Can We Trust Profiling Results?

Understanding and Fixing the Inaccuracy in Modern Profilers

Hao Xu
hxu07@email.wm.edu
College of William and Mary

Qingsen Wang
qwang06@email.wm.edu
College of William and Mary

Shuang Song
songshuang1990@utexas.edu
The University of Texas at Austin

Lizy Kurian John
ljohn@ece.utexas.edu
The University of Texas at Austin

Xu Liu
xl10@cs.wm.edu
College of William and Mary

ABSTRACT

Profilers are an indispensable component in modern software stack of data centers and supercomputers. Profilers collect detailed performance data during program execution and guide code optimization across the entire software stack. The *accuracy* of the profiling result proves to be vital for one to effectively gain performance insights. Unfortunately, inaccuracy may arise due to measurement techniques or hardware limits, which can waste optimization efforts.

However, there are few studies in evaluating the accuracy of modern profiling techniques. In this paper, we study performance monitoring units (PMU) based statistical sampling, one of the profiling techniques widely adopted by many state-of-the-art profilers. While PMU sampling based profilers are efficient in collecting performance data, they suffer from inaccurate instruction measurement due to the intrinsic limit in the PMU design. To understand and fix the instruction profiling inaccuracy, we propose a novel 3-step approach. First, we investigate multiple modern architectures and quantify the PMU instruction profiling inaccuracy in these architectures with mathematical modeling. Second, we design a systematic framework to evaluate the impact of PMU inaccuracy to the profiling results. Finally, we propose a software-based technique to rectify the measurement inaccuracy raised by PMU and demonstrate its effectiveness.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Software and its engineering** → **Runtime environments**.

KEYWORDS

PMU, Call path profiling, Statistical sampling, Accuracy.

ACM Reference Format:
Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. 2019. Can We Trust Profiling Results?: Understanding and Fixing the Inaccuracy in Modern Profilers. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3330345.3330371

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330371

1 INTRODUCTION

Computer systems have become increasingly complex. It is challenging to tune software on a specific hardware platform for high performance [45, 46]. To address such a problem, modern computer systems provide performance monitoring units (PMU) [2, 30, 42], which are able to monitor more than a hundred performance events, such as CPU cycles, cache misses, floating point operations, and many others. Performance tools utilize PMUs to identify performance inefficiencies, attribute them to different code segments, and provide optimization guidance for hardware designers, compiler developers, and application programmers.

PMU sampling (also known as event-based sampling) is widely used in mainstream performance tools, such as Oprofile [13], Linux Perf [38], HPCToolkit [5], and Intel VTune [3]. These tools configure the PMUs to a preset value $MAX - P$, where MAX is the maximum value a 64-bit PMU register can represent, while P is the sampling period predefined for the monitored event. When the event occurs P times, the PMU triggers an overflow interrupt, known as a sample. The monitored program suspends its execution and switches to the signal handler routine installed by the performance tool. In the signal handler, the performance tool is able to associate the sample with the program code obtained from the signal context. Furthermore, Perf can create an internal buffer to store all samples and use `poll()` to read the buffer to further lower the overhead. Performance tools based on PMU sampling typically have low overhead with reasonable sampling periods and thus are more desirable in production.

There has been a large volume of work [3, 5, 13, 39, 40, 44] showing the usefulness of the sampling-based tools in guiding performance diagnosis and optimization. Instruction per cycle (IPC) is one of the most important evaluation metrics during performance analysis, which largely relies on accurate instruction and cycle profiling results. However, there is a lack of systematic study about the accuracy of sampling-based instruction measurement. The common knowledge about the PMU sampling is that hot procedures (i.e., procedures with more samples) tend to have more reliable profiling results, while the measurement results of cold procedures (i.e., procedures with fewer samples) come with more noises. Since performance tools usually focus on hot procedures more than cold ones, such inaccuracy is tolerable. However, this common knowledge is derived from theoretical analysis or intuition [36], with

Table 1: Exclusive function-level instruction counts for two hot procedures in `PowerGraph PageRank` [17]. The ground truth result is collected by CCTLib.

Procedure	VTune	Perf	HPCToolkit	Ground Truth
`gather`	1.40e12 (50.1%)	1.37e12 (49.6%)	1.63e12 (59.3%)	9.32e11 (33.8%)
`execute_gathers`	1.36e12 (49.3%)	1.39e12 (50.4%)	1.13e12 (40.7%)	1.83e12 (66.2%)

little work in quantifying the accuracy in practice. We show that state-of-the-art tools based on this common knowledge can produce misleading results.

We use Intel VTune, Linux Perf, and HPCToolkit to collect retired instructions via PMUs on a real code base—`PowerGraph` [17], one of the most popular graph engines, as discussed in Section 6.1. Table 1 shows the profiling results in two hot procedures—`gather` and `execute_gather`. These two procedures account for more than 80% of CPU cycles of the entire graph processing phase. We can see that all three tools report similar results but significantly different from the ground truth, which is collected with CCTLib [10], a tool based on binary instrumentation. This profiling result incorrectly identifies the hotspot, potentially wasting optimization efforts.

This inaccurate measurement arises due to the hardware limit in handling the PMU samples. There is always a time delay between the PMU overflow interrupt and the signal delivery to the performance tool, which is known as "skid" [15]. Such skid pervasively exists in most architectures, such as x86, POWERPC, and ARM, and is the major source of the inaccuracy. However, the impact of the skid to the measurement accuracy has not been extensively studied. Prior work [11] relies on hardware support to minimize the skid, such as the precise event-based sampling (PEBS) [2] and last branch record (LBR) [4] in Intel processors. However, PEBS and LBR are not generally available in all processors. Moreover, even PEBS is not guaranteed to be accurate, which will be described in Section 2.

To address the limitations in existing approaches, we systematically study the instruction measurement accuracy of performance tools based on the PMU sampling. We will show the provenance of the inaccuracy by identifying the inaccurate, misleading profiling results generated by state-of-the-art performance tools even for hot procedures. We then design an emulation based algorithm to measure the skid cycle duration on different CPU architectures. With measured skid cycle duration, we propose a software-based fixing technique that can be employed by other tools to eliminate such inaccuracy with the utilization of control flow graph. In summary, we make the following contributions in this paper:

- We show the instruction measurement inaccuracy in state-of-the-art performance tools by comparing with the ground truth and summarize the characteristics of victims of the inaccurate measurement from the software perspective.
- We design and implement a measurement technique to quantify the skid in various architectures, as the root cause of the inaccuracy from the hardware perspective.
- We propose a novel software scheme without any extra hardware support to quantify the measurement inaccuracy by solving an optimization problem.
- We evaluate our techniques on several benchmarks and applications, which shows significant improvement on the accuracy of instruction profiles.

We organize the paper as follows. Section 2 shows the background and motivation of this paper. Section 3 describes our approach to quantifying the skid in various CPU architectures. Section 4 elaborates on the methodology of fixing the measurement inaccuracy with a pure software technique. Section 5 and 6 depict our evaluation and case study on real applications and benchmarks, respectively. Section 7 offers related work. Finally, Section 8 presents some conclusions of the paper.

2 BACKGROUND

This section introduces the background knowledge of existing profilers based on PMU sampling, reveals the provenance of inaccurate measurement, and discusses our scheme of obtaining ground truth for the accuracy study.

2.1 Profilers with PMU Sampling

Hardware Performance Monitoring Units (PMU). CPU's PMUs offer a programmable way to count hardware events such as retired instructions, CPU cycles, cache misses, etc.. A PMU can trigger an overflow interrupt once a preset number of occurrences of an event is reached. A profiler, running in the address space of the monitored program, can handle the interrupt and attribute the measurement "appropriately". We refer to a PMU counter overflow as a "sample". PMUs pervasively exist in CPU processors from different vendors, such as Intel, AMD, IBM, and ARM.

Linux Perf_events: Linux provides APIs to configure, enable, and disable PMUs under thread granularity (e.g., `perf_event_open` [38], `ioctl`). Once a PMU event counter overflows, the Linux kernel signals the corresponding thread with the details of the event (e.g., instruction pointer). The signal handler in the user space then examines the event information and attributes proper measurements. Some PMU facilities, such as Intel's precise event-based sampling, allocate a kernel buffer to record multiple samples and allow tools to read the buffer via `poll()`.

Profiling Mechanisms: Many tools utilize call path profling [18], a profiling technique in which runtime events (e.g., cache misses) are attributed to the full call path seen at the time of the event. Call path profiling offers insightful details in complex applications with deep call chains. The *calling context* of an event is a set of active procedure frames when the event happens. A calling context begins at a process or thread entry function such as `main` and ends at the instruction pointer (IP) of the instruction that triggers the event. With the calling context, tools are able to associate samples with all functions in the call chain. The accumulated metrics from all the callees are known as inclusive metrics, while metrics not accumulated from the callees are known as exclusive metrics.

Provenance of Inaccurate Measurement. In out-of-order processors, the PMU counter overflow interrupts triggered by the monitored event may be significantly delayed. Such delay is called "skid", which typically is indeterminate and not fixed [11, 15]. Figure 1 illustrates how skid incurs inaccurate measurement. For simplicity,

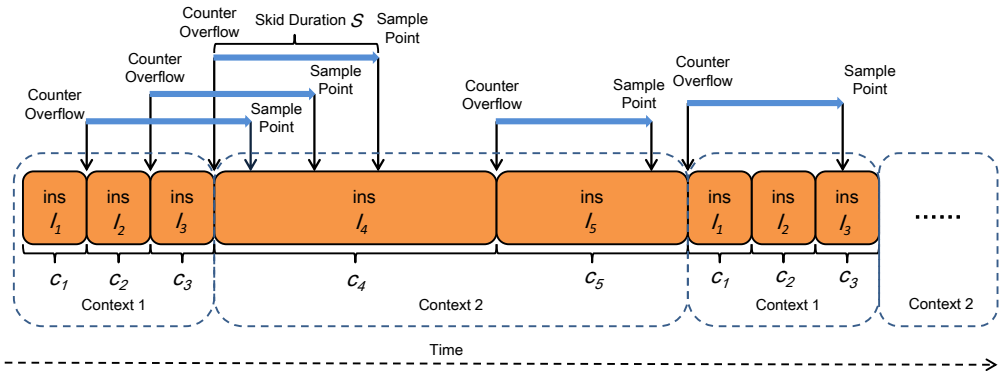
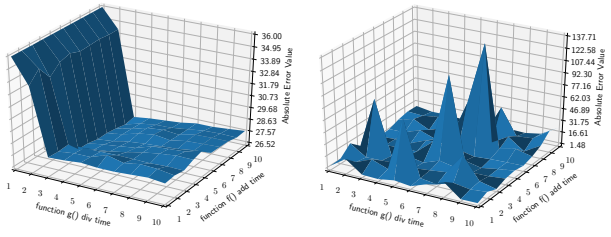


Figure 1: Skid effects on instruction profiling result. A loop consists of 5 instructions $I_1, I_2, ..., I_5$, which has different cycle duration $c_1, c_2, ..., c5$, respectively. Due to the skid effects, it takes CPU a fixed length of time S to address the active instruction. For example, after retired instruction I_1 triggers the instruction counter overflow, the CPU will take a short period of time S to find I_4 is active and then treat it as sample point. Every instruction has the same probability causing the counter overflow, but every instruction does not have the same probability to be treated as sample points. Instruction I_4 is more likely to be selected as sample points, since its cycle duration is significantly larger than those of other instructions. As a result, it will cause instruction sample point mis-attribution at context level.



(a) PEBS **(b)** Traditional PMU Sampling

Figure 2: Comparison of instruction profiling inaccuracy loss. Z-axis value indicates the total mis-attributed instruction number at the function level.

a context (e.g., a procedure, a loop) only consists of two or three instructions (I) and each instruction has its own execution time (c). Due to the skid effects, samples occurring at a context can be mis-attributed to its adjacency.

2.2 Limitation of Precise Event Based Sampling

Modern CPU processors provide various precise sampling mechanisms to alleviate or eliminate the skid, such as Intel's precise event-based sampling (PEBS) [2] and AMD's instruction-based sampling (IBS) [15]. These mechanisms use specific PMU registers to record the precise instruction pointers (IP) that trigger PMU counter overflows. However, not all CPU vendors (e.g., ARM) support these precise mechanisms. Moreover, these mechanisms do not always provide reliable profiling results. For IBS, PMU needs to tag an instruction at the issuing point to monitor its execution in the pipeline. If this instruction is not retired due to the speculative execution, PMU will not capture any sample in this period. PEBS suffers from shadow effect [2]. When the PMU selects an instruction being retired in the pipeline to report, there can be multiple candidates. Yet PEBS is more likely to report the one with the highest execution latency, leading to biased profiling results.

To show PEBS does not produce reliable results, we compare the profiles generated by PEBS and traditional PMU. We adopt the micro-benchmarks, introduced in Section 3. Every micro-benchmark contains two functions `g()`, `f()` with N_g and N_f instructions, respectively. We quantify the profiling accuracy loss in functions `g()` and `f()`, which account for $N_g + E$ and $N_f - E$ instructions (E is the accuracy loss) from the measurement. Figure 2 plots the error E of PEBS and traditional PMU for all micro-benchmarks with different sizes of `g()` and `f()`. Traditional PMU produces consistent and predictable inaccuracy due to skid effects, while PEBS incurs unpredictable profiling results that are difficult to reason with the ground truth.

Thus, for general applicability, we target the widely-used traditional PMUs, rather than specialized precise PMUs.

2.3 Ground Truth Profiler

We cannot rely on PMUs to collect the ground truth. Alternatively, one may use simulators [8, 9, 37, 47] to measure the hardware-related events (e.g., cycles, cache misses). However, since it is difficult to accurately simulate every feature of a processor, simulators may not truly produce ground truth for these events. We observe that software-related events such as retired instruction and floating point operations, do not depend on hardware. We are able to adopt a software method to collect ground truth profile of these software-related events, regardless of hardware platforms. Thus, we develop a tool that uses a pure software method to measure the number of retired instructions of procedures in their calling contexts. We build the tool based on CCTLib [10], a Pin tool that is able to determine the calling context of each monitored instruction in a parallel program. We design a client tool atop CCTLib to count the number of retired instructions in each procedure within its calling context.

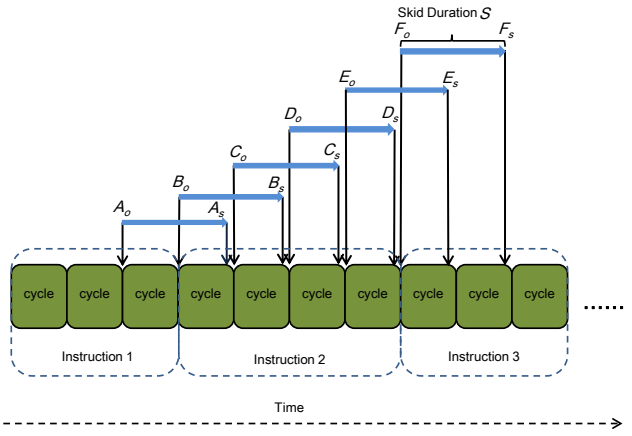


Figure 3: Skid effect on cycle profiling result. The skid duration S is less than 2 cycles and every cycle has the same probability to cause the counter overflow. There are three instructions (1, 2, 3) executed in order. Two different kinds of time points are defined: 1) **counter overflow point**, a finished cycle causing the hardware performance counter to overflow; 2) **sample point**, a time point when active instruction is blamed as the cause of performance counter overflow. Each counter flow point has a corresponding sample point (e.g., the counter overflow pointer A_o has its own sample point A_s.). The time difference of each pair of counter overflow point and sample point is skid duration S measured in cycles.

3 QUANTIFYING SKID EFFECT

In this section, we introduce a mathematical model to quantify the skid effect for a simple program that only consists of a *simple loop of instructions*.

Definition 3.1. **Simple Loop**[1] **of Instructions**: A repeatedly executed loop consists of a fixed number of instructions, which contains no conditional branches except the one for loop control.

The mathematical model relies on the following assumptions:

- The skid effect can be quantified in CPU cycles [12].
- Each instruction takes a fixed number of cycles on average, i.e., its Cycle Per Instruction (CPI) stays the same.
- CPU can issue multiple instructions at the same time.
- When sampled by CPU cycles, each cycle has the same chance to overflow the hardware event counter, regardless of the instruction being currently executed.

Figure 3 illustrates the skid effect of a simple loop under instruction profiling, which helps us draw the two conclusions:

CPU Cycle profiling result for each instruction is not affected by skid effects. As shown in Figure 3, Instruction 2 with a duration of 4 cycles should trigger 4 counter overflows (C_o, D_o, E_o, F_o) and own 4 samples[2]. With a skid of 2 cycles, the sample points E2 and F2 are finally attributed to Instruction 3. In the meantime,

[1]It is also called *Simple Cycle*. We deliberately use *Simple Loop* to avoid any confusion from CPU cycles.

[2]More accurately, it is $4p$ samples instead of 4, where p is the probability of a cycle to trigger an overflow.

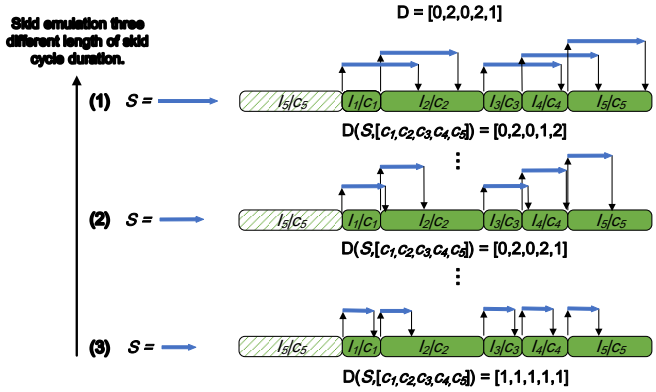


Figure 4: Skid effect emulation with three different skid values on a simple loop consisting of five instructions. Each emulation generates an instruction distribution $\mathbf{D}(S, [c_1, c_2, c_3, c_4, c_5])$ that is a vector of the number of samples of all instructions, while $\mathbf{D}$ is the actual instruction profiling result.

another two sample points A2 and B2 triggered by previous instructions are attributed to Instruction 2, which makes Instruction 2 still own 4 sample points. The number of sample points within an instruction always stays constant whatever the skid duration is. Thus, the skid effect does not affect the cycle profiling result [11]. Instead, a sampling profiling with a period of T and a skid S is equivalent to the one with a period of $T + S$ without any skid. This property can be used to estimate the CPI value of each instruction from cycle profiling result. To minimize the overhead of profiler, T is usually significantly greater than S, which is usually very small.

CPU cycle profiling result is not affected by instruction-level parallelism (ILP). CPU has its own mechanism to blame which instruction from all instructions on the fly as the sample point, when performance counter are employed for sampling CPU cycles. Each instruction has its own probability to be blamed at the sample point. Such probability is not affected by skid effect and up to CPU PMU design. Since the skid effect only increases the sampling period and the sampling period does not alter the profiling result, instruction-level parallelism (ILP) with skid effect will not introduce any further effect on CPU cycle profiling.

In the next few subsections, we will first explain how skid effect emulation can generate relative instruction distribution in a simple loop of instructions, and then design an algorithm to measure the CPU cycle duration of the skid. .

3.1 Skid Effects Modeling in a Simple Loop

We use a simple loop with 5 instructions I_1, I_2, I_3, I_4 and I_5 to illustrate our skid effect model. They have their CPI values, which are c_1, c_2, c_3, c_4 and c_5 respectively. Figure 4 shows skid effect emulations with different skid CPU cycle duration S. Each instruction has the same probability to cause the performance counter overflow, when PMU is adapted to sample instructions with a fixed sampling rate. When a retired instruction causes the counter overflow, it takes S CPU cycles (skid duration) to stop and then attributes to the instruction on the fly as the sample point. Then we can construct a mapping

from the instruction causing counter overflow to the instruction at the sample point. With the CPI information, this mapping for fixed skid CPU cycle can generate the relative sampled instruction distribution $\mathbf{D}(S, [c_1, c_2, c_3, c_4, c_5])$. In Figure 4, we emulate the skid effect under three different skid values, thus generating three relative sampled instruction distributions. For example, skid emulation (2) generates a distribution of $\mathbf{D}(S, [c_1, c_2, c_3, c_4, c_5]) = [0, 2, 0, 2, 1]$, which is closest to the actual profiling result $\mathbf{D} = [0, 2, 0, 2, 1]$.

For a simple loop with a specific CPI vector $\mathbf{c}$, a specific skid duration S will generate a corresponding instruction distribution $\mathbf{D}(S, \mathbf{c})$ upon skid effect emulation. We rely on this property to design an algorithm to measure the skid duration S for specific CPU, which will be explained in the next subsection. Formal mathematical modeling is provided in Section 1 in our complement document [1].

3.2 Measurement of CPU Skid Duration

The skid model of a simple loop introduced in the previous subsection involves three variables:

- S, skid duration in cycles,
- $\mathbf{c}$, a vector of cycle duration of instructions, equivalent to $\{c_1, c_2, ..., c_N\}$,
- $\mathbf{D}$, a vector of the number of samples of all instructions, i.e., the profiling result.

As CPU skid changes the instruction distribution of a simple loop other than the total number of sampled instructions, we have

$$\|\mathbf{D}\|_1 = \|\mathbf{D}(S, \mathbf{c})\|_1 = \|\mathbf{D}(0, \mathbf{c})\|_1 .$$

Since the cycle profiling result of a simple loop is immune to the skid effect, we can estimate each instruction's CPI by the cycle profiling result. More specifically, if the loop is executed N_e times, we can derive the CPI value of an instruction i by

$$c_i = \frac{C_i}{N_e},$$

where C_i is the total number of cycles executing instruction i from the cycle profiling result. After applying this equation to all the instructions, we can get the CPI values of all instructions, which constitute a vector $\mathbf{c}$.

Knowing $\mathbf{c}$, we can further obtain $\mathbf{D}(S, \mathbf{c})$ under different values of S as described in Section 3.1. To quantify how close our emulation result $\mathbf{D}(S, \mathbf{c})$ is from the actual instruction sampling result $\mathbf{D}$, we introduce an error metric $Error(\mathbf{C}, S)$, where

$$Error(\mathbf{C}, S) = \|\mathbf{D}(S, \mathbf{c}) - \mathbf{D}\|_2.$$

If $Error(\mathbf{C}, S)$ is small enough, we can assume the skid value S chosen is correct.

There may be more than one "correct" skid value for one simple loop. Thus we adopt several similar but different simple loops to eliminate other possible values. Listing 1 shows a mini-benchmark template that is used to measure the skid duration S. [3] It consists of two functions in a simple loop. By altering the number of `add` operations P and `div` operations Q from 1 to 10, respectively, we end up with $M = 10 \times 10 = 100$ programs in total. Then we collect all programs' cycle profiling results (denoted as $\{\mathbf{C}_1, \mathbf{C}_2, ..., \mathbf{C}_M\}$) and instruction profiling results (denoted as $\{\mathbf{D}_1, \mathbf{D}_2, ..., \mathbf{D}_M\}$) to

[3]The mini-benchmark code and scripts to collect the data are in https://github.com/xuhao417347761/ics19-minibenchmark.git.

```
1 void g(){
2   a/=i; // P div operations
3   ...}
4 void f(){
5   a+=i; // Q add operations
6   ...}
7 int main(){
8   for (i = 1; i<=NUM_TIME; i++) {
9     g();
10    f();
11  }
12 }
```

Listing 1: Mini-benchmark template for measuring skid cycle duration. A for-loop in `main` calls two functions `g()` and `f()`, which contains P `div` operations and Q `add` operations, respectively.

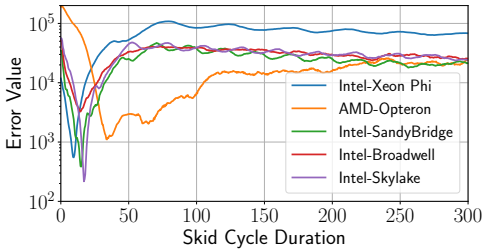


Figure 5: Measurement of skid cycle duration for multiple platforms. Y-axis represents the sum of errors described in the equation, while X-axis denotes the candidate skid duration from 0 to 300 cycles. We search for the skid duration value, which makes the error sum minimum.

estimate the CPI vectors for all programs on the same CPU (denoted as $\{\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_M\}$). By choosing an arbitrary skid value S and emulating the skid effect on all the programs, we can get $Error(\mathbf{C}, S)$ of all the programs. The correct skid duration S of this CPU should minimize the sum of $Error(\mathbf{C}, S)$, i.e.,

$$\arg\min_S \sum_{m=1}^{M} Error(\mathbf{C}_m, S). \quad (1)$$

Algorithm 1 describes how we measure the skid duration on a specific platform. We exhaustively set S from 0 to 300 (Line 3). Under each value of S, we emulate the skid effect on all the programs and sum up the error as shown in Equation 1 (Line 4-8). The value of S is only kept when the corresponding error sum is the minimal value seen so far (Line 9-12).

Figure 5 plots the error sums when S changes from 0 to 300 under five platforms (AMD-Opteron, Intel-SandyBridge, Intel-Broadwell, Intel-Skylake and Intel-Xeon Phi). We can always find a global minimum point for every platform, where its S value makes the error sum minimal. The skid duration measurement process is a one-time job for a specific CPU platform, which takes around 10 minutes to finish running all mini-benchmarks and searching for the optimal value. As we discussed earlier, the S value at the minimal point is the skid duration of this platform. The skid duration of all Intel CPUs is less than 20 cycles, while AMD-Opteron has a much longer skid, which is 34 cycles.

Besides the arithmetic instructions, we have also developed other mini-benchmark suite with *heavy memory access instructions*. The skid measurement results are quite close to Figure 5. We will release

Algorithm 1: Measurement of Skid Duration

Input: Cycle profiling result $\{\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_M\}$, instruction profiling result $\{\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_M\}$
Output: Skid duration S

```
min ← 0;
minErrorTemp ← ∞;
for S ← 0 to 300 with step of 0.5 do
    errorSum ← 0;
    for m ← 1 to M do
        c ← C_m / N_e ;
        errorSum ← errorSum + ||D(S, c) − D_m||_2;
    end
    if errorSum < minErrorTemp then
        minErrorTemp ← errorSum ;
        min ← S ;
    end
end
S ← min ;
```

all the mini-benchmarks for the skid duration measurement once the paper gets accepted.

After we obtain the skid value S of a platform, we can use it to emulate the skid effect happening in any simple loop, which should resemble the actual instruction profiling result. However, we encounter a challenge when applying the skid effect emulation to control flow graph with multiple branches. PMU does not quantify the execution frequency of every branch, which makes it impossible to estimate CPI of each instruction (one of the skid emulation inputs). We elaborate on how to eliminate skid effect for on complex control flow graphs with multiple branches in the next section.

4 NULLIFYING SKID EFFECT ON INSTRUCTION PROFILING

In this section, we explain how to eliminate the skid effect on instruction profiling results for real programs by obtaining the skid duration S from Section 3. Unlike instructions in simple loops, instructions from a real program usually belong to a control flow graph with many branches. To apply our skid emulation with measured skid CPU cycle duration S, we decompose the control flow graph into several simple loops and aggregate the skid effects on these simple loops. An optimization problem is formulated to obtain the simple loop frequencies with measured skid duration in cycles. In this section, we will show the intuition and all the formal mathematical description is in the complement material [1].

4.1 Control Flow Graph Decomposition

Extracting Instruction Control Flow Graph. We rely on static analysis [5] to extract the basic-block level control flow graph of the whole program. Since instructions within the same basic block are executed in a deterministic order, we are able to deduce the corresponding instruction control flow graph, which is the result generated after step (1) in Figure 7. In practice, we only focus on the code blocks with large amounts of instructions, which are most interesting to profiler users.

Decomposing Control Flow Graph into Simple Loops. A control flow graph is a directed graph, which contains several simple loops. Previous work [20] proves that a directed graph can be decomposed into simple loops. The example shown in Figure 7 describes a complete process of control flow graph decomposition. We rely on a Python library `networkx` to generate simple loops (simple cycles) for instruction control flow graph. Every instruction is treated as a node, while branches are treated as directed edges. Each simple loop generated by decomposition is with a fixed set of instructions in order and its own execution frequency. Their execution frequencies can be adapted to calculate the CPI of each instruction. In the next subsection, we will explain how we formulate an optimization problem based on the connection between simple loops execution frequencies and instructions' CPI.

4.2 Obtain Simple loop Frequency to Recover Instruction Profile

Every simple loop has its own execution frequency. The instruction profile is up to execution frequencies of these simple loops. We cannot directly derive the CPI values of a simple loop from the CPU cycle profiling result as the number of times executed (or execution frequency) is unknown. Seeking the execution frequency of all simple loops is then formulated as an optimization problem consisting of six steps as shown in Figure 6. We keep using the control flow graph in Figure 7 as an example to explain our methodology.

(1) *Obtain Cycle Per Instruction.* With a fixed set of simple loop frequencies, we are able to obtain execution frequency for each instruction. Like simple loop, skid effect with fixed CPU cycle duration does not affect the cycle profiling distribution in the control flow graph [12]. Since CPU cycle profiling result can provide CPU cycle counts for each instruction, cycle per instruction information for each instruction can be obtained by dividing CPU cycle counts by instruction counts. As a result, we are able to obtain the CPI information for each simple loop to emulate the skid.
(2) *Emulate Skid Effect on Simple Loops.* Skid emulation on each simple loop produces the instruction distribution.
(3) *Aggregate Instruction Distribution.* We aggregate all the instruction distribution with their frequency weights to generate the instruction distribution for the entire control flow graph. The emulation error is the difference between the generated instruction distribution and the actual instruction sampling result.
(4) *Generate New Simple Loop Frequencies.* By investigating the emulation error, we are able to obtain a cost function to evaluate how close the summation of emulations on all simple loops to the instruction profile generated by sampling based profilers. We formulate an optimization problem to find the best solution for all simple loop frequencies based on this cost function. To avoid brute-force enumerating all possible values of simple loop frequencies, we adopt the Gibbs Sampling method [16] in polynomial time complexity to search the best simple loop frequency vector.
(5) *Update Simple Loop Frequencies.* When the emulation error is still large, we start a new iteration by updating the simple loop frequencies in steps (1) and (3) with more accurate ones obtained from the last step.

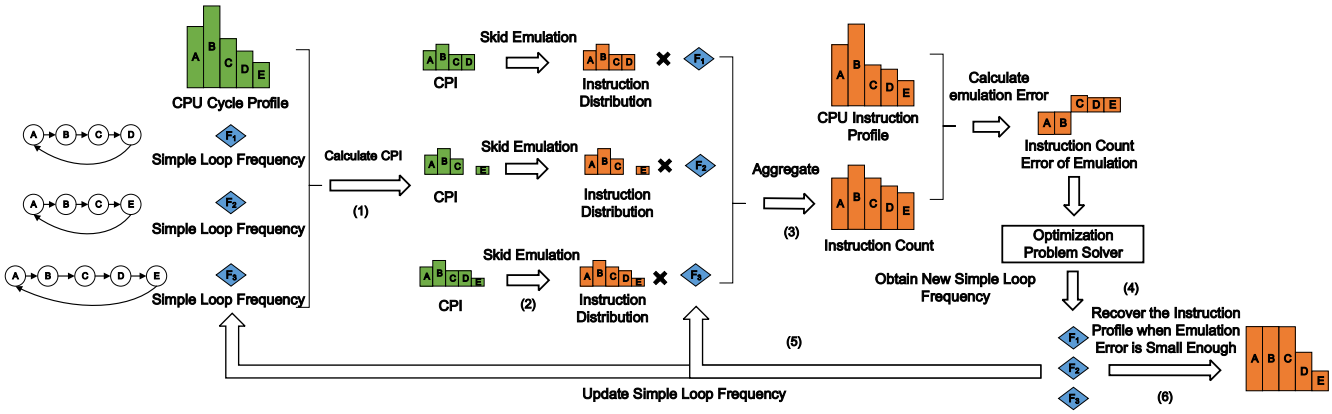


Figure 6: Recover instruction profile of control flow graph in Figure 7. There are six steps for recovering process. CPU cycle profile and instruction profile are provided by sampling based profile. Step (1) to (5) constitutes a closed iterative loop to improved the recovering quality.

Table 2: Evaluation platform configurations.

Microarchitecture	Intel-SandyBridge	Intel-Broadwell	Intel-Skylake	AMD-Opteron	Intel-Xeon Phi
Processor	Xeon E5 4650@2.7GHz	Xeon E5-2650 v4@2.2GHz	Xeon E3-1240 v5@3.5GHz	Opteron 6168@1.6GHz	KNL 7210@1.30GHz
SMT × # Cores	2 × 8	2 × 12	2 × 4	1 × 12	2 × 64
L1/L2/L3 Cache, Memory	32KB/256KB/20MB, 64GB	64KB/256KB/30MB, 128GB	64KB/256KB/8MB, 64GB	64KB/512KB/10MB,16GB	32KB/32MB/-,32GB
Compiler	gcc 4.8.5 -O3	gcc 4.8.5 -O3	gcc 5.4.0 -O3	gcc 4.8.5 -O3	gcc 4.8.5 -O3

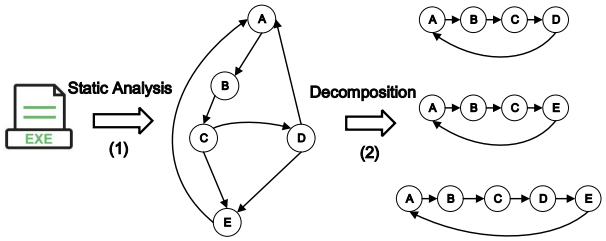


Figure 7: Control flow graph with multiple branches decomposition. After static analysis, we obtain a instruction level control flow graph with 5 instructions (A, B, C, D, E). Decomposition generate 3 different simple loops: A → B → C → D, A → B → C → D, A → B → C → E.

(6) *Output Recovered Instruction Profile.* When the emulation error is significant smaller than initial emulation error and does not converge to smaller value with additional iterations, the iteration process is stopped. Then We are able to obtain the instruction profile close enough to the ground truth, combined with the structure of the simple loops. We implement this algorithm using MATLAB. The optimization problem can be solved in less than 1 minute. After solving this optimization problem, we can get each instruction's profiling result inside the problematic loop. Then each function's profiling result can be successfully recovered.

Our method only requires to profile the program by sampling cycles and instructions and the optimization problem can be solved offline. The algorithm's time complexity grows linearly with the number of simple loops and the number of iterations. For the problematic loops of real applications we have studied, the simple loop number generated from the control flow graph decomposition is less than ten.

5 EVALUATION

In this section, we evaluate the accuracy of instruction profiles collected from 23 applications (21 from SPEC CPU2006 benchmark suite [34] and 2 real applications) on five platforms. We also evaluate our methodology on problematic applications. Our study demonstrates that our method is able to rectify the skid effect effectively. Our study indicates the instruction profile of application with small hot functions is more likely to be mis-attributed at function level.

5.1 Evaluation Setup

The experiments are performed on five platforms, whose configurations are shown in Table 2. We run SPEC CPU2006 integer and floating-point benchmarks with reference inputs [30, 42]. For real applications like `PowerGraph` and `libsvm`, we use their representative datasets as inputs. Two profiling tools HPCToolkit [5] and CCTLib [10] are adapted to profile each application[4]:

- HPCToolkit is an integrated suite of tools for collecting measurement and analysis of program performance, based on statistical **sampling** hardware performance counters. HPCToolkit incurs low overhead during the whole sampling process, but its profiling result may suffer from skid effects [5].

[4]We have explained in the previous section about inconsistent profiling result of PEBS techniques. Besides, AMD-Opteron and Intel-Xeon Phi do not support PEBS. Thus, we do not choose PEBS as our baseline to prove the effectiveness of our method.

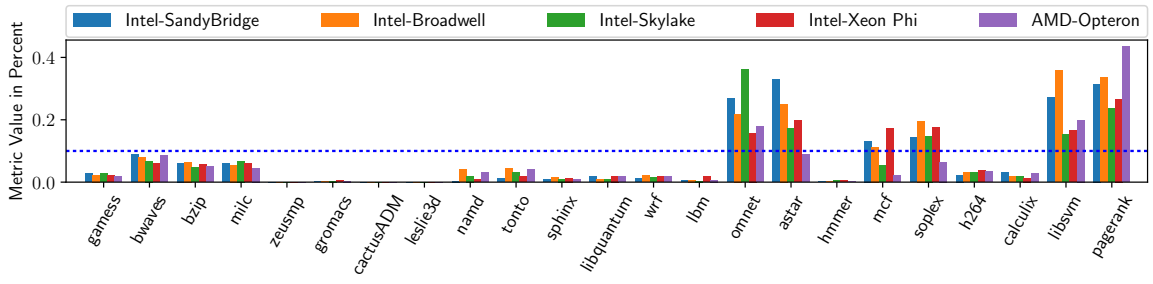


Figure 8: The value $\epsilon_{hpctoolkit}$ for chosen applications on five platforms. Smaller value is better.

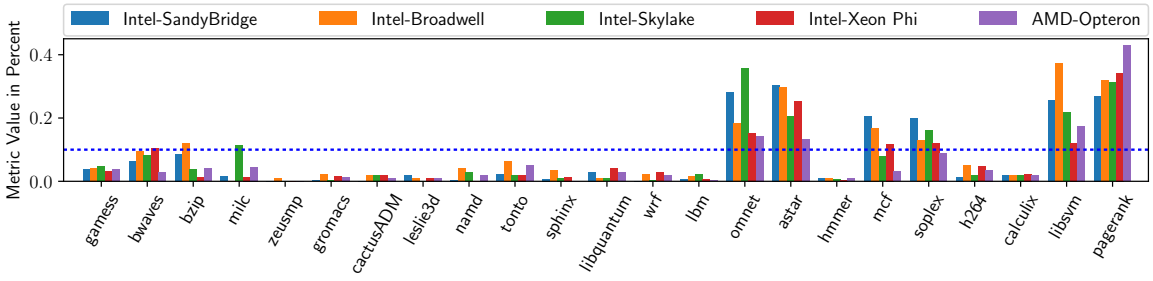


Figure 9: The value ϵ_{Perf} for chosen applications on five platforms. Smaller value is better.

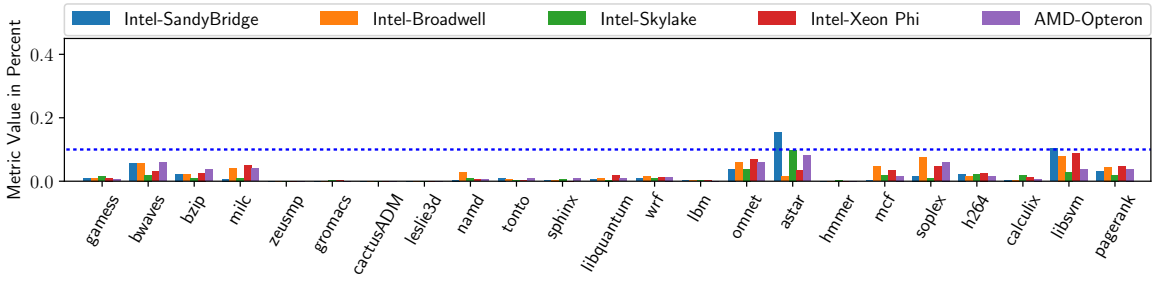


Figure 10: The value ϵ_{fix} for chosen applications on five platforms. Smaller value is better.

- CCTLib [10] is an instrumentation tool based on Intel Pin [25]. It instruments every instruction instance dynamically, however, with significantly higher overhead than HPCToolkit. We treat its result as the **ground truth** to identify mis-attribution problem from the HPCToolkit profiling result.

Our method recovers instruction profiling result at the basic block level. However, we present all the profiling result in *function level*, since function level profiling result is more straightforward for users and helps users more effectively pinpoint hotspots and optimize problematic code blocks.

5.2 Effectiveness Provenance

After processing instruction profiling result generated by sampling based profiler HPCToolkit, our algorithm can generate fixed (recovered) instruction profiling result. We use this property to identify the application with problematic instruction profiling result at the function level. A metric ϵ is defined to quantify the difference of sampling based instruction profile from ground truth instruction profiling result provided by CCTLib. Since we only care about hot functions in each application, here we only focus on the top 10 functions with most instruction counts executed in recovered instruction profiling result, these functions are denoted as set Φ. We define $\pi_{\phi,h}$ and $\pi_{\phi,c}$ as the instruction profiling result (in percent) of function ϕ from HPCToolkit and CCTLib, respectively. Then we define $\epsilon_{hpctoolkit}$ as:

$$\epsilon_{hpctoolkit} = \sum_{\phi \in \Phi} |\pi_{\phi,h} - \pi_{\phi,c}|.$$

If $\epsilon_{hpctoolkit}$ is close to 0, the HPCToolkit profiling result is close to the ground truth, which means that HPCToolkit produces an accurate instruction profile. The $\epsilon_{hpctoolkit}$ values of all applications are summarized in Figure 8. Most of the applications have a very ideal sampling based instruction profile while several applications with $\epsilon_{hpctoolkit} > 0.1$ are problematic. The profiling result of `omnet` is severely mis-attributed on all five machines, as well as `libsvm` and `pagerank`. Another two applications `astar` and

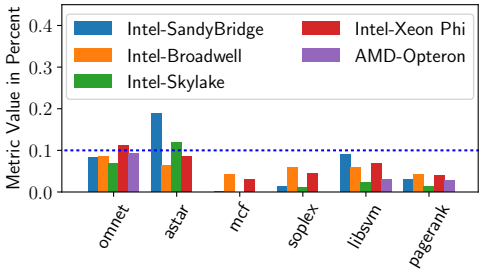


Figure 11: The value $\epsilon_{fix,bb}$ for chosen applications on five platforms. Smaller value is better.

`soplex` fail to get accurate profiling results on four machines except AMD-Opteron. As to `mcf`, only AMD-Opteron and Intel-Skylake produce accurate results.

To demonstrate the effectiveness of our approach, we define another metric ϵ_{fix}:

$$\epsilon_{fix} = \sum_{\phi \in \Phi} |\pi_{\phi,f} - \pi_{\phi,c}|,$$

where $\pi_{\phi,f}$ denotes the instruction profiling result (in percent) of function ϕ after applying our approach on HPCToolkit profiling results $\pi_{\phi,h}$. As shown in Figure 10, most of the applications have a ϵ_{fix} below than 0.1, indicating that our recovered profiling result is quite close to the ground truth. Besides, low ϵ_{fix} in figure 10 indicates our approach generates accurate instruction profile for non-problematic applications. Finally, our method is able to achieve an average error reduction ($\epsilon_{hpctoolkit} - \epsilon_{fix}$) of nearly 0.178, which is a significant improvement on instruction profile accuracy.

Profiling Accuracy of Linux Perf. Same as $\epsilon_{hpctoolkit}$ for HPCToolkit, we also define metric ϵ_{perf}:

$$\epsilon_{perf} = \sum_{\phi \in \Phi} |\pi_{\phi,p} - \pi_{\phi,c}|,$$

, where $\pi_{\phi,p}$ represents the instruction profiling result (in percent) of function ϕ collected by Perf. Figure 9 indicates Perf profiling result are very close to that of HPCToolkit.

Profiling Accuracy at Basic Block Level. We collect all basic blocks belong to functions set ϕ, which are denoted as **B**. Sample as $\pi_{\phi,h}$ and $\pi_{\phi,c}$, we also define $\pi_{b,f}$ and $\pi_{b,c}$ as the instruction profiling result (in percent) of basic block b from HPCToolkit with our approach and CCTLib, respectively. Then we define $\epsilon_{fix,bb}$ as:

$$\epsilon_{fix,bb} = \sum_{b \in \mathbf{B}} |\pi_{b,f} - \pi_{b,c}|,$$

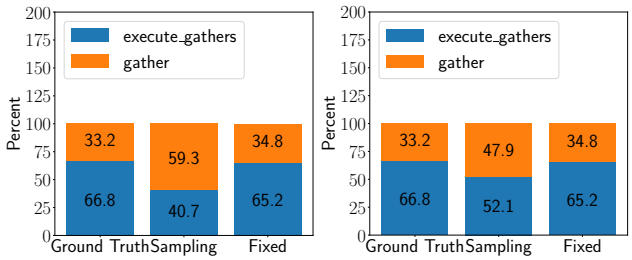
Figure 11 shows the $\epsilon_{fix,bb}$ value of six problematic applications' profiling accuracy. Most of the applications have a good basic block level instruction profiling accuracy with our approach ($\epsilon_{fix,bb} < 0.1$). $\epsilon_{fix,bb}$ of `omnet` and `astar` are higher than other applications, since they have more branches within the hot functions. Our technique improves the profile accuracy on the basic block level, which results in accuracy improvement on the function level.

6 CASE STUDY

In this section, we evaluate a few benchmarks with high $\epsilon_{hpctoolkit}$ value, seen in Figure 8. For each application, We select functions with largest $\epsilon_{hpctoolkit}$, use our algorithm to fix the problematic instruction profile, and discuss the root cause of mis-attribution. Due to the page limit, we only show the fixed results of `Pagerank`, `astar` and `hmmer` on two different Intel CPUs: Intel-Skylake and Intel-SandyBridge. The rest of results are provided in Section 3 of complement material [1]. Our study shows that the mis-attribution at function level of a problematic application is caused by heavy-weight instructions located near hot small functions.

6.1 PowerGraph Pagerank

`PowerGraph` is a high performance graph processing framework written in C++ [17]. We evaluate a representative data analytic application `PageRank` [29, 31–33], from it. The input graph data are transformed from the Amazon product purchasing network with 0.4 million vertices and 3.3 million edges [21]. We configure `PageRank` to run 50 iterations and only consider the application execution part as the profiling domain (pre-processing part is excluded in profiling). As the graph data are very large and cannot be entirely loaded into the cache, instructions of loading data from main memory are quite heavy, causing a severe problem of instruction mis-attribution in the profiling result.



(a) pagerank, Intel-SandyBridge **(b)** pagerank, Intel-Skylake

Figure 12: Fixed result of 2 mis-attributed functions' instruction profile in `PageRank` for Intel-SandyBridge and Intel-Skylake.

Fixed results are shown in Figure 12a and Figure 12b for Intel-Skylake and Intel-SandyBridge, respectively. The function `gather` is overestimated in the instruction profiling result. Users may underestimate the memory loading operations' overhead of the whole system, when they rely on sampling based profiling result to calculate IPC. We effectively recover the three functions' sampling based instruction profiling result inside the loop from `execute_gathers` (line 9 in Listing 2), with a low metric ϵ_{fix}'s value close to 0.

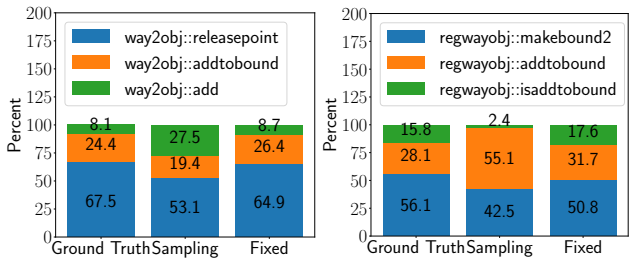
6.2 SPEC CPU2006 astar

The benchmark `astar` is an application based on a 2D path-finding library for game AI development [19]. The profiling result shows that over 38% of the instructions are mis-attributed on Intel-SandyBridge. Listing 3 shows function `releasepoint` and function `addtobound` in `Way2_.cpp`, and function `add` in `Arrays.cpp`. These three functions account for 41.3% of the total instructions. Function `addtobound` is called in a nested for-loop belonging to function `releasepoint`.

```
1 //callee gather called by execute_gathers
2 double gather(icontext_type & context, const vertex_type&
        vertex, edge_type& edge) const{
3   return (edge.source().data()/edge.source().num_out_edges());
4 }
5 ...
6 // caller execute_gathers's loop in synchronous_engine.hpp
7 execute_gathers(const size_t thread_id) {
8   ...
9   foreach(local_edge_type local_edge, local_vertex.in_edges()){
10    edge_type edge(local_edge);
11    if(accum_is_set) {
12    accum += vprog.gather(context, vertex, edge);
13    } else {
14      accum = vprog.gather(context, vertex, edge);
15      accum_is_set = true;
16    }
17    ++edges_touched;
18  }
19  ...
20 }
```

Listing 2: Code of problematic loop in `PowerGraph Pagerank`.

The function `addtobound` is very small, invoking `add` near the exit. The `mod` computation is very cycle-consuming in `addtobound` at line 10. Figure 13a plots the normalized instruction profiling result for `releasepoint`, `addtobound` and `add`. Here `add` is over-attributed more than 3 times, while `releasepoint` and `addtobound` are under-estimated. After applying our algorithm, we obtain an instruction profile on the function level very close to the ground-truth generated by CCTLib.



(a) `astar`, Intel-SandyBridge **(b)** `astar`, Intel-Skylake

Figure 13: Fixed result of 3 mis-attributed functions' instruction profile in `astar` for Intel-SandyBridge and Intel-Skylake.

For Intel-Skylake machine, the instruction profiling result of the nested loop (Listing 3, line 16) is not significantly affected. However, there are still problematic loops with such mis-attribution (`makebound2`, `addtobound` and `isaddtobound` from `RegWay_.cpp`, shown in Figure 13b). The `addtobound`, compared with ground truth. Our algorithm also effectively fixes it.

6.3 SPEC CPU2006 hmmer

The application `hmmer` focuses on searching patterns in DNA sequences with hidden markov models [19]. We profile it with reference input. Its $\epsilon_{hpctoolkit}$ value indicates that the instruction profile at function level is not significantly mis-attributed. However, we find that the profiling result for store instructions[5] is quite inconsistent at line level. A loop in `fast_algorithms.c`, shown in Listing 4, contributes nearly 99% of all store instructions. There is only one store instruction in line 137 and 134. Since line 137 is conditional executed, the attributed value of line 137 must be smaller than that of line 134. However, the actual sampling result

[5]The corresponding PAPI event name is `PAPI_SR_INS`.

```
1 // callee flexarray<eobj>::add in Arrays.cpp
2 template <class eobj> inline void flexarray <eobj>::add(const
      eobj& e){
3   if (elemqu==maxelemqu) doubling(true);
4   ep[elemqu]=e;
5   elemqu++;
6 }
7 // callee addtobound in Way2_.cpp
8 void way2obj::addtobound(i32 x, i32 y){
9   i32 boundnum;
10  boundnum=((filltact+movetime(x,y))%(maxmovetact+1));
11  boundar[boundnum].add(pointt(x,y));
12 }
13 // caller releasepoint in Way2_.cpp
14 void way2obj::releasepoint(i32 px, i32 py){
15  ...
16  for (y=y1; y<=y2; y++)
17    for (x=x1; x<=x2; x++)
18      if ((x!=px)||(y!=py))
19        if (waymap[x+y*mapsizex].fillnum==fillnum) {
20          ...}
21      else if (isaddtobound(x,y))
22            addtobound(x,y);
23  ...
24 }
```

Listing 3: Code of problematic functions in `astar`

```
133 for (k = 1; k <= M; k++) {
134   mc[k] = mpp[k-1]   + tpmm[k-1];
135   ...
136   if (k < M) {
137     ic[k] = mpp[k] + tpmi[k];
138     if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
139     ic[k] += is[k];
140     if (ic[k] < -INFTY) ic[k] = -INFTY;
141   }
142 }
```

Listing 4: Code for for-loop in fast_algorithms.c from `hmmer` in SPEC CPU2006.

Table 3: `PAPI_SR_INS` instruction counts for code line of `hmmer`.

Code line	HPCToolkit	CCTLib
134	5.23e09	1.35e10
137	1.43e10	1.35e10

shown in Table 3 reveals that the attributed value of line 137 is even greater than that of line 134, which seems unreasonable. Compared with the ground truth, the value of line 134 is under-attributed by more than 50%.

Since we are able to obtain the execution frequency of each basic block, we can recover store instruction profiling result for each code line. Our method reports the same result as the one from CCTLib. The execution frequencies of three basic blocks within this loop are very close. Such mis-attribution is not caused by heavy instructions, but skid effect also incurs mis-attribution at code line level.

6.4 Summary and Discussion

The applications aforementioned show that instruction profiles of a small function frequently called can be easily mis-attributed caused by skid effect, especially when there are heavyweight instructions near the exit of it. The heavyweight instructions can be further categorized into two types:

- Heavyweight memory load instructions.
- Heavyweight arithmetic operations like `mod` and `div`.

When users utilize hardware sampling to profile applications, they have to be aware of such inaccuracy within small hot functions. Our method is able to rectify this limitation of sampling based profiling regardless of the CPU architecture beneath.

6.5 Effectiveness on Other Hardware Events

We have already shown that our algorithm can successfully recover sampling-based instruction profile from mis-attribution, caused by skid effect. For other instruction-related hardware events (e.g. load or store instruction), static analysis is able to characterize each instruction as the instruction type we are interested in. For instance, we can rely on static analysis to determine a specific instruction is memory load instruction or not [35]. Based on this, recovered instruction profile at function level by our algorithm can also successfully deliver the recovered specific instruction type profile at function level. For CPU cycle event, we have explained the CPU cycle events are not affected by skid effects of sampling based profiler, when skid CPU cycle duration is a fixed value.

Hardware-related Events. Hardware-related events like cache miss or memory access, are not evenly distributed on all instructions [44, 48]. There is a probability for each instruction on the control flow graph causing specific these hardware-related events. Modern profilers cannot provide ground truth result for these hardware-related events. Thus we are unable to verify the accuracy of the recovered profile of these events. For these events, the probability of each instruction causing counter overflow is not the same. For instance, every instruction has a different probability causing a cache miss. We are not able to determine the very instruction that triggers the counter overflow based on the mapping from an overflow point to its sample point directly. With all branch execution frequencies, we can also formulate an optimization problem to recover the probability of each instruction causing specific hardware-related event, based on skid effect emulation. With this kind of probability, hardware-related event profile is able to be recovered. Transforming our method to recover hardware related event profile remains an important direction of future work.

7 RELATED WORK

In this section, we review some previous work on edge profiling and handling inaccurate hardware event sampling.

Edge Profile Prediction by Heuristics. Ball et al. [7] propose to collect edge frequency profiles by optimally inserting monitoring code, which incurs acceptable overhead. They also obtain some heuristics about predicting the edge frequency profiles. Moreover, Wu et al. [43] show that such branch frequency heuristics can guide static profiling to obtain estimation of edge execution frequencies. Anderson et al. [6] introduce a two-step framework, correlating the sample count of instructions with execution frequencies heuristics, to improve the prediction accuracy of execution frequencies. Yet our research goes beyond heuristics by building a mathematical model on skid effects in a complex control flow graph.

Accuracy Enhancement of Sampling Based Profiling. Much work has been done to tame inaccurate hardware event sampling result. Levin et al. [22] propose that constructing an edge profile from basic block sample counts can be formalized as a Minimum Cost Circulation problem. Chen et al. [12] extend the Minimum Cost Circulation model by adding additional performance counters to improve the quality of sampling profiles. They apply supervised learning techniques to minimize the skid effect on sampling profiles. A later study [41] by Wu et al. points out that varying the sampling rate does not improve the accuracy of collected profiling result. In a previous exploration [14], Dimakopoulou et al. have studied event scheduling optimization in the Linux kernel to minimize hardware performance counter corruption. Mytkowicz et al. [27] have studied the accuracy of multiple java profilers and found that only sampling at yield points, which is a JVM mechanism for supporting maintenance operations, incurs bias on profiling result. Lim et al. show that intelligently selecting how events are multiplexed based on their rate of change can improve profiler's accuracy [24]. Moreover, Mathur et al. quantify the error caused by events multiplexing and propose new estimation algorithms to improve accuracy [26]. All of these works do not quantify the CPU hardware flaw's impact (skid effect) on profile accuracy. In this paper, we propose a mathematical model to quantify the skid effect in loop-based programs, measure skid duration for different CPUs, and then formulate an optimization problem for control flow graph to eliminate the mis-attribution caused by the skid effect.

Architecture Support for Profiling Accuracy. Intel x86 provides Last Branch Records (LBRs) [4] to continuously record the most recent branches, which can help count basic block execution frequency [23]. Works by Chen et al. [11] and Nowak [28] have proved the effectiveness of instruction profiling in basic block level by utilizing LBRs. However, users must sample event of branch instruction, when user adopts LBR to calculate each basic block's frequency. The sampling result of taken branches could also be affected by skid effects. LBRs cannot eliminate the skid effect, since skid effect is caused by hardware flaw in CPU design. Moreover, not all CPU vendors provide such branch logging facility as Intel, e.g., AMD or ARM. Modern CPUs can often support more advanced forms of sampling, such as Intel's Precise Event-Based Sampling (PEBS). PEBS tries to keep the skid small [39], directly supported by hardware. Even though Nowak [28] claims that PEBS could obtain more accurate profiling result over the standard hardware event sampling method, we find that PEBS still suffers from skid effect, and its sampling result cannot be quantified by an mathematical model with a constant skid. Our method eliminates the skid effect from the software side regardless of the hardware architecture beneath.

8 CONCLUSIONS

This paper describes a framework that rectifies the instruction profile inaccuracy at function level. We design a measurement approach to quantify the skid effect on five different CPUs. Furthermore, we invent a novel software scheme to minimize the skid effect and recover the instruction profiling result from the mis-attribution. We study several CPU2006 benchmarks and real applications to demonstrate the effectiveness of our approach on different CPU architectures. We foresee our scheme can be integrated into modern profilers as an important component to produce accurante measurement.

ACKNOWLEDGEMENT

This project is partially supported by National Science Foundation (NSF) under grant numbers 1618620, 1725743, and 1745813 and a Google Faculty Research Award.

REFERENCES

[1] Complement material. https://github.com/simon4173/ics_complement_materials/blob/master/Complement_ICS19.pdf.

[2] Intel 64 and ia-32 architectures software developer's manual. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html. [Accessed: 10-22-2018].

[3] Intel Vtune. https://software.intel.com/en-us/intel-vtune-amplifier-xe. [Accessed: 08-12-2017].

[4] An introduction to last branch records. https://lwn.net/Articles/680985/. [Accessed: 10-24-2018].

[5] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[6] 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? In *ACM SIGOPS Operating Systems Review*, volume 31, pages 1–14. ACM, 1997.

[7] Thomas Ball and James R Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[9] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 52. ACM, 2011.

[10] Milind Chabbi, Xu Liu, and John Mellor-Crummey. Call paths for pin tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 76. ACM, 2014.

[11] Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang Li, Stephane Eranian, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for precise and versatile feedback directed optimizations. *IEEE Transactions on Computers*, 62(2):376–389, 2013.

[12] 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 *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 42–52. ACM, 2010.

[13] William E Cohen. Tuning programs with oprofile. *Wide Open Magazine*, 1:53–62, 2004.

[14] Maria Dimakopoulou, Stéphane Eranian, Nectarios Koziris, and Nicholas Bambos. Reliable and efficient performance monitoring in linux. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 34. IEEE Press, 2016.

[15] Paul J Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. *Advanced Micro Devices*, 2007.

[16] Alan E Gelfand, Susan E Hills, Amy Racine-Poon, and Adrian FM Smith. Illustration of bayesian inference in normal data models using gibbs sampling. *Journal of the American Statistical Association*, 85(412):972–985, 1990.

[17] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[18] Robert J Hall. Call path profiling. In *Proceedings of the 14th international conference on Software engineering*, pages 296–306. ACM, 1992.

[19] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[20] Donald B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[21] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5, 2007.

[22] Roy Levin, Ilan Newman, and Gadi Haber. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 291–304. Springer, 2008.

[23] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30:18, 2009.

[24] Robert V Lim, David Carrillo-Cisneros, W Alkowaileet, and I Scherson. Computationally efficient multiplexing of events on hardware counters. In *Linux Symposium*, pages 101–110. Citeseer, 2014.

[25] 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 *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[26] Wiplove Mathur and Jeanine Cook. Toward accurate performance evaluation using hardware counters. In *ITEA Modeling and Simulation Workshop*, pages 23–32, 2003.

[27] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Evaluating the accuracy of java profilers. In *ACM Sigplan Notices*, volume 45, pages 187–197. ACM, 2010.

[28] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. Establishing a base of trust with performance counters for enterprise workloads. In *USENIX Annual Technical Conference*, pages 541–548, 2015.

[29] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[30] R. Panda, S. Song, J. Dean, and L. K. John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282, Feb 2018.

[31] S. Song, M. Li, X. Zheng, M. LeBeane, J. H. Ryoo, R. Panda, A. Gerstlauer, and L. K. John. Proxy-guided load balancing of graph processing workloads on heterogeneous clusters. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 77–86, Aug 2016.

[32] S. Song, X. Zheng, A. Gerstlauer, and L. K. John. Fine-grained power analysis of emerging graph processing workloads for cloud operations management. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2121–2126, Dec 2016.

[33] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. Start late or finish early: A distributed graph processing system with redundancy reduction. *Proc. VLDB Endow.*, 12(2):154–168, October 2018.

[34] SPEC Corporation. SPEC CPU2006 benchmark suite. http://www.spec.org/cpu2006. 3 November 2007.

[35] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. Redundant loads: A software inefficiency indicator. *arXiv preprint arXiv:1902.05462*, 2019.

[36] Nathan R. Tallent. *Performance Analysis for Parallel Programs: From Multicore to Petascale*. Ph.D. dissertation, Department of Computer Science, Rice University, March 2010.

[37] Rafael Ubal, Julio Sahuquillo, Salvador Petit, and Pedro Lopez. Multi2sim: A simulation framework to evaluate multicore-multithreaded processors. In *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)*, pages 62–68. IEEE, 2007.

[38] Vincent M Weaver. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, volume 13, 2013.

[39] Vincent M Weaver. Advanced hardware profiling and sampling (pebs, ibs, etc.): Creating a new papi sampling interface. 2016.

[40] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. Watching for software inefficiencies with witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 332–347. ACM, 2018.

[41] Bo Wu, Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, Raul Silvera, and Graham Yiu. Simple profile rectifications go a long way. In *European Conference on Object-Oriented Programming*, pages 654–678. Springer, 2013.

[42] Q. Wu, S. Flolid, S. Song, J. Deng, and L. K. John. Invited paper for the hot workloads special session hot regions in spec cpu2017. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 71–77, Sep. 2018.

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

[44] Hao Xu, Shasha Wen, Alfredo Gimenez, Todd Gamblin, and Xu Liu. Dr-bw: identifying bandwidth contention in numa architectures with supervised learning. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 367–376. IEEE, 2017.

[45] Maotong Xu, Sultan Alamro, Tian Lan, and Suresh Subramaniam. Optimizing speculative execution of deadline-sensitive jobs in cloud. In *ACM SIGMETRICS Performance Evaluation Review*, volume 45, pages 17–18. ACM, 2017.

[46] Maotong Xu, Sultan Alamro, Tian Lan, and Suresh Subramaniam. Chronos: A unifying optimization framework for speculative execution of deadline-critical mapreduce jobs. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 718–729. IEEE, 2018.

[47] Matt T Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34. IEEE, 2007.

[48] Gangyi Zhu and Gagan Agrawal. A performance prediction framework for irregular applications. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 304–313. IEEE, 2018.