Taming Hardware Event Samples for Precise and Versatile Feedback Directed Optimizations

Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang Li, Stephane Eranian, Wenguang Chen, and Weimin Zheng

Abstract—
Feedback-directed optimization (FDO) is effective in improving application runtime performance, but has not been widely adopted due to the tedious dual-compilation model, the difficulties in generating representative training data sets, and the high runtime overhead of profile collection. The use of hardware-event sampling overcomes these drawbacks by providing a lightweight approach to collect execution profiles in the production environment, which naturally consumes representative input. Yet, hardware event samples are typically not precise at the instruction or basic-block granularity. These inaccuracies lead to missed performance when compared to instrumentation-based FDO. In this paper, we use Performance Monitoring Unit (PMU) based sampling to collect the instruction frequency profiles. By collecting profiles using multiple events, and applying heuristics to predict the accuracy, we improve the accuracy of the profile. We also show how emerging techniques can be used to further improve the accuracy of the sample-based profile. Additionally, these emerging techniques are used to collect value profiles, as well as to assist a lightweight inter-procedural optimizer. All these profiles are represented in a portable form, thus they can be used across different platforms. We demonstrate that sampling-based FDO can achieve an average of 92% of the performance gains obtained using instrumentation-based exact profiles for both SPEC CINT2000 and CINT2006 benchmarks. The overhead of collection is only 0.93% on average, while compiler-based instrumentation incurs 2.0%–351.5% overhead (and 10x overhead on an industrial web search application).

Index Terms—Sample Profile, Feedback Directed Optimization, Performance Counter, Last Branch Record.

1 Introduction

Many compiler optimizations, such as procedure inlining, instruction scheduling, and register allocation benefit from dynamic information e.g. basic block frequency and branch taken / not-taken ratios. This information allows a compiler to optimize for the frequent case, rather than using probabilistically estimated frequencies, or assuming that all code is equally likely to execute. Profiling is used to provide this feedback to a compiler.

The traditional approach to profile-guided optimization involves three steps. First, we compile the application with special flags to generate an instrumented version of the program (*instrumentation build*). Next, we run the instrumented application with training data to collect the profile. Finally, we recompile the application using the profile, which can help the compiler make better optimization decisions (*feedback-directed optimization (FDO) build*).

Unfortunately, there are four shortcomings in this approach. First, it requires compiling the application twice. For applications with long build times, doubling the build time can significantly degrade programmer productivity.

Second, the instrumentation and optimization builds are tightly coupled, this prevents flexible use of the profile data. For example, GCC requires that both builds use the same inline decisions and similar optimization flags to ensure that the control-flow graph (CFG) profiled in the instrumentation build matches the CFG annotated with the profile data in the FDO build. During the FDO build, profiles are used for inline decisions. However, the compiler cannot find a callsite specific profile to annotate the inlined callees. Instead, it can only use the scaled callee profile to annotate the cloned instances. In addition, GCC requires that the source files remain unchanged between two builds. However, observations indicate that many of the code changes between releases are in the cold paths, and this does not affect the behavior of the hot code. The tight coupling between the two builds prevents the previous profiles from being reused.

Third, collecting profiles requires an appropriate execution environment and *representative* input. For example, profiling a transaction processing application may require an elaborate database setup and a representative set of queries to exercise the application. Creating such an environment and identifying a set of representative input can be very difficult.

- *Dehao Chen, Robert Hundt, Xinliang Li and Stephane Eranian are with the Google Inc, 1600 Amphitheatre Parkway, Mountain View, California, 94043. E-mail: {dehao, rhundt, davidxl, eranian}@google.com*
- *Neil Vachharajani is with Pure Storage Inc, 650 Castro Street, Mountain View, California, 94041. E-mail: nvachhar@gmail.com*
- *Wenguang Chen and Weimin Zheng are with Department of Computer Science and Technology, Tsinghua University, Beijing, China, 100084. E-mail: {cwg, zwm-dcs}@tsinghua.edu.cn*

Fourth, the instrumented profile collection run typically incurs significant overhead (reported to range from 9% to 105% [5], [6], but has been observed to be as much as 10x on an industrial web search application) due to the additional instrumentation code executed. While scaling down inputs may ameliorate the problem, for the profiles to be useful, they must accurately reflect the application's real usage. Crafting an input that is sufficiently scaled down to facilitate fast and easy profiling while retaining high fidelity to the real workload is difficult. The problem is exacerbated by constant application changes potentially making old profiling inputs inapplicable to new versions of the application. Furthermore, the high runtime overhead can alter the critical path of time critical routines, e.g., OS kernel codes, for which getting an instrumentation-based profile is not easily possible in the first place.

These limitations often lead developers to avoid FDO compilation and forgo its associated performance benefits. Several workarounds exist to make instrumentation-based FDO easier to adopt. For example, binary instrumentation can be used to decouple the instrumentation build and the FDO build. However, since its overhead is significantly higher than compiler instrumentation, binary instrumentation is hard to deploy in actual production environments. Synchronous sampling [4], [24] using instrumentation has been proposed to reduce overhead. However, it needs to insert extra code to switch between the normal and the monitoring states. The inserted code still incurs overhead that is not acceptable for production systems.

To overcome these limitations, we propose skipping the instrumentation step altogether, and rely instead on sampling events generated by the performance monitoring units (PMU) of modern processors to obtain estimated execution profiles. The sample data does not contain any information about the intermediate representation (IR) used by the compiler. Instead, source position information in the debug section of unstripped binaries is used to correlate the samples to the corresponding basic blocks during the FDO build.

Using sampling to collect profile, together with using source position information to correlate profile has two key benefits.

1) Since source position information is used to correlate the profile to the program being compiled, this approach eliminates the tight coupling between the instrumentation and FDO builds. Profiles collected on older versions of a program can be used by developers as long as the behavior of the hot code doesn't change and source file haven't been changed too much, thus eliminating the need for dual compilation in the normal work-flow. This benefit can also apply to binary level instrumentation-based FDO.
2) The overhead of profile collection is significantly lower since no instrumentation code is inserted, and is typically in the range of 2% or less.

The low overhead of profiling together with a loose coupling between the profiling build and the FDO build offer compelling usage scenarios. For example, in an Internet company, profile collection can occur by infrequently attaching it to standard binaries running on production systems. The data collected can be stored in a profile database for future FDO builds. This usage model further eliminates any potential discrepancy between profile input data and actual usage patterns observed in the deployed application. Since the profile is collected in the production environment, real-world workloads become natural input for the profile collection runs. Thus, the collected profile has a very high fidelity to actual behavior, and programmers do not need to manually forge any training input.

Using hardware performance monitoring events to estimate execution profiles is, however, not a panacea. To prevent performance monitoring from slowing down the processor's execution, many tradeoffs are made in the design of modern PMUs; these lead to imprecise sample attribution. Specifically, the instruction address that the PMU associates with an event is often not the true address where the event occurred. To complicate matters further, the distance between the instruction that caused an event and the instruction to which event is attributed is typically variable. Our experiments show that even when using advanced PMU features (e.g., Precise Event-Based Sampling (PEBS) mode on Intel Core 2 processors), events aggregate on particular instructions and are missing on others. While these phenomena may not be problematic for performance debugging, they create significant challenges for using sample profiles in FDO. For example, when profiling values of registers, if a target instruction never receives samples, we are unable to collect the value distribution information for that particular instruction.

In this paper, we present a sampling-based framework for FDO. In traditional compilers, such as GCC and Open64, FDO uses two types of profiles: edge/basic block frequency profiles and value profiles, which are used to drive many feedback directed optimizations. Our framework focuses on both these profiles, and uses several different sampling techniques to collect them. Though researchers have proposed using PMU-based sample profiles to drive other optimizations such as data prefetching [1], these techniques are mainly adopted in dynamic optimizers, which are not the focus of this paper. However, our framework enables these optimizations to be easily adopted in FDO.

We first introduce using sampling approach to collect edge profiles. We show the artifacts observed in sample-based profiles, and propose two approaches to improve its accuracy. Then, we show how sampling can be used to derive value profiles. Finally we evaluates our approach.

We summarize the primary contributions of this work below:

1) We build a framework to collect portable profiles in a lightweight manner. We also build infrastructure in the GCC compiler to effectively use these profiles.
2) We identify hardware effects that negatively influence sample distribution. We propose a heuristic approach, based on sampling multiple hardware events, that mitigates the systematic bias introduced by these hardware

effects.

3) We propose algorithms that use branch history information to collect an edge/basic block frequency profile. This information is used for indirect call promotion, and to guide module grouping in a lightweight inter-procedural optimizer.
4) We use precise event based sampling to collect value profiles; and use program slicing to extend the sampling scope to allow for more effective sampling-based value profiling.
5) Finally, we present an evaluation of the efficacy of the proposed approach. We present results from an implementation of sampling-based FDO in the GCC compiler. Overall, we show that PMU sampling-based FDO, combined with the proposed smoothing heuristics, can achieve 92% of the performance gains obtained using instrumentation-based FDO for SPEC2000 benchmarks. However, sampling-based FDO, on average, incurs only 0.9% to 1.8% profiling overhead as compared to the 90.5% profiling overhead (2x to 10x on an industrial web search application) incurred by compiler-based instrumentation.

The rest of the paper is organized as follows: Section 2 describes hardware event sampling and explains how it can be used to generate the profiles for FDO compilation. Section 3 shows the problems in using general sampling approaches to collect frequency profile, and the heuristics used to improve the accuracy. Section 4 describes the algorithm to use branch information to get the frequency profile, and how branch information can be used to assist a lightweight inter-procedural optimizer. Section 5 describes how to use precise event based sampling to collect value profiles. Section 6 then describes the experimental evaluation of PMU sampling-based FDO. Section 7 describes related work in the area. Finally, Section 8 concludes the paper.

2 Using PMU-based Sampling for FDO Compilation

This section describes how sampling works with most modern performance monitoring units and how PMU sampling can be used to guide feedback directed optimizations.

2.1 Hardware Event Sampling

The performance monitoring unit on a modern micro processor is usually organized as a collection of counters that can be configured to increment when certain *hardware events* occur. For example, counters can be configured to increment on each clock cycle, each time an instruction retires, for every L2 cache miss, etc. The raw contents of these counters can be dumped at program exit to get summary information about how the program executed. Alternatively, the counters can be used for sampling. In this mode, the PMU is configured to generate an interrupt whenever a counter overflows. When the interrupt triggers, performance monitoring software can record the system state (e.g., the program counter (PC), register contents, etc.). This recorded data forms the sample profile for the application.

2.2 Using the Profile in a Compiler

In the following parts of this paper, several methods are used to collect the sample profile. Overall, the output of these methods is a mapping from the binary instruction to its corresponding profile, including frequency profile and value profile. Because this mapping is maintained at instruction level, we refer to it as instruction-level profile. For the sample-based profile to be usable by a compiler, the instruction-level profile must be converted into a profile annotated onto the compiler's intermediate representation (IR). To achieve this, the instruction-level samples are first attributed to the corresponding program source line using the source position information present in the debug information. The execution frequency for each source line is stored in the feedback data file.

During the FDO build, a compiler reads the frequency profile data to annotate the CFG. Each basic block consists of a number of IR statements. The source line information associated with the individual IR statements is used to determine the list of source lines corresponding to a basic block. The basic block sample count is then determined by the frequency of source lines corresponding to it. Theoretically, the frequency of all source lines corresponding to a basic block should be the same. However, as will be discussed in Section 6.2, source correlation can be skewed. A voting algorithm (e.g., average or max) is designed to assign the most reliable frequency as the basic block sample count.

For the value profiles, if a source line is found to contain possible value profiling optimization opportunities, compiler reads in the corresponding value profile from the instruction profile, and uses it to rebuild the histogram information of the value and optimize the code.

By using source line information to record profiles, the coupling between the binary used for profile collection and the FDO build is greatly relaxed. This allows effective re-use of the collected profiles. For example, when there are minor source code changes between profile collection and the FDO build, if the source code change does not affect too much of the frequently executed code, the list of source code changes (change-list descriptions) can be used to update the profile recorded to better match the source code being compiled with FDO. One thing to note, this approach is not limited to the sampling-based FDO. The profiles collected using binary instrumentation tools such as PIN can also be represented in the same format to drive a decoupled FDO build. However, because of the excessive overhead incurred by instrumentation, it's not chosen to collect profiles in this paper.

2.3 Constructing Edge Profile

Due to errors and noise in sampling, the frequency profile obtained via sampling may not be consistent. That is to say, for a given basic block, its sample count will not always equal the sum of the sample counts of its successor or predecessor edges. To make the counts consistent and to obtain an edge profile from the basic block profile, we

translate the problem into an instance of the minimum cost flow (MCF) problem. In our implementation, we use MCF twice. First, before creating the sample feedback file, an MCF prepass is performed on instruction level profile. During the prepass, a binary level CFG is built for each procedure, the instruction level profile is annotated on the CFG, and MCF is used to refine the profile (detailed in Section 3.3). This refined profile is used to create the profile feedback file. Second, after reading the profile feedback file, compiler uses MCF to translate the basic block profile into an edge profile. One thing to note is, if we use the branch information to derive the frequency profile, as described in Section 4, the profile itself is already accurate enough. Thus the first step is omitted in this approach. The details of formulating the basic block to edge profile conversion problem as an MCF problem can be found in the literature [17], [22]. Here, we describe a few salient details.

An instance of the MCF problem consists of a graph $G = (V, E)$, where each edge has a capacity and a cost function. The objective is to assign a flow to each edge such that for each edge, (a) the flow is less than the edge's capacity, (b) for a given vertex, the sum of the flows on incoming edges equals the sum of the flows on outgoing edges, and (c) that over the whole graph, the sum of the costs is minimized.

For profile smoothing, the graph used in MCF is known as the residual graph and it is based on a function's CFG. Each basic block is split into two nodes, the incoming edges to the block connect to the first node in the pair, and the outgoing edges originate at the second node in the pair. The two nodes are connected with a forward and reverse edge. Sending flow through the forward edge corresponds to increasing the basic block count, and sending flow through the reverse edge corresponds to decreasing the basic block count. Since a solution to MCF seeks to minimize cost, the solution can be biased in favor of raising a particular block's weight by assigning its forward edge a low cost. Similarly, one can bias in favor of lowering a block's weight by assigning its reverse edge a low cost. Additionally, the solution can be biased towards altering a specific block's weight by giving its forward and reverse edges a lower cost. We exploit this property of MCF in Section 3.3.

3 Collecting Frequency Profile

Frequency profile is among the most important profiles for FDO compilation because most of the backend optimizations can benefit from it. In this section, we present a general approach to collect frequency profile. Section 3.1 introduces the general approach. Though this approach can be applied to most of the architectures, it suffers from significant inaccuracy problems, which can be summarized as the anomalies described in Section 3.2. Section 3.3 proposes a heuristic to improve the quality of the instruction profile. Another approach is proposed in Section 4, which utilizes an emerging technique, namely LBR, to collect more accurate frequency profile.

Fixed Sample Period		**Random Sample Period**		**PEBS**		
Abs.	Norm.	Abs.	Norm.	Abs.	Norm.	**Loop**
267	0.52	577	1.13	1554	3.01	`00: add $0x1,%rdx`
142	0.28	95	0.19	0	0.00	`04: or $0x2,%rdx`
1212	2.35	237	0.46	0	0.00	`08: add $0x3,%rdx`
272	0.53	532	1.04	447	0.87	`0c: or $0x4,%rdx`
0	0.00	523	1.02	1438	2.79	`10: add $0x5,%rdx`
1252	2.43	475	0.93	66	0.13	`14: or $0x6,%rdx`
269	0.52	502	0.98	1	0.00	`18: add $0x7,%rdx`
149	0.29	454	0.89	46	0.09	`1c: or $0x8,%rdx`
1197	2.32	512	1.00	504	0.98	`20: add $0x9,%rdx`
9	0.02	498	0.98	1402	2.72	`24: or $0xa,%rdx`
327	0.63	487	0.95	3	0.01	`28: add $0xb,%rdx`
48	0.09	724	1.42	116	0.22	`2c: or $0xc,%rdx`
1504	2.92	633	1.24	1833	3.55	`30: add $0xd,%rdx`
266	0.52	565	1.11	19	0.04	`34: or $0xe,%rdx`
141	0.27	762	1.49	260	0.50	`38: add $0xf,%rdx`
1219	2.36	999	1.96	1675	3.25	`3c: or $0x10,%rdx`
268	0.52	532	1.04	35	0.07	`40: add $0x1,%esi`
0	0.00	0	0.00	0	0.00	`43: cmp %rcx,%rdx`
1255	2.43	591	1.16	398	0.77	`46: jbe 0`
515.63		510.42		515.63		**Average**
541.21		222.45		677.56		**StdDev**

Fig. 1. The sample counts measured on an Intel Core 2 for a loop consisting of one basic block.

3.1 General Approach

Sampling a counter that increments each time an instruction retires (e.g., `INST_RETIRED` on x86 processors) provides a natural way to estimate the instruction profile. Each time the counter overflows, the PC is recorded. In the literature, this approach has been called *frequency-based* sampling [28]. An alternative to this approach is *time-based* sampling [28], where processor cycles, rather than instructions, are counted. Unfortunately, time-based sampling biases the sample towards basic blocks that take longer to run than others. Thus in this section, PMU-based sampling is adopted to collect the frequency profile.

3.2 Problems Observed

Sampling is a statistical approach and therefore its results are not exact. However, we observe hardware induced problems that go well beyond plain statistical inaccuracies. For example, consider the loop shown in Figure 1. The loop is comprised of one basic block that iterates 104166667 times. If the loop is sampled using a sampling period of 202001, then one would expect each instruction in the loop's body to receive approximately $\frac{104166667}{202001} = 515.67$ samples. The two columns of numbers labeled Fixed Sample Period in the figure show the actual samples collected on an Intel Core 2 machine. The first column shows the raw count for each instruction and the second shows the count normalized by the expected count (i.e., 1.0 is the correct count, < 1.0 means the instruction was undersampled, and > 1.0 means the instruction was oversampled). We can see from this data that the sample counts vary by a factor of 2–3 from what they ought to be. In this section, we describe these artifacts, and posit causes for these anomalies. Section 3.3 will then introduce an approach to achieve more accurate profiles. We observed similar effects on a variety of architectures from Intel and AMD.

3.2.1 Synchronization

If one selects a period that is *synchronized* with a piece of the application, a few instructions will receive all of the samples. For example, if a loop contains k dynamic instructions per iteration, and the sampling period is selected as a multiple k, then only one instruction in the loop will be sampled.

Randomization can avoid synchronization. Instead of using a constant sampling period, the PMU is configured so the number of events between samples is the user provided sampling period plus a randomly chosen delta. After each sample, a new random delta is selected. Since the number of events between each sample is not constant, periodic properties in the program being measured do not skew the sample.

Additionally, our empirical results show that random sampling improves the uniformity of samples even in the absence of synchronization. In the example in Figure 1, there are 19 instructions in the loop and the sampling period used was 202001 which is not a multiple of 19. Consequently, the unexpected results should not be due to synchronization. However, when random sampling is used, one obtains the results shown in the two columns labeled Random Sample Period in the figure. With randomization, the samples are more uniformly distributed. The average number of samples per instruction changed because the average sampling period was 204080 (rather than 202001) due to randomization. However, notice that random sampling reduced the standard deviation by a factor of almost 2.5.

Further experiments reveal that non-random sampling leads to a form of pseudo-synchronization. Although a particular sampling period is requested, due to skid (described in the next section) that is variable, yet systematic, the actual sampling period is ultimately partially synchronized with the loop. While this can be mitigated through careful non-random adjustment of the sampling period for the particular code in the example, random sampling proves more effective when dealing with code with complex control flow and with varying amounts instruction-level parallelism.

3.2.2 Sample Skid

Ideally the PC reported when a counter overflows would be the PC associated with the instruction that triggered the overflow. Unfortunately, the reported PC is often for an instruction that executes many cycles later. This phenomenon is referred to as *skid*. For example, previous work shows that on an Alpha 21064, the recorded PC corresponds to the instruction that is at the head of the instruction queue 6-cycles after the one that triggered the overflow [12]. On an Intel Core 2 machine, we observed a similar phenomenon. The reported PC corresponds to the instruction that is at the head of the instruction queue some number of cycles (often approximately 30-cycles) after the one that overflows the counter.

When sampling the `CPU_CLK_UNHALTED` event, which is distributed evenly among every cycle, the skid merely shifts the profile by a certain amount of cycles. The sample count attributed to each instruction is still proportional to the total amount of cycles it consumes. As a result, this phenomenon does not affect the precision of the collected profile [3]. However, for `INST_RETIRED` event sampling, the effects of skid are important. Figure 2 shows how this effect interacts with a long latency instruction. Because long latency instructions sit at the head of the instruction queue for long periods of time, they are sampled disproportionately more than other instructions. Consequently, instructions that trigger long stalls such as cache or TLB misses will have abnormally higher sample counts compared to other instructions in the same basic block. We refer to this as the *aggregation effect*. These additional samples should have been attributed to instructions *after* the stalled instruction, however since they accumulate on the stalled instruction, instructions in the shadow of the stalled instruction frequently have unusually low sample counts. We refer to this as the *shadow effect*.

INST_RETIRED	CPU_CLK_ UNHALTED	DTLB_MISS	Source
1957	5801	0	m = m + i;
1958	5965	0	m = m + i;
1942	5764	0	m = m + i;
3947	11634	0	x = rand() % size;
68551	340252	1047	m = m + test_v[x];
38	2042	0	m = m + i;
105	5835	0	m = m + i;
13	5846	0	m = m + i;
7	5813	0	m = m + i;
3	5901	0	m = m + i;
3040	5912	0	m = m + i;
2027	5875	0	m = m + i;
2057	5883	0	m = m + i;

Fig. 2. Aggregation Effect due to long latency instructions measured on an Intel Core 2.

Previous work suggests accounting for this phenomenon by approximating the amount of time that an instruction spends at the head of the instruction queue [3]. Unfortunately, estimating this quantity on a modern out-of-order, superscalar processor with a deep cache hierarchy is difficult. In the next section, we show how measuring other performance counters can be used to help correct for this bias.

Modern Intel x86 processors provide *precise event based sampling* (PEBS) which guarantees that the address reported for a counter overflow corresponds to a dynamic instruction that caused the counter to increment. Provided sufficient delay between two back-to-back events, the address reported corresponds to the instruction immediately after the one that overflowed the counter [11]. Unfortunately, when measuring instruction retirement, as the two columns labeled PEBS in Figure 1 show, sampling with PEBS actually yields *lower* accuracy than sampling without PEBS. This occurs due to bursts of instruction retirement events near the counter overflow. These instructions will not be sampled, once again leading to asymmetric sampling. Since PEBS does not support randomized sampling periods, non-PEBS sampling with randomized sampling periods appears to be a more promising approach.

AMD processors, on the other hand, provide *instruction-based sampling* (IBS) which is similar to the ProfileMe approach [12]. Unfortunately, this facility only allows sam-

pling instructions fetched (which include instructions on mispredicted paths) or μops retired (which are at a finer granularity than ISA instructions). Since the number of μops per instruction is unknown, using IBS also proves problematic [13].

3.2.3 Multi-Instruction Retirement

On most modern superscalar processors, more than one instruction can retire in a given cycle. For example, on Intel's Core 2 processor, up to four instructions can retire each cycle. Unfortunately, the interrupt signaling the overflow of a performance counter happens immediately before or after a group of committed instructions, and the performance monitoring software records only one PC associated with the group. Consequently, if a set of instructions always retire together, only one instruction in the group will have samples attributed to it, and these samples will be the aggregation of all the samples for the instructions it retired with. For example, in Figure 1, observe that the `cmp` instruction receives no samples. While the precise cause cannot be known, it is likely because it commits with the instruction immediately following it (due to fused compare and branch in the processor backend). Further, since the other instructions are data-dependent, the instruction with address `0x30` will execute approximately 30-cycles later, and the data shows that it has accumulated additional samples. We find similar effects on other x86 architectures such as AMD.

Fortunately, as Figure 1 shows, this aggregation is frequently contained within a single basic block due to the serialization caused by branches. Consequently, while the sample counts for individual instructions may show significant variation due to this effect, the *basic block* profiles derived by averaging these samples across each block's instructions exhibit significantly less variability.

3.3 Improving Profile Precision

From the previous section, it may seem that profiles derived from PMU sampling will be fraught with inaccuracies. However, as Levin et al. show [17], MCF is an effective algorithm to derive completely consistent basic block and edge profiles from potentially inaccurate basic block profiles. However, as they also demonstrate, the quality of the derived profiles heavily depend on the specific cost functions used in MCF. In general, if the sample counts for a particular basic block are accurate, the corresponding edges in the residual graph used during MCF should be assigned a high cost. Conversely, if the sample count is inaccurate, depending on whether the sample count is too high or too low, the corresponding forward or reverse edge in the residual graph should have a lower cost. Based on the observation that basic blocks are often missed during profiling (and therefore have a profile that is too small), prior work uses a fixed cost for all edges, with forward edges having a significantly lower cost than reverse edges. This section details an alternate approach for assigning edge costs. By sampling multiple performance counters, one can compute a confidence in the accuracy of the profile for a basic block, and estimate if the sample count is too high or too low. As our results indicate, adjusting the cost functions used in MCF according to these predictions significantly improves the quality of the derived profiles.

In our approach, we use heuristics to predict the confidence level of the instruction retired profile for a specific basic block. High confidence means that the basic block sample count is predicted to be close to the real execution count. Basic blocks with low confidence are further divided into two categories, blocks where the sample count is predicted to be larger(smaller) than the true execution count. The basic block classification information is used by the edge cost functions in the MCF algorithm to help make better smoothing decisions.

As was described earlier, there are two principal biasing effects in the INST_RETIRED based profile: the aggregation effect and the shadow effect. Recall that the aggregation effect leads to larger sample counts, and the shadow effect leads to smaller sample counts. However, both these effects usually coexist for a single basic block. Consequently, the goal of the heuristic is to determine which effect, if any, is dominant for a particular basic block.

Recall that aggregation occurs for long-latency instructions. For a fixed skid, D, a unit-latency instruction will be sampled if the instruction that retired D cycles earlier overflowed the performance counter. However, since an instruction with latency L remains at the head of the instruction window between times t and $t + L - 1$, it will be sampled if the counter overflowed anywhere between D and $max(D - L - 1, 1)$ cycles before the instruction issued. Consequently, an instruction's chance of getting sampled increases proportionally to its latency. To model this aggregation, a compiler must estimate the latency of each instruction. However, it is hard to measure latency since stall events are not attributed to the correct instruction due to skid. However, our observations show that most aggregation is caused by instructions that stall for significant amounts of time (e.g., stalling due to a DTLB miss). Events measuring these long stalls are generally unaffected by skid and therefore are attributed to the instruction that caused the overflow of the performance counter. Consequently, the heuristic to model aggregation is restricted to events that lead to significant stalls. The set of such events is selected once when a compiler is being tuned for a specific architecture.

For each such event e, the average stall duration (obtained from processor manuals), stall_duration_e, multiplied by the sample count for the event, $\text{count}_{e,i}$, gives the total number of cycles that a particular instruction i stalled due to event e. Summing over all such stall events for all instructions in a basic block gives us an aggregation factor, A.

$$A = \sum_e \text{stall\_duration}_e \times \left(\sum_{i \in \text{BB}} \text{count}_{e,i} \right)$$

The shadow effect can be modeled by comparing the total number of cycles spent in a basic block (as measured

by sampling CPU_CLK_UNHALTED) to the number of instruction retired events attributed to the block. The difference between these two sample counts is the shadow factor, S. Recall, that in the CPU_CLK_UNHALTED event based profile, sample count should have proper attribution. Consequently, if S is large, two possibilities exist. First, the basic block could legitimately have experienced high CPI. Alternatively, its instruction retirement samples could have been shadowed. In the first case, A should also be large. Consequently if $S \gg A$ then it is likely that the block's samples have been shadowed. In our implementation, if $S - A$ is greater than twice the raw basic block count, the block is classified as under-sampled. Conversely, if $A > S$ and A is a significant fraction of the total number of cycles spent in the block, then it is likely that the block has aggregated too many instruction retirement samples[1]. In our implementation, if $A > S$ and A accounts for more than 50% of the cycles spent in the block, it is classified as over-sampled.

Based on this classification, an MCF prepass is performed on the profile at the instruction level, with adjusted cost function for basic blocks that are predicted to be over-/under-sampled. For over-sampled block, its corresponding forward edge in the residual graph is set as the maximum cost in the CFG, while its reverse edge is set to 0 (and vice-versa for under-sampled basic blocks).

4 Sampling the Last Branch Record

In recent Intel micro-processors, a series of registers are designated to record the last few branches taken, namely Last Branch Record (LBR). This is an extension of the Branch Trace Buffer (BTB), which exists on almost all modern processors. In the Intel Core 2 processor, these registers record the last 4 taken branches, while in the Intel Core i7 (code named Nehalem), the last 16 taken branches are recorded.

In PMU-based sampling, these registers can also be recorded in the interrupt handler. To prevent branches inside the interrupt handler from being recorded, the hardware freezes LBR registers whenever overflow is triggered. In this section, we describe how LBR samples can help derive more accurate instruction frequency profile. We also show how this approach can be used to assist a lightweight inter-procedural optimizer and indirect call promotion.

4.1 Using LBR to Collect Edge Profiles

In our approach, each time an overflow triggers an interrupt, we record all the branches in the LBR buffer. Ideally, each taken branch would be sampled in proportion to its frequency. On Intel Core 2 processors, there is an event called $\texttt{BR\_INST\_RETIRED}_{taken}$ that can serve this purpose. Unfortunately, on the Intel Nehalem processors, this event is not available. However, there is an alternative event called $\texttt{BR\_INST\_EXEC}_{taken}$, which counts in all taken branches including those mis-predicted branches. By subtracting the profile derived from the event $\texttt{BR\_MISP\_EXEC}_{taken}$, which takes effect when a non-taken branch is mis-predicted, we can approximate the original branch retired taken event.

4.2 Using Edge Profiles to Calculate Instruction Profiles

Each taken branch recorded by the LBR corresponds to an edge in the control flow graph (CFG) of the program, and consequently an edge profile can be derived from the LBR samples. However, this profile will only contain frequency information for taken branch edges. We use Algorithm 1 to derive the unknown frequency information for fall through edges. We use the same algorithm to calculate the frequency of each basic block, which is then used to derive the instruction frequency profile.

Algorithm 1 Deriving the basic block frequency from the taken branch frequency

Require: Basic block are labeled according to their topological order
1. **for** $i = 1$ to BB_Count **do**
2. $BB_i.Freq$ = 0
3. **for all** $edge$ in $BB_i.preds$ **do**
4. $BB_i.Freq$ += $edge.Freq$
5. **end for**
6. $BB_i.succs(fall\_through).Freq$ = $BB_i.Freq$
7. **for all** $edge$ in $BB_i.succs$ **do**
8. $BB_i.succs(fall\_through).Freq$ -= $edge.Freq$
9. **end for**
10. **end for**

In this algorithm, the basic blocks are initially labeled by their topological order. The frequencies of the incoming edges are summed to derive the frequency of the basic block. Then the frequency of the basic block, after subtracting the frequencies of all taken out-going edges, is used to find the frequency of the fall-through edge. This algorithm propagates samples of each basic block along the topological order, and finally derives the frequencies of all basic blocks and edges.

The proof of correctness of this algorithm is as follows: note that for each basic block, there should be at most one incoming fall-through edge and at most one out-going fall-through edge, which have unknown frequency in the original profile. First, the basic blocks are labeled in the order they appear in the binary. Assume that for BB_i, all the incoming edges have known frequency. The frequency of BB_i can be easily calculated by summing up frequencies of all its incoming edges. Once the frequency of BB_i is known, the frequency of its only out-going fall-through edge can be calculated because the frequencies of all out-going taken edges can be derived from the LBR profile. So far we can infer that all the incoming edges of BB_{i+1} should have known frequency because the only possible unknown incoming edge is the fall-through edge from BB_i to BB_{i+1}, for which the frequency has already been calculated. In addition, the frequencies of all incoming

1. The aggregation factor A may over-estimate the number of cycles spent in a basic block due to stalls if some of the stalls are overlapped. In such cases, our heuristic may assert that a block has aggregated too many samples when in fact it has not. Our experience has shown that this mischaracterization occurs rarely.

edges of the first basic block (entry block) should be known in the LBR data because they're all function calls. As a result, Algorithm 1 gives the frequency of all edges and basic blocks in one pass of the control flow graph. The time complexity can be represented as $O(N + E)$, where N represents the total number of basic blocks, and E is the total number of edges.

4.3 Using the LBR Profile to Directly Collect an Instruction Profile

In the LBR profile, whenever the counter triggers an overflow, N back-to-back taken branches are recorded. Algorithm 2 can be used to derive an instruction profile directly (without first constructing an edge profile). The idea here is to increment the sample count for instructions between the destination of one taken branch and the source of the subsequent taken branch.

Algorithm 2 Using back-to-back taken branch pair to derive the instruction profile

```
1. for all entry in Sample_file do
2.     for i = 1 to Total_record_per_entry − 1 do
3.         for all inst between entry.record[i].target and
           entry.record[i + 1].source do
4.             inst.count + +
5.         end for
6.     end for
7. end for
```

In this algorithm, all consecutive taken branch pairs in the LBR are examined. The frequency of all instructions between the destination of the first branch and the source of the second branch are incremented. In the end, the total count of each instruction represents the frequency of the instruction.

To prove the correctness of this algorithm, we first label the basic blocks in the binary according to their topological order. We define a path from instruction I_i to I_j as maximum non-taken path P_{ij}, if and only if I_i is the target of a taken branch, and no branch is taken along this path except I_j. Since all the branches in P_{ij} are fall-through branches, we can easily describe P_{ij} as $I_i, I_{i+1}, \ldots, I_j$. In the LBR profile, all instructions between each back to back taken branch pair can be considered a maximum non-taken path.

The total execution count of an instruction is equal to the sum of all maximum non-taken paths that spans it. Following the algorithm, the total sample count of an instruction is equal to the total number of sampled back-to-back edges that span the instruction. Since each taken branch will have an equal opportunity to trigger the counter overflow, each maximum non-taken path will have an equal probability of being sampled. Thus the sample count of a maximum non-taken path (the instructions between a back-to-back branch pair) is proportional to the actual invocation count of that maximum non-taken path. This proves that algorithm 2 can derive the correct instruction profile.

4.4 Accuracy of the LBR-based Instruction Profiles

The LBR profile is also collected using PMU-based sampling, which could potentially suffer from the problems described in Section 3.2. We use random sampling to mitigate the synchronization problem. Current processors can only retire at most 1 branch instruction each cycle, therefore, an LBR-based profile won't have the multi-retirement issue. For the sample skid problem, the LBR-based profile is affected much less than the traditional method because of the following reasons:

1) The skid for LBR sampling is only 10 cycles, which is one third that of the traditional sampling approach.
2) $\texttt{BR\_INST\_RETIRED}_{taken}$ events are far less frequent than `INST_RETIRED` events. Also, most often the intervals between two events are greater than the skid itself.
3) Since the LBR contains multiple entries, the sampling naturally occurs in a burst. This produces more accurate profiles.

As a result, LBR-based sampling can derive much more accurate profiles than the traditional sampling-based approach. This is further verified by the accuracy evaluations in Section 6.1.

The approaches described in Section 4.2 and Section 4.3 are two orthogonal techniques to collect instruction profiles. The accuracy of these two approaches is similar. Thus for later evaluations, we'll only use the approach described in Section 4.3 to demonstrate the quality of LBR-based approaches.

4.5 Sampling-based LIPO

When writing a program, programmers usually divide it into several modules. In C/C++, each .c/.cc file is considered to be one module. Generally, the compiler compiles the program module by module, and the linker links the compiled modules to produce an executable file. In traditional inter-procedural optimization (IPO), a compiler reads in all the modules to carry out a whole-program analysis, which is usually extremely expensive and not scalable.

LIPO [18] is a technique aimed at using a lightweight approach to perform IPO. The basic idea is to identify modules that have large degree of affinity. When compiling inter-procedurally, instead of compiling the whole program, LIPO still compiles the original modules one by one. During compilation of each module, a compiler only reads in the relevant modules as auxiliary modules to assist in compilation of the main module. This method is lightweight in the sense that the compilation model is the same as the traditional intra-module compilation, and it uses the traditional linker, which does not require significant linking time. In addition, whenever a file is modified, only the relevant modules need to be rebuilt. Unfortunately, this lightweight approach is built on top of heavy-weighted instrumentation. Thus a significant amount of overhead is incurred in collecting the profile.

The most important task for LIPO is to build the module grouping policy, i.e. identifying auxiliary modules for each compilation module. This is done by instrumenting each call instruction to get the frequency of the call-edge. If function A frequently calls B, and they're in different modules, LIPO adds the module of B as the auxiliary module of A. This auxiliary module relationship is transitive. As a result, modules along a hot call path will all be included to form an auxiliary module set. In our approach, instead of using instrumentation to collect the edge frequency, we use the LBR profile collected in the previous steps to derive the call-edge frequency. Then we apply the same heuristics that LIPO used to derive the module grouping policy.

4.6 Using LBR-based Sampling to Collect Indirect Call Profiles

LBR can record the frequency of each taken branch including function calls. We use this information to guide an indirect call promotion optimization. First, the binary is disassembled to find all indirect call instructions. For each indirect call instruction, all possible targets are recorded by reading the LBR entries. By looking up the target address in the symbol table, one can easily find the potential targets of an indirect call. A histogram is constructed to identify the most frequent targets of an indirect call, and these are stored as the indirect call profile.

5 Sampling-based Value Profiling

Value profile-based optimizations attempt to compute the distributions of the values for some specific computations. If some values dominate the whole distribution, a compiler will generate multiple versions of the code to handle the special values.

5.1 Value Profiling in GCC

In GCC, there are 3 different types of value profiles:

1) **Indirect Call Promotion** If GCC finds that the indirect call is directing to some functions most of the time, the callsites are cloned, and the expected callees are promoted to direct calls. In this way, these functions have an opportunity to be inlined into the caller. As was mentioned in Section 4.6, by sampling LBR registers, one can collect indirect call profiles.
2) **Stringop Optimization** If the size of the string operations (e.g. memcpy, memset) is constant most of the time, GCC tries to generate multiple versions of the call according to the frequently occurring values. In this way, later compiler optimizations can choose the most efficient way to expand these stringops. For example, if the size is small most of the time, GCC may use inlined assembly code to reduce the call overhead.
3) **Div/Mod Optimizations** The divide and mod operations are expensive. If the dividend is frequently found to be some constant, GCC can use shift operations to reduce the complexity of the computation.

One thing to note is that the current implementation of value profiling in GCC only records the one most frequent value, and may cause the wrong information to be recorded. This is designed to reduce the overhead. One can choose to use more sophisticated algorithms to get more accurate profiles, but this approach could incur significant overhead.

5.2 Using PEBS to Collect Value Profiles

In recent Intel processors, a Precise Event Based Sampling (PEBS) mode is provided. In this mode, when an event triggers an overflow, the system enters a special state; the next time the event occurs, the system captures the current state immediately, and records it in the memory buffer. This buffer is flushed to the disk once it is full or when the program terminates. PEBS guarantees that the state recorded is strictly concurrent with the instruction that triggers the event. In AMD micro-processors, a technique called Instruction Based Sampling (IBS) can perform similar tasks as PEBS sampling.

In PEBS mode, all values within one sample are strictly correlated to the instruction pointer recorded in it. This is important for value profiling since the values need to be mapped back to the source code through the debug information of the recorded instruction address. For non-PEBS-based sampling, only the instruction pointer is recorded by hardware, and thus other register values might be modified by instructions inside the interrupt handler. Using PEBS, we can record the value profiles for stringop optimization and div/mod optimization.

During the sample collection phase, the `INST_RETIRED` event is used to sample the PEBS profile, and thus each instruction should have an equal opportunity to be sampled. We record all register values for each sample. When the collection finishes, the offline analysis tool is used to collect the value profiles. This happens in two steps:

First, the tool disassembles the binary to find potential target instructions and registers. For the stringop operations, we check if it is a call instruction to a stringop functions. Following the x86_64 calling convention, we record the register that saves the actual parameter to represent the size of the stringop. For div/mod operations, we record the dividend register.

Next, the tool clusters all samples by the value of the PC register. A histogram is constructed to compute the few most frequent few values for each instruction found in the previous step. The ratios of the most frequent values are recorded as the value profile of the instruction.

Since the sampling period is controllable, and is usually kept below a threshold, the overhead of recording the system state is negligible.

5.3 Program Slicing to Extend the Value Profiles

Unfortunately, in PEBS sampling, some target instructions are not sampled, even though they are in a hot path. This can be explained by the shadowing and multi-retiring effects as discussed in Section 3.2.

To overcome this problem, we propose the use of program slicing to extend the value profiles beyond the target instruction itself. After looking at the instructions that dominate or post-dominate the target instruction, we can utilize extra samples to assist the target instruction. Since instructions after the target instruction could be the destination of other branches, slicing is only performed in the backward direction. The slicing algorithm traverses backwards from the target instruction until the value of the target register is killed. The samples associated with these instructions are used to derive the value profiles of the target instruction.

6 Evaluation

To evaluate the sample profile, we need to compare it to a correct profile. As was described in Section 1, instrumentation incurs a large overhead which may affect program behavior. However, when the program has deterministic behavior, one can use instrumentation to derive correct edge/basic block frequency profiles and value profiles. For this section, we chose to use the deterministic SPECCPU benchmarks for evaluation. The sample profile is compared to the instrumented profile, which is deemed correct for all SPECCPU benchmarks.

We first evaluate the precision of the sample profiles by comparing the overlap measures, and then show how improved source correlation can be used to improve precision. Additionally, we evaluated the effectiveness of sampling-based FDO by comparing the runtime performance of sample-FDO builds with instrumented-FDO builds. Finally we evaluate the overheads of different profiling mechanisms, and discuss the pros and cons of our approach.

In this section, all binaries were produced using GCC version 4.4.3 targeting x86_64. Our sampling-based FDO framework was also built on top of this compiler. The sample profiles were collected using perfmon2 on an Intel Core2 i7 920 2.67GHz machine with a sampling period of 1 million. Random sampling, with a randomization mask of 0xFFF, was used to improve the quality of the samples. With these parameters, a sample was taken after every 1,000,000 + (`rand()` & 0xFFF) instructions retired. All runtime performance measurements were performed on the same machine that was used to collect profiles.

6.1 Precision of the profile

We used the *degree of overlap* metric [17] to evaluate the quality of the profiles independent of the FDO optimizations with which they will be used. The degree of overlap metric compares the similarity of two edge profiles annotated onto a common CFG. The definition is as follows:

$$\mathrm{PW}(e, W) = \frac{W(e)}{\sum_{e' \in E} W(e')}$$

$$\mathrm{overlap}(W_1, W_2) = \sum_{e \in E} \min\left(\mathrm{PW}(e, W_1), \mathrm{PW}(e, W_2)\right)$$

where W is a map from edges to weights, E is the set of edges in the CFG, and PW computes the normalized weight of an edge. If two profiles agree exactly, the overlap is equal to 1 (or 100%), the sum of the normalized edge weights over the CFG. Conversely, if the profile weights differ for some edge, since the minimum of the two is selected the overlap will decrease. Consequently, the overlap can vary between 0% and 100%.

Figure 3 shows the overlap between the sample profiles and the instrumented profiles for the SPEC CINT2000. The overlap is measured at the binary level, derived by comparing sampled profiles to edge profiles that are derived using Pin [19]. We evaluate binary level overlap to isolate the PMU sampling precision problem from source correlation problems (see Section 6.2), and show how refinements can improve the precision incrementally. The first bar shows the quality of the raw profiles (converted to an edge profile using static profile heuristics [27]). On comparing the first and second bars, we see that, on average, the MCF algorithm (as presented in the literature [17]) improves the overlap by 8.46% compared to static estimation. Comparing the second and third bars, we see that by classifying basic blocks as over-/under-sampled using multiple PMU profiles, precision can be further improved by 7.67%. The fourth bar shows the potential of our refinement approach of classifying blocks as over-/under-sampled using perfect profiles (obtained from Pin) rather than using additional hardware events. Comparing the third and fourth bars shows that our approach performs only 1.21% worse (82.3% vs 83.5%) than using perfect profiles for basic block classification. The LBR-based profile, as shown in the last bar, achieves much better accuracy, achieving an average of 96.95% overlap.

To estimate the potential for further improvement of the non-LBR based approach, we computed the function-level overlap of the sampled profile and the true function profiles obtained using Pin. Function-level overlap is defined identically to edge overlap except that W is a mapping from a procedure to its weight. Since the heuristics used to infer edge profiles from the sample profiles are intra-procedural, the function-level overlap is an upper bound to the edge overlap. The function-level overlap was measured to be 88.03%, thus the smoothed edge profile obtained using our algorithms is within 10% of optimal. The imprecision in the function-level profile can be explained by aggregation/shadow effects crossing procedure boundaries. The overlap when using a more aggressive compiler inline heuristic (which reduces the chances of aggregation/shadowing across procedure boundaries) increases the function-level overlap to 92.37%.

6.2 Issues with Source Position Information

In addition to the challenges imposed by issues inherent to hardware-event sampling, there are other challenges that arise due to inaccuracies in the source position information used to correlate samples to the GCC IR. These challenges, along with our enhancements to the GCC source information, are outlined in this section.

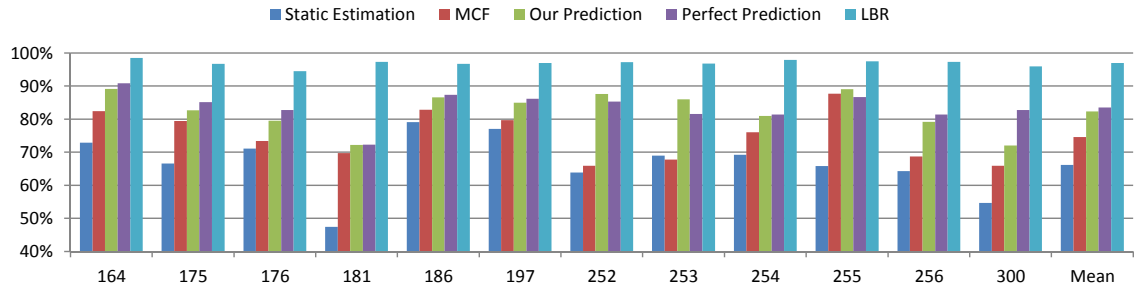


Fig. 3. Edge overlap measures for SPEC CINT2000 benchmarks.

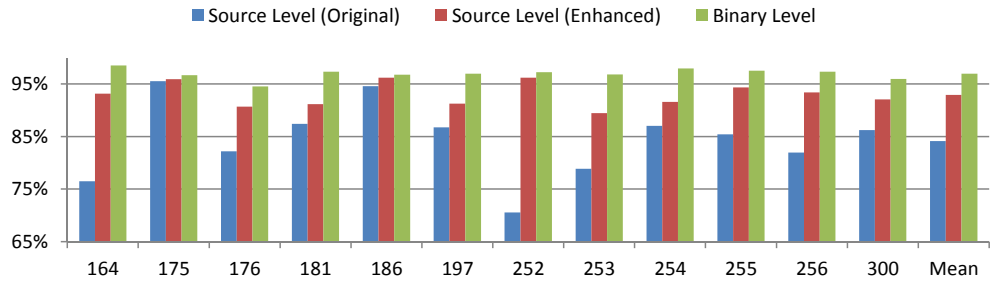


Fig. 4. Edge overlap measures of LBR-based profile at both source and binary level for SPEC CINT2000 benchmarks.

6.2.1 *Insufficient Source Position Information*

One line of source code can embody multiple basic blocks (e.g., consider any use of the ternary `?:` operator). In our current implementation, a patch in GCC and the GNU binutils is applied to distinguish different basic blocks that are mapped to the same line of code. If instructions in different basic blocks are mapped to the same source line, the source information of an instruction is represented by a triplet $(filename, lineno, discriminator)$. The $discriminator$ is a distinct number used to denote the residing basic block of an instruction.

6.2.2 *Over/Under Sampling Due to Optimization*

Optimizations such as loop unrolling etc., cause some statements to be duplicated in different basic blocks in the optimized binary used for profile collection. Because multiple basic blocks in the binary correspond to one basic block in the GCC IR, the profile normalization strategy will cause the profile for these basic blocks to be too low. Conversely, optimizations like if-conversion promote conditionally executed code to unconditionally executed code. This increases the likelihood that it will be sampled thus causing its profile count to be too high. In our implementation, we have special bookkeeping for optimizations that duplicate code. Additionally, when optimizations move an instruction out of its original basic block, we abandon the profile of that instruction to ensure the correctness of the recorded profiles.

6.2.3 *Quantitative Comparison*

Figure 4 shows a quantitative comparison between the source level profile and binary level profile. The profiles were all collected using LBR-based sampling. The source level overlap decreased to 84.13%, whereas the same profile shows an overlap of 96.95% at binary level. However, with our enhancements to the source position information, the overlap measure improves to 92.93%, which is only 4.72% worse than the binary level profile.

6.3 Effectiveness of the framework

Figure 5 shows the speedup obtained by using FDO over a baseline binary compiled without FDO. The baseline and FDO binaries were all compiled using GCC with the `-O2` flag. In this figure, the instrumented FDO has incorporated both value profile and LIPO, and thus represents the peak performance that GCC can achieve.

On average, using profiles collected on an Intel Core2 i7 processor, sampling-based FDO with our refinements provides an absolute speedup of 14.24%. This speedup is due to a series of components in the framework. Initially, using traditional PMU-based sampling to collect frequency profile, as described in [9], approximately 4.35% speedup can be achieved. If we use LBR to collect the profile, the speedup increases to 7.38%. After enabling the lightweight inter-procedural optimizer, the speedup reaches 11.85%. And finally, enabling the value profile-based optimizations boosts the speedup to 14.24%, which is 91.8% of the peak speedup of GCC using instrumentation-based FDO.

Detailed investigation into several benchmarks revealed that most of the performance gap between sampling-based FDO and instrumentation-based FDO can be attributed to the following major causes:

1) Source correlation issues.
   For the 252.eon, the gap between the sample FDO and the instrumented FDO lies in the missing inline information for some frequently invoked call-sites. This

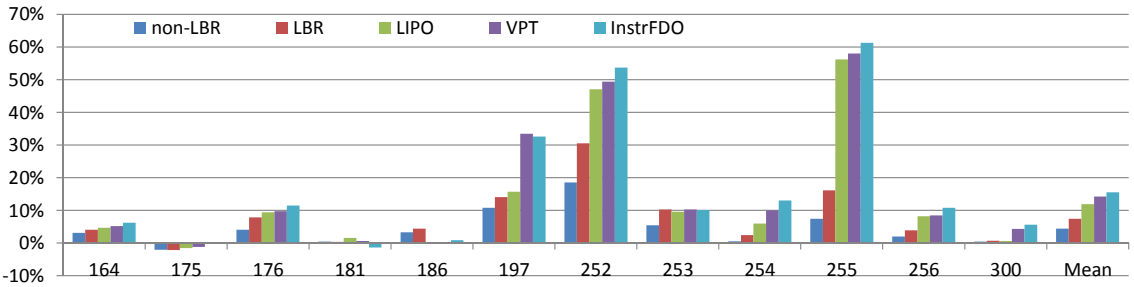


Fig. 5. Speedup of SPEC CINT2000. SampleFDO achieves an average speedup of 14.24%, which is 91.8% of the speedup of instrumented FDO.

is because in the current GCC implementation, only `filename` and `lineno` information is embeded in each level of the inline stack. The `discriminator` is only available at the top of the stack, but not at other levels. As a result, it cannot distinguish between two inlined callsites that reside in the same source line. The performance would no doubt improve once the `discriminator` is better supported in GCC.

2) Statistical sampling issues.
Sample FDO is a statistical approach, which means that sample count for each instruction is scaled down proportionally to avoid the overhead. This will sometimes cause problems when the frequency is too low to get any samples. For example, in 254.gap, a callsite is only invoked for around $1,000$ times. Though not very frequent, it is still beneficial to inline it because the callee is hot, and inlining this callsite can enable other backend optimizations to further optimize the callee. GCC has a heuristic to inline these infrequent but beneficial callsites. However, this heuristic is only effective when the callsite is executed at least once. For the sample FDO, using a sampling period of 1 million, this callsite is too infrequent to get any samples, and thus it is deemed "not executed". As a result, the heuristic fails to inline this callsite, causing performance loss of the 254.gap benchmark.
3) Errors in the profile.
As shown in Section 6.1, the LBR-based approach can already obtain very accurate profiles, but this approach still has an error ratio of less than 5%. This could cause problems when accurate information is vital for compiler optimizations. Further study of an industrial application shows that loop unrolling is one such optimization. The instrumentation-based profile can derive exact loop trip counts, while the trip counts derived using the sample-based profile are sometimes off by a small amount. As a result, with instrumented FDO, a loop may be fully-unrolled for most frequent situations, while in sampled FDO it may have some "left-over" iterations that degrade the performance. Tuning the compiler's unrolling heuristics for the sample-based behavior could potentially ameliorate this problem.

One thing to note is that the above evaluations are not cross-validated. However, they are good indicators of the effectiveness of our approach because FDO (both instrumented and sample-based) performs best when the input data used for profile collection is also used for performance evaluation. To make the evaluation complete, we cross-validated the performance improvements on SPEC CINT2006 benchmarks. "Train" data sets are used to collect both sampled and instrumented profiles. These profiles are used in the FDO builds and performance is measured using the "Ref" data sets. As a comparison, the non-cross-validated setup is also performed on SPEC CINT2006 benchmarks, in which "Ref" data sets are used to collect both sampled and instrumented profiles, and the performance is measured using "Ref" data sets. As shown in Figure 6, for the non-cross-validated version, sampling-based FDO can achieve 91.9% speedup of instrumentation-based FDO. In the cross-validation, sampling-based FDO can achieve 85.2% speedup of instrumentation-based FDO, which is less than the non-cross-validated version because "Train" data sets run for a much shorter period of time than "ref" data sets; this introduces more statistical errors for less sampled instructions. However, as soon as the program is sampled for enough time, the sampling-based approach can always derive representative profile for FDO.

TABLE 1
Overhead of profile collection using different approaches, with sampling rate at 1 million.

Benchmark	Our Approach	GCC -fprofile-generate	PIN
164.gzip	1.7%	40.1%	51x
175.vpr	0.9%	15.6%	44x
176.gcc	1.1%	66.2%	87x
181.mcf	0.6%	2.0%	14x
186.crafty	1.8%	103.9%	95x
197.parser	0.8%	52.2%	62x
252.eon	1.0%	240.6%	71x
253.perlbmk	-0.5%	165.6%	128x
254.gap	1.0%	351.5%	101x
255.vortex	1.6%	208.8%	150x
256.bzip2	0.6%	41.3%	45x
300.twolf	1.4%	50.7%	31x
Geomean	0.9%	90.5%	62x

6.4 Profiling Overhead

We also evaluated the overhead incurred by profile collection. As shown in Table 1, on the SPEC benchmark-

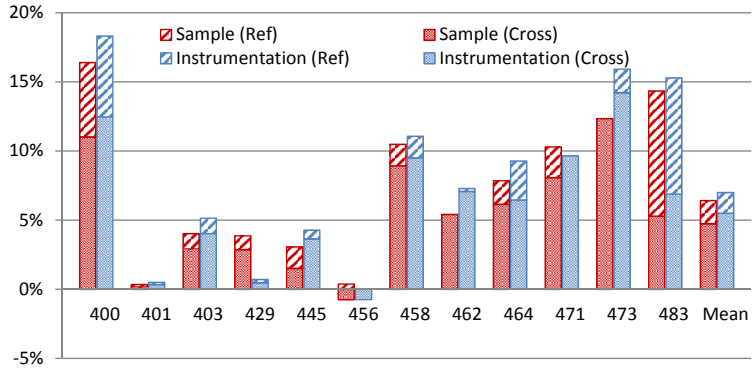


Fig. 6. Cross-validation of the speedup for SPEC CINT2006 benchmarks. The sample profile collected using "Ref" input data can obtain an average speedup of 6.42%, which is 91.9% of the instrumentation-based approach. The sample profile collected using "Train" input data can obtain an average speedup of 4.69%, which is 85.2% of the instrumentation-based approach.

s, using a sampling rate of 1 million, the overhead of PMU-based sampling never exceeds 1.8% and averages 0.9%. Compiler-based instrumentation incurs an overhead between 2.0% and 351.5%, and dynamic instrumentation tools, such as Pin [19], incur an overhead between 14x and 150x. On an industrial web search application, the compiler-based instrumentation suffered a 10x overhead, compared to just over 2% overhead when profiled using hardware PMU sampling.

6.5 Discussion

As was demonstrated in the performance evaluation, sample FDO shows speedup that is competitive with the instrumentation-based approach. However, these two approaches of collecting profiles are not equivalent. First, instrumentation may insert extra code to collect flexible profiles that is hard to archive by the sampling-based approach. For example, path profiling can be collected using instrumentation. Though LBR can also be used to collect a partial path profile, because of the limited branch entries, it cannot collect the full path profile. Second, sampling the PMU can provide some information that instrumentation cannot get. For example, by sampling cache miss events, sample FDO can derive the locality information and perform aggressive locality optimizations. In this paper, we focus on traditional profiles that have been adopted by instrumented FDO. The evaluation is also intentionally biased in favor of instrumented FDO by choosing deterministic benchmarks. We leave the study of profiles that sample FDO can use to out-perform instrumented FDO for future work.

7 Related Work

In a recent paper, Levin, Newman, and Haber [17] use sampled profiles of the instruction retirement hardware event to construct edge profiles for feedback-directed optimization in IBM's FDPR-Pro, post-link time optimizer. The samples can be directly correlated to the corresponding basic blocks without using source position information, as this is done post-link time. As is done in this paper, the problem of constructing a full edge profile from basic block sample counts is formalized as a Minimum Cost Circulation problem. In this paper, we extend their work by applying sampling to higher level compilation (as opposed to post-link optimization) and show how sampling additional performance counters can improve the quality of sample profiles.

Others have proposed sampling approaches without relying on performance counters. For example, the Morph system [28] collects profiles via statistical sampling of the program counter on clock interrupts. Alternatively, Conte et al. proposed sampling the contents of the branch-prediction hardware using kernel-mode instructions to infer an edge profile [10]. In particular, the tags and target addresses stored in the branch target buffer (BTB) serve to identify an arc in an application, and the branch history stored by the branch predictor can be used to estimate each edge's weight. Both of these works require additional information to be encoded in the binary to correlate instruction-level samples back to a compiler's IR rather than using source position information present in unstripped binaries. Additionally, neither work investigates the intrinsic bias of the sampling approach nor attempts to correct the collected profiles heuristically.

Other profiling methods build on ideas from both program instrumentation and statistical sampling. For example, Traub, Schechter, and Smith propose periodically inserting instrumentation code to capture a small and fixed number of the branch's executions [25]. A post-processing step is used to derive traditional edge profiles from the sampled branch biases collected. Their experiments show that the derived profiles show competitive performance gains when compared with using complete edge profiles to drive a superblock scheduler. Rather than dynamically modifying the binary, others have proposed a similar framework that performs code duplication and uses compiler-inserted counter-based sampling to switch between instrumented and non-instrumented code in a controlled, fine-grained manner [16]. Finally, stack sampling has been used, without the use of

any instrumentation, to implement a low-overhead call path profiler [14].

Similarly, there have been proposals that combine instrumentation and hardware performance counters. Ammons, Ball, and Larus proposed instrumenting programs to read hardware performance counters [2]. By selecting where to reset and sample the counters, the authors are able to extract flow and context sensitive profiles. These profiles are not limited to simple frequency profiles. The authors show, for example, how to collect flow sensitive cache miss profiles from an application.

Besides the frequency profiling, instrumentation based value profiling has been proven to be useful [8], [15], and has been adopted in some compilers such as GCC. However, the instrumentation based approach suffers from excessive overhead and potential inaccuracy introduced by replacement policy. Sampling based value profiling is proposed to pursue better efficiency and flexibility [7]. However, it still incurs an average overhead of around 10%.

Not surprisingly, performance counter sampling has also been used in the context of just-in-time (JIT) compilation. For example, Schneider, Payer, and Gross sample cache miss performance counters to optimize locality in a garbage collected environment [23]. Like our work, the addresses collected during sampling have to be mapped back to the source code (in their case, Java bytecode). However, since their optimizations were implemented in a JIT, they simply augmented the information stored during dynamic compilation to perform the mapping.

Specialized hardware has also been proposed to facilitate PMU-based profiling. ProfileMe was proposed hardware support to allow accurate instruction-level sampling [12] for Alpha processors. AMD adopts the ProfileMe approach in the Opteron processors. As discussed in Section 3.2, it cannot produce profiles accurate enough for compiler use. Merten et al. also propose specialized hardware support for identifying program hot spots [20]. Unfortunately, the hardware they propose is not available in today's commercial processors.

Orthogonal to collecting profiles, recent work has studied the stability and accuracy of hardware performance counters [21], [26]. In that work, the authors measured the total number of instructions retired across a range of benchmarks on various x86 machines running identical binaries. Their results show that subtle changes to the heap layout, the number of context switches and page faults, and differences in the definition of one instruction can lead to substantial variability in even the total number of instructions retired as reported by the performance counters. Unfortunately, the authors do not study the artifacts in sampling the performance counters, and the results on the aggregate data do not explain the anomalous behavior observed in our experiments.

8 Conclusion

We designed and implemented a framework to use hardware event sampling and source position information to drive feedback-directed optimizations. Both frequency profile and value profile are implemented, making sampling-based FDO achieving good overlap with the true execution frequencies and competitive speedups when compared with the instrumentation-based approach. Moreover, sampling-based FDO provides better portability and usability while incurring negligible overhead. Our experiments show that the proposed techniques are feasible for production use on out-of-order platforms.

References

[1] Ali-Reza Adl-Tabatabai, Richard L. Hudson, Mauricio J. Serrano, and Sreenivas Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 267–276, New York, NY, USA, 2004. ACM.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1997. ACM.

[3] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, 1997.

[4] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.

[5] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[6] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[7] M. Burrows, U. Erlingsson, S-T. A. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl. Efficient and flexible value sampling. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 160–167, New York, NY, USA, 2000. ACM.

[8] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 259–269, Washington, DC, USA, 1997. IEEE Computer Society.

[9] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for fdo compilation. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 42–52. ACM, 2010.

[10] Thomas M. Conte, Burzin A. Patel, Kishore N. Menezes, and J. Stan Cox. Hardware-based profiling: An effective technique for profile-driven optimization. *International Journal of Parallel Processing*, 24(2):187–206, 1996.

[11] Intel Corporation. *Volume 3B: System Programming Guide, Part 2*, 2008.

[12] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[13] Paul J. Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. Technical report, Advanced Micro Devices, Inc., November 2007.

[14] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM.

[15] Freddy Gabbay and Avi Mendelson. Can program profiling support value prediction? In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 270–280, Washington, DC, USA, 1997. IEEE Computer Society.

[16] Nick Gloy, Zheng Wang, Catherine Zhang, J. Bradley Chen, and Michael D. Smith. Profile-based optimization with statistical profiles. Technical report, Harvard University, April 1997.

[17] Roy Levin, Ilan Newman, and Gadi Haber. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *HiPEAC'08: Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, pages 291–304, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] David Xinliang Li, Raksit Ashok, and Robert Hundt. Lightweight feedback-directed cross-module optimization. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 53–61. ACM, 2010.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[20] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen-mei W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 136–147, Washington, DC, USA, 1999. IEEE Computer Society.

[21] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of java profilers. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 187–197, New York, NY, USA, 2010. ACM.

[22] Vinodha Ramasamy, Paul Yuan, Dehao Chen, and Robert Hundt. Feedback-directed optimization in gcc with estimated edge profiles from hardware event sampling. In *GCC Developers' Summit*, 2008.

[23] Florian T. Schneider, Mathias Payer, and Thomas R. Gross. Online optimizations driven by hardware performance monitoring. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 373–382, New York, NY, USA, 2007. ACM.

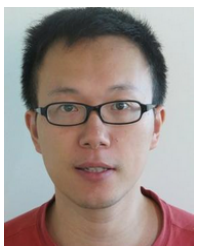
[24] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 269–280, New York, NY, USA, 2010. ACM.

[25] Omri Traub, Stuart Schechter, and Michael D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard University, June 2000.

[26] Vincent M. Weaver and Sally A. McKee. Can hardware performance counters be trusted? In *IEEE International Symposium on Workload Characterization*, pages 141 – 150, September 2008.

[27] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11, New York, NY, USA, 1994. ACM.

[28] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automatic profiling and optimization. *SIGOPS Operating Systems Review*, 31(5):15–26, 1997.



Dehao Chen Dehao is a software engineer at Google. He is working on performance of Google applications, developing scalable mechanisms to make feedback directed optimizations useful in the production environment. He received the B.S in Huazhong University of Science and Technology, M.S and Ph.D in Tsinghua University.

Neil Vachharajani Neil Vachharajani is a Software Engineer for Pure Storage where he is working on next generation high performance storage systems. Previously, he was a compiler engineer at Google investigating application performance for warehouse scale computers. His research interests include I/O performance, compiles, computer architecture, and programming languages focused on concurrency and multi-core architectures. Neil received his Ph.D. from Princeton University where he was an NSF graduate fellow. Neil also holds a BSE in Electrical Engineering and an MA in Computer Science from Princeton University.

Robert Hundt Robert is a Tech Lead at Google. He is working on GMail and datacenter performance, building fleet-wide in-depth analysis tools and infrastructure. He is heavily involved in compiler and performance research. Robert has a Diplom Univ. in computer science from the Technical University in Munich.

Xinliang Li David Li leads the compiler optimizer group in Google. He is working on advanced Profile Directed Optimizations and highly scalable cross module optimizations that can be deployed in Google's build environment. He received the B.S and M.S degrees (both in EE) from Wuhan University and McMaster University respectively.

Stephane Eranian Stephane Eranian is a software engineer in the Linux kernel team at Google. For several years now, he has been working on improving the Linux kernel support for hardware-based performance monitoring. He is the author of the perfmon2 subsystem. Nowadays, he is a contributor to the perf_event subsystem. Prior to joining Google, Stephane was at HPLabs where he worked on the port of Linux to the Intel Itanium processors. He is the co-author of the book entitled "IA-64 linux kernel design and implementation".

Wenguang Chen Wenguang Chen received the B.S. and Ph.D. degrees in computer science from Tsinghua University in 1995 and 2000 respectively. He was the CTO of Opportunity International Inc. from 2000-2002. Since January 2003, he joined Tsinghua Univeristy. He is now a professor and dupty head in Department of Computer Science and Technology, Tsinghua University. His research interest is in parallelizing compiler, programming model, computer system performance analysis and distributed computing.

Weimin Zheng Weimin Zheng received the masters degree from Tsinghua University in 1982. He is now a professor in the Department of Computer Science and Technology at Tsinghua University. His research interests include parallel and distributed computing, compiler technique, grid computing, and network storage.