2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

CounterMiner: Mining Big Performance Data from Hardware Counters

Yirong Lv
HICAS
Shenzhen Institute of Advanced Technology, CAS
Shenzhen, China
lyr060813@163.com

Bin Sun
Information Engineering College
Capital Normal University
Beijing, China
brad_sun@126.com

Qinyi Luo
Department of Computer Science
University of Southern California
Los Angles, USA
qingyiluo@usc.edu

Jing Wang
Information Engineering College
Capital Normal University
Beijing, China
jwang@cnu.edu.cn

Zhibin Yu
HICAS
Shenzhen Institute of Advanced Technology, CAS
Shenzhen, China
zb.yu@siat.ac.cn

Xuehai Qian
Department of Computer Science
University of Southern California)
Los Angles, USA
xuehai.qian@usc.edu

***Abstract*—Modern processors typically provide a *small* number of hardware performance counters to capture a *large* number of microarchitecture events [1]. These counters can easily generate a huge amount (e.g., GB or TB per day) of data, which we call *big performance data* in cloud computing platforms with more than thousands of servers and millions of complex workloads running in a "$24/7/365$" manner. The big performance data provides a precious foundation for root cause analysis of performance bottlenecks, architecture and compiler optimization, and many more. However, it is challenging to extract value from the big performance data due to: *1)* the many unperceivable errors (e.g., outliers and missing values); and *2)* the difficulty of obtaining insights, e.g., relating events to performance.**

In this paper, we propose *CounterMiner*, a rigorous methodology that enables the measurement and understanding of big performance data by using data mining and machine learning techniques. It includes three novel components: *1)* using data cleaning to improve data quality by replacing outliers and filling in missing values; *2)* iteratively quantifying, ranking, and pruning events based on their importance with respect to performance; *3)* quantifying interaction intensity between two events by residual variance. We use sixteen benchmarks (eight from CloudSuite and eight from the Spark [2] version of HiBench) to evaluate CounterMiner. The experimental results show that CounterMiner reduces the average error from 28.3% to 7.7% when multiplexing 10 events on 4 hardware counters. We also conduct a real-world case study, showing that identifying important configuration parameters of Spark programs by event importance is much faster than directly ranking the importance of these parameters.

***Index Terms*—performance, big data, computer architecture, performance counters, data mining**

I. Introduction

Modern processors typically provide 4-8 hardware counters to measure hundreds of crucial events such as cache and TLB misses [1]–[4]. These events can generally reveal root causes and key insights about the performance of computer systems. Therefore, performance counter based analysis is applied in a wide range of applications, including task scheduling [5], [6], workload characterization [7]–[14], performance optimization of applications [15]–[18], compiler optimization [19]–[21], architecture optimization [22], and many more. A number of programable performance measurement tools have therefore been developed, including PAPI [23], VTune [24], Perfmon [25], Oprofile [26], and many others [27], [28].

However, there is a fundamental tension between accuracy and efficiency when using a *small* number of hardware counters to measure a *large* number of events. On the one side, the one counter one event (OCOE) approach can achieve high *accuracy* because a number of events are measured by the same number of hardware counters at a time. Obviously, OCOE becomes inefficient with more than one hundred and up to fourteen hundreds of measurable events [29].

This motivates the multiplexing (MLPX) approach, which improves the measurement *efficiency* by scheduling events from a fraction of execution to be counted on hardware counters and extrapolating the full behavior of each event from its samples. However, MLPX incurs large measurement errors due to time-sharing and sampling [4], [30]–[32].

In cloud computing era, this problem is exaggerated for two reasons. First, resolving the measurement errors of MLPX is a *requirement*. A modern cloud computing platform usually consists of more than thousand of servers and millions of complex workloads running services with *diverse* characteristics in a "$24/7/365$" manner. The major companies have good incentives to understand the performance behavior because a small performance improvement (e.g., 1%) can result in millions of dollars of savings [7]. In this context, randomly selecting and measuring a small number of events with OCOE is not sufficient because the cloud services' performance may be affected by the unmeasured events. To measure a large number of events, using MLPX and handling its measurement errors are *mandatory*. Prior work shows that the errors cannot

[1]We use events to represent microarchitecture events throughput the paper.
[2]We use Spark to represent Apache Spark throughput the paper.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00056

be effectively avoided *during* the sampling [33], [34].

Second, it becomes more difficult to *extract insights* of performance behavior. The events of server processors in a cloud computing platform can generate a huge amount of data, leading to *big performance data*. For example, GWP (Google-wide-Profiler) [7] could generate several GBs of performance data per day. If we treat the data equally, the high event dimensionality (typically > 100) incurs extremely high cost.

In this paper, we propose *CounterMiner*, a rigorous methodology that enables the measurement and understanding of the big performance data with data mining and machine learning techniques. It includes three components: *1)* data cleaner, which improves the counter data quality by replacing outliers and filling in missing values *after* the sampling of MLPX, which is complementary to [33], [34]; *2)* importance ranker, which iteratively quantifies, ranks, and prunes events based on their importance with respect to performance; *3)* interaction ranker, which quantifies interaction intensity between two events by residual variance.

We use sixteen benchmarks that eight from CloudSuite [35] and eight from the Spark version of HiBench [36] to evaluate CounterMiner. The experimental results show that CounterMiner reduces the average error from 28.3% to 7.7% when multiplexing 10 events on 4 hardware counters. We also conduct a real-world case study, showing that identifying important configuration parameters of Spark programs by event importance is much faster than directly ranking the importance of these parameters.

Moreover, CounterMiner reveals a number of interesting findings: *1)* the event of *stall cycles due to instruction queue full* is the most important event for most cloud programs; *2)* the branch related events interact with other events the most strongly; *3)* there is a *'one-three significantly more important'* law ('one-three SMI' for short) which concludes that generally one to three events of a benchmark are significantly more important than others with respect to performance; *4)* a number of noisy events of a modern processor can be definitely removed; *5)* there are common important events related to branches, TLBs (instruction, data, and second-level TLBs), and remote memory and remote cache operations; *6)* the eight Spark benchmarks from HiBench surprisingly show more diversity than those from CloudSuite when we look at the top ten important events. These findings are valuable to guide cross-layer performance optimizations of architecture and applications of cloud systems.

The rest of this paper is organized as follows. Section II discusses the background and motivation. Section III presents our CounterMiner framework. Section IV describes the experimental methodology. Section V provides the results and analysis. Section VI discusses the related work, and Section VI concludes the paper.

II. Background and Motivation

A. Hardware Counters

Every modern processor has a logical unit called Performance Monitoring Unit (PMU) which consists of a set of hardware counters. A counter counts how many times a certain event occurs during a time interval of a program's execution. The number of counters may vary across microarchitectures or even processor modes within the same microarchitecture. For example, ARM Cortex-A53 MPCore processor has six counter registers [2] while recent Intel processors have three fixed counters (which can only measure specific events such as clock cycles and retired instructions) and eight programmable counters per core (four per SMT thread if it is enabled) [1].

The events that can be measured by hardware counters are predefined by processor vendors. The number of events may also be significantly different for different generations of processors within the same microarchitecture, let alone different microarchitectures. For instance, the basic Ivy-Bridge model defines only 338 events whereas the high-end IvyTown chips support 1423 events [29]. In summary, the number of events greatly outnumbers that of hardware counters (more than one hundred vs. less than ten).

There are two ways to program hardware counters to capture events. In the one counter one event (OCOE) approach, one hardware counter is only programmed for counting one event during the whole profiling period of a program. OCOE is accurate because one counter is dedicated to one event at a time. However, OCOE is ineffective because one usually needs to measure a large number of events to identify the root causes of performance bottlenecks for an unknown system [37]. The number of events that can be simultaneously measured in OCOE is equal to or less than that of the available hardware counters of a processor.

Multiplexing (MLPX) is developed to improve the measurement efficiency by letting multiple events timeshare a single hardware counter [4], [38]. In MLPX, events are scheduled to be sampled during a fraction of execution. Based on the samples, the full behavior of each event is extrapolated [30].

However, large measurement errors occur with MLPX because information may be lost when the event does not happen during a sampled interval [30], [33], [34] but happens during a un-sampled interval. Mathur *et al.* [38] reported that higher than 50% of errors were observed when the SPEC CPU 2000 benchmarks were profiled with MLPX.

B. Motivation

1) Measurement Errors: while some prior works such as [14], [15] employ OCOE to measure performance, MLPX is *mandatory* when a large number of events need to be sampled, e.g., for emerging workloads in cloud computing. Moreover, quantifying measurement errors is a fundamental challenge due to the sampling nature of MLPX.

We conduct the following experiments to observe the errors caused by MLPX. We firstly run a set of cloud computing programs and measure their events by OCOE. Since different runs of the same program in cloud computing platforms may take different times, which is caused by the non-deterministic nature of modern operating systems, the different time series for the same event may have different lengths. The traditional approaches such as calculating the Euclidean or Manhattan

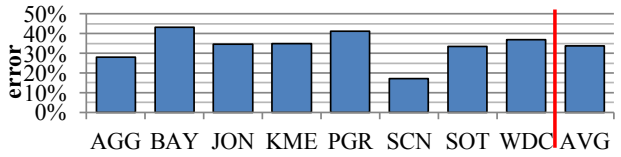

Fig. 1. The measurement errors caused by MLPX. WDC-wordcount, PGR-pagerank, AGG-aggregation, JON-join, SCN-scan, SOT-sort, BAY-Bayes, KME-kmeans, DAA-DataAnalytics, DAC-DataCaching, DAS-DataServing, GPA-GraphAnalytics, IMA - In-memoryAnalytics, MES-MediaStreaming, WSH-WebSearch, WSG-WebServing. AVG-average error.

distance between two vectors cannot calculate the performance behavior differences [39] because they require the two vectors have the same length. In order to compute the distance (difference) between two time series, we must "wrap" the time axis of one (or both) sequences to achieve a better alignment. Dynamic time wrapping (DTW) [40] is a technique for efficiently achieving this wrapping. It employs a dynamic programming approach to align one time series to one another so that the distance measurement is minimized. We therefore employ DTW to calculate the distance between two event time series as follows:

$$dist = DTW(S_1, S_2) \tag{1}$$

where S_1 and S_2 are the first and second time series for the same event, respectively. Note that the length of S_1 is not necessarily equal to that of S_2.

First, we calculate the DTW between two time series collected by OCOE, denoted by $dist_{ref}$.

$$dist_{ref} = DTW(S_{ocoe1}, S_{ocoe2}) \tag{2}$$

where S_{ocoe1} and S_{ocoe2} are the two time series for a certain event of the same program collected by OCOE. Due to the accuracy of OCOE and non-deterministic nature of OS, $dist_{ref}$ is a nonzero value but is theoretically close to zero.

Second, we compute the DTW between one time series collected by MLPX and one by OCOE, represented by $dist_{mea}$.

$$dist_{mea} = DTW(S_{mlpx}, S_{ocoe}) \tag{3}$$

where S_{mlpx} and S_{ocoe} are the time series collected by MLPX and OCOE, respectively. S_{mlpx} and S_{ocoe} are for the same program with the same event. Theoretically, $dist_{mea}$ is larger than $dist_{ref}$ due to MLPX.

Finally, we define the error caused by MLPX as follows:

$$error = |1 - \frac{dist_{ref}}{dist_{mea}}| \times 100\% \tag{4}$$

This error definition roughly quantifies how close the time series generated by MLPX is to the one obtained by OCOE. Since S_{ocoe1} and S_{ocoe2} are both collected by OCOE for the same program with the same event, $dist_{ref}$ is a very small value that is close to zero. In contrast, $dist_{mea}$ is larger than $dist_{ref}$. Thus, $error$ reflects the error caused by MPLX.

Figure 1 shows that the errors caused by MLPX for the event ICACHE.MISSES for 16 programs (the program description is presented in Section IV-B). We can see that the minimum and maximum errors are 8.8% and 43.3%, respectively. The

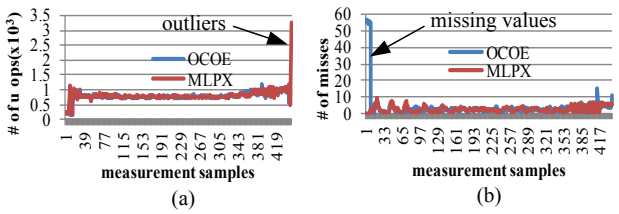

Fig. 2. The outlier and missing value examples in benchmark *WordCount*. (a) The outliers in the time series of event IDQ.DSB_UOPS (the # of uops delivered to Instruction Decode Queue from the Decode Stream Buffer path). (b) The missing values in the time series of event ICACHE.MISSES (the # of instruction cache misses per 1K instructions).

average error achieves 28.3%. For other events, we observe similar or even higher errors.

By carefully analyzing the errors, we find two root causes: *outliers* and *missing values*. Figure 2 (a) shows an example of outliers. In the end of the time series of event IDQ.DSB_UOPS collected by MLPX, the number of UOPS is $4.2\times$ of the normal numbers collected by OCOE. Such outliers will significantly "improve" the overall results if they are taken into account. On the other hand, Figure 2 (b) illustrates an example of missing values. By using OCOE, we observe a large number of instruction cache misses at the beginning of the time series of event ICACHE.MISSES. This is reasonable because at the beginning of a program execution, the instruction cache is empty (a.k.a "cold cache") and a large number of misses must happen. However, these instruction cache misses are not observed by MLPX. These examples indicate that MLPX may indeed cause large errors. Moreover, simultaneously measuring more events in MLPX generally exacerbates the problem, as shown in Figure 3. These errors, however, are difficult to be removed by event scheduling [33], [34] and estimation algorithms [38]*during* the sampling procedure. This motivates us to reduce them by data cleaning techniques *after* the sampling procedure of MLPX, which is very important for mining the CPU big performance data in an off-line manner.

2) Are all events equally important?: Due to the large number of measurable events compared to the available hardware counters (e.g., $207\times$ for HaswellX [29]), MLPX is therefore *mandatory*, but it comes at a high cost. Figure 3 shows that the MLPX errors increase with more events measured simultaneously with the limited hardware counters (red line indicates the trend). To lower the errors, the number of events measured by the same counter during a program's execution needs to be limited. However, this makes the same program need to run many times to measure all the events of the processor being used.

On the other side, measuring all events may not be necessary. First, even if we measure all the events accurately in an ideal scenario at the same time, most values are zeros, which is inconceivable. Second, the optimization overhead would be extremely high when considering all the events of a processor. In fact, to improve performance, an OS can only leverage a small number of events to provide feedback to a runtime system for optimization [29].

Therefore, it is crucial to identify a subset of events that are

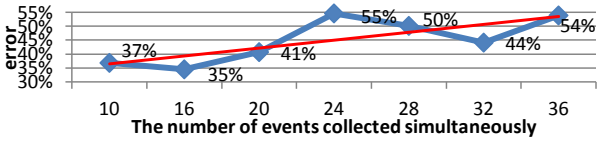

Fig. 3. The error variation with the number of events (represented by the numbers along with X axis) measured simultaneously.

strongly relevant for a given architecture or workload [29]. It can reduce both measurement and optimization overhead. The key to achieve this goal is to quantify the *importance of events*.

III. CounterMiner Methodology

CounterMiner is a methodology designed to mine the big performance data collected from hardware counters. It reduces the measurement errors of MLPX by leveraging data cleaning techniques rather than traditional event scheduling [33], [34] and estimation algorithms [38]. CounterMiner is complementary to [34] [33] [38] because CounterMiner does that *after* (not during) the sampling. Moreover, it quantifies the importance of events and the interactions between two events with respect to performance. Figure 4 shows the workflow of CounterMiner. It consists of four components: data collector, data cleaner, importance ranker, and interaction ranker.

A. Data Collector

The data collector is in charge of sampling the values of events when a program is running. It supports two modes: OCOE and MLPX. The data collector can be any available counter profiling tools such as Perf [41] and Permon2 [25]. In this paper, we use the Linux Perf [41] because it is available in all Linux distributions.

We take the sampled values of an event as a time series since it is important to observe the time varying behaviors of the event [30]. We formally describe the time series as follows:

$$TS_{ei} = \{V_{i1}, V_{i2}, ..., V_{ij}, ..., V_{in}\} \quad (5)$$

where TS_{ei} is the time series for the i^{th} event of a program, V_{ij} is the j^{th} value of the i^{th} event, and n is the total number of sampled values for TS_{ei}. The important feature as well as challenge of these time series is that their lengths may be significantly different even for the same event of the same program because of the non-deterministic behavior of a modern OS. This property makes storing and analyzing these time series challenging.

We store the collected time series in a database management system (DBMS) which can be any popular DBMS such as MySQL and SQL Server. In CounterMiner, we employ SQLite because it seamlessly integrates with Python which is the program language we used for data analysis. In the database, we design a two-level table organization. The first level tables store information including the name of a program, the names of the measured events, the execution times of the program, and the names of the second-level tables. The second-level tables store the time series for the measured events for a program at each run. Note that these two level tables may need to be re-initialized when CounterMiner is applied on a different microarchitecture.

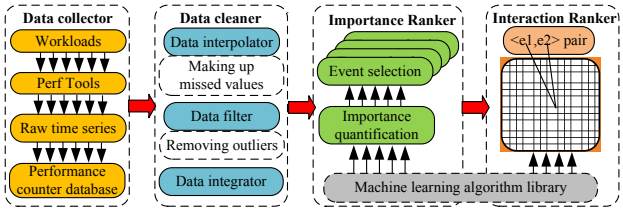

Fig. 4. Block Diagram of CounterMiner.

B. Data Cleaner

The outliers and missing values of events are the main error sources of the collected data with MLPX. Data cleaner replaces the outliers by normal values and fills in the missing values. To evaluate the accuracy of the data cleaner, we take the values of events collected by OCOE as golden references.

1) Replacing Outliers: To determine outliers, we first perform a rigorous statistic testing (see Section IV-C for the testing tool) on the value distributions of all 229 events. We find that, only for 100 events, their values follow Gaussian distribution. We further investigate the value distributions of the other 129 events and find they all show long tail distribution but with different levels. In order to find out the long tail distributions, we perform the statistic testing on them using several different distribution functions such as logistic and Gumbel (general extreme value - GEV) distributions. We find the GEV distribution best fits the long tail distributions, which has been confirmed by [15].

After knowing the value distributions of the events, we design the criterion to replace the outliers. We employ a general technique from data mining to determine outliers by the following equation:

$$threshold = mean + n \times std \quad (6)$$

where $threshold$ is the criterion for replacing outliers, $mean$ and std are the mean value and the standard deviation of a series of values of an event, respectively; n is a control variable which needs to be determined according to the distribution or user requirements such as the percentage of data within the threshold. According to [42], if a data series obeys Gaussian distribution, n equals 3.

Since the values of a large number of events do not obey the Gaussian distribution but instead long tail distribution. We need to determine n by controlling the percentage of data within the threshold. In this study, we specify that 99% of the collected data for events of a program are within the threshold because there are not many outliers based on our confirmed observation. Table I shows the percentage of data within the threshold with different n values. We see that when n is 5, the percentages of data within the threshold for all programs exceed 99%, we therefore set n to 5.

When an outlier is found, we use the median value of the interval where the outlier locates to replace the outlier. The interval length is calculated as follows.

$$L = \frac{Max(TS_{ei}) - Min(TS_{ei})}{Roundup(Sqrt(Count(TS_{ei})), 0)} \quad (7)$$

Benchmarks	n=3	n=5	Benchmarks	n=3	n=5
wordcount	98.8%	99.3%	DataAnalytics	99.3%	99.8%
pagerank	98.6%	99.4%	DataCaching	99.1%	99.8%
aggregation	99.3%	99.8%	DataServing	99.4%	99.8%
join	98.9%	99.8%	GraphAnalytics	99.2%	99.8%
scan	99.3%	99.8%	In-mAnalytics	99.2%	99.8%
sort	98.9%	99.5%	MediaStreaming	99.3%	99.8%
bayes	98.6%	99.8%	WebSearch	98.8%	99.7%
kmeans	99.1%	99.8%	WebServing	99.1%	99.5%

TABLE I
THE COVERAGE RATE WITH DIFFERENT VALUES OF n.

with $Max(TS_{ei})$ and $Min(TS_{ei})$ the maximum and minimum values in TS_{ei}, respectively, $Count(TS_{ei})$ the number of values in TS_{ei}, $Sqrt$ the square root, and $Roundup$ the round up value.

2) Filling in Missing Values: To fill in missing values, we first classify the event values into two categories: zero values and none-zero values. Based on our observation, zero values are highly likely to be missing values. However, it is still possible that the values of a certain event at some points are indeed zeros. It is therefore difficult to distinguish them from the missing values. To address this issue, we check the maximum and minimum values of the event in the past. If the minimum value is zero and the maximum value is less than 0.01, we consider that the zero value for the event is not a missing value. The rationale is that the maximum value of an event is only 0.01 which is very close to zero, and even the actual value is not zero, the error would not be high.

For the none-zero value category, we employ *KNN (K-Nearest Neighbor)* algorithm [43] to fill in missing values. For example, a series of data for ICACHE.MISSES is as follows:

$$\{X_1, X_2, X_3, X_4, X_5, 0, X_7, ..., X_i, ..., X_m\} \tag{8}$$

where X_i is the i^{th} value of ICACHE.MISSES, m is the total number of values, and 0 is the missing value. We use *KNN* regression to calculate the missing value and it is represented by the average value of the *k* nearest neighbors. The determination of k is important because it affects the accuracy. In this study, we tried several values from 3 to 8 based on [42] and find that $k = 5$ is accurate enough to represent the missing value.

C. Importance Ranker

In order to quantify the importance of events with respect to IPC (Instructions Per Cycle), an accurate performance model needs to be constructed. The inputs of the model are event values and the output is IPC, which can be represented by:

$$IPC = perf(e_1, e_2, ..., e_i, ..., e_n) \tag{9}$$

where e_i is the value of the i^{th} event, and n is the total number of events. As discussed before, n is typically larger than 100 and up to 1423 for modern processors. For the processors used in this paper, n is 229. It is extremely challenging to build an accurate performance model with such a large number of input parameters. Analytical modeling techniques are not suitable for this case because they allow only a few parameters (e.g., less than 5). Moreover, analytical models are not accurate when dealing with complex cases. Statistical modeling techniques are either not accurate with high dimensional inputs because they make an unrealistic assumption that the relationship between the input parameters are linear. However, complex nonlinear relationships typically exist between events of modern processors.

To solve this problem, machine learning techniques are applied to construct accurate models with high dimensional inputs. To mitigate over-fitting, we use an ensemble learning algorithm, Stochastic Gradient Boosted Regression Tree (SGBRT) [44], to construct the model. The key insight is that SGBRT combines a number of tree models in a stagewise manner, where each one reflects a part of the performance. The final model is called ensemble model. With performance model constructed, we can leverage the model to quantify the importance of events and their interactions. Clearly, a more accurate performance model results in more precise event importance quantification.

For a single tree T in an ensemble model, one can use $I_j^2(T)$ as a measure of importance for each event e_j, which is based on the number of times e_j is selected for splitting a tree weighted by the squared improvement to the model as a result of each of those splits [45]. This measure of importance is calculated as follows:

$$I_j^2(T) = nt \cdot \sum_{i=1}^{nt} P^2(k), \tag{10}$$

where nt is the number of times e_j is used to split tree T, and $P^2(k)$ is the squared performance improvement to the tree model by the k^{th} split. In particular, $P(k)$ is defined as the relative IPC error which is $(IPC_k - IPC_{k-1})/IPC_{k-1}$ after the k^{th} split. If e_j is used as a splitter in R trees in the ensemble model, the importance of e_j to the model equals:

$$I_j^2 = \frac{1}{R} \sum_{m=1}^{R} I_j^2(T_m). \tag{11}$$

To make the results intuitive, the importance of an event is normalized so that the sum across all events adds up to 100%. A higher percentage indicates stronger influence of the event on performance.

After the importance of each event with respect to performance is obtained, we rank them in a descending order. Then, we remove the 10 least important events and take the remainders as input parameters to construct a performance model by using the SGBRT algorithm again. We conduct the same procedure on the new model to quantify the importance of the remaining events and rank them. This procedure may *iteratively* repeat several times until we obtain the Most Accurate Performance Model (MAPM) for a program and we call this procedure Event Importance Refinement (EIR). The event importance obtained by MAPM is the most accurate [45].

D. Interaction Ranker

After we obtain a set of important events, we construct a linear regression model per pair of the events and consider

the *residual variance* of the model as an indication for interaction intensity. The intuition is that if two microarchitecture events are orthogonal (i.e., they do not interact), the residual variance will be small because the linear model will be able to accurately predict the combined effect of both events. If on the other hand, the microarchitecture events interact substantially, this will be reflected in the residual variance being significantly larger than zero, because the linear model is unable to accurately capture the combined effect of the event pair. The linear regression model is trained for each pair of important events by setting the values of all other events to their respective means. This process is repeated for each possible event pair. The residual variance or interaction intensity for a particular event pair is computed as:

$$v = \sum_{i=1}^{n} (p_i - \overline{p})^2, \tag{12}$$

where p_i is the performance (IPC) predicted by the linear regression model, $\overline{p}$ is the observed performance, and n is the number of predictions. Zero indicates that there is no interaction between two microarchitecture events, and a higher value indicates a stronger interaction.

To indicate the importance of interactions among all possible pairs, we normalize interaction intensity against the other pairs as follows:

$$I_i = \left(\frac{v_i}{\sum_{j=1}^{n} v_j} \right) \times 100\%, \tag{13}$$

where I_i is the importance of the i^{th} event-pair interaction and v_i is the i^{th} event-pair interaction intensity. After the normalization, we can tell how much more/less important an event pair is compared to another event pair, — reflecting the relative interaction intensity of event pairs.

IV. Experimental Methodology

A. Experimental Cluster

Our experimental cluster consists of four Dell servers, one serves as the master node and the other three serve as slave nodes. Each server is equipped with 12 Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz eight-core processor with Haswell-E microarchitecture and 64GB PC3 memory. The OS of each node is SUSE Linux Enterprise Server 12. The OS of the whole cluster is Mesos1.0. Although our experimental cluster is small compared to a cloud platform, we believe that evaluating CounterMiner in this environment is still sufficient, because it is essentially a performance analysis methodology. In a real cloud platform, CounterMiner can easily work with the Google Wide profiler (GWP) to provide more meaningful results. In addition, it can be integrated with cluster management tools such as Quasar [46] and other cloud computing researches such as [47]–[49].

Benchmarks	Framework	Input data
wordcount	Spark 2.0	generated by RandomTextWriter.
pagerank	Spark 2.0	hyperlinks with Zipfian.
aggregation	Spark 2.0	hyperlinks with Zipfian.
join	Spark 2.0	hyperlinks with Zipfian.
scan	Spark 2.0	hyperlinks with Zipfian.
sort	Spark 2.0	generated by RandomTextWriter.
bayes	Spark 2.0	words with Zipfian.
kmeans	Spark 2.0	numbers with Gaussian.
DataAnalytics	Hadoop	a Wikimedia dataset.
DataCaching	Memcached	a Twitter dataset.
DataServing	Cassandra	a YCSB dataset.
GraphAnalytics	GraphX	a Twitter dataset.
In-mAnalytics	SparkMLlib	a move-rating dataset.
MediaStreaming	Nginx	a synthetic video dataset.
WebSearch	Apache Solr	a set of crawled websites.
WebServing	Elgg	generated by Faban.

TABLE II
THE EXPERIMENTED BENCHMARKS.

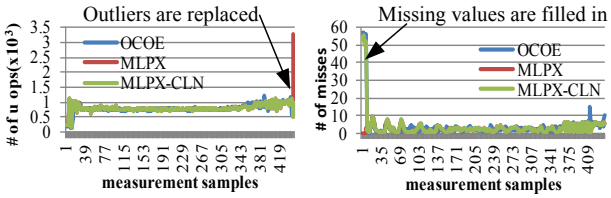

Fig. 5. The event cleaning examples. (a) The replaced outliers in the time series of IDQ.DSB_UOPS. (b) The filled in values in the time series of ICACHE.MISSES. MLPX-CLN represents the cleaned time series.

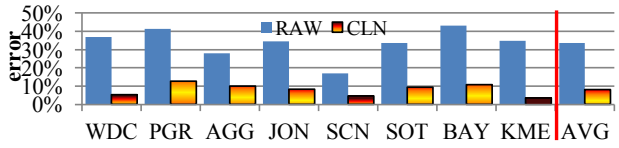

Fig. 6. The measurement error comparison between before and after our data cleaning approach is employed.

B. Benchmarks

We employ CloudSuite 3.0 [35] and select eight programs from HiBench with version of Spark 2.0 [36] (a.k.a "Spark-Bench") to evaluate CounterMiner in this study. CloudSuite 3.0 is a benchmark suite for cloud services and it consists of eight applications based on their popularity in today's datacenters. HiBench with Spark 2.0 consists of a broad set of MapReduce-like programs implemented by Spark 2.0. The benchmarks are listed in Table II. The benchmarks in CloudSuite use different frameworks while the ones in HiBench employ the same framework but represent four categories of applications including websearch, SQL, machine learning, and microbenchmarks.

C. Modeling Tools

We use Python, a freely available high-level programming language, to perform our *SGBRT* modeling and *KNN* modeling. The Python version is 2.7 and we use *scikit-learn 0.19.0* which is a machine learning algorithm library implemented in Python to construct our performance models. To perform a rigorous statistic testing for the value distribution of events, we employ SciPy [50] which is an open-source software for mathematics, science, and engineering. In which, we use

Abbr.	Event	Description
ISF	ILD_STALL.IQ_FULL	stall cycles due to IQ is full.
ISL	ILD_STALL_LCP	counts cycles where the decoder is stalled on an inst with a length changing prefix (LCP).
BRE	BR_INST_EXEC.ALL_BRANCHES	counts all near executed branches (not necessarily retired).
IPD	INST_RETIRED.PREC_DIST	Precise inst retired event with HW to reduce effect of PEBS shadow in IP distribution.
BRB	BR_INST_RETIRED.ALL_BRANCHES	counts the number of retired branch instructions.
BMP	BR_MISP_RETIRED.ALL_BRANCHES	mispredicted macro branch instructions retired.
PI3	PAGE_WALKER_LOADS.ITLB_L3	counts the number of extended page table walks from ITLB hit in the L3.
ITM	ITLB_MISSES.WALK_DURATION	counts the number of cycles while PMH (page miss handler) is busy with the page walk.
DSP	DTLB_STORE_MIS.PDE_CACHE_MIS	DTLB store misses with low part of linear-to-physical address translation missed.
DSH	DTLB_STORE_MIS.STLB_HIT	store ops that miss the first TLB level but hit the second and do not cause page walks.
MMR	MEM_LD_UOPS_L3_MIS.R_M	Retired load uops whose data source was remote DRAM.
MUL	MEM_UOPS_RETIRED.ALL_LOADS	Load uops retired to architected path with filter on bits 0 and 1 applied.
URA	UOPS_RETIRED.ALL	Counts the number of micro-ops retired..
URS	UOPS_RETIRED.RETIRE_SLOTS	counts the number of retirement slots used in each cycle.
MCO	MACHINE_CLEARS.MEMORY_ORDERING	counts the number of machine clears due to memory order conflicts.
MSL	MEM_UOPS_RETIRED.STLB_MISLD	load uops with true STLB (second level TLB) miss retired to architected path.
MLL	MEM_UOPS_RETIRED_LOCK_LOAD	load uops with locked access retired to architected path.
PDM	PAGE_WALKER_LOADS.DTLB_MEMORY	number of DTLB page walker hits in memory.
TFA	TLB_FLUSH.STLB_ANY	count number of STLB (second TLB) flush attempts.
LAA	LD_BLKS_PARTIAL.ADDR_ALIAS	false dependencies in MOB (memory order buffer) due to partial compare on address.
LSF	LD_BLKS.STORE_FORWARD	loads blocked by overlapping with store buffer that cannot be forwarded.
BRC	BR_INST_RETIRED.CONDITIONAL	counts the number of conditional branch instructions retired.
BNT	BR_MISP_RETIRED.NEAR_TAKEN	number of near branch instructions retired that were mispredicted and taken.
LMH	MEM_LD_UOPS_L3_MIRE.R_H	retired load uops whose data sources were a remote HitM responses.
UEP	UOPS_EXECUTED_PORT.PORT_0	cycles during which uops are dispatched from the Reservation Station (RS) to port 0 (on the per-thread basis).
MLH	MEM_LOAD_UOPS_RETIRED.L1_HIT	retired load uops with L1 cache hits as data sources.
MST	MEM_UOPS_RETIRED.STLB_M_ST	store uops with true STLB miss retired to architected path.
IM4	ITLB_MISSES.WALK_COMPLD_4K	code miss in all TLB levels causes a page walk that completes (4K).
IMC	ITLB_MISSES.WALK_COMPLD	misses in all ITLB levels that cause completed page walks.
LHN	LD_UOPS_L3_H_R.X_N	retired load uops which data sources were hits in L3 without snoops required.
CAC	CYC_ACT.CYC_LDM_PEND	cycles with pending memory loads.
C2P	CYC_ACT.STAL_L2_PEND	number of missed L2.
LUO	LSD.UOPS	Number of uops delivered by the LSD.
PLM	PAGE_WALKER_LD.DTLB_MEM	number of DTLB page walker loads from memory.
OTS	OTHER_ASSISTS.AVX_TO_SSE	number of transitions from AVX-256 to legacy SSE when penalty applicable.
I4U	IDQ.ALL_MITE_CYCLES_4_UOPS	counts cycles MITE is delivered four uops. Set Cmask = 4.
ORA	OFFCORE_REQUESTS.ALL_DATA_RD	data read requests sent to uncore (demand and prefetch).
BAA	BACLEARS.ANY	number of front end re-steers due to BPU (Branch Prediction Unit) misprediction.
LRC	L2_RQSTS.CODE_RD_MISS	number of instruction fetches that missed the L2 cache.
MIE	MOVE_ELIMINATION.INT_ELIMINATED	number of integer move elimination candidate uops that were eliminated.
ORO	OFCORE_REQ_OUSTAND.AL_D_RD	Offcore outstanding cacheable data read transactions in SQ (Super Queue) to uncore.
IDU	IDQ.DSB_UOPS	the number of of uops delivered to IDQ (instruction dispatch queue) from DSB (decode stream buffer) path.
LRA	L2_RQSTS.ALL_RFO	counts all L2 store RFO (read for ownership) requests.

TABLE III
EVENT NAME AND DESCRIPTION FOR THOSE APPEARED IN THE 10 MOST IMPORTANT EVENT LIST OF EACH BENCHMARK.

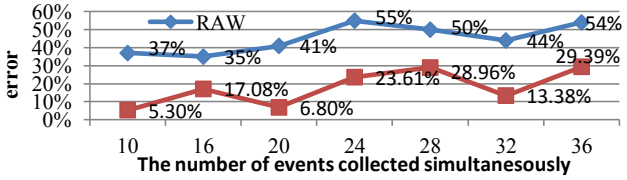

Fig. 7. The measurement error comparison between before and after our data cleaning approach is employed when different number of events are measured simultaneously by MLPX.

scipy.stats.anderson to perform Anderson-Darling test for data coming from a particular distribution.

V. RESULTS AND ANALYSIS

A. Error Reduction

Figure 5 shows the cleaning results for the outlier and missing value examples shown in Figure 2. MLPX-CLN represents the cleaned times series for the corresponding events. The benchmark is *wordcount* in these two examples. Figure 5 (a) illustrates that the outliers are correctly replaced and Figure 5 (b) shows that most missing values are filled in.

Figure 6 compares the measurement errors before (blue bars) and after (red bars) applying our data cleaning techniques on the times series of ICACHE.MISSES for the sixteen benchmarks. We see that our data cleaner significantly reduces the errors caused by MLPX. In particular, the average error is reduced from 28.3% to 7.7% thanks to data cleaning.

Figure 7 illustrates the behavior of the data cleaner when we increase the number of events measured by MLPX at a time. We made several interesting observations. *1)* The data cleaner significantly reduces the errors caused by MLPX in each case. With 10 events, the error is reduced to only 5.3%. *2)* The data cleaner accurately follows the error trend when number of simultaneously measured events increases. *3)* Although no error is larger than 30% in all cases, some errors are high such as 23.6% in the case of 24 events. This indicates that we can not measure too many (e.g., 24) events at the same time even with data cleaner. As a general recommendation, the number should not be larger than 20.

B. Important Events

As mentioned in Section III-C, the event importance obtained by MAPM (the most accurate performance model) is the most accurate [45]. We therefore perform the EIR (event importance refinement) procedure aiming to get MAPM before

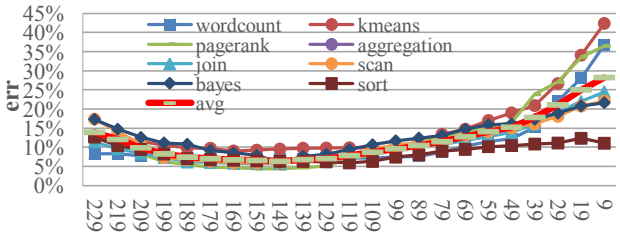

Fig. 8. The error variation of the models when we reduce the number of events used as their inputs for HiBench benchmarks. The X axis represents the numbers of events.

we show the results for important microarchitecture events. During EIR, we employ a number of training examples (m) to train the model and use one-quarter of m unseen test examples to evaluate the model accuracy. The error of the models is defined as follows.

$$err = \frac{|IPC_{meas} - IPC_{pred}|}{IPC_{meas}} \times 100\% \qquad (14)$$

where IPC_{meas} is the measured IPC of a program and IPC_{pred} is the IPC predicted by the performance model.

Figure 8 show the error variation when we perform the EIR for HiBench benchmarks. We see that considering more events may not necessarily result in higher performance model accuracy. For the experimented processors and benchmarks, the average error is 14% when we take all 229 events as the model inputs, while the lowest average error is only 6.3% when around 150 events are taken as the model input parameters. This indicates that the exhausted list of events of modern processors may contain a large number of noisy events. Processor vendors could leverage the proposed approach to systematically select proper events for their processors.

However, the accuracy of performance models decreases when we further reduce the number of input events of the models after they achieve the highest accuracy (e.g., 150 events in this study), as shown in Figure 8. When the number of events decreases to 99, the average performance model error increases to 9.6%, but is still very low. When we decrease the number to 59, the average error further increases to 14% which is the same as that of models with all 229 events. This implies that using only 59 events can achieve the same results of performance analysis by using 229 events. The benchmarks from CloudSuite show the similar results. We therefore do not show the figure due to limited space.

Figure 9 and 10 show the importance ranking of events obtained by MAPM for the benchmarks from HiBench and from CloudSuite, respectively. The Y axis represents event importance and the X axis denotes the abbreviations of events which are shown in Table III. Due to space limit, we show the 10 most important events for each benchmark.We highlight four key findings as follows.

First, *the importance of one to three events of a benchmark is significantly higher* than that of other events of the same benchmark, and this is true for all benchmarks, — both HiBench and CloudSuite. For example, the three most important events of the benchmark *wordcount* are ISF, BRE, and ORA. Their importance exceeds 5% while those of the other events are less than 2.2%. We call this phenomenon *one-three significantly more important law (one-three SMI law)*. Note that this is different from Pareto principle because the accumulated importance of 20% of events is not around 80%. The one-three SMI law indicates that there is always an opportunity to optimize cloud programs significantly more efficiently by first tuning the parameters related to the top one to three events than by tuning other ones. We will demonstrate this in a case study in Section V-D.

Second, *the importance rank of events may vary across benchmarks*. For instance, the most important event for *wordcount* is ISF while that for *pagerank* is BRE. This indicates that different benchmarks have different characteristics at the microarchitecture level.

Third, the *common most important events* for all the experimented cloud benchmarks are related to instruction queue, branch, TLBs (instruction TLB, data TLB, and second level TLB), memory load, and remote memory or cache access. The second insight indicates that application-specific tuning is needed at application level whereas the third one indicates that common optimization approaches for different applications are also effective at lower-level, e.g., microarchitecture or compiler. For example, enlarging the length of instruction queue of processors used in cloud or enhancing the performance of the memory sub-system of cloud servers may improve the performance of most cloud services significantly because our results show that ISF (stall cycles due to instruction queue is full) is the most important event for most experimented benchmarks.

Fourth, we surprisingly find that the *eight benchmarks from HiBench show more diversity than those from CloudSuite* based on the 10 most important events for each benchmark. Only four important events (MUL, MLL, DSP, and DSH) of CloudSuite benchmarks are not included in those of the eight HiBench benchmarks while thirteen important events (ORA, URA, URS, BRC, BAA, LRC, IMC, IM4, CAC, ORO, IDU, LRA, and OTS) of the HiBench programs are not in those of the eight CloudSuite benchmarks. The common belief is that, the benchmarks from CloudSuite should be more diverse than the ones from HiBench because the CloudSuite benchmarks use different frameworks such as Hadoop, Spark, and MemCached while the HiBench benchmarks only use Apache Spark. Our counter-intuitive results indicate that, to achieve application diversity, different frameworks may not be more important than the algorithms and codes of benchmarks. Moreover, our results indicate that more diverse benchmarks need to be included in CloudSite3.0.

C. Important Event Interactions

Figure 11 and Figure 12 show the interaction intensity ranks for the eight Spark benchmarks from HiBench and all the benchmarks from CloudSuite, respectively. Again, we only show the 10 most important interactions of event pairs. The Y axis represents the importance of interaction intensity of event pairs and the X axis denotes the event pairs.

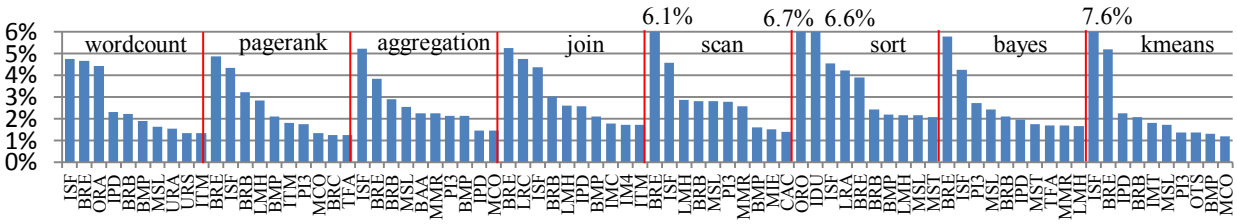

Fig. 9. The importance rank of the eight Spark benchmarks from Hibench when we employ the events which can construct the most accurate performance models. Y axis represents the importance of events and the X axis denotes the abbreviations of the events.

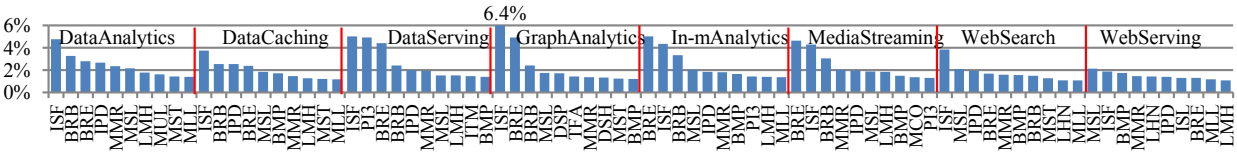

Fig. 10. The importance rank of the eight benchmarks from CloudSuite when we employ the events which can construct the most accurate performance models. Y axis represents the importance of events and the X axis denotes the abbreviations of the events.

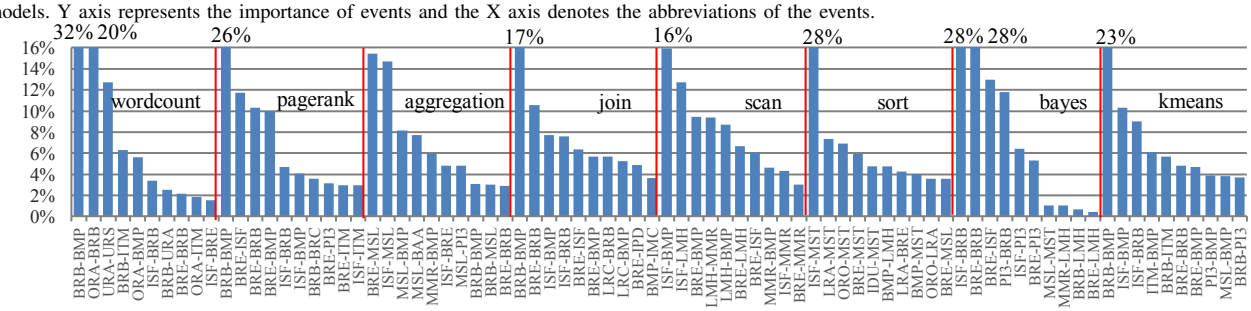

Fig. 11. The interaction rank of the important event pairs for the eight Sark benchmarks from HiBench. The Y axis represents the importance of interaction intensity of event pairs. The X axis represents abbreviations of event pairs. XXX-YYY denotes an event pair.

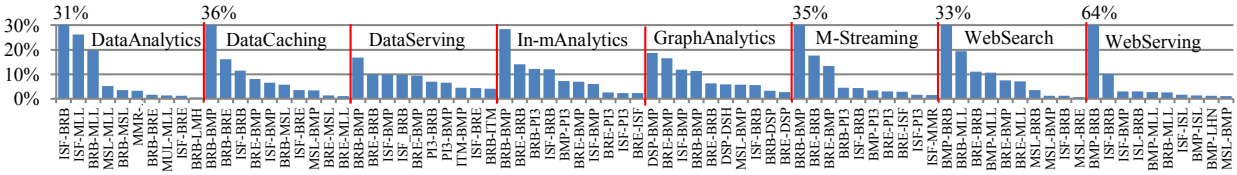

Fig. 12. The interaction rank of the important event pairs for the benchmarks from CouldSuite. The Y axis represents the importance of interaction intensity of event pairs. The X axis represents abbreviations of event pairs. XXX-YYY denotes an event pair.

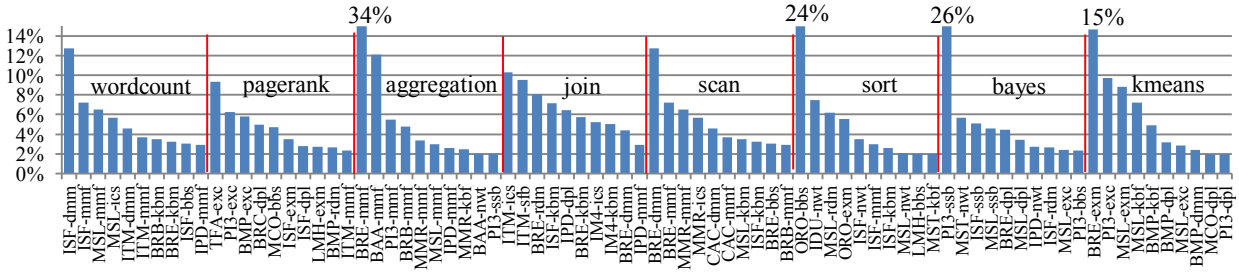

Fig. 13. The interaction rank of spark configuration parameter and event pairs. The Y axis represents the importance of the interaction of configuration parameter and event pairs and the X axis denotes the abbreviations of the parameter and event pairs. In XXX-YYY, XXX represents an event, and YYY represents a configuration parameter.

First, we see that all benchmarks have one or two dominant pairs of events which interact with each other more strongly than other event pairs. This indicates that we can focus on analyzing the dominant interaction pairs with limited time budget. Second, the branch related events interact strongly with other events. In the 160 most important interaction pairs for the 16 benchmarks (10 most important interaction pairs for each), one branch related event is involved in 98 interaction pairs. Two branches related events are involved in 36 interaction pairs. It means that 83.4% of the 160 most important interaction pairs contain branch related events. These results imply that branch related events are critical for cloud computing environment. Moreover, the pair BRB-BMP (see Table III) appears in 12 of the 16 experimented

benchmarks and it is ranked as the most important interaction pair in 10 benchmarks (*wordcount*, *pagerank*, *join*, *kmeans*, *DataCaching*, *DataServing*, *In-memoryAnalytics*, *MediaStreaming*, *WebSearch*, and *WebServing*). This indicates the number of successfully retired branch instructions (BRB) and that of mispredicted but finally successfully retired branch instructions (BMP) interact strongly in most benchmarks. This is because a small BRB surely results in a small BMP and a large BMP is definitely caused by a large BRB.

Another interesting phenomenon is that the the events in dominant interaction pairs of benchmarks from CloudSuite interact much more strongly with each other than those in the dominant pairs of benchmarks from HiBench, see Figure 11 and Figure 12. This indicates that a benchmark containing more software tiers results in stronger interactions between events than a benchmark only implementing algorithms. For example, *WebServing* has four tiers: the web server, the database server, the memcached server, and the clients. The interaction intensity of its dominant interaction pair achieves 64%. In contrast, *GraphAnalytics* only implements the pagerank algorithm on a Spark Library GraphX. The interaction intensity of its dominant interaction pair is only 19%.

Knowing the importance of interaction pairs is important for performance analysis. First, it can explain why one event value changes significantly when the other event value is changed in the same interaction pair. Second, it can explain why performance variation is larger with the change of two event values at the same time than that with the change of one of two event values.

D. Case Study

This case study shows an usage example of CounterMiner. After we know the important events, we first leverage our interaction intensity quantification approach to determine which configuration parameters of the Spark framework strongly interact with the important events. We then tune two configuration parameters, e.g., A and B, which tightly correlate with a more important and a less important event, respectively. Finally, we observe the performance variation when we tuning the two parameters. Note that the default values of Spark configuration parameters can be found at [51].

Figure 13 shows the importance of interactions between a Spark configuration parameter and an event. The Spark configuration parameter names and their abbreviations are shown in Table IV. For each benchmark, we see that there exists one or two pairs of a Spark configuration parameter and an event whose interaction intensities are much stronger than other pairs. This implies that we should tune the configuration parameter in the strongest interaction pair first, which more likely leads to more performance gain. Second, the most important pair of interaction between a Spark configuration parameter and an event varies across benchmarks. It is because of different characteristics between benchmarks and indicates that different configuration parameters should be tuned first for different programs for efficiently optimizing performance.

Abbr.	Configuration Parameter
bbs	spark.broadcast.blockSize.
dpl	spark.default.parallelism.
dmm	spark.driver.memory.
exc	spark.executor.cores.
exm	spark.executor.memory.
ics	spark.io.compression.snappy.blockSize.
kbf	spark.kryoserializer.buffer.
kbm	spark.kryoserializer.buffer.max.
mmf	spark.memory.fraction.
nwt	spark.network.timeout.
rdm	spark.reducer.maxSizeInFlight.
sfb	spark.shuffle.file.buffer.
ssb	spark.shuffle.sort.bypassMergeThreshold.

TABLE IV
THE NAMES AND ABBREVIATIONS OF SPARK CONFIGURATION PARAMETERS THAT ARE INTERACT WITH THE IMPORTANT EVENTS STRONGLY.

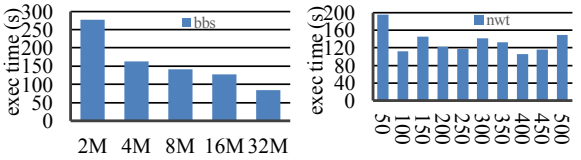

Fig. 14. Execution time (in second) optimization for *sort* by tuning *bbs* and *nwt*. *bbs* tightly correlates with the most important event (ORO) of *sort*. *nwt* tightly correlates with the less important event I4U. See Table III for bbs and nwt, Table IV for ORO and I4U.

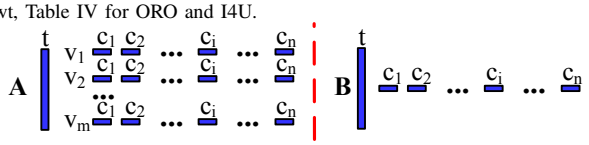

Fig. 15. An illustration of profiling times.c_i denotes the value of the i^{th} configuration parameter; t represents the execution time of a benchmark; v_m denotes the m^{th} value of a certain event.

We study an example to demonstrate how to optimize the performance of Spark programs by using our event importance quantification. As shown in Figure 13, the most important interaction pair of benchmark *sort* is ORO-bbs and Figure 9 shows that ORO is the most important event of *sort*. Looking at Table IV, we know *bbs* corresponds to *spark.broadcast.blockSize*. We then choose an event not in the 10 most important event list and find the Spark configuration parameter that is tightly correlated with it. In this example, we choose I4U and the corresponding configuration parameter is *nwt* (*spark.network.timeout*). We tune *spark.broadcast.blockSize* and *spark.network.timeout* separately and compare the performance variation. Figure 14 shows the results. We see that, the execution time reduction is significantly larger when tuning *bbs* than tuning *nwt*. Specifically, average execution time variation is 111.3% when tuning *bbs* while that is only 29.4% by tuning *nwt*. These results confirm that CounterMiner can indeed provide the "handle" to users to optimize performance more quickly and efficiently.

Nevertheless, one may think that it is unnecessary to identify the important parameters of Spark programs by using our event importance quantification (method A). Instead, one can quantify the importance of parameters directly by using our importance ranker (method B). We argue that it is *untrue*, because method B *takes much longer time* than method A. We use Figure 15 to illustrate the two methods. To quantify the importance of either configuration parameters or events, we need to collect a number of training examples. For method B,

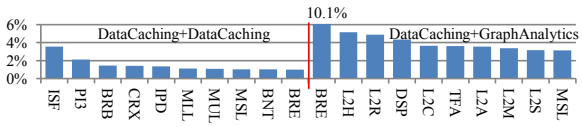

Fig. 16. The importance rank of events for co-located workloads: 'DataCaching + DataCaching' and 'DataCaching + GraphAnalytics'.

we have to run a benchmark k times with k different configurations to collect k training examples because we can collect the execution time of a program only after it completes its execution. In contrast, for method A, we can collect m training examples for an event during *one* execution of a benchmark because we sample a number of values for the event during one run of the benchmark with a certain configuration. Method A therefore needs a much smaller number of benchmark runs than method B.

Taking *pagerank* as an example, we have to run it 6000 times to collect 6000 training examples to build a performance model with around 90% of accuracy by method B. Then we identify the important configuration parameters. For method A, we only run the benchmark 60 times to build a performance model as a function of events with 90% of accuracy. To find the tightly coupled configuration parameter and event pairs, we need to additionally run the benchmark 1520 times. Therefore, we need to run *pagerank* 1580 times in total to identify its important configuration parameters by method A, which is nearly only $1/4$ as the time needed for method B.

E. Co-located Workloads

We now demonstrate how to use CounterMiner with co-located benchmarks running on a shared cluster, which is a typical scenario in cloud computing environment. We consider two cases. *1)* An application is submitted to the cluster to run when the same application is running. *2)* An application is submitted to the cluster to run when another application is running. These are two typical cases people use cloud platforms. In this study, we use '*DataCaching* + *DataCaching*' and '*DataCaching* + *GraphAnalytics*' to demonstrate the first and second case, respectively.

Figure 16 shows the importance ranking of events for the two cases. Note that CounterMiner can not show the importance ranking for individual benchmarks in this context because hardware counters and events are shared resources among co-located benchmarks. The left part shows the importance ranking of events for 'DataCaching + DataCaching'. Compared with Figure 9, we see that the most important event is still ISF with the similar importance of 3.7%. However, the importance order and events in the other top 9 important event list of 'DataCaching + DataCaching' are only slightly different from those of 'DataCaching'. This indicates that two 'DataCaching' programs do not interfere with each other severely.

In contrast, we see that the importance order of events and events of 'DataCaching + GraphAnalytics' are significantly different from both of those of 'DataCaching' and 'GraphAnalytics'. This indicates that 'GraphAnalytics' churns the execution of 'DataCaching' severely. More interestingly, 6 L2 cache related events are ranked in the top 10 important event list for 'DataCaching + GraphAnalytics'. No L2 cache related events have been in the 10 most events for both 'DataCaching' and 'GraphAnalytics'. This indicates that the mixed running of these two benchmarks cause a lot of L1 cache misses for both instruction and data caches, which should be avoided.

As observed above, we see that CounterMiner can capture not only significant churns but also small ones in the context of co-located workloads running in cloud platforms.

VI. Related Work

A. Counter Data Management and Analysis

Google recently developed a profiling infrastructure, named Google-Wide Profiling (GWP), to provide performance insights for cloud applications [7]. GWP employs a two-level sampling technique (sample machines and sample time intervals within a machine for profiling) to collect counter data in Google data centers. Huck *et al.* first developed a framework to manage performance data [28] and then proposed to leverage statistics techniques such as clustering to analyze the performance data of parallel machines [52]. Dong *et al.* proposed to use statistical techniques such as PCA (Principle Component Analysis) to extract important features from performance counters [53]. CounterMiner differs from these studies with twofold: *1)* CounterMiner proposes data cleaning techniques to clean the counter data collected by MLPX; *2)* CounterMiner not only extracts important events but also directly quantifies the importance of an event with respect to performance. The related studies that use PCA or random linear projection can implicitly tell the important events as a form of principle components or projected metrics but can not explicitly quantify how important an event is. This hinders one to directly leverage the important events to optimize application performance.

B. Error Reduction

The measurement errors caused by MLPX have been observed for nearly two decades [29]–[31], [33], [34], [38], [54]–[56] and several approaches were proposed to reduce them. In MLPX, the values of unsampled time intervals of an event are usually estimated by linear interpolating a value between the predecessor and successor intervals. Mathur *et al.* tried to develop a fine-grained estimation algorithm which divides the time interval into several sub-intervals. They found that performing a linear interpolation estimation for each sub-interval results in better accuracy [38]. Weaver *et al.* found that experimental setup significantly affects the accuracy of measurements by hardware counters and they therefore provided corresponding suggestions to reduce the errors [31].

Recently, Lim *et al.* propose a scheduling algorithm that schedules n events on m counters ($n > m$) (vs. the traditional round-robin algorithm) to improve the measurement accuracy [34]. The key idea is to monitor the most recent three values of an event for determining whether another event should be scheduled to monitor on a counter. If the values of an event are not significantly different, another event will

be scheduled and vice-versa. Dimakoupoulou *et al.* found that the measurement error of MLPX increases when Intel hyper-thread is enabled. They then proposed a dynamic event scheduling algorithm based on graph matching to reduce the errors [33]. These studies try to reduce errors *before* or *during* the performance measurement. In contrast, our approach decreases the errors *after* the performance measurement has been completed.

C. Counter Applications

1) Workload Characterization: Recently, Kanev *et al.* leveraged hardware counters to profile a warehouse-scale computer [14] and they found a number of interesting observations. For example, the instruction locality of emerging cloud computing workloads is getting weaker and therefore the instruction cache needs to be redesigned. Ferdman *et al.* employed hardware counters to characterize a group of scale-out workloads and released the CloudSuite [9]. Later on, Yasin *et al.* performed a deep characterization by using hardware counters for the CloudSuite [11], [12]. Jia *et al.* use performance counters to characterize data analysis workloads in datacenters [8]. Wang *et al.* characterized big data workloads for internet services [10]. Xiong *et al.* employed performance counters to characterize the big data analysis in city transportation industry and they proposed a transportation big data benchmark suite [13].

2) Architecture and Compiler Optimization: Kozyrakis *et al.* employ hardware counters and other tools to analyze how large-scale online services use resources in data centers and then they provide several insights for server architecture design in data centers [22]. Chen *et al.* leveraged hardware-event sampling to generate edge profiles to perform feedback-directed optimization for application runtime performance [19]. Moseley *et al.* used hardware counters to optimize compilers and show speedups between 32.5% and 893% on selected regions of SPEC CPU 2006 benchmarks [20].

3) Application Optimization: Chen *et al.* used hardware counters to observe the cache behavior and then proposed a task-stealing algorithm for multisocket multicore architectures [5]. Blagodurov *et al.* developed a user level scheduling algorithm for NUMA multicore systems under Linux by analyzing information from hardware performance counters [6]. By observing the CPI (Cycle Per Instruction) collected from hardware counters, Zhang *et al.* proposed a CPU performance isolation strategy for shared compute clusters [15]. Tam *et al.* firstly carefully analyzed the L2 cache miss rate behavior of commodity systems and subsequently proposed an algorithm to approximate the L2 miss rate curves which can be used for online optimizations [16]. He *et al.* proposed to leverage fractals to approximate the L2 miss rate curves with much lower overhead [17]. Based on hardware counters, Blagodurov *et al.* proposed an algorithm to manage the contention of NUMA multicore systems [18].

Although CounterMiner does not focus on workload characterization and optimization for architectures, compilers, and applications, it provides an important step stone toward these goals. After CounterMiner cleans the performance counter data, these approaches can achieve better results. After CounterMiner quantifies the importance of microarchitecture events, these approaches can be more efficient.

VII. Conclusions

This paper proposes CounterMiner, a methodology that enables the measurement and understanding of the big performance data with three novel techniques: *1)* using data cleaning to improve data quality by replacing outliers and filling in missing values; *2)* iteratively quantifying, ranking and pruning events based on the importance with respect to performance; *3)* quantifying interaction intensity between two events by residual variance. For various applications, experimental results show that CounterMiner reduces the average error from 28.3% to 7.7% when multiplexing 10 events on 4 hardware counters. The real-world case study shows that identifying important parameters of Spark programs by event importance is much faster than directly ranking the importance of parameters.

Acknowledgment

The authors would also like to thank the anonymous reviewers for their valuable comments. This work is supported by the national key research and development program under Grant No. 2016YFB1000204, Shenzhen Technology Research Project (Grant No. JSGG20160510154636747), outstanding technical talent program of CAS, and NSFC under (Grant No. 61672511 and 61702495). The research is also partially supported by the National Science Foundation grants NSF-CCF-1657333, NSF-CCF-1717754, NSF-CNS-1717984, and NSF-CCF-1750656. Zhibin Yu is the corresponding author. Contact: zb.yu@siat.ac.cn.

References

[1] I. Coorporation, "Intel 64 and ia-32 architectures developer's manual," 2017. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html

[2] A. Coorporation, "Arm cortex-a53 mpcore processor technical reference manual," 2014. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf

[3] I. Advanced Micro Devices, "Bios and kernel developer's guide for amd family 10h processors," 2013. [Online]. Available: http://support.amd.com/TechDocs/31116.pdf

[4] J. M. May, "Mpx: Software for multiplexing hardware performance counters in multithreaded programs," in *Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, 2001.

[5] Q. Chen, M. Guo, and Z. Huang, "Cats: Cache aware task-stealing based on online profiling in multi-socket mukti-core architectures," in *Proceedings of the International Conference on Supercomputing*, 2012.

[6] S. Blagodurov and A. Fedorova, "User-level scheduling on numa multicore systems under linux," in *Proceedings of Linux Symposium*, 2011.

[7] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE Micro*, vol. 30, no. 4, pp. 65–78, 2010.

[8] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[9] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[10] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: a big data benchmark suite from internet services," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[11] A. Yasin, Y. Ben-Asher, and A. Mendelson, "Deep-dive analysis of the data analytics workload in cloudsuite," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[12] A. Yasin, "A top-down method for performance analysis and counters architecture," in *Proceedings of the IEEE International Symposium on Performance Analysis of Sytems and Software (ISPASS)*, 2014.

[13] W. Xiong, Z. Yu, L. Eeckhout, Z. Bei, F. Zhang, and C. Xu, "Shenzhen transportation system (szts): A novel big data benchmark suite," *Journal of Supercomputing*, vol. 72, pp. 4337–4364, 2016.

[14] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[15] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2:cpu performance isolation for shared compute clusters," in *Proceedings of European Conference on Computer Systems (EuroSys)*, 2013.

[16] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "Rapidmrc: Approximating l2 miss rate curves on commodity systems for online optimizations," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[17] L. He, Z. Yu, and H. Jin, "Fractalmrc: Online cache miss rate curve prediction on commodity systems," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[18] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for numa-aware contention management on multicore systems," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2011.

[19] D. Chen, N. Vachharajani, R. Hundt, S. wei Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng, "Taming hardware event samples for fdo compilation," in *Proceedings of the International Symposium on Code Generation and Optmization (CGO)*, 2010.

[20] T. Moseley, D. Grunwald, and R. Peri, "Optiscope: Performance accountability for optimizing compilers," in *Proceedings of the International Symposium on Code Generation and Optmization (CGO)*, 2009.

[21] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[22] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server engineering insights for large-scale online services," *IEEE Micro*, vol. 30, no. 1, pp. 8–19, 2010.

[23] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, no. 0, pp. 189–204, 2000.

[24] Intel, "Intel vtune amplifier," 2017. [Online]. Available: https://software.intel.com/en-us/intel-vtune-amplifier-xe/

[25] P. Team, "Perfmon2: the hardware-based performance monitoring interface for linux," 2017. [Online]. Available: http://perfmon2.sourceforge.net/docs_v4.html

[26] O. Team, "Oprofile," 2017. [Online]. Available: http://oprofile.sourceforge.net/news/

[27] L. Adhianto, S. Banerjee, M. Fagan, M. Krental, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, pp. 1–7, 2008.

[28] K. A. Huck, A. D. Malony, R. Bell, and A. Morris, "Design and implementation of a parallel performance data management framework," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2005.

[29] G. Zellweger, D. Lin, and T. Roscoe, "So many performance events, so little time," in *Proceedings of the ACM Asia-Pacific Workshop on Systems (APSys)*, 2016.

[30] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan, "Time interpolation: So many metrics, so few registers," in *Proceedings of IEEE International Symposium on Microarchitecture*, 2007.

[31] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted," in *Proceedings of IEEE International Symposium on Workload Characterization*, 2008.

[32] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong," in *Proceedings of the 14th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, 2009.

[33] M. Dimakopoulou, S. Eranian, N. Koziris, and N. Bambos, "Reliable and efficient performance monitoring in linux," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2016.

[34] R. V. Lim, D. Carrillo-Cisneros, W. Y. Alkowaileet, and I. D. Scherson, "Computationally efficient multiplexing of events on hardware counters," in *Proceedings of the Ottawa Linux Symposium*, 2014.

[35] D. Ustiugov, Z. Tian, M. Sutherland, A. Pourhabibi, H. Kassir, S. Gupta, M. Drumond, A. Daglis, M. Ferdman, and B. Falsafi, "Cloudsuite 3.0: A benchmark suite for cloud services," 2017. [Online]. Available: http://cloudsuite.ch//pages/download/

[36] Intel, "Sparkbench: The big data micro benchmark suite for spark 2.0," 2016. [Online]. Available: https://github.com/intel-hadoop/HiBench/blob/master/docs/run-sparkbench.md

[37] T. Moseley, N. Vachharajani, and W. Jalby, "Hardware performance monitoring for the rest of us: A position and survey," in *Proceedings of IFIP International Conference on Network and Parallel Computing (NPC)*, 2011.

[38] W. Mathus and J. Cook, "Toward accurate performance evaluation using hardware counters," in *Proceedings of the ITEA Modeling and Simulation Workshop*, 2003.

[39] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[40] D. J. Berndt and J. Clifford, "Using dynamic time warping to find pattern in time series," in *Proceedings of the AAAI-94 Workshop on Knowledge Discovery in Databases (KDD)*, 1994.

[41] linux community, "perf: Linux profiling with performance counters," 2015. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page#perf:_Linux_profiling_with_performance_counters

[42] J. Han, M. Kamber, and J. Pei, *Data Mining — Concetps and Techniques*, 3rd ed. New York: Morgan Kaufmann, 2012.

[43] N. Altman, "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, February 2012.

[44] J. H. Friedman, "Stochastic Gradient Boosting," *Computational Statistics and Data Analysis*, vol. 38, no. 4, pp. 367–378, October 2002.

[45] J. H. Friedman and J. J. Meulman, "Multiple Additive Regression Trees with Application in Epidemiology," *Statistics in Medicine*, vol. 22, no. 9, pp. 1365–1381, May 2003.

[46] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 1–17.

[47] ——, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Proceedings of International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 1–12.

[48] ——, "Hcloud: Resource-efficient provisioning in shared cloud systems," in *Proceedings of Twenty First International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 1–15.

[49] ——, "Bolt: I know what you did last summer... in the cloud," in *Proceedings of Twenty Second International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 1–15.

[50] S. Team, "Scipy: An open-source software for mathematics, science, and engineering," 2018. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/index.html

[51] A. S. Team, "Hive performance benchmarks," 2018. [Online]. Available: https://spark.apache.org/docs/latest/configuration.html

[52] K. A. Huck and A. D. Malony, "Perfexplorer: A performance data mining framework for large-scale parallel computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2005.

[53] D. H. Ahn and J. S. Vetter, "Scalable analysis techniques for microprocessor performance counter metrics," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2002.

[54] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou, "Experiences and lessons learned with a portable interface to hardware performance counters," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.
[55] G. Ammons, T. Ball, and J. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *ACM SIGPLAN Notices*, vol. 32, 01 1999.
[56] V. M. Weaver, D. Terpatra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.