) of a run on input i. Exact profiles can be obtained through a full profiling. Let SP'_i be the profile obtained by sampling. Before comparing the two profiles, we multiply each counter in SP'_i by the ratio between the sum of the counters in B_i and that in SP'_i so that the two profiles are at the same scale for comparison. We denote the scaled sampled profile with SP_i. We use $SP_i[j]$ and $B_i[j]$ to denote the counter values of the jth item (e.g., basic block frequency counter) in the sampled and exact profiles, respectively. For the purpose of explanation, we use basic block frequency profiles as our example in the following discussion. In such a profile, each item corresponds to the frequency of a basic block being accessed.

The definition of the accuracy should quantify the similarity between SP_i and B_i. Following common practices, we define the *accuracy* (Acc) of a basic block counter as follows:

$$Acc_i[j] = 1 - \frac{|SP_i[j] - B_i[j]|}{max(SP_i[j], B_i[j])}$$

The use of *max* in the denominator is to normalize the accuracy to the range of [0, 1]. We use two definitions for the overall accuracy of a profile. An *unweighted accuracy* (UAcc) is just an arithmetic average of all basic blocks' accuracies. It treats each basic block equally. A *weighted accuracy* (*WAcc*) of a profile is a weighted average as follows:

$$WAcc_i = \sum_j Acc_i[j] \times \frac{B_i[j]}{\sum_j B_i[j]}$$

where, the weights are proportional to the significance of a basic block in the program in terms of its access frequency.

Correlation Metrics Among the different variations of commonly used correlations metrics, we find the *Spearman's rank correlation coefficient* (called *rank coefficient* in short) suiting our needs. Let X and Y represent two ordered lists of values. For instance, in our experiment, we set X to be the list of accuracies of a number of sampled profiles, and Y be the list of speedups brought by FDO based on those profiles. Elements in both X and Y are sorted in an ascending order of their values. The position of an element in the ordered list is called the *rank* of that element. The rank coefficient is defined as follows:

$$r = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}$$

,
where, $x[i]$ and $y[i]$ are the ranks of $X[i]$ and $Y[i]$ in X and Y respectively.

For example, consider the following case. We have four sampled profiles of a program. Their accuracies are shown as the X column and their FDO speedups as the Y column in Table 3. The ranks of each profile in the two lists are shown in the two rightmost columns. In the definition of the correlation metric, x_i and y_i represent the ranks; when $i = 2$, they equal 1 and 4 respectively. The symbols, $\bar{x}$ and $\bar{y}$ in the rank coefficient formular, represent the average of the ranks in X and Y. They both equal 2.5 in this example. The rank coefficient between X and Y is -0.4.

Table 3. Example for illustration of rank coefficient

profile-ID	X	Y	X's rank	Y's rank
1	0.89	1.1	3	3
2	0.93	1.08	1	4
3	0.90	1.2	2	1
4	0.86	1.15	4	2

Recall that the questions we try to answer are whether a higher sampling rate leads to a more accurate profiles and hence more benefits from FDO. The rank coefficient fits our needs as it assesses how well the relationship of two variables fits in a monotonic function. In comparison, the standard Pearson coefficient measures whether two variables form a linear relation, which is a property unnecessarily stronger than what we need.

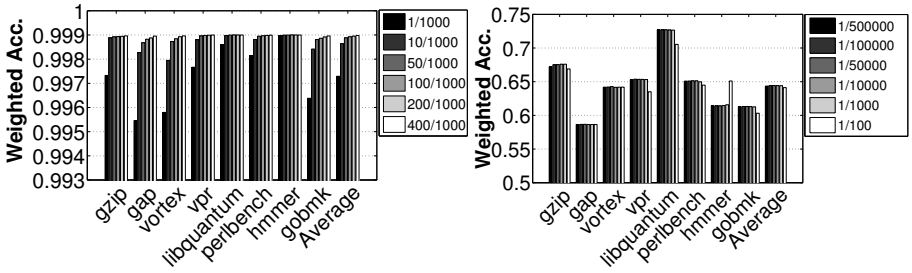

Fig. 2. (a) Weighted accuracy of bursty sampling. (b) Weighted accuracy of uniform sampling.

The value of a rank coefficient is always between -1 and 1, with a value close to 1 implying a strong co-increasing relation between X and Y, and a value close to -1 implying that the two variables' values are taking an opposite trend.

Sampling Rate and Profile Accuracy Figures 2 and 3 report the weighted and unweighted accuracies of the sampled profiles of all benchmarks when different sampling rates are used. The profiles for the bursty sampling have a close-to-perfect weighted accuracy across all sampling rates, while the profiles for the uniform sampling have an average 64% accuracy. The intuition behind the large accuracy disparity is that because each time the uniform sampling checks only one instruction, a larger basic block gets some larger chance to be sampled than a smaller basic block does if the two blocks actually have the same frequencies of being executed. The issue is less serious in bursty sampling. For bursty sampling, block size may cast some influence on which block the sampling period starts from, but the influence is much weaker to the overall accuracy because within a sampling period, the size of a basic block less affects the chance for it to get sampled. These results echo some previous observations on the two sampling methods [5, 11]. The unweighted accuracy difference is smaller, but bursty sampling still outperforms uniform sampling in general.

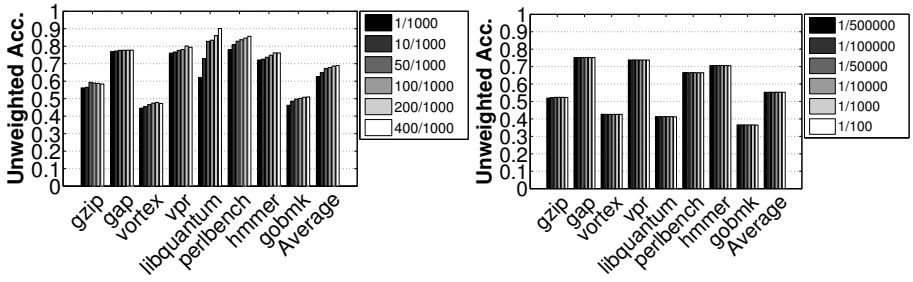

Fig. 3. (a) Unweighted accuracy of bursty sampling. (b) Unweighted accuracy of uniform sampling.

Table 4 provides the rank coefficients between sampling rate and profile accuracies. When weighted accuracy is used, the coefficients are all 1 for bursty sampling, indicating the very strong correlation between sampling frequency and profile accuracy. In other words, the profile accuracy will definitely increase when we use a higher sampling rate. When unweighted accuracy is used, the correlations are slightly lower, but still close to one for most programs.

Uniform sampling shows much weaker correlations with profile accuracy. The average of the rank coefficient is only -0.22 when weighted accuracy is used. Two programs, *hmmer* and *libquantum*, are exceptions. For *hmmer*, a higher sampling rate leads to more accurate profiles, while for *libquantum*, the trend is the opposite. Overall, a higher rate of uniform sampling does not lead to a more accurate profile. The reason for the weak correlations comes from the same source (the effects of basic block sizes) for the low profile accuracy mentioned earlier in this section. To further understand the severity of the effects, we extend the sampling rate to some large values (20%, 40%, 80%) that are rarely used in

actual runtime sampling. The results show that even at such a level of sampling rates, the correlations are no much stronger than what Table 4 has shown.

Table 4. Rank correlation coefficients between sampling frequency and profile accuracy

Program	Weighted Bursty	Unweighted Bursty	Weighted Uniform	Unweighted Uniform
gzip	1	0.43	0.09	0.66
gap	1	1	-0.6	0.94
vortex	1	0.83	-0.03	-0.94
vpr	1	0.94	-0.43	-0.37
libquantum	1	1	-0.94	0.086
perlbench	1	1	-0.49	-0.6
hmmer	1	0.94	1	-0.66
gobmk	1	1	-0.37	-0.43
Median	1	0.97	-0.4	-0.4

These results provide two insights. First, they confirm and further reinforce that bursty sampling is more suitable for program profiling than uniform sampling. Second, the weak correlations of uniform sampling suggest that the shortcoming of uniform sampling for program profiling is deeply inherent in the method, and can hardly be overcome by an increase in sampling rate. Given that uniform sampling is still the most commonly used runtime profiling method in today's systems, these insights hopefully will prompt developers to revisit the sampling methods they select.

Profile Accuracy and FDO Benefits Figure 4 reports the speedups FDO produces on the full profiles and profiles collected through bursty sampling at three sampling rates. Recall that for bursty sampling, higher sampling rates always lead to more accurate profiles. However, the bars in Figure 4 show a quite irregular pattern in the speedups as sampling rate increases. While *gap*, *vortex* and *perlbench* follow the intuitive trend of benefiting more from more accurate profiles, all other benchmarks show an opposite trend sometimes—degraded performance from more accurate profiles. The extreme case on *libquantum* even shows 75% more speedup from the lowest sampling rate than from the highest sampling rate. Overall, the FDO shows the best effectiveness on the exact profiles, 17% more speedups than on the best sampled profiles.

Table 5 reports the rank coefficients between profile accuracies and the speedups. No benchmarks have near 1 correlation coefficient. Only two benchmarks (*vortex, perlbench*) have coefficients larger than 0.8 on bursty sampling when weighted accuracy is used. So for them, higher bursty sampling rates are likely to bring better optimizations. But for most benchmarks, there is only weak or no correlation between profile accuracy and the usefulness for FDO. The program *gap* even has a coefficient of -0.85 on uniform sampling, indicating a largely monotonic decreasing relation between profile accuracy and usefulness for FDO.

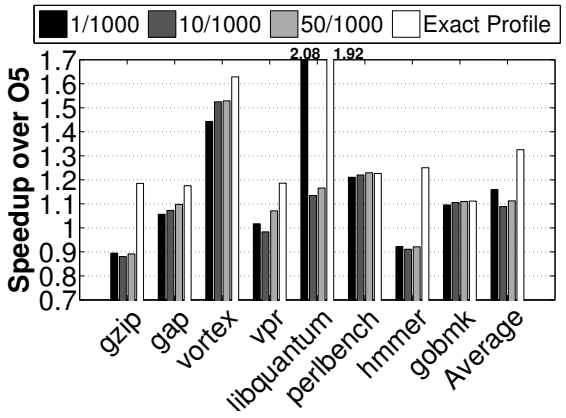

Fig. 4. Speedup comparison between sampled profiles of three sampling rates(1/1000, 10/1000 and 50/1000) and exact profiles.

Short Summary Current FDO optimization systems are constructed mostly on a common perception that larger sampling rates tend to lead to better performance. This section debunks the intuition by first showing that higher sampling frequency does not necessarily give us more accurate profiles, and it depends on the sampling method used. This indicates that we should be more careful about the design of sampler. More surprisingly, we show that there are very weak correlations between the accuracy of a profile and its usefulness for FDO, no matter which sampling method is used. It does not mean that we can just feed the compiler with randomly generated profiles for good FDO-driven performance. As results show, the best performance mostly still come from the exact profiles for most benchmarks. The findings suggest that current understanding to how profiling errors influence FDO is preliminary; some deep analysis into the results are necessary, as given in the next section.

Table 5. Rank Correlation coefficients between profile accuracy and performance

Program	Weighted Bursty	Unweighted Bursty	Weighted Uniform	Unweighted Uniform
gzip	-0.14	-0.29	-0.15	0.58
gap	0.75	0.75	-0.85	0.34
vortex	0.88	0.59	0.08	-0.8
vpr	0.62	0.79	-0.01	-0.07
libquantum	-0.08	-0.08	-0.5	0.16
perlbench	0.82	0.82	0.51	0.63
hmmer	0.47	0.41	0.11	-0.76
gobmk	0.41	0.41	-0.42	-0.59
Median	0.55	0.5	-0.08	0.05

3 Demystification and Profile Rectification

The previous section showed that for most benchmarks, there exists only very weak correlation between profile accuracy and its usefulness for FDO. However, we observe that sampled profiles do not perform as well as exact profiles, which means sampling errors do play an important role. After analyzing the influence of various types of errors, we identify two kinds of sampling errors that critically affect the FDO benefits: *zero-count errors* and *inconsistency errors*. In this section, we first present some analysis results on how these two kinds of errors impair the effectiveness of FDO, and then show that both types of errors can be fixed through a simple profile rectification, and finally report the dramatic speedup increment the rectification helps FDO generate.

3.1 Deep Analysis on Profile Errors

Zero-count Errors The first type of errors is zero-count errors, referring to the case when a counter in a sampled profile equals zero but its value in the full profile is not. For the purpose of explanation, we will concentrate our discussion on basic block frequency profiles.

Sampling, by nature, misses some parts of a program execution. But basic blocks that have a small value in the exact profile are especially easy to be missed completely by the sampler. Given the 20-80 rule (i.e., commonly 20% of a program is responsible for about 80% of its execution), most basic blocks are relatively cold, and hence have some good probabilities to get missed by the sampler, causing zero-count errors.

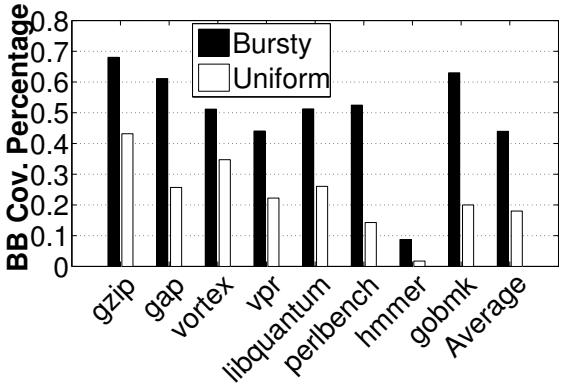

Fig. 5. Basic block coverage comparison between exact profiles and sampled profiles

Figure 5 shows the basic block coverage of the sampled profiles[4]. The *coverage* is defined as the percentage of the non-zero counters in the exact profile that

[4] Without noting, the results in this and following figures are similar across sampling rates, and the results at the lowest sampling rate is used.

also have non-zero values in the sampled profile. This metric shows how well the sampled profile represents the coverage pattern of the exact profile. We observe an average of 56% basic block coverage reduction by bursty sampling. In the worst case shown on *hmmer*, more than 92% of basic blocks are completely missed by the sampler. The coverage by uniform sampling is even worse, only 18%.

Through a detailed analysis of the influence of zero-count errors on the various optimizations in FDO, we find that two optimizations, function inlining and loop optimizations, are influenced the most. As Section 2.2 has mentioned, function calling frequencies are inferred from basic block frequencies in XLC. If the basic block containing a function call has zero frequency, the recompilation totally ignores the corresponding call edge. If all basic blocks invoking a function have zero frequencies, all the profile information of that function is ignored. This implies that the counter values of calling basic blocks play an important role in making inlining decisions, which is supported by Figure 6. It reports the number of function inlinings the FDO does when it uses a sampled profile, normalized by the number when it uses the full profile. On average, the zero-count errors cause the FDO to miss 79% inlining opportunities.

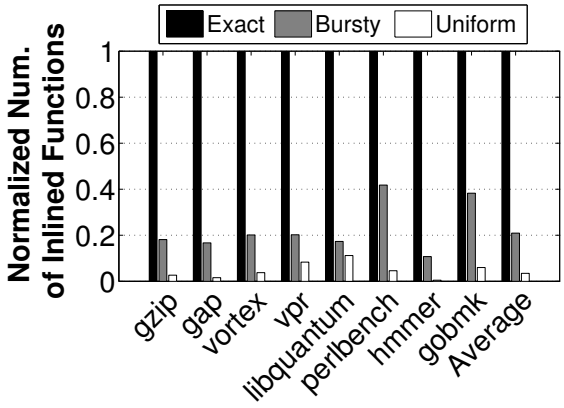

Fig. 6. Normalized number of inlined functions.

The second type of transformation, loop optimizations, also leverages profile information heavily. For example, in XLC, the iteration counts of all loops are calculated through the basic block counters in loop body and that of the loop preheader. If a loop's preheader's counter value is 0, its iteration count is annotated as "unknown". Since iteration count is one of the most important parameter in most loop transformations (e.g., loop versioning, loop unrolling, etc.), false information on loop iteration count may seriously impair the transformation quality. However, due to the fact that loop preheader is usually executed much less frequently than its corresponding loop body, it is quite possible that although the sampler obtains a reasonable profile of the loop body, it can not take

advantage of it because of a zero counter value of the loop preheader. Figure 7 shows the percentage of loops which have "unknown" iteration counts when the sampled profile is used, while non-zero iteration counter when the exact profile is used. On average, only 33% of loops derived their iteration count information from the sampled profile. This percentage dropped to 3% for uniformly sampled profiles.

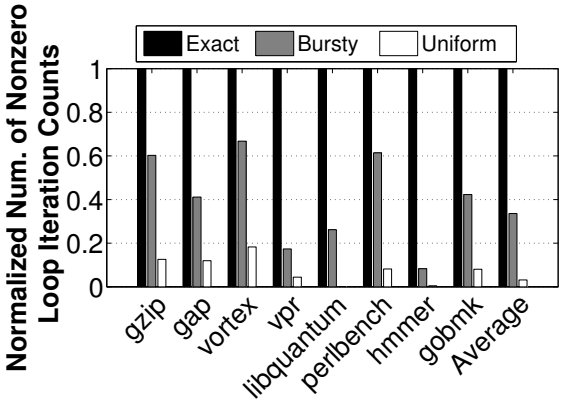

Fig. 7. Normalized number of loops having non-zero iteration counts.

We now try to rectify the zero-count errors to show their impact on performance. Figure 8 reports the results when the zero counters in the sampled profile are set to the values of their counterparts in the exact profile. That is, this rectification replaces the zero counters in the sampled profile with perfect information and hence completely removes the zero-count errors. We observe an increase of 10% and 19% performance improvement for bursty and uniform sampling respectively. It echoes the results of basic block coverage and inlining decision difference, and shows that zero-count errors are one of the main sources leading to reduced FDO benefit.

Inconsistency Errors Zero-count errors mainly happen on cold events, while inconsistency errors also happen on warm or hot events. An inconsistency error refers to the case when two counters of two basic blocks in the same function have different values in the sampled profile but have the same value in the full profile.

For hot events, both sampling methods can get pretty good approximation of their values, which is reflected by the very high weighted accuracy reported in Section 2. However, a decent approximation cannot prevent inconsistency errors from happening.

To help quantify the amount of inconsistency errors in the sampled profiles, we introduce a concept called *consistency score*. Let G represent a group of

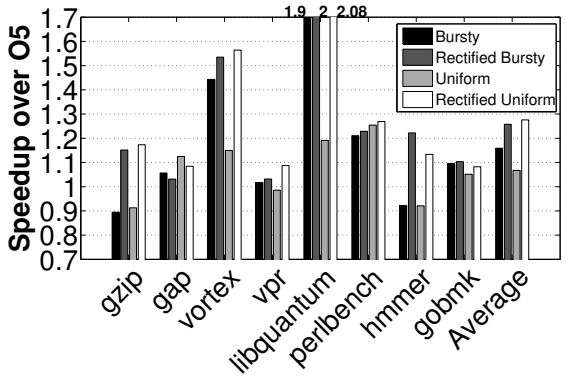

Fig. 8. Performance improvement after fixing zero-count errors with exact profiles.

basic blocks in an exact profile that have the same counter values, and G' be the largest subset of G that have identical counter values in a sampled profile. The *consistency score* of G in the sampled profile is $\frac{|G'|}{|G|}$. So the score must fall between 0 and 1; the higher it is, the better is the consistency preserved in the sampled profile. The overall consistency score of a sampled profile is just the average of the consistency scores of all the consistent groups in the corresponding exact profile.

Figure 9 reports the consistency scores of the sampled profiles. On average, the profiles have consistency scores of 0.47 and 0.01 for bursty and uniform sampling respectively, suggesting that the sampling methods cannot preserve the consistency relation among counters well.

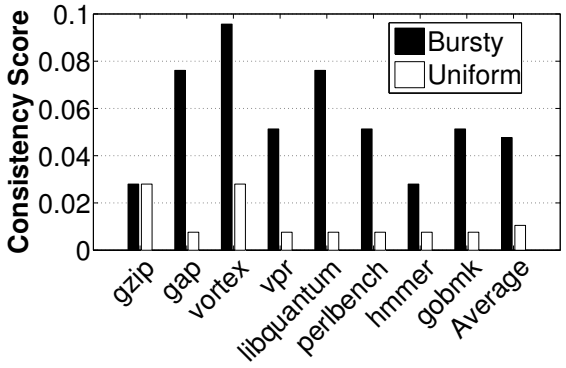

Fig. 9. Consistency scores of all benchmarks.

We study the potential performance gain by leveraging exact profiles to help rectify the inconsistency errors in the sampled profiles. We identify all the basic

block groups of each function in the exact profile that have the same counter value. Then, we set the counters in each group of the sampled profiles to their average. In this way, we maintain the equality relationship without changing the sampled profile's accuracy much. Figure 10 shows an improvement of up to 34% for uniform sampling on *vortex*, demonstrating the large potential of fixing inconsistent basic block counters. For bursty sampling, we have an outlier *libquantum*, for which the rectification degrades the performance by 90%. A plausible reason is that as the rectification is applied to its inconsistency errors only, the rectified profile somehow forms some serious conflict with the zero-count errors remaining in the profile. Such an inference comes from an observation the next subsection (Figure 13) will show: The degradation is completely reversed when zero-count errors are also fixed.

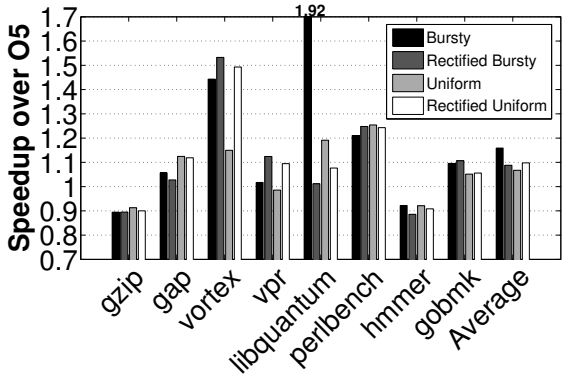

Fig. 10. Performance improvement after fixing inconsistency errors with exact profiles.

A detailed analysis shows that the primary influence of the inconsistency errors is also on function inlining. The XLC compiler makes inlining decisions based on function hotness, size, and other factors. It first chooses the functions whose calling frequencies exceed a threshold to inline. If two functions have the same frequency, it chooses the smaller one to inline. So consider two functions, A and B (assuming A is much larger than B), have the same frequency in the exact profile but different in the sampled one (A has a larger frequency than B). The inconsistency error may hence cause A rather than B to be inlined in the FDO on the sampled profile. As A is quite large, inlining it could cause many other functions to fail to get inlined because of the limit on the size of the resulting function. We observed a large degree of differences in inlining decisions of FDO before and after the inconsistency errors are fixed, especially on programs *vortex* and *vpr* (details skipped for lack of space).

It is worth noting that both types of errors have very limited influence on the weighted accuracy of a profile: zero-count errors are on cold blocks, which have small weights in the accuracy calculation; inconsistency errors happen on

warm and hot events, but being inconsistency does not prevent those events from being sampled enough times to get a good approximation of their exact values. For unweighted accuracy, zero-count errors play some more substantial role in the calculation, which explains why the unweighted accuracies are much lower than the weighted ones in Section 2. However, the unweighted accuracy still cannot well capture the actual effects of inconsistency errors. These reasons explain why there is no strong correlations between the accuracy of a profile and its usefulness for FDO, despite that profile errors—more specifically, the zero-count and inconsistency errors—affect FDO substantially.

3.2 Simple Profile Rectification

We analyzed the two critical types of sampling errors and showed the performance potential after fixing them with exact profiles. However, in reality we do not have exact profiles in hand when recompiling the programs. We consider two alternative options.

The first is through static analysis. By purely analyzing the program, it tries to find out which basic blocks will be executed for sure and which basic blocks must have the same execution frequency. Recall that the XLC tries to find out straight lines of basic blocks, and only instruments the first basic block in each straight line. This instrumentation optimization is a kind of static analysis. It not only reduces instrumentation overhead, but also maintains the equality relationship among the basic blocks in each straight line. However, to rectify these two types of errors, more sophisticated static analysis is necessary. Furthermore, the conservativeness of static analysis may also form some barriers for the rectification. For two basic blocks that in practice almost always have the same counter values, static analysis cannot give such a conclusion if there is no way to prove that they must have the same counter values. A probabilistic rectification is possible to bring more benefits than conservative static analysis for the nature of program optimizations.

In this work, we choose a second option, which uses a training profile (on a smaller input) to rectify sampled profiles. The rectification is simple. We assign 1 to the counters of the basic blocks, which are covered in the training profile but missed in the sampled profile. There are some other options, to use the exact counter value in the training profile or its scaled version. However, our experiments show that the minimal value change (from 0 to 1) is sufficient. We then identify all consistency groups (e.g., basic blocks that have equal counter values) in the training profile, and maintain the relation in the sampled profile by setting the counter of every block to the average counter value of the consistency set that block belongs to.

We justify this solution by answering three questions in the rest of this section. First, are the basic block coverage pattern and counter equality pattern hold across different ref inputs? Second, is the relatively small training input similar enough with ref input in terms of these two patterns? Third, does the FDO performance from the rectified sampled profiles outperform the performance of just using training profiles?

Pattern Stableness across Ref Inputs We use a training profile for the rectification based on our claim that once collected, it can be used to rectify many future sampled profiles of production runs. To support this claim, we need to show that the basic block coverage and counter equality patterns are stable across ref inputs. We use 3 ref inputs for each program, and collect their exact profiles. We quantify the basic block coverage pattern stableness by calculating the basic block coverage percentage for each pair of the three exact profiles and get their average. Similarly, to quantify the counter equality pattern stableness, we just replace the basic block coverage percentage with the consistency score. As Figure 11 (a) shows, the basic block coverage among ref inputs is reasonably stable with a minimum of 70% and an average of 89%. The counter equality pattern similarity is a bit less (on average 85%) due to the difficulty in maintaining it in different profiles.

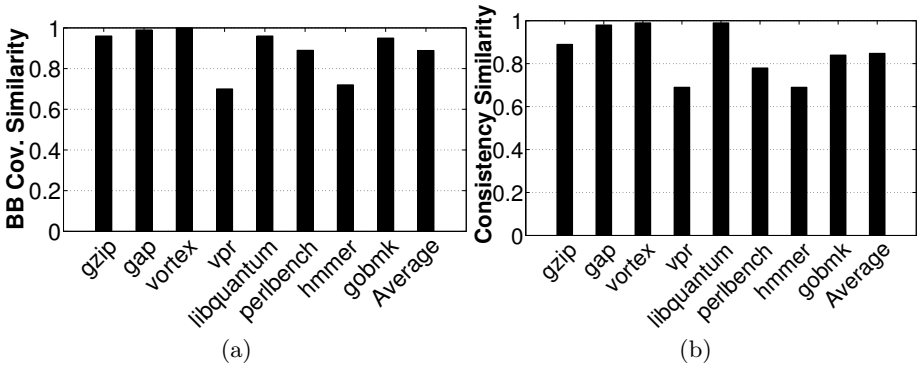

Fig. 11. (a) Basic block coverage pattern similarity across ref inputs. (b) Basic block counter equality pattern similarity across ref inputs.

Pattern Stableness between Train and Ref Inputs Figure 12 (a) reports the basic block coverage percentages in the training and ref profiles. We observe that on average 87% basic blocks executed on the ref input are also executed on the training input. Figure 12 (b) shows that training profiles' basic block equality patterns are very similar to that of exact profiles, with a consistency score of 83% on average.

The stableness of the patterns across inputs suggests the promise of using training profiles for profile rectification. It is tempting to wonder why not just simply use the training profile as the approximated ref profile for FDO. The reason is that although the training profile carries some value patterns applicable to other profiles, the exact values it contains differ significantly from ref profiles for the high sensitivity of profile values to program inputs. Directly using training profiles for FDO may hence result in some less desirable performance, as we show next.

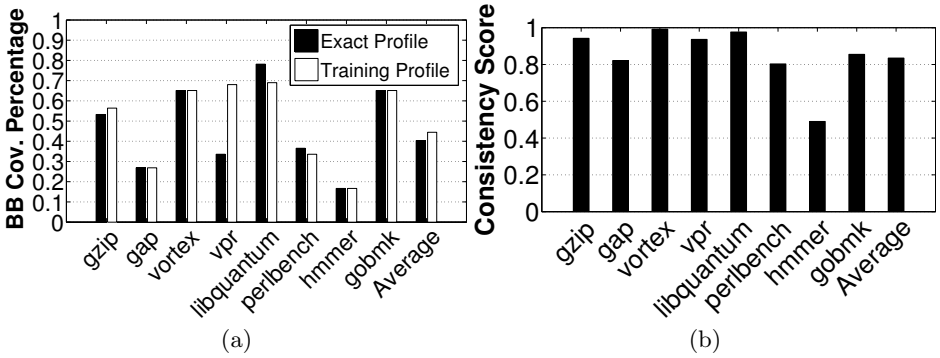

Fig. 12. (a) Basic block coverage pattern similarity between training profiles and exact profiles of ref inputs. (b) Counter equality pattern similarity between training profiles and exact profiles of ref inputs.

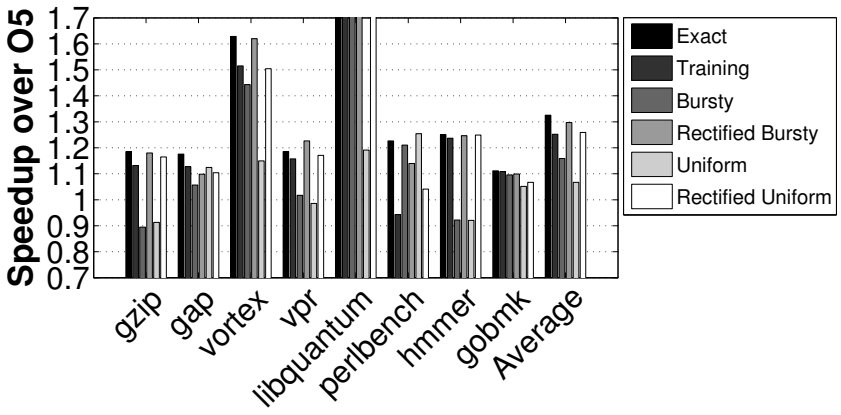

Fig. 13. Speedup comparison of FDO based on exact profiles, training profiles and rectified sampled profiles. (Most bars of "libquantum" are out of the range of the graph; their values are respectively 2.08, 2.09, 1.92, 2, 2.01.)

Performance Results We use the sampled profiles of the lowest sampling frequencies for evaluation. As Figure 13 shows, the sampling error rectification based on training profiles perform very well by obtaining 92% and 81% of the full FDO benefit from exact profiles for bursty and uniform sampling respectively. Compared to the sampled profiles, the FDO performance benefit recovery from the rectification is 59% and 43% for the two sampling methods. We also include the FDO benefit from purely using training profiles. On average, the rectified sampled profiles of bursty sampling brings 4.6% more speedup than training profiles, which shows the usefulness of training profiles for rectification. For the specially input-sensitive program *perlbench*, the training profile gives even 6% slowdown, while the sampled profiles—rectified or not—produce 1.04X to 1.25X

speedup. It demonstrates that the basic block counter distribution could be very different between training profiles and exact profiles, and so the recompilation based on training profiles may optimize the program in a way not suitable for the ref inputs. We also observe that the rectification reduces FDO benefit by 7% and 21% for bursty and uniform sampling, respectively. This anomaly, along with several other cases in which exact profile does not produce the best performance, implies the imperfect implementation of the FDO due to the complexity in program optimizations, rather than some inherent rules.

3.3 Results from Gcc

Despite the different implementations between Gcc and XLC, most of the insights reported on XLC hold on Gcc. A prominent difference is that Gcc tends to have a smaller degree of speedups by its FDO than those by the FDO of XLC. The reason probably comes from the relatively less sophistication of its FDO implementation.

Figure 14 reports the speedups when bursty sampling of different sampling rates are used. (Vortex is elided as it cannot run through the modified Gcc for some unknown reasons.) The settings include the cases of the highest static compilation, FDO on the exact profiles and fixed sampled profiles. The results show that three benchmarks (*gzip, libquantum, perlbench*) have considerable speedups from FDO when the full profiles are used. For all of them, the rectified profiles help materialize most of the potential of FDO. The four sampling rates, although differing by up to 100 times, do not show much different influence on the FDO benefits when the profile is rectified. On program *gap*, the rectified profiles offers even higher speedups than the full profile. The reason is due to the imperfect design of FDO as mentioned in Section 3.2.

3.4 Discussions

The results in this section indicate that simple profile rectifications go a long way: Despite the simplicity of the profile rectifications through training profiles, the rectified profiles—at even the lowest sampling rate—can help tap into most of the potential of FDO.

Second, when the two kinds of rectifications are applied, speedups replace the performance degradation seen in Section 3.1 when only inconsistency errors are rectified. It suggests that the two kinds of value patterns have some subtle relations among each other. Fixing them together can help avoid some conflicts subsumed by the relations.

Finally, using a training profile helps explore the potential of profile rectification in this experiment, but after getting the insights that simple rectification to the two types of errors is sufficient, one may choose some other ways to do the rectification. For instance, one could combine sophisticated static program analysis with lightweight profiling on some ambiguous branches to identify the two kinds of value patterns of counters. Combined with cross-production run lightweight profiling [19], the method may provide more seamless integration

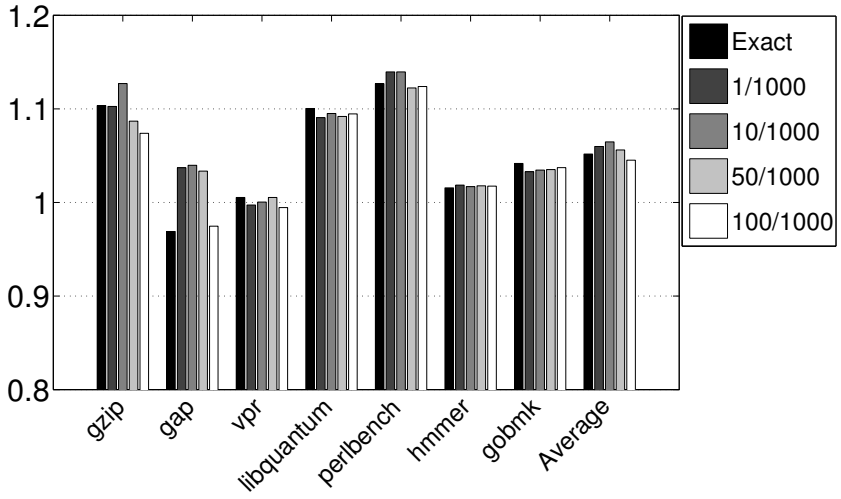

Fig. 14. Speedup by Gcc.

with the JIT-based runtime engines. Detailed research in this direction is future work.

4 Related Work

Levin and others proposed the use of a Minimum Cost Circulation algorithm to adjust an incomplete edge profile of a control flow graph [14] for post-link optimization. The basic idea is to find minimum adjustment to the edge and basic block weights (i.e., frequencies) in a sampled profile such that after the adjustment, the weights meet the constraints defined by the control flow graph, that is, the sum of the weights of all incoming edges of a block equals the sum of the weights of all outgoing edges of the block. A later study by Chen and others extended the idea to higher level compilation and explored the usage of extra hardware performance counters for alleviating sampling errors [10]. Our work concentrates on the two special types of sampling errors, namely zero-count and inconsistency errors. Both types of errors impose some challenges to the prior approach. For zero-count errors, consider a loop with its preheader block executed once and the loop body executed 1000 times. Due to the inaccuracy in sampling, the sampled frequencies could be 0 for the preheader block, 92 for the loop body and 93 for the loop back edge. The previous algorithm may adjust the back edge to 92 to meet the constraint on incoming and outgoing edges of the loop header block. But that adjustment fails to correct the zero-count error of the preheader block. The algorithm is even less effective in fixing inconsistency errors. The algorithms may be able to adjust the frequencies such that two blocks that are dominator and postdominator of each other have the same frequency. But as Section 2.2 mentions, such kind of relations are already being explored by many

compilers by default. Most inconsistency errors we observed happen on blocks across functions or branches. They often reflect dynamic patterns rather than static invariants. Although they are hard to capture by the prior algorithm on static control flows, they are fixable through the statistical rectification method proposed in this work.

In a previous exploration [16], Mytkowicz and others have studied existing profilers and showed that they failed to agree with each other on the identification of hot functions. They found the sources of incorrectness and proposed a prototype of a random sampler to remove the biases in the previous implementations of random samplers. The only study we have found directly on the relation between profile sampling and the usefulness of profiles for FDO is by Langdale and others [13]. In that study, the authors have used only uniform sampling on machines a decade old. More importantly, the authors used compilers with quite preliminary FDO implementation: The full potential of the FDO on exact profiles is only about 3% speedups. Because of all these limitations, most conclusions from that work are out of date and even contrary to what we observe on modern compilers and machines (e.g., the results on busty sampling.) To the best of our knowledge, the study presented in this current paper is the first systematic study on the relations among sampling, profile accuracy, and profile usefulness *on modern compilers, machines, and sampling methods.* Moreover, we are not aware of previous proposals of the two types of profile error rectification. The novel insights this study obtains are multi-fold on many aspects, including sampling for profiling, FDO, profile rectification, and cross-input stableness of value patterns in profiles.

Many FDO-related studies focus on efficient instrumentation. Knuth and Stevenson show in [12] that they only need to instrument a minimum number of edges of the control flow graph and calculate the counters for all other edges in an offline analysis. Ball and Larus [7] propose an efficient path profiling technique, which encodes each path into a non-negative integer and uses it as an index to update global counters efficiently. In [20], the authors separate interesting paths and profile them with low overhead. Some other studies focus on reducing profiling overhead through sampling. Bond et al. [8] identifies that Ball's path profiling algorithm overhead bottleneck is counter update, and uses sampling to reduce the overhead to provide continuous profiling. Arnold and others [6] reduce instrumentation overhead of a JAVA JIT compilation system by creating a fully instrumented copy for each function and periodically switching execution to that copy to collect profile information.

Many researchers propose to take advantage of different kinds of profiling information for various optimizations. Pettis and Hansen [17] leverage execution counter profile to order procedures and position BBs within each procedure. By exploring the correlations among BBs, this optimization improves greatly code cache performance and reduces branch penalty. Chang and others [9] have implemented an inter-file inliner that automatically uses profile information. Wu [22] explored memory load profiles to find stride patterns and identify the responsible load instructions for prefetching. Rajagopalan and others [18] profile event-based

programs to identify commonly occurring event sequences, and reduce the overhead from function indirections. One of our previous papers [21] finds correlations among program behaviors through multiple profiles and applies the technique to runtime version selection.

With the trend of adding more kinds of hardware counters in modern machines, we have seen increasing interests in exploring those hardware resources. Ammons et. al [4] attach hardware counter information on calling context sensitive paths. Adl-Tabatabai and his colleagues [3] leverage hardware counters to smartly inject prefetching instructions in a JIT compilation system.

5 Conclusion

This paper presents a systematic exploration on the relations among sampling rates, profile accuracy, and profile usefulness for FDO. The exploration covers seven factors in four levels. It reveals some counter-intuitive relations, the most prominent of which are that higher sampling rates (within a typical sample rate range) do not lead to more accurate profiles when uniform sampling is used, and more importantly, no matter which sampling method is used, the accuracy of the profiles does not show a strong correlation with their usefulness for FDO. The paper then describes a detailed analysis and points out that two types of sampling errors, *zero-count errors* and *inconsistency errors*, play an essential role in restraining the power of FDO. Based on empirically confirmed cross-input stableness of two kinds of value patterns in profiles, the paper presents a simple way to rectify the two types of errors through statistical patterns. The dramatic enhancement of the FDO benefits concludes that statistical rectification of the two types of errors in a profile is promising in tapping into the full potential of FDO. It also suggests that with profile rectification, sampling rate (and hence sample overhead) can be significantly lowered without hurting the FDO benefits. In addition, the study exposes some subtle relations among the rectification of the two types of errors, and meanwhile, reinforces that bursty sampling is superior to uniform sampling for collecting profiles for FDO.

These multi-fold novel insights provide the first principled understanding in effective collection of profiles for FDO. They may help advance the profile collection in modern runtime systems, and open up many new opportunities for modern program optimizations.

Acknowledgment

We owe the anonymous reviewers of ECOOP'13 our gratitude for their helpful comments to the paper. This material is based upon work supported by the National Science Foundation under Grant No. 0811791 and CAREER Award, DOE Early Career Award, and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, DOE, or IBM.

References

1. Jikes RVM. http://jikesrvm.org.
2. SPEC CPU benchmarks. http://www.spec.org/benchmarks.html.
3. A. Adl-Tabatabai, R. Hudson, M. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *PLDI*, 2004.
4. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, 1997.
5. M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *CGO*, 2005.
6. M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, 2001.
7. T. Ball and J. R. Larus. Efficient path profiling. In *Micro*, 1996.
8. M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In *MICRO*, pages 130–140, 2005.
9. P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. Hwu. Profile-guided automatic inline expansion for c programs. *Software Practice and Experience*, 22(5), 1992.
10. D. Chen, N. Vachharajani, R. Hundt, S. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. Taming hardware event samples for fdo compilation. In *CGO*, 2010.
11. M. Hirzel and T. M. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proceedings of ACM Workshop on Feedback-Directed and Dynamic Optimization*, Dallas, Texas, 2001.
12. D. Knuth and F. Stevenson. *BIT Numerical Mathematics*, 13(3):313–322.
13. G. Langdale and T. Gross. Evaluating the relationship between the usefulness and accuracy of profiles. In *Proc. Workshop on Duplicating, Deconstructing, and Debunking*, 2003.
14. R. Levin, G. Haber, and I. Newman. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *HiPEAC*, 2008.
15. H. Mousa and C. Krintz. HPS: Hybrid profiling support. In *PACT*, 2005.
16. T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. In *PLDI*, 2010.
17. K. Pettis and R. C. Hansen. Profile guided code positioning. In *PLDI*, 1990.
18. M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Profile-directed optimization of event-based programs. In *PLDI*, pages 106–116, 2002.
19. K. Tian, E. Zhang, and X. Shen. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *OOPSLA*, 2011.
20. K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *POPL*, 2007.
21. B. Wu, Z. Zhao, X. Shen, Y. Jiang, Y. Gao, and R. Silvera. Exploiting inter-sequence correlations for program behavior prediction. In *OOPSLA*, 2012.
22. Y. Wu. Efficient discovery of regular stride patterns in irregular programs. In *PLDI*, 2002.