Taming Hardware Event Samples for FDO Compilation

Dehao Chen[1] Neil Vachharajani[2] Robert Hundt[2] Shih-wei Liao [2] Vinodha Ramasamy
Paul Yuan[3] Wenguang Chen[1] Weimin Zheng[1]
[1]Tsinghua University, [2]Google, [3]Peking University
chendh05@mails.tsinghua.edu.cn, {nvachhar, rhundt, sliao}@google.com, vinodha23@gmail.com,
yingbo.com@gmail.com, {cwg,zwm-dcs}@tsinghua.edu.cn

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 to generate estimated edge profiles overcomes these drawbacks. 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 multiple hardware event profiles and supervised learning techniques to generate heuristics for improved precision of basic-block-level sample profiles, and to further improve the smoothing algorithms used to construct edge profiles. We demonstrate that sampling-based FDO can achieve an average of 78% of the performance gains obtained using instrumentation-based exact edge profiles for SPEC2000 benchmarks, matching or beating instrumentation-based FDO in many cases. The overhead of collection is only 0.74% on average, while compiler based instrumentation incurs 6.8%–53.5% overhead (and 10x overhead on an industrial web search application), and dynamic instrumentation incurs 28.6%–1639.2% overhead.

Categories and Subject Descriptors C.4 [*PERFORMANCE OF SYSTEMS*]: Reliability, availability, and serviceability; C.4 [*PERFORMANCE OF SYSTEMS*]: Modeling techniques; D.3.4 [*PROCESSOR*]: Optimization; D.3.4 [*PROCESSOR*]: Compilers

General Terms Algorithms, Design, Performance

Keywords Feedback-Directed Optimization, Sampling Profile, Performance Counters

CGO'10, April 24–28, 2010, Toronto, Ontario, Canada.

1. Introduction

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

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

Unfortunately, there are several 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, thereby preventing reuse of previous profile collection. For example GCC requires that both builds use the same inline decisions and similar optimization flags to ensure that the control-flow graph (CFG) that is profiled in the instrumentation build matches the CFG that is annotated with the profile data in the FDO build.

Third, collecting the profiles requires the 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.

Fourth, the instrumentation build of an application typically incurs significant overhead (reported as 9% to 105% [3, 4], but observed to be as much as 10x on an industrial web search application) due to the additional instrumentation code that is 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. To overcome these limitations, we propose skipping the instrumentation step altogether, and instead rely on sampling events generated by the performance monitoring units (PMU) of modern processors to obtain estimated edge profiles. The sample data does not contain any information on 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.

This approach has two key benefits. First, 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, thus eliminating dual compilation in the normal workflow. Second, the overhead of profile collection is significantly lower since no instrumentation code is inserted, 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 use cases. For example, in an Internet company, profile collection can occur by infrequently attaching 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.

Using hardware performance monitoring events to estimate execution profiles is, however, not a panacea. First, sampling provides instruction frequencies, rather than edge frequencies, and it has been shown that it is not possible to transform statement profiles into exact edge profiles in general [15]. Second, to avoid having performance monitoring slow the processor's execution, many tradeoffs are made in the design of modern PMUs leading to imprecise sample attribution. The instruction address associated with an event by the PMU is often not the true address at which 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.

In this paper, we present methods to mitigate these problems and use heuristics to derive relative basic block and edge frequency count estimates from the sample profiles. Below, we summarize the primary contributions of this work.

1. We develop a machine learning approach to identify the hardware event most closely correlated to the true execution frequency of program instructions.
2. We identify hardware effects which negatively influence sample distribution, namely synchronization, sample skid, and aggregation/shadow effects.
3. We introduce a hueristic approach, based on sampling multiple hardware events, that mitigates the systematic bias introduced by these hardware effects. Specifically, we show how sample profiles from ancillary hardware events can be used to predict which basic blocks are over/under-sampled, and how this prediction can be used to tune parameters in MCF, the algorithm used to smooth inconsistencies in the primary sample profile.
4. We build a framework to study the limits of the accuracy that can be achieved with the currently available sampling quality and intra-procedural analysis scope.
5. Finally, we present an evaluation of the efficacy of the proposed approach. We present results from an implementation of sample-based FDO in the GCC compiler. Overall, we show that PMU sampling-based FDO, combined with the proposed smoothing heuristics, can achieve 78% of the performance gains obtained using instrumentation-based FDO for SPEC2000 benchmarks. However, sampling-based FDO, on average, incurs only a 1% profiling overhead (2.47% in the worst case) as compared to the 22% profiling overhead (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 estimate a basic block profile and derive an estimated edge profile. Section 3 then describes anomalies observed in the raw PMU samples. Section 4 proposes several heuristics to improve the quality of the edge profiles inferred from the raw data. Section 5 then describes the experimental evaluation of PMU sampling-based FDO. Section 6 describes related work in the area. Finally, Section 7 discusses conclusions and future work in sampling-based FDO.

2. Inferring Profiles with the PMU

This section describes how sampling works with most modern performance monitoring units, and how PMU sampling can be used to devise an edge profile for an application.

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.

Sampling a counter that increments each time an instruction retires (e.g., `INST_RETIRED` on x86 processors) provides a natural way to estimate a basic block profile. Each time the counter overflows, the PC is recorded. Then, for each basic block, the sample counts for all the instructions in the basic block are summed and normalized by the number of instructions in the block. This guarantees that large basic blocks do not receive higher profile weights than smaller blocks. In the literature, this approach to sampling has been called *frequency-based* sampling [22]. An alternative to this approach is *time-based* sampling [22], 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. Section 4.1 compares both approaches and confirms the hypothesis that the frequency-based approach most closely approximates the true basic block frequencies. The remainder of section 4 examines how other counters may be used to correct for anomalies observed in the frequency-based sample profiles.

2.2 Using the Profile in the Compiler

For the sampling-based profile to be usable by the 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, the compiler reads the 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 5.2, 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.

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 reuse of the collected profiles. For example, when there are minor source code changes between profile collection and the FDO build, 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.

2.3 Constructing Edge Profiles

Due to errors and noise in sampling, the basic block counts obtained via sampling may not be consistent. That is to say, for a given block, its sample count will not always equal the sum of the sample counts of its successor or predecessor blocks. 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 4). This refined profile is used to create the profile feedback file. Second, after reading the profile feedback file, the compiler uses MCF to translate the basic block profile into an edge profile. The details of formulating the basic block to edge profile conversion problem as an MCF problem can be found in the literature [12, 16]. 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. However, 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 higher cost. We exploit this property of MCF in Section 4.

3. 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

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**

Figure 1. The sample counts measured on an Intel Clovertown for a loop consisting of one basic block.

of numbers labeled Fixed Sample Period in the figure show the actual samples collected on an Intel Clovertown 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 4 will then introduce various approaches to achieve more precise profiles at both the basic block level and CFG level. We observed similar effects on a variety of architectures from Intel and AMD.

3.1 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

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;

Figure 2. Aggregation Effect due to long latency instructions measured on an Intel Clovertown.

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 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 [7]. On an Intel Clovertown 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 using time-based sampling, this phenomenon is not important as it only skews the sampling period [2]. However, for frequency-based 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*.

Previous work suggests accounting for this phenomenon by approximating the amount of time that an instruction spends at the head of the instruction queue [2]. 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 [6]. 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 [7]. Unfortunately, this facility only allows sampling 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 [8].

3.3 Multi-Instruction Retirement

On most modern superscalar processors, more than one instruction can retire in a given cycle. For example, on Intel's Clovertown 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 committing 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 preceding it (they are not data dependent) or 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.

4. 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 MCF is an effective algorithm to derive completely consistent basic block and edge profiles from potentially inaccurate basic block profiles [12]. 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.

4.1 Choosing the Profiles

As was discussed in Section 2, there are two primary approaches for obtaining a sample profile using hardware-event sampling, the frequency-based approach and the time-based approach. More generally, any of the myriad hardware events exposed by the PMU can be used to derive a sample profile. Consequently, it is unclear which event is best for estimating the execution count of basic blocks. We propose using machine learning during compiler tuning to find the most relevant events automatically. We use linear support vector regression (SVR) [11] to quantify how various hardware events correlate with the execution count of a basic block. SVR is similar to the common least-squares linear regression, but uses a different cost function for evaluating the deviation of predictions [18]. SVR is applied to a training set of hardware event values and the exact execution counts of basic blocks obtained through instrumentation (note, the instrumentation is only necessary when *training* the regression model). Given a training set with the true execution frequency of a basic block, and the normal-

Event	Mask	Counter Incremented	Weight
INST_RETIRED	None	when an instruction retires	0.43
INST_RETIRED	PEBS	when an instruction retires	0.272
INST_RETIRED	0	when no instruction retires in a cycle	-0.1247
INST_RETIRED	4	when 4 instructions retire in 1 cycle	-0.1887
CPU_CLK_ UNHALTED	None	each CPU cycle	0.2131
DTLB_MISS	None	when there is an DTLB miss	-0.1124
L1I_MISS	None	when there is an L1 I-Cache miss	0.0092

Table 1. Events and related weights from the SVM regression model.

ized values of various sampled performance counters, SVR attempts to find a vector of weights such that

$$F \approx \sum_i w_i c_i + b$$

where F is the true execution frequency of a block, w_i is the weight for the i^{th} sampled event, c_i is the corresponding sample count, and b is a constant offset. The absolute value of a weight signifies how well the particular sampled event correlates with the true execution frequency; the sign of the weight indicates whether the correlation is positive or negative.

Table 1 shows the results of applying this approach with the SPEC CINT2000 benchmarks used as training data. Four different hardware events were sampled, and the INST_RETIRED event was configured with 4 different masks leading to a total of 7 different profiles.

As expected, sampling the INST_RETIRED event with randomization has the best correlation to the true execution frequency of a basic block. The DTLB miss event has a negative weight because it leads to many cycles of stall, and consequently leads to aggregation effects. Other events such as zero and multiple instruction retirements result in a negative factor because of the aggregation effect. The CPU_CLK_UNHALTED profile has a positive factor, but it is less significant than random sampling of the INST_RETIRED event since, as was discussed earlier, time-based sampling correlates with execution time not execution frequency. As the micro-benchmark from Section 3 showed, using precise event based sampling (PEBS) on the instruction retired event has a lower positive factor than the corresponding event without PEBS.

The automatically trained model shows which events could serve as the principal ones to sample to estimate basic block frequencies, and it also provides information about which events can be used to supplement the principal profile. Unfortunately, the SVM model cannot directly be used to convert a collection of profiles into a basic block frequency estimate because of the regression constant b. This constant implies that the estimated frequency of a block is non-zero even if no sample (across all the measured events) was attributed to it. Since there are many blocks in a program which truly do not get executed, using the model directly would yield poor results. The next section describes an alternate strategy for using additional hardware events to supplement the primary count.

4.2 Classifying Basic Blocks

Since the instruction retired event with random sampling showed highest correlation to the actual execution frequency of a basic block, we chose it as the base profile to estimate basic block counts. Here, we present 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 $D-L-1$ cycles before the instruction issued. Consequently, an instruction's chance of getting sampled increases proportionally to its latency. To model this aggregation, the 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 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 instruc-

tion retired events attributed to the block. The difference between these two sample counts is the shadow factor, S. Recall, that the delay in attribution does not affect time-based sampling, implying that the CPU_CLK_UNHALTED 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 binary level profile, with adjusted cost function for basic blocks that are predicted to be over-/under-sampled. For over-sampled blocks, 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).

5. Experimental Results

We evaluated the framework described in the previous sections by comparing the quality of refined sample profiles to raw sample profiles and instrumentation profiles. Additionally, we evaluated the performance of sampling-based FDO by comparing the runtime performance of sample-FDO builds with instrumented-FDO builds. All binaries were produced using GCC version 4.3.2 targeting an x86_64. The sample profiles were collected using perfmon2 on an Intel Core2 Quad 2.4 GHz machine with a prime sampling period of 202001. 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 202001 + (`rand()` & 0xFFF) instructions retired. All runtime performance measurements were run on the Intel Core2 Quad 2.4GHz machine used to collect profiles.

5.1 Precision of the profile

We used the *degree of overlap* metric [12] 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 first four bars are measured at binary level, which are derived by comparing sampled profiles to edge profiles derived using Pin [13]. We evaluate binary level overlap to isolate the PMU sampling precision problem from source correlation problems (see Section 5.2), 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 [21]). Comparing the first and second bar shows that, on average, the MCF algorithm (as presented in the literature [12]) improves the overlap by 17.8% when compared with static estimation. Comparing the second and third bars shows that by classifying basic blocks as over-/under-sampled using multiple PMU profiles, the precision can be further improved by 5.5%. The fourth bar shows the potential of our refinement approach by 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 is only 2.3% worse (80.5% to 82.8%) than using perfect profiles for basic block classification. However, as shown by the 5th bar, when the profile is transformed to the source level and used to annotate the CFG in GCC, the precision decreases by 4.6% due to source correlation problems (see Section 5.2).

To estimate the potential for futher improvement, 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 the 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 91.7%, making the smoothed edge profile obtained using our algorithms 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 94.1%. These results suggest that inter-procedural smoooothing algorithms may be a promising avenue of future research.

[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, if at all.

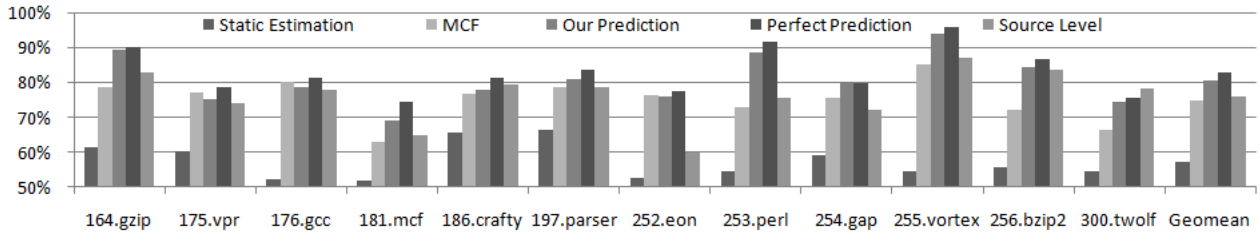

Figure 3. Edge overlap measures for SPEC CINT2000 benchmarks. Sampled FDO reaches an overlap of 80.5%.

Finally, to experimentally verify that using INST_RETIRED as the primary profile is best, we measured the overlap using other hardware events. Since our enhancements to MCF are tuned specifically for the INST_RETIRED hardware event, for fairness, the results presented here were obtained using the original MCF algorithm. As expected, using INST_RETIRED with randomization achieved the best average overlap (75.1%), while using PEBS is slightly worse (74.2%), and using CPU_CLK_UNHALTED achieved an average overlap of (72.6%) even with randomization.

5.2 Issues with Source Position Information

In addition to the challenges imposed by issues inherent to hardware-event sampling, there are other challenges due to inaccuracies in the source position information used to correlate samples to the compiler's IR. These challenges are outlined here.

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, samples originating from instructions corresponding to such lines of code will be attributed to all of the corresponding basic blocks in the compiler's IR instead of the specific block for the instruction. We currently use source formatting to mitigate this issue, and, in the future, will rely on basic-block discriminators[2] to distinguish the different code regions.

Missing/Incorrect Source Position Information Source formatting does not help in cases where there is incorrect source position information. For example, even if each clause in a `?:` expression is on a separate line, GCC attributes all the code for the expression with the first line. In other cases, source position information is completely lost during optimization [16].

Over/Under Sampling Due to Optimization Optimizations such as loop unrolling etc., lead to some statements being duplicated in different basic blocks in the optimized binary used for profile collection. Because the multiple basic blocks in the binary correspond to one basic block in the compiler's 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.

[2] Currently being implemented in GCC.

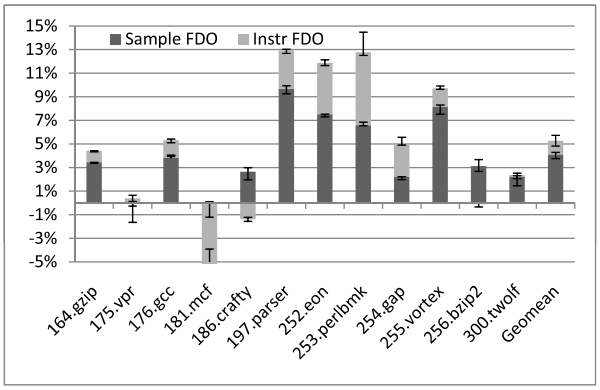

Figure 4. Speedup for SPEC CINT2000 benchmarks. Sampled FDO achieves 78% of instrumented FDO.

5.3 Effectiveness of the framework

The true measure of quality for the profiles is how well they enable feedback-directed optimizations. Figure 4 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.

On average, using profiles collected on an Intel Core2 processor, sample-based FDO with our refinements provides an absolute speedup of 4.106%. This is 78% of the speedup obtained by instrumentation-based FDO.

For some benchmarks (e.g. 186.crafty), sample-based FDO outperforms its instrumentation-based counterpart. Since many feedback-directed optimizations in GCC are driven by threshold based heuristics, this difference is not surprising as subtle differences in the profile can lead to substantially different optimization decisions.

Detailed investigation into several benchmarks revealed that most of the performance gap between sample-based FDO and instrumentation-based FDO can be attributed to source correlation problems. For 181.mcf, instrumentation based FDO suffers a significant performance loss compared to the baseline binary. Code layout decisions change a conditional back edge jump in the baseline binary to a conditional loop exit followed by an unconditional back edge jump in an important loop. The latter code sequence suffers from sig-

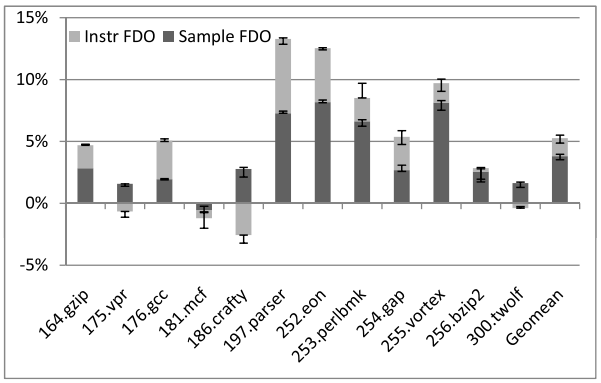

Figure 5. Cross-validation of the speedup for SPEC CINT2000 benchmarks. Sampled FDO achieves 72% of instrumented FDO.

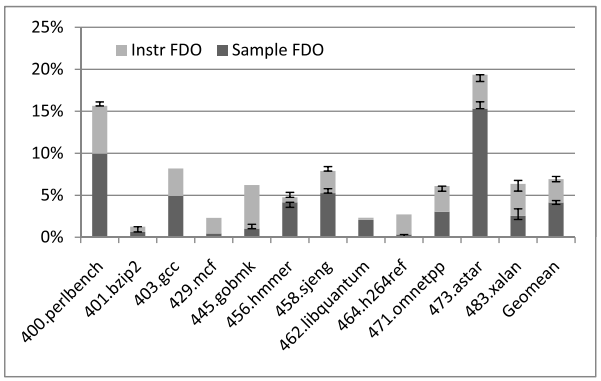

Figure 6. Cross-validation of the speedup for SPEC CINT2006 benchmarks. Sampled FDO achieves 60% of instrumented FDO.

nificantly higher branch misprediction leading to the performance degradation. For the 252.eon and 253.perlbmk, the gap between the instrumentation-based approach and sampling-based approach is due to frequent use of the ternary(?:) operator in one of the hottest functions in the benchmark. Unfortunately, samples for all instructions participating in the statement will be allocated to a single source line (even after source formatting) even though it corresponds to several basic blocks. These benchmarks' performance would no doubt improve with better source position information. Further study of an industrial application shows that loop unrolling is another important source for the performance gap between the two approaches. The instrumentation based profile can derive exact loop trip counts, while the trip counts derived using the sampling based profile is often 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 unroll heuristics for the sample based behavior could potentially ameliorate this problem.

One thing to note is that the above evaluations were 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. However, to make the evaluation complete, we cross-validated the performance improvements on both SPEC CINT2000 and 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 shown in Figures 5 and 6, sampling based FDO can achieve 72% and 60% speedup of instrumentation based FDO, respectively.

We also evaluated the overhead incurred by profile collection. Using a sampling rate of 202001, the overhead of sampling ranges from 0.44% to 2.47%, averaging 0.74%. On the SPEC benchmarks, compiler-based instrumentation incurs an overhead between 6.8% and 53.5%, and dynamic instrumentation tools, such as Pin, incur an overhead between 28.6% and 1639.2%. 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. Related Work

In a recent paper, Levin, Newman, and Haber [12] 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 [22] 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 [5]. 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 the 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. We do however believe that edge sampling is a promising approach and are evaluating extending our infrastructure using hard-

ware support for branch recording (for example the LBR stack on Intel Core2 processors) to enable the approach with unmodified commodity operating systems.

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 [19]. 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 [10]. Finally, stack sampling has been used, without the use of any instrumentation, to implement a low-overhead call path profiler [9].

Similarly, there have been proposals that combine instrumentation and hardware performance counters. Ammons, Ball, and Larus proposed instrumenting programs to read hardware performance counters [1]. 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.

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 [17]. 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 [7] for Alpha processors. Merten et al. also propose specialized hardware support for identifying program hot spots [14]. 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 [20]. 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

7. Conclusion and Future Work

We designed and implemented a framework to use hardware event sampling and source position information to drive feedback-directed optimizations. By using multiple profiles and supervised learning to refine the profile precision, sampling-based FDO can achieve 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, and the precision/performance can be further improved with more precise source position information.

The results presented here represent an initial implementation. Our ongoing work is exploring the possibility of using algorithms other than MCF to refine the precision of the profile in the CFG. We are also investigating heuristics to avoid precision loss due to code duplicating optimizations.

Further, while our current implementation focuses on generating edge profiles, we plan on exploring using other types of profiles, such as cache miss profiles to guide code- and data-layout optimizations, and branch misprediction profiles to guide if-conversion. Ultimately, we believe these additional profiles facilitated by hardware event sampling will significantly improve the profitability of feedback-directed optimization.

8. Acknowledgments

We want to thank all the reviewers for their insightful reviews and suggestions, which are integrated into the final version of this paper. We would like to thank all the people on the Google compiler team. Special thanks to Stephane Eranian for his help in analyzing the behaviour of PMU based sampling.

References

[1] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. *Proc. of SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997.

[2] 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.

[3] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems*, 1994.

[4] Thomas Ball and James R. Larus. Efficient Path Profiling. *Proc. of ACM/IEEE International Symposium on Microarchitecture*, IEEE Computer Society, 1996.

[5] 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.

[6] Intel Corporation. Volume 3B: System Programming Guide, Part 2. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2008.

[7] 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. *Proc. of ACM/IEEE International Symposium on Microarchitecture*, IEEE Computer Society, 1997.

[8] Paul J. Drongowski. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. Advanced Micro Devices, Inc., November 2007.

[9] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-Overhead Call Path Profiling of Unmodified, Optimized Code. *Proc. of International Conference on Supercomputing*, Arvind and Larry Rudolph, eds., ACM, Cambridge, Massachusetts, June 2005.

[10] Nick Gloy, Zheng Wang, Catherine Zhang, J. Bradley Chen, and Michael D. Smith. Profile-Based Optimization with Statistical Profiles. Harvard University, Cambridge, Massachusetts, April 1997.

[11] Steve R. Gunn. Support Vector Machines for Classification and Regression. Ph.D. Thesis, University of Southampton, 1998.

[12] Roy Levin, Gad Haber, and Ilan Newman. Complementing Missing and Inaccurate Profiling using a Minimum Cost Circulation Algorithm. *Proc. of International Conference on High Performance Embedded Architectures and Compilers*, Göteborg, Sweden, January 2008.

[13] 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. *Proc. of SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, Illinois, June 2005.

[14] 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. *Proc. of International Symposium on Computer Architecture*, IEEE Computer Society, Atlanta, Georgia, 1999.

[15] R. L. Probert. Optimal Insertion of Software Probes in Well-Delimited Programs. *IEEE Transactions on Software Engineering.*, 8(1):34–42, 1982.

[16] Vinodha Ramasamy, Paul Yuan, Dehao Chen, and Robert Hundt. Feedback-Directed Optimization in GCC with Estimated Edge Profiles from Hardware Event Sampling. *Proc. of GCC Developers' Summit*, Ottawa, Canada, June 2008.

[17] Florian T. Schneider, Mathias Payer, and Thomas R. Gross. Online Optimizations Driven by Hardware Performance Monitoring. *Proc. of SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[18] Alex J. Smola and Bernhard Scholkopf. A Tutorial on Support Vector Regression. *Statistics and Computing*, 14(3):199–222, 2004.

[19] Omri Traub, Stuart Schechter, and Michael D. Smith. Ephemeral Instrumentation for Lightweight Program Profiling. Harvard University, Cambridge, Massachusetts, June 2000.

[20] Vincent M. Weaver and Sally A. McKee. Can Hardware Performance Counters Be Trusted? *Proc. of IEEE International Symposium on Workload Characterization*, IEEE Computer Society, Seattle, Washington, September 2008.

[21] Youfeng Wu and James R. Larus. Static Branch Frequency and Program Profile Analysis. *Proc. of ACM/IEEE International Symposium on Microarchitecture*, IEEE Computer Society, 1994.

[22] 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.