2017 IEEE International Parallel and Distributed Processing Symposium

SimProf: A Sampling Framework for Data Analytic Workloads

Jen-Cheng Huang[1], Lifeng Nai[2], Pranith Kumar[2], Hyojong Kim[2], and Hyesoon Kim[2]

[1]Oracle Corporation
[2]Georgia Institute of Technology
{*tommy24, lnai3, pranith, hyojong.kim, hyesoon.kim*}*@gatech.edu*

***Abstract*— Today, there is a steep rise in the amount of data being collected from diverse applications. Consequently, data analytic workloads are gaining popularity to gain insight that can benefit the application, e.g., financial trading, social media analysis. To study the architectural behavior of the workloads, architectural simulation is one of the most common approaches. However, because of the long-running nature of the workloads, it is not trivial to identify which parts of the analysis to simulate.**

In the current work, we introduce SimProf, a sampling framework for data analytic workloads. Using this tool, we are able to select representative simulation points based on the phase behavior of the analysis at a method level granularity. This provides a better understanding of the simulation point and also reduces the simulation time for different input sets. We present the framework for Apache Hadoop and Apache Spark frameworks, which can be easily extended to other data analytic workloads.

***Keywords*—performance modeling; sampling; architectural simulation;**

I. Introduction

Data analytic workloads are gaining importance because of the rapid increase in the volume of data being collected. Enterprise companies are looking to gain valuable insights using data analytics. For example, using social media analysis, a company can adjust its pricing and promotion on the fly for optimal results. To process the huge data volume, data analytic workloads are typically built on top of a computing framework, e.g., Apache Hadoop [1], Apache Spark [2] to scale out to multiple nodes. In addition, the framework handles how the workloads are scheduled onto multiple nodes and provides reliability to tolerate node failures.

Architectural simulation is the most common approach to understand the workload behavior. However, because of the slow simulation speed, it is not possible to simulate entire workloads. Therefore, various approaches have been proposed to sample the instructions to be simulated [3], [4]. For server workloads, so far mostly transaction-based workloads are assumed, e.g., SpecWeb [5], which have a large number of short and atomic transactions. To simulate this server workload, the common practice is to simulate a time interval that covers a sufficient number of transactions, e.g., 10 seconds [6], [7], [8]. However, simulating a single interval is not suitable for data analytic workloads for various reasons.

From an accuracy perspective, a single interval cannot represent the behavior of the entire workload. Data-analytic workloads typically have several stages, each of which has multiple tasks being executed. Furthermore, the task length of the workload can be much longer than transaction-based workloads because of the complexity of the queries being processed. The duration of a task varies from few hundred milliseconds to tens of seconds while more tasks are spawned to process larger data sets.

From an efficiency perspective, even a 10-second interval demands a long simulation time e.g., 20 days for simulating 10 seconds of a 64-core hadoop-based data analytic workload [9]. Other optimization techniques, such as check-pointing and parallel simulation, help reduce the simulation time, but the space overhead and the simulation complexity still remain as challenges that need to be overcome.

In this work, we propose SimProf, which is the first sampling framework to generate simulation points for data analytic workloads based on *phase analysis*. SimProf has the following key features: First, it runs natively on a real machine to achieve high profiling speed. It uses the standard profiling interfaces, such as Java Virtual Machine Tools Interface (JVMTI) [10] and perf_event kernel API, that can be easily integrated into any functional simulator. Second, it identifies the phases using call stacks and applies statistical sampling methods to select the final simulation points, which can sample the phases that do not have homogeneous performance. Third, it detects input-insensitive phases, whose performance is not input dependent. When simulating multiple inputs with different characteristics, the simulation time can be further reduced by skipping input-insensitive phases. Fourth, the simulation points contain the method-level information making it easier for the architects to analyze the program behavior. Finally, the validation is done against the performance of a real machine rather than a simulator to demonstrate the robustness of the proposed SimProf technique.

Our contributions are as follows.

- We propose the first sampling framework, SimProf, based on phase analysis for data analytic workloads, which are built on top of the commonly used computing frameworks like Apache Hadoop and Spark.
- We propose to apply stratified random sampling, a statistical sampling approach to select the representative simulation points by taking into account the factors that cause performance variation within a phase.
- We propose an input sensitivity test to reduce simulation points when exploring multiple inputs. This helps eliminate the simulation points whose performance is not input sensitive.

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.118

IEEE computer society

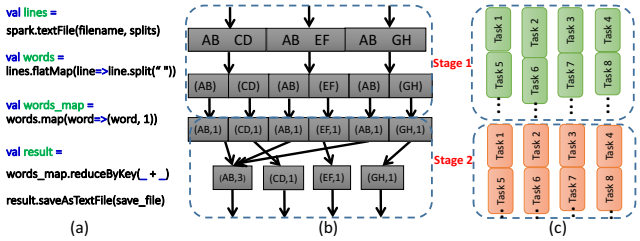

Fig. 1. WordCount Example

- SimProf provides the method-level information of each phase, which may help architects to understand the performance bottlenecks of data analytic workloads with managed runtime, e.g., Java.

II. Background and Motivation

A. Apache Spark

Apache Spark is a computing framework for distributed computing. It leverages in-memory computation to provide performance up to 100 times faster than Hadoop for some applications. Several libraries, such as Spark SQL and GraphX are built on top of the core functionality in Spark to support different types of workloads. We explain the various phases in the execution of a data analytic workload.

Figure 1 shows how the benchmark `wordcount` is implemented and executed in Spark. Figure 1(a) shows the source code of `WordCount` written in Scala [11]. Line 1 is to read the files from HDFS [1], which is a distributed file system, and to put all lines into `lines`, which is a Resilient Distributed Dataset (RDD). An RDD is an immutable collection of data elements. Multiple operations can be performed on an RDD, such as map and reduce operations. After applying an operation on every element in an RDD, the original RDD is transformed to a new RDD. For example, each line between lines 2 and 4 applies an operation, and each operation results in a new RDD (`words`, `words_map` and `result`). Finally, the final RDD, `result`, is saved to a text file.

Figure 1(b) shows the execution flow. For the first three operations, the processing of a data element in an RDD can be done in parallel, but for the fourth operation (`reduceByKey`), the processing of a data element in `words_map` also involves the data elements with the same key. So, it has to wait until the first three operations are completed before proceeding. Therefore, the first three operations form the first stage, while the other two operations form the second stage. In each stage, multiple tasks are spawned, as shown in Figure 1(c). Each task processes a "subset" of an RDD and applies multiple operations, e.g., three operations in stage 1. If stages have dependencies as in this example, the tasks in different stages are serialized although the tasks within the same stage can be executed concurrently.

B. Phase Behaviors

We define "phase" as a set of sampling units that execute similar code. A sampling unit is a fixed number (e.g., 100M) of instruction interval within a thread. Non-contiguous sampling units could belong to the same phase.[1] We categorize the phase behaviors into intra-stage and inter-stage.

- **intra-stage**: Multiple operations are executed within each task. Depending on the duration of each operation, several phases could be formed. One example is that data processing and data IO can be in different phases. Initially, a task may be busy processing the data and generate the outputs to a memory buffer. When the memory buffer is close to full, it flushes the data to the disk to be used by the next stage. In addition, the framework operations used for data movement between nodes- like data shuffling - also occur within the stage.
- **inter-stage**: The tasks in different stages may belong to different phases since they could execute different operations, e.g., map and reduce operations.

Phase analysis is a commonly used technique for selecting simulation points. The idea is to group the sampling units that have similar execution into a phase. For each phase, a simulation point is selected to represent the behavior of the phase. However, the commonly-used tool of selecting simulation points, e.g., SimPoint [4], is not suitable for data analytic workloads for the following reasons.

First, most data analytic workloads are written in high-level languages, such as Java, Python or Scala, which use managed runtime to achieve platform independence. However, SimPoint collects the basic block counts (BBC) as the code signature while collecting BBC for Java applications is not trivial and does not provide the method-level information. The method-level information can provide useful insights for the architects to understand the architectural behaviors. Furthermore, collecting BBC incurs high profiling overhead, the range of which causes anywhere from 100% to 400% slowdown [12]. Finally, although porting the workloads to other languages, such as C++, is possible, it requires nontrivial porting effort and whether the behavior is similar to the Java-based implementation is unknown.

Second, using the code signature solely to classify program behavior is insufficient. We corroborate with the previous study [13] that the performance of server workloads could be dominated by other factors, such as the *data access behaviors*, while the same code may have heterogeneous performance because of different last-level-cache miss rates. Thus, without being aware of the data-sensitive behavior within a phase, a single simulation point is insufficient to represent the performances of other points of the phase.

Third, the data diversity, which can be characterized, in "4V" [14] , is significant for the data analytic workloads. Previous tools did not consider the input data impact on performance since only a few inputs exist for the conventional benchmark suites, such as SPEC CPU and Parsec. However, for the data analytic workloads, the data synthesizer is usually used to synthesize data sets with different volumes based on the real-world seed inputs while retaining the characteristics

[1]Essentially a phase is the same as a cluster in this paper, but we use the term phase because it is commonly used in program profiling contexts.

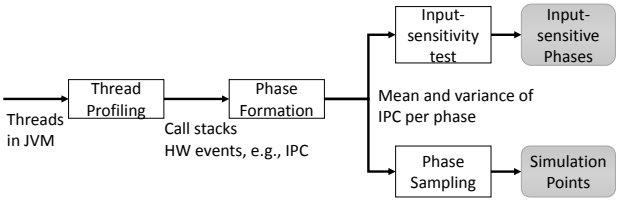

Fig. 2. SimProf Overview

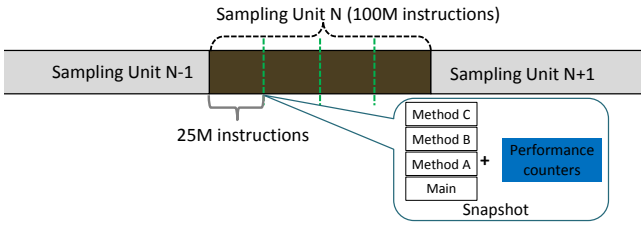

Fig. 3. Snapshots in a sampling unit

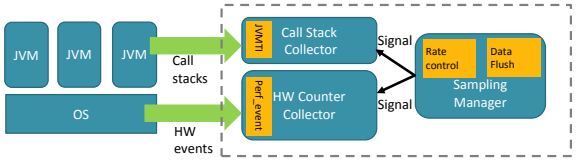

Fig. 4. Thread profiler in SimProf

of the inputs. Thus, it is important to understand how the performance of a phase changes by inputs. We will show that the number of simulation points can be reduced significantly by simulating only the input-sensitive phases.

Our goal is to sample only one executor thread based on its phase behavior. The design is based on the observation that in each execution stage, executor threads are executing the same code. On the other hand, since different computing frameworks may adopt different programming models, the sampling framework should be general enough to be applied to any computing frameworks. For example, in Apache Hadoop [1], the users define the map and reduce operations applied on the data while in Spark, the users can define the types of data collections (RDD types) as well as the operations, which are not limited to map and reduce operations, e.g., union operation.

III. SimProf

Figure 2 shows an overview of the SimProf framework. *Thread Profiling* profiles every executor thread within a JVM to retrieve the call stacks and the hardware counter events. *Phase Formation* vectorizes the call stacks and clusters them into phases. *Phase Sampling* leverages the statistical sampling approach, stratified random sampling, to select the simulation points based on phases. *Input-Sensitivity test* uses the mean and variance of the IPC measured of each phase to detect input-sensitive phases and architectural behavior sensitive to the inputs.

A. Thread Profiling

To detect the code executed in a sampling unit, the call stacks within the unit are collected. A call stack is a record of the active stack frames at a certain point in time during the execution. Additionally, the hardware counters, such as IPC and cache miss rate, are collected for validation and sampling. To reduce the profiling overhead, we periodically take snapshots of the call stacks in a sampling unit, as shown in Figure 3.

Figure 4 shows the infrastructure for thread profiling. The call stack collector retrieves the call stacks of JVM through the JVM tool interface (JVMTI), which is the standard profiling interface available in all Java implementations. The hardware counter collector retrieves the hardware counter values through the `perf_event` interface provided by the Linux kernel. Both counters are controlled by the sampling manager, which controls the sampling rate, e.g., the frequency of a snapshot. It is also responsible for flushing the outputs of the collectors to the files since the collectors initially output to the memory for fast collection speed.

The profiling infrastructure is attached to each JVM while SimProf profiles on a per-executor-thread basis in the JVM. Depending on the execution model of the profiling target framework, the lifetime of an executor thread can be different. In Spark, the lifetime of an executor thread is equal to the total time of a job. Profiling only one executor thread can cover different stages. However, in Hadoop, the lifetime of an executor thread is equal to the lifetime of a task that it executes. So, the executor thread dies when its task is finished. In this case, the profiler merges the profiled results from the executor threads running on the same core to mimic a long running executor thread in Spark.

Our design leverages the standard profiling interfaces so that it is applicable to any Java-based computing framework, not limited to Spark or Hadoop. The sampling unit size and the frequency of a snapshot can be tuned based on the users' need. Empirically, we use a large sampling unit size, 100M instructions to avoid the simulation start-up effect, e.g., cold cache. The frequency of a snapshot affects the profiling overhead. The high frequency can slow the application and skew the results of hardware counters while the low frequency may miss important call stacks executed in a sampling unit. We take snapshots every 10M instructions to have negligible profiling overhead while having a sufficient number of call stacks.

B. Phase Formation

Phase formation groups the sampling units into a phase if they have similar call stacks. Phase formation contains two steps. First, it converts the call stacks in one sampling unit into a feature vector. Second, it clusters the feature vectors into phases.

Figure 5 shows an example call stack that represents the IO routine commonly seen in Spark. From levels 1 to 4, it represents the starting methods of an executor thread. Then it is followed by the task routine, which indicates that a

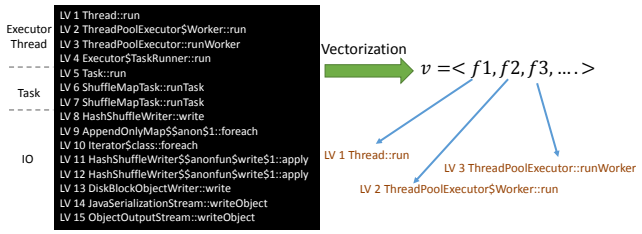

Fig. 5. Convert call stacks into a feature vector

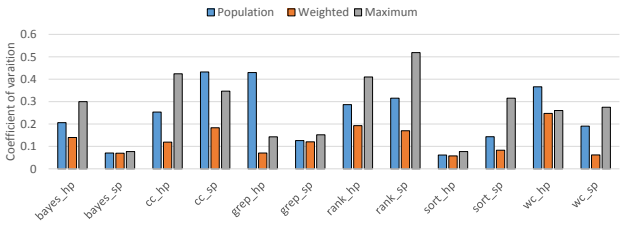

Fig. 6. Coefficient of variation of CPIs

map task is currently executed. Finally, several IO-related methods, such as object serialization and disk writing, are called. Note that the figure only shows only one call stack while one sampling unit typically contains multiple call stacks. All methods appearing in the call stacks in one sampling unit need to be counted and converted into a feature vector. Each dimension of the vector represents a method while the value of the dimension is the frequency of the method appearing in the sampling unit. In addition, the number of dimensions in a feature vector is equal to the number of unique methods in the entire job execution so that all feature vectors have the same number of dimensions.

However, this type of feature vector has the following problems. First, a feature vector can easily have thousands of dimensions because a large number of methods are called, which often results in a high clustering time. Second, the high dimensions often have challenges in identifying the hot functions that have the most impact on performance.

To select the important methods (features), we use a linear regression to identify the methods that are highly correlated with performance i.e., IPC. The univariate linear regression test [15] selects highly correlated top K methods. We set K as 100 to balance the clustering speed while still capturing important methods. For example, in Figure 5, the starting methods of the executor thread and the tasks are eliminated in the feature vector since those are common in most feature vectors and have no significant impact on performance.

We use the k-means clustering algorithm to group the sampling units into clusters (phases). To determine the number of phases k, we score the fitness of each k between 1 and 20 using the silhouette coefficient. Then, we choose the smallest k, which has at least 90% of the highest score among all k.

1) Phase Homogeneity Analysis: To understand how similar the performance is in each phase, we calculate the coefficient of variation (CoV) of cycles-per-instruction (CPI). The CoV is a good metric to indicate how homogeneous (i.e., how much the performance is uniform in a phase) the collection of sampling units is. A higher CoV means that the sampling units have a higher CPI variation.

Figure 6 shows that the population/weighted/maximum CoV of CPIs. We calculate the weighted CoV, which is the CoV of each phase weighted by the number of sampling units in the corresponding phase. The maximum CoV is the CoV of the phase that has the highest CoV. The population CoV is the CoV of all sampling units. The purpose of the figure is to demonstrate how well the phase formation performs. In an ideal case, the phase formation results in a low performance variation of each phase. In this case, the weighted CoV is low even if the population CoV is high. By contrast, if the phase formation is not useful, it cannot separate high-performance phases from low-performance phases. In this case, the weighted CoV remains high when the population CoV is high. The figure shows that the weighted CoVs from all benchmarks are always lower than the population CoVs, so we can safely conclude that our phase formation separates phases with distinct performances. However, phase formation does not necessarily construct all phases to be homogeneous, as we see from the maximum CoV. We summarize the reasons for non-homogeneous phase behavior as follows.

- **Data access pattern:** For these key-values applications, if the reduce stage exists, it involves the sorting algorithm, which sorts the key-value pairs by the key since the reducer needs to process the data in the *per-key basis*. The common sorting algorithm used is quicksort, which sorts the keys recursively from small to large partitions. Although the sampling units execute the same sorting algorithm, some of them have a lower cache miss rate because of sorting the small partitions, while others have a higher miss rate because of sorting the large partitions. For example, `wc_hp` spends a significant amount of time in sorting keys (words) before the count of each word can be calculated. Thus, the non-homogeneous sorting phase results in a high weighted CoV.
  Another type of changing data access pattern is the reduce operation, which combines the values of the same key. For all key-value pairs, the values with the same key may not be next to each other leading to random accesses. Thus, for sort and reduce operations, the LLC cache miss rates could affect the overall IPC.
- **OS scheduling:** We observe that sometimes the executor thread can be scheduled to other cores by the OS. For the sampling units that involve the newly scheduled threads, higher D-cache misses could occur leading to a higher CPI.
- **Phase Interleaving:** The phase behavior of executor threads can be interleaved in a random order depending on how tasks are scheduled within threads. Even though the same phase behavior is observed in two different sampling units, they may have diverse performances since they are interleaved with different phases of other executor threads.
- **Executed code difference:** To make a phase have homo-

geneous behavior, the sampling units in a phase need to have nearly identical call stacks to ensure that they have similar behavior. However, in some cases, since the large size of code is executed, a phase could have sampling units that have different code leading to performance differences.

Because non-homogeneity can occur within a phase, instead of assuming that all phases have homogeneous behavior and selecting one sampling unit for each phase, we need a reliable way to select a set of simulation points within a phase to better represent the behavior of the entire phase.

C. Phase Sampling

The phase sampling selects a number of sampling units within each phase as the final simulation points. To deal with the performance variation within a phase, we use the optimal allocation [16] to determine the sample size of each phase and use simple random sampling to select sampling units for each phase. This is a statistical sampling approach, known as stratified random sampling [16]. We refer to the "sample" as the set of selected sampling units (simulation points) and the "sample size" as the number of these units.

The idea behind optimal allocation is to have a larger sample in the phase that has a higher IPC variance. Since the hardware counter events of every sampling unit, such as IPC, can be obtained from the frontend data collector (Section III-A), the variance of each phase also can be measured.

Eq. 1 shows the sample size of a phase h determined by the optimal allocation. n is the overall sample size. For each phase h, N_h is the total number of sampling units, n_h is the sample size and σ_h is the standard deviation of phase h. The sample within each phase is randomly selected. We call the selected sampling units the final "simulation points." We use the sampling unit ID to represent each simulation point. On the other hand, the cluster center of each phase is also saved and will be used for the input sensitivity test.

$$n_h = \frac{n \times (N_h \times \sigma_h)}{\sum_{i \in phases}(N_i \times \sigma_i)},\ h \in phases \tag{1}$$

The sampling error of the simulation points is represented as the confidence interval, which can be calculated as follows. Eq. 2 shows how the confidence interval is computed. Eq. 3 shows the margin of an error defined by the confidence level $(1-\alpha)$ and the standard error (SE). Users can specify the required the confidence level while the standard error depends on the sampling technique. We assume the distribution of the means (the average CPI of the simulation points of a phase $(\overline{CPI_h})$) is normal because of the central limit theorem. In this case, α can be expressed in z-score form.

$$CI = sample\_mean \pm margin\_of\_error \tag{2}$$

$$margin\_of\_error = \alpha \times SE \tag{3}$$

Eq. 4 shows the standard error (SE) of stratified sampling. Eq. 5 shows s_h, which is the standard deviation of the phase. We use the collected CPIs of each sampling unit to get s_h.

$$SE = \frac{1}{N} * \sqrt{\sum_{h \in phases} [N_h^2 \times (1 - \frac{n_h}{N_h}) \times \frac{s_h^2}{n_h}]} \tag{4}$$

$$s_h = \sqrt{\frac{1}{n_h - 1} \sum_{i=1}^{n_h} (CPI_i - \overline{CPI_h})} \tag{5}$$

Based on the above analysis, first, users choose the sample size n that fits in their simulation time budget. Second, SimProf picks the simulation points based on the sample size. Third, users simulate the simulation points and estimate the sampling error. If the error is higher than their requirement, they can increase the sample size and repeat the procedure until the sampling error is acceptable.

Since SimProf uses the large size of sampling units, the simulation time can still be significant, users can combine other sampling approaches, e.g., systematic sampling [3] to reduce the simulation time of each simulation point. Exploring the reduction of the simulation time by combining SimProf and systematic sampling will be our future work.

D. Input Sensitivity Test

Algorithm 1 shows the procedure of the input sensitivity test of reference inputs. Initially, one input is assigned as the training input and then the rest of the inputs become the reference inputs. For each reference input, the unit classification classifies the sampling units of the reference input into phases. For each phase, we perform the input sensitivity test of the corresponding phase by comparing their performances and determining whether the performance of the phase changes by inputs.

```
# Assign one input as the base input, and the others
      are reference inputs
assign_inputs()
for inp in reference_inputs:
    # Classify sampling units into phases
    total_phases = phase_classification()

    for phase in total_phases:
        # Check if the phase is input sensitive
        if phase not in input_sensitive_phases and
            phase_sensitivity_test(phase) is True:
            input_sensitive_phases.add(phase)
```

Algorithm 1. Input sensitivity Test

1) Unit Classification: The unit classification classifies the sampling units from a reference input into phases using the cluster (phase) centers of the training input. Even though the program takes different inputs, similar methods are still executed, resulting in the same set of phases. However, because of the different length of phases and/or input-sensitive phases, the performance of the program varies by inputs.

The procedure of classifying a sampling unit is as follows.

- A sampling unit is vectorized into a feature vector based on the profiled call stacks. (Section III-B)

TABLE I
EVALUATED BENCHMARKS

Benchmark	Abbrev.	Type	Input Size	Frameworks
Sort	sort	Microbench	10G text	Hadoop, Spark
WordCount	wc			
Grep	grep			
NaiveBayes	bayes	Machine Learning		
Connected Components	cc	Graph Analytics	2^{24} nodes	
PageRank	rank			

- The feature vector is classified into the phase where the center of the phase has the minimum distance to the feature vector.

The unit classification ensures that sampling units that execute similar program methods are belong to the same phase between the training and the reference inputs.

2) Phase Sensitivity Test: The mean and standard deviation tests are used for the phase sensitivity test. For each phase, we collect the mean and standard deviation of the sampling units within the phase. Then, we compare the mean and standard deviation between those of the reference inputs and those of the train inputs respectively. If any of them are larger than 10%, then the phase is considered as *input sensitive*, as shown in Eq. 6. For a given phase, if all the reference inputs do not pass the input sensitivity test, then the phase is considered *input insensitive*. When exploring the micro-architectural behavior of multiple inputs, one can skip the input-insensitive phases while focusing on simulating the input-sensitive phases.

$$\text{if}(|\frac{\mu_{\text{train}}-\mu_{\text{ref}_i}}{\mu_{\text{train}}}| > 10\% \text{ or } |\frac{\sigma_{\text{train}}-\sigma_{\text{ref}_i}}{\sigma_{\text{train}}}| > 10\%); \text{then Pass} \quad (6)$$

Note that the input sensitivity test does not select the inputs for the test. As the number of inputs is infinite, it is impossible to select the representative inputs. However, one can use the knowledge to reduce the simulation space by categorizing the inputs and selecting one input from each category. For example, if an input is a graph, the categories can be based on the number of vertices and their connectivity. Selecting inputs is based on one's need and is out of the scope of this paper.

After the input sensitivity test, we can easily trace the methods that show input-sensitive/input-insensitive behavior using the information of the method encoded in the phase centers. For example, we can analyze the phase centers and retrieve the methods (features) that have the highest weight, indicating that the method is the most commonly seen in this phase.

IV. EVALUATION

A. Platform and Benchmarks

Our testbed is the machine with Intel(R) Core(TM) i7-4820K and 40G DDR3-1333 MHz memory.

Table I shows the evaluated benchmarks from BigDataBench [17]. Each workload has implementations on two frameworks: SPARK and Hadoop. We use "_[hp]" and "_[sp]" to represent the Hadoop and Spark frameworks respectively. The purpose of evaluating both frames is to test the robustness of SimProf and analyze how phase behavior looks on different frameworks. Since Hadoop is disk-IO intensive when the default configuration is used, we apply common optimizations, such as increasing the memory buffer size of mappers and compressing the output of the mappers before writing to the disk, to improve the performance and make it closer to the real-world settings. The inputs are generated using the data synthesizer provided in BigDataBench.

B. Comparisons

We evaluated the following sampling approaches. Note that SRS and SimProf are probabilistic sampling techniques for which the range of the CPI error can be quantified as the confidence interval.

- **Single N-second simulation point (SECOND):** This is one of the most common approaches used for simulating data center workloads. The original use was for the transaction-based workloads, such as OLTP [18] and SpecWeb [5]. In those workloads, only a few type of transactions exist and each has a short running time. In that case, when N is sufficiently large, e.g., 10 seconds, it is able to cover the entire transaction behaviors.
- **Simple Random Sampling (SRS):** SRS randomly selects sampling units into the sample in which each sampling unit has an equal chance to be selected. The advantage is that SRS does not need to know the running code in each sampling unit so it can save the profiling overhead.
- **Single point for each phase (CODE):** This is a SimPoint-like approach that uses only call stacks to cluster phases and selects the sampling unit closest to the center of the cluster as the simulation point.
- **SimProf:** This uses the stratified random sampling approach. The maximum number of phases is 20 and the threshold of the Silhouette coefficient is 90%.

C. Accuracy and Sample Size Results

To evaluate the sampling accuracy. We run each workload until its completion to collect the CPI of each sampling unit. Oracle CPI is the average CPIs of all sampling units. The sampling errors are computed by comparing the differences between the predicted CPIs with sampling and the oracle CPIs.

Figure 7 shows a comparison of sampling errors of the different sampling approaches. The sample size is 20 for SRS, CODE and SimProf, while SECOND uses the 10-second interval as the sample size. SRS sometimes has a higher errors than SECOND since different sampling units have higher CPI variations. Although SECOND may not have high CPI errors, in most cases, the sample is not representative since it does not cover all the execution stages. For example, SECOND is not able to cover the reduce stage for all Hadoop workloads. CODE also has a higher error than SimProf since purely using the code signature does not reduce the CPI variation of each phase because of other factors that could affect the performance. The average errors of SECOND, SRS, and

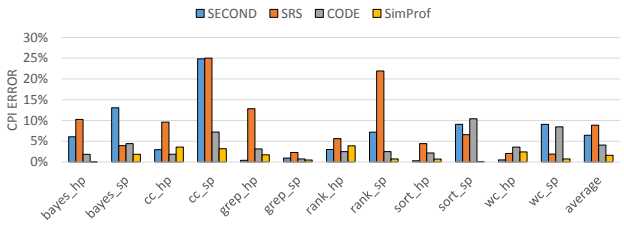

Fig. 7. The sampling errors of CPI of different sampling approaches

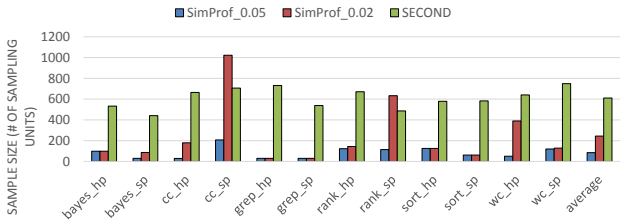

Fig. 8. The comparison of the sample size (number of sampling units) between SimProf and SECOND.

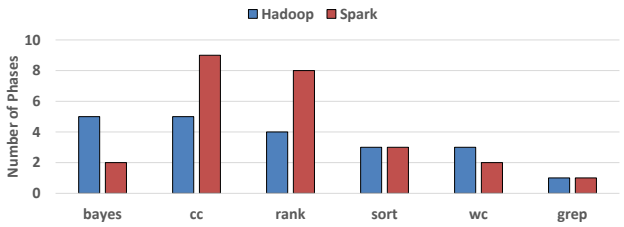

Fig. 9. Number of phases

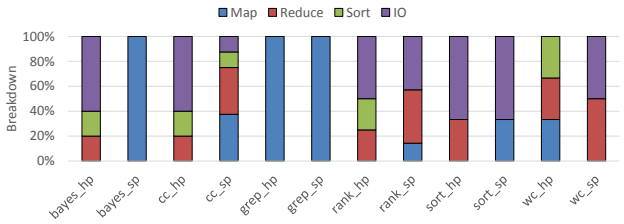

Fig. 10. Phase Type Distribution

CODE are 6.5%, 8.9%, and 4.0% respectively, whereas the average error of SimProf is only 1.6%.

Another advantage of SimProf is having the bound of sampling errors thanks to the stratified random sampling. Figure 8 shows the required sample size of SimProf for the 99.7% confidence interval with sampling errors of 5% and 2%. Since SECOND uses the time duration as the sample size, the actual number of sampling units may differ by benchmarks. (Each sampling unit has 100M instructions.) In most benchmarks, the required sample size is less than SECOND while achieving much lower CPI errors except `cc_sp` and `rank_sp`. These two workloads have more phases and the CPI variations of these phases are also high. The average sample sizes of SimProf_0.05, SimProf_0.02 and SECOND are 85, 244 and 611 respectively. This demonstrates that SimProf not only selects more representative samples that leads to a higher accuracy, but also has a smaller sample size that leads to higher efficiency.

D. Phase Analysis

Figure 9 shows the number of phases in each category of the workloads. It is interesting to see that the number of phases of Spark-based workloads has a much wider range (1 for grep and 9 for cc) than that of Hadoop-based workloads. Some benchmarks, such as `cc` and `rank`, have many more phases since they use more operations and data collection types for the optimization purpose, e.g., GraphX libraries, while in Hadoop implementations, only one to two map and reduce operations are defined, leading to fewer phases.

Since we use the workloads that operate on key-value pairs, we can categorize the phase types based on the dominant operations of the phase: (1) map, (2) reduce, (3) sort, and (4) IO types. Sort indicates the key-sorting operation, such as quicksort while the IO includes the operations that read/write to the HDFS due to insufficient memory space. Since operations are often very tightly coupled, the boundary between them is sometimes blurred. For example, a phase may have both map and IO operations active since each sampling unit in the phase contains both operations. In this case, the type of the phase depends on the dominant operation.

Figure 10 shows the breakdown of different phase types, while the weight depends on the number of sampling units in the type of the phase. Sort operations appear in all Hadoop-based workloads, except `grep_hp` and `sort_hp`. The purpose of the sort operations is to reduce the number of mapper outputs for the optimization purpose. Before the mapper outputs are flushed to the disk, they are sorted and merged by keys so that a reducer receives only one copy instead of N copies where N is equal to the number of mappers on the node. By default, Spark-based workloads do not enable this option so the sorting operations are not seen. In general, Hadoop-based workloads spent more time on IO operations instead of doing actual work than Spark-based workloads, which could be one of the reasons Hadoop-based workloads have a lower performance than Spark-based workloads.

Figure 11 shows how the optimal allocation distributes the simulation points between phases. The sample size ratio of a phase represents the number of simulation points in the phase divided by the total number of simulation points. The figure also shows the CoV of CPI, and the weight of the phase. All three numbers as the range between 0 and 1. Phase 0 has the highest phase weight (29.1%) and the sample size ratio in which 35% of the simulation points belong to this phase. The high CPI variation is caused by the `aggregateUsingIndex` operation, defined in the GraphX library, which performs a reduce operation. By contrast, although Phase 1 has the second-highest phase weight (18.9%), only 5% of the simulation points belong to this phase since the CPI variation of the phase is low. The phase is dominated by the `mapPartitionsWithIndex` operation, which sequentially converts the lines from an input file to the key-

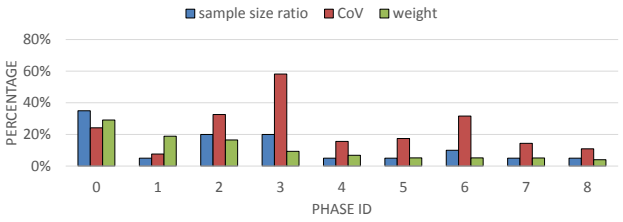

Fig. 11. The sample size ratio of each phase of `cc_sp` distributed based on the optimal allocation. (The phases are sorted by the weight)

TABLE II
EVALUATED INPUTS

Input Name	Input Type	
Google	Web graph	training input
Facebook	Social Network	reference inputs
Flickr	Online communities	
Wikipedia	Online encyclopedia	
DBLP	Computer science bibliography	
Stanford	Web graph	
Amazon	Product co-purchasing networks	
Road	Road Networks	

value pairs. The nature of sequential accesses leads to a low CPI variation. Thus, the optimal allocation allocates the simulation points based on both CPI variations and phase weights. Since SimProf captures both performance and call stacks, the information is used by the optimal allocation to determine the best partition of the simulation points.

E. Input Sensitivity Analysis

Table II shows the graphs downloaded from the SNAP website [19] that we use to analyze how phase behaviors change by inputs. Since most of the original graphs are small and have a different number of nodes, we synthesize the Kronecker graphs [20] that have connectivity similar to the original graph. The resulting graphs have a number of nodes between 2^{20} and 2^{24}. One input is selected as the training input, while the rest are the reference inputs. To select the representative inputs, we try to select the ones that are likely to lead to distinct performances of the workloads to cover a wide range of inputs. For graph-analytic workloads, we can select the graphs with diverse topology as the representative inputs. For text-based workloads, it is more challenging to identify the representative inputs since it is more benchmark-dependent. For example, for WordCount, the inputs with different frequencies of words should be used, while for Sort, the inputs with different ordering between words should be used. Since the criteria of representative inputs for text-based workloads requires more analysis on the content of the documents, we evaluated the graph analytic workloads for now, leaving the text-based workloads for future work.

Figure 12 shows the percentage of simulation points that are in input sensitive phases. This indicates the sample size for reference inputs. For training inputs, all simulation points need to be simulated, but for the reference inputs, the simulation points in the input insensitive phases can be skipped. Using the proposed input sensitivity analysis, the sample size is reduced

Fig. 12. The input-sensitive sample size

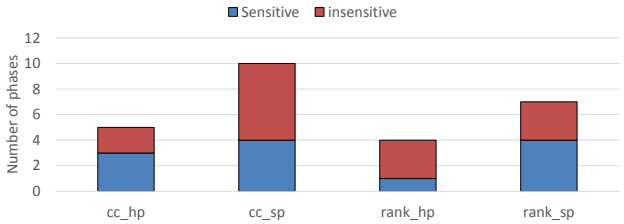

Fig. 13. The number of input-sensitive and input-insensitive phases

from 20% up to 45%. On average, 33.7% of the sample size can be reduced.

Figure 13 shows the number of input sensitive and insensitive phases. For most workloads, the number of input insensitive phases is at least 40% of the total number of phases. We found that the input sensitive phases are likely to have some operations that have time-varying performance. For example, in `cc_sp`, the phase with the `aggregateUsingIndex` operation have different performances at different execution stages. The performances of the phase change by inputs as well. In addition, a high CPI variation within a phase is not necessarily an input sensitive phase since the variation could exhibit some pattern while the pattern does not change by inputs, e.g., the quicksort algorithm.

F. Framework Comparisons

Here, we compare the phase behaviors of the WordCount benchmark on the Hadoop and Spark frameworks. In the following figures, the sampling units are sorted by phase IDs. Blue dots correspond to the left y-axis and represent the CPI of the sampling units while the red lines correspond to the right y-axis and represent the phase IDs of the sampling units.

Figure 14 shows the Spark implementation of WordCount. From programmers' view, the reduce operation should occur at the second stage, as shown in Figure 1. However, by analyzing the call stacks, we found that the majority of the reduce operation occurs in the first stage. The phase with the majority of the sampling units is dominated by the `combineValuesByKey` operation of the `Aggregator` class, which calls not only the reduce operation, but also the map and IO operations, e.g., the first three operations in Figure 1. Then the reduce operation is performed as hashing the key and inserting the key-value pairs into a large in-memory map. If the key already exists in the in-memory map, the newly inserted value is merged with the value in the map. This is the so-called *map-side reduce* optimization used to reduce the size of the mapper

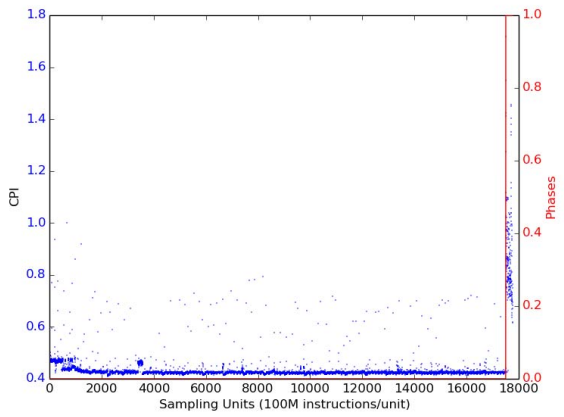

Fig. 14. WordCount - Spark implementation

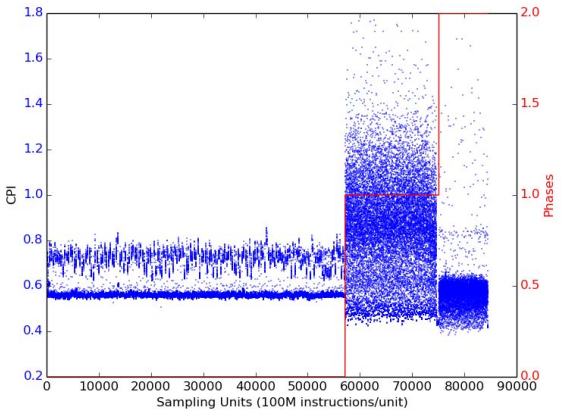

Fig. 15. WordCount - Hadoop implementation

output. Since these operations are tightly coupled, they belong to the same phase. Finally, the second phase reduces the output from the first phase and saves the output to the HDFS. Since the map-side reduce already takes care of the majority of the reducer work, the second phase occupies only 1% of the total sample size. In terms of CPI variations, we expect the first phase, which has the reduce operation, to have a high CPI variation; however, it shows fairly stable performance. This is likely due to multiple operations being merged together, e.g., map, IO, while the reduce operation is not the dominant operation of the phase. The second phase, which contains the HDFS IO operations, has a higher CPI variation because of different data types being accessed. Figure 15 shows the Hadoop implementation of WordCount. The first phase is dominated by the map operation of the `TokenizerMapper` class, which generates the key-value pairs where the key is a word, and writes them to the memory buffer. Since the map operations have a good cache locality, the phase has a high performance and a relatively low CPI variation. The second phase is dominated by the combine operation of the `NewCombinerRunner` class, which performs the map-side reduce operation. Different from the Spark implementation, the combine operation is not coupled with other operations; therefore, the CPI variation is higher in this phase. The third phase is dominated by the sort operation, which uses the quicksort algorithm. The purpose is to reduce the number of mapper outputs sent to a reducer. Because of the recursive nature of the function, the CPI variation is high.

V. Related Work

Sampling Methodology: The sampling approaches can be divided into phase-based sampling and statistical systematic sampling. For phase-based sampling approaches, the most popular one is SimPoint [4] (or PinPoint [21] with Pin tool), which uses basic blocks (BBVs) to form a phase. Perelman [22] et al. used statistical analysis to determine the number of phases. Annavaram et al. [13] found fuzzy correlation between the code signatures and the performances for server workloads.

The fundamental difference between SimProf and the other phase-based sampling is in the basic assumption about phase classifications and performance. Others assume that performance differences between sampling units can be classified with only executed code information. Thus, with a sufficient number of phases, the performance within a phase is homogeneous. However, since the others did not account for the memory access patterns and other factors, which could have more dominant impact than code for data center workloads, simply increasing the number of clusters does not result in having more homogeneous performance in each phase, which becomes the over-fitting problem. In SimProf, we use hardware counters to detect the performance variance of a phase resulting from the non-captured features, such as random access patterns. Then, we use simple random sampling within each phase based on the degree of variance.

For statistical systematic sampling, Wunderlich et al. [3] proposed systematic sampling, SMARTS, which periodically includes a sampling unit into the sample, and Wenisch et al. [6] applied the technique for server workloads. Ardestani et al. [23] and Carlson et al. [24], [25] use the systematic programming for multi-threaded applications. They periodically sample the execution regarding progressed time rather than the instruction count. Compared to SimProf, the advantage of systematic sampling is that the profiling overhead can be saved since the executed code does not need to be captured. However, the code executed in each sampling unit is not considered, which could result in a higher error than SimProf with the same sample size. In addition, SMARTS [3] uses much smaller sampling unit sizes, e.g., 10K instructions, making functional simulation between sampling units (i.e., functional warning) necessary to alleviate the cold cache impact. Because the functional warming is slow, the small sampling unit size is not suitable for simulating large scale applications.

Several works evaluated the phase-based and systematic sampling approaches. Yi et al. [26] compared the SimPoint and SMARTS and found that SMARTS provides better accuracy, while SimPoint is faster. Annavaram et al. [13] proposed

Extended Instruction Pointer Vector (EIPV), a sampled code signature to identify phases, but found that the correlation between the code and the performance is not strong. Wunderlich et al. [27] applied stratified random sampling using BBVs and found that it does not provide much benefit over simple random sampling when the sampling unit size is small. However, they did not account for the cost of functional warming using small sampling unit sizes. In SimProf, we identified that stratification using code can help separate the methods with high CPI variance from those with low variance and the sample size can be smaller than in simple random sampling. Furthermore, SimProf is a real machine based profiling tool with a fast speed in selecting simulation points. Such information can help simulators to decide where to simulate, making the phase-based sampling approach a viable option for studying data center workloads.

Input Set Selection: Prior work on input sets focused on selecting the representative input sets. Eeckhout et al. [28] used PCA to select the representative program-input pairs. They characterize each program-input pair using 20 program characteristics, such as instruction mix and cache miss rates. Breughe et al. [29] used BBV to select the representative inputs for microprocessor design space exploration. Hsu et al. [30] found that the procedure coverage and microarchitecture behavior is different between the training and reference sets in SPEC. In SimProf, the input-sensitive test is an orthogonal approach. With the representative inputs, using the input sensitive test can reduce the simulation time further by skipping the phases whose performance does not change by inputs.

VI. Conclusion

SimProf is the first sampling framework for data analytic workloads based on modern computing frameworks, such as Apache Spark and Hadoop. It identifies phases using call stacks and applies statistical sampling methods to select the final simulation points, which can sample the phases that do not have homogeneous performances. It achieves a 1.6% sampling error of CPI with only 20 simulation points, each of which has 100 million instructions. This sample size is less than 5% of that of single 10-seconds interval. In addition, it detects input-insensitive phases, whose performance does not change by inputs. When simulating multiple inputs with different volumes, the number of simulation points can be further reduced by 33.7% on average as a result of skipping input-insensitive phases.

Acknowledgment

We gratefully acknowledge the support of National Science Foundation (NSF) CAREER CCF 1054830. We would like to thank HPArch members and the reviewers for their comments and suggestions. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF.

References

[1] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *HotCloud'10*, 2010.

[3] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *ISCA '03*, 2003.

[4] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS X*, 2002.

[5] R. Hariharan, N. Sun, and S. Microsystems, "Workload characterization of specweb2005," 2006.

[6] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, Jul. 2006.

[7] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out processors," in *ISCA '12*, 2012.

[8] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *ISCA '13*, 2013.

[9] E. PARSA, *Rigorous and Practical Server Design Evaluation*, Georgia Institute of Technology, 2015.

[10] Oracle, "JVM TI Reference," WWW page, 2007.

[11] M. Odersky and M. Zenger, "Scalable component abstractions," in *OOPSLA '05*, 2005.

[12] M. Bach, M. Charney, R. Cohn, T. Devor, E. Demikovsky, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal, "Analyzing parallel programs with pin," vol. 43, no. 3, pp. 34–41, March 2010.

[13] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. Hankins, and B. Davies, "The fuzzy correlation between code and performance predictability," in *MICRO-37*, Dec 2004, pp. 93–104.

[14] "The four v's of big data," in *IBM big data & analytics hub*.

[15] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[16] S. Wikipedia and L. Books, *Sampling (Statistics): Census, Sample, Stratified Sampling, Sampling Bias, Statistical Unit, Opinion Poll, Statistical Survey, Margin of Erro*. General Books LLC, 2011.

[17] L. W. et al., "Bigdatabench: A big data benchmark suite from internet services," in *HPCA*, 2014.

[18] C. A. Curino, D. E. Difallah, A. Pavlo, and P. Cudre-Mauroux, "Benchmarking OLTP/Web databases in the cloud: The OLTP-bench framework," in *CloudDB '12*, 2012.

[19] J. Leskovec and R. Sosič, "SNAP: A general purpose network analysis and graph mining library in C++," http://snap.stanford.edu/, Jun. 2014.

[20] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, Mar. 2010.

[21] H. Patil, R. S. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *MICRO-37*, 2004.

[22] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," ser. PACT '03, 2003.

[23] E. K. Ardestani and J. Renau, "ESESC: A fast multicore simulator using time-based sampling," in *HPCA*, 2013.

[24] T. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *ISPASS*, 2013.

[25] T. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout, "Barrierpoint: Sampled simulation of multi-threaded applications," in *ISPASS*, 2014.

[26] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins, "Characterizing and comparing prevailing simulation techniques," in *HPCA*, 2005.

[27] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "An evaluation of stratified sampling of microarchitecture simulations," in *WDDD*, 2004.

[28] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere, "Workload design: Selecting representative program-input pairs," in *PACT*, 2002.

[29] M. B. Breughe and L. Eeckhout, "Selecting representative benchmark inputs for exploring microprocessor design spaces," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 37:1–37:24, Dec. 2013.

[30] W. C. Hsu, H. Chen, P. C. Yew, and H. Chen, "On the predictability of program behavior using different input data sets," in *InterACT*, 2002, pp. 45–53.