VarCatcher: A Framework for Tackling Performance Variability of Parallel Workloads on Multi-Core

Weihua Zhang, Xiaofeng Ji, Bo Song, Shiqiang Yu, Haibo Chen, *Senior Member, IEEE*, Tao Li, Pen-Chung Yew, *Fellow, IEEE*, and Wenyun Zhao

Abstract—The non-deterministic nature of multi-threaded workloads running on multi-core platforms often leads to notable performance variability from run to run. Such variability makes experimental results prone to misinterpretations or misguided claims. To deal with such variability, statistical inference methods are usually used to summarize the experimental results with certain confidence levels by running the experiments or measurements a large number of times. However, such statistical results are often too vague or too simplistic. They are not sufficient to help users understand the causes of such variability, and allow more in-depth analysis on the results or reproduce the results for validation during design space exploration. To allow better analyzability and reproducibility, we propose a framework to tackle such variability, called VarCatcher. The key to VarCatcher is to characterize a parallel execution using *Parallel Characteristics Vector* (PCV). A clustering-based approach is then used to group runs with similar execution characteristics that can later be used to analyze results in-depth, to customize different evaluation strategies, reproduce the result for variability, to determine the impact of features, or to assist performance diagnosis. We have built a prototype of VarCatcher that includes a user-level toolset for runtime monitoring and measurements using the Intel Processor Trace feature on commodity Intel processors as well as an architecture extension with very low runtime overheads (around 3 and 0.01 percent accordingly). Several case studies confirm that VarCatcher enables several appealing features such as in-depth result analysis, customized evaluation strategies, and reproducibility.

Index Terms—Variability, parallel application, evaluation, multi core

1 Introduction

Performance variability is a phenomenon where performance results of the same workload change noticeably between different execution runs with the same input set [1]. With the advent of multi-core processors, performance variability is exacerbated by parallel workloads running on such machines. The inherent non-deterministic nature of parallel execution often leads to significant diverse behavior among different runs with pronounced variability in performance.

To deal with such increased performance variability, averaging methods are generally introduced for performance evaluation with variability. A quick survey of the papers published in architecture-centric conferences (HPCA, ISCA, and MICRO) in the past three years shows that about 90 percent of the papers use a simple arithmetic average, typically with only three to five execution runs for each benchmark, and without a confidence guarantee. This makes it difficult to draw proper conclusions with confidence when multiple designs are being compared and evaluated during design space exploration. For example, the performance differences between two cache designs are 4.84 and 9.54 percent for the first and the second 10-runs accordingly (Section 2.1).

More sophisticated statistical inference schemes such as normality-based arithmetics [1], [2], non-parametric testing [3], or visual tests [4] can be used to summarize performance results with a pre-determined confidence level. However, such statistical results are still too vague and too simplistic to help users reason about the causes of such variability. For example, the performance differences for the above cache design example are 4.08, 9.24 and 0.83 percent when using normality, hypothesis testing and visual tests accordingly, which is very hard for users to reason about the causes behind the summarized performance numbers. Besides, they are insufficient to reproduce the experimental results for validation when necessary.

In many cases, analyzability [5] and reproducibility [6] are critical to performance study and architectural design exploration. System designers sometimes would like to know the main causes of such performance variability, and to tackle such variability either to eliminate it or to exploit it. They may want to determine the impact of some design features on

- *W. Zhang, X. Ji, B. Song, and S. Yu are with Software School, Shanghai Key Laboratory of Data Science, and Parallel Processing Institute, Fudan University, Shanghai 200433, China. E-mail: {zhangweihua, songb11, sqyu14, xfji11}@fudan.edu.cn.*
- *H. Chen is with the Institute of Parallel and Distributed Systems, Shanghai Jiaotong University, Shanghai 200240, China. E-mail: haibochen@sjtu.edu.cn.*
- *T. Li is with the Department of Electrical and Computer Engineering, University of Florida, FL 32611. E-mail: taoli@ece.ufl.edu.*
- *P. Yew is with the Department of Computer Science and Engineering, University of Minnesota, Twin Cities, MN 55455. E-mail: yew@cs.umn.edu.*
- *W. Zhao are with the Shanghai Key Laboratory of Data Science School of Computer Science, and Parallel Processing Institute, Fudan University, Shanghai 200433, China. E-mail: wyzhao@fudan.edu.cn.*

Manuscript received 15 Apr. 2016; revised 5 Aug. 2016; accepted 2 Sept. 2016. Date of publication 26 Sept. 2016; date of current version 15 Mar. 2017. Recommended for acceptance by G. Tan. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2016.2613524

performance, identify abnormal behavior for performance diagnosis, or to customize result summarization. They would also like to reproduce the results for validation if needed.

In this paper, to improve analyzability and reproducibility, we first analyze the main causes of variability using *execution patterns* that are based on execution paths (i.e., program control flow) of the parallel workload and the relative execution speed of the cores. Such an analysis leads to two main observations. First, parallel execution often exhibits a significant number of different execution patterns, and the performance of different execution patterns could vary substantially. Second, the distribution of execution patterns across multiple runs could be quite random, i.e., the pattern distribution among the first 100 runs could be substantially different from that in the next 100 runs.

Based on the analysis of the execution patterns, we propose a framework, called VarCatcher, to tackle such variability for better analyzability and reproducibility. This analysis can be carried out in conjunction with existing statistical inference methods. VarCatcher first collects the execution patterns from different runs. A clustering technique such as K-means [7], [8] is then used to group runs with similar execution patterns. Performance variability caused by distinct execution patterns is then analyzed and evaluated.

In VarCatcher, a core-oriented Parallel Characteristics Vector (PCV) is used to capture the variability in execution paths (i.e., control flow) on each core. To account for varied execution speed among different cores, PCV is captured in a globally defined time interval across all cores. The time interval is defined in clock cycles, e.g., every 100 K cycles. To eliminate equivalent execution patterns, we devise and apply *core alignment* and *interval alignment* before final clustering because similar runs with the same execution pattern may be identified as dissimilar during clustering.

An important requirement for all runtime tools is to minimize their intrusion and perturbation during the program execution. It is done in two ways. First, we leverage Intel Processor Trace [9], an emerging performance monitoring feature on Intel processors, to non-intrusively capture relevant runtime features with low overhead (around 3 percent). To further reduce space and offline processing overhead, we propose some architectural extensions and have integrated them into a cycle-accurate simulator to show their effectiveness in reducing space and processing overhead. An offline analysis tool then accepts and analyzes the runtime collected data to draw final conclusions.

To study the effectiveness of such a framework, we built a prototype of VarCatcher and use it in five different scenarios. They include an in-depth performance analysis using PCVs, a customized performance evaluation scheme, a reproducibility study, a design space exploration, and a case of performance diagnosis. The case studies confirm that VarCatcher allows more in-depth, flexible and reproducible evaluation of parallel workloads and interpretation of the performance data.

In summary, this paper makes the following contributions.

- An approach to characterize the execution of parallel workloads (Section 3.1) that takes into account the program execution paths (i.e., control flow) in a core and the relative execution speed among cores (Section 4).
- A clustering-based approach to reasoning about performance variability in parallel execution using execution patterns as well as core and interval alignments to improve clustering accuracy (Section 3.2).
- An evaluation framework empowers users with better analyzability and reproducibility for performance evaluation of parallel workloads. (Section 5).

The rest of this paper is organized as follows. In Section 2, the motivation and the major motivation are discussed. Section 3 gives out the overall design, including how to extract execution features and how to group multiple runs into the same clusters. Section 4 discusses the basic implementation and the experimental results. Section 5 gives out several case studies. We discuss the related works in Section 6. Finally, we summarize our paper in Section 7.

2 A Motivating Example and Some Observations

2.1 A Motivating Example

We use a simple design space exploration as an example in which two cache designs are being evaluated. One design uses a 128 KB shared L2 cache (marked as Design-1) and the other uses a 1 MB shared L2 cache (marked as Design-2), both are on a four-core system.[1]) We use *Word Count (WC)*, a parallel benchmark program in Phoenix-2 [10], [11] with the reference input to evaluate both cache designs. The simulation is done on Transformer [12], a cycle-accurate parallel architecture simulator. We first run *WC* 10 times on Design-1 (i.e., 128 KB L2), the performance differences among 10 runs can be up to 39.65 percent in cycles, i.e., the best execution time versus the worse execution time can differ by 39.65 percent. We then run *WC* 10 times *twice* on each design, and use the simple arithmetic average of their execution time for each 10 runs. The performance difference between the two cache designs in the first 10 runs is 4.84 percent, and it changes to 9.54 percent in the second 10 runs. Even though both experiments give the same conclusion that shows larger L2 cache gives better performance, such large variability in performance gives little confidence on the conclusions in more complex design space explorations.

More sophisticated statistical inference methods have been used to summarize such experimental results using a higher confidence level. Statistical approaches such as normality-based schemes [1], [2], [13], non-parametric testing schemes [3], [14] and visual test schemes [4] are among the commonly used. Normality-based schemes are based on the Central Limit Theorem. They first run a few samples and use their results to estimate the number of runs needed to achieve a certain confidence level. In our L2 design example, we need at least 121 runs to achieve a 95 percent confidence level, and the performance improvement is 4.08 percent. According to the analysis in [3], the runs of multiple applications do not always satisfy the Central Limit Theorem for normal distribution even many execution runs are conducted. Non-parametric testing schemes (a type of

1. We have conducted other evaluations, such as different memory access latencies, INT ALU latencies and different ALU numbers, all of which can get similar conclusions. For brevity, we only use this configuration to show the example and demonstrate the effectiveness of VarCatcher.

TABLE 1
Performance Differences Using Different Statistical Approaches

Average-1	Average-2	Normality	Hypothesis testing	Visual Tests
4.84%	9.54%	4.08%	9.24%	0.83%

hypothesis testing), such as Mann-Whitney U test or Wilcoxon Rank-Sum Test [3], [14], do not make any assumption about such distributions, and depend only on the number of runs with a pre-determined confidence level. Based on such a method [14], the performance improvement is 9.24 percent for our L2 design example with a 95 percent confidence level. Visual Test schemes first calculate the confidence interval of performance for each design, and then check the overlapping of two intervals to see whether there is a difference. They require the width of confidence interval be above a certain threshold for comparison. For our L2 design example, 40 runs on each design can achieve 3 percent confidence interval width [4], and the performance improvement is 0.83 percent.

All these statistical inference methods focus only on result summarization as shown in Table 1. There is no further detail available to reason about such performance variability, or to reproduce the results of validation. From this perspective, statistical inference is inadequate in its analyzability and reproducibility in particular for large design space exploration.

2.2 Some Observations

To see if we could achieve better analyzability and reproducibility, we first look at why *WC* behaves differently in different runs on the multi-core platforms. First, we found that immediately after the beginning of its execution, *WC* forks a worker thread on each core to do the job. These threads have to wait to be scheduled after their creation. In some extreme cases, some threads may have already finished their jobs before other threads begin. Second, the workload may not be evenly distributed among these worker threads. The tail thread (i.e., the last thread) is often assigned more workload than others. Third, thread migration and contention on shared resources happen non-deterministically in different runs because of the nature of parallel execution and the underlying system.

Those three factors are quite common in parallel execution, and they dominate the performance variability in different runs. Depending on the goals of the performance studies, those factors can be exploited and manipulated to study various scheduling schemes, resource management strategies, performance diagnosis, among others.

However, in many performance studies such as architectural design space exploration, users may want to control such factors in order to minimize or eliminate such performance variability. There have been several strategies proposed to minimize the impact of these factors, such as adding barrier instructions in the source code, setting thread affinities to cores, changing workload allocation, and using shared memory access tables [15]. By applying those strategies to all benchmarks in Phoenix-2, we are able to reduce the performance (i.e., execution time) variability from 17 to 1 percent, and 22 to 0.45 percent on its CoV's (coefficient of variance, the ratio of the standard deviation to the mean).

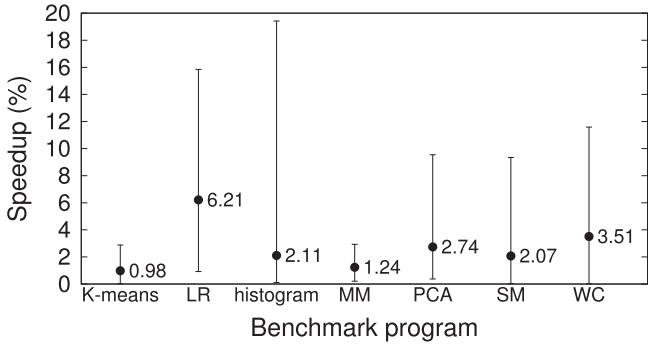


Fig. 1. The variability of evaluation results from different execution patterns for each application on two different designs. For each application, the average performance difference evaluated from different execution patterns and the range of these performance differences are given out.

The analysis shows that such factors are the most dominant causes of performance variability. They significantly impact the execution path (i.e., control flow) on each core, and the relative execution speed among multiple cores. The *execution patterns* of parallel execution, which are primarily determined by the execution paths and relative execution speed, can thus be used to characterize the performance variability in parallel execution. Different execution patterns of multiple runs are defined as either the execution paths on a certain core are different or the relative execution speed among cores varies.

By studying the execution patterns of all benchmarks in Phoenix-2 [10], [11], we have the following observations.

Observation 1. There are multiple execution patterns that cause performance variability across multiple runs. We run each benchmark 200 times using the same configuration. For the tested seven benchmarks, there are on average 18.86 execution patterns among 200 runs. The largest number of execution patterns is 25, and the smallest is 14. The average CoV of the performance variability among different execution patterns is more than 49 percent, which explains the notable performance variability among different runs even using the same configuration. Fig. 1 also shows the average performance and the performance variability among different execution patterns for the example in Section 2.1. The performance of one pattern may be 65.26 percent better than the performance of another pattern of the same application.

Observation 2. The distribution of such execution patterns can be notably different in different sets of runs. For example, we found a specific execution patterns in WC appeared in 8 runs of the first 100 runs, and appeared in 17 runs of the next 100 runs. Such phenomenon explains why the statistical inference methods without specifying a confidence level can lead to misleading results.

Based on these observations, we can see that statistical inference may not be the most effective way to deal with performance variability in parallel execution, since such approaches only focus on result summarization. By identifying and managing execution patterns instead, we can go a step further over statistical approaches to tackle such performance variability for better analyzability and reproducibility in addition to result summarization.

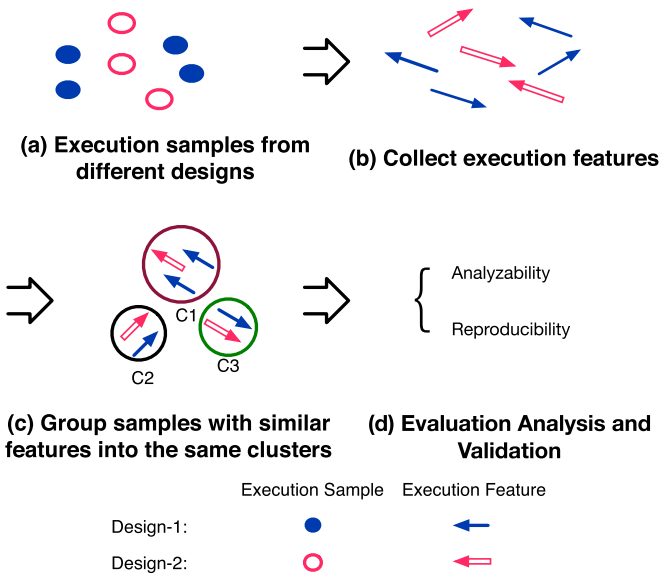


Fig. 2. An overview of workflow for VarCatcher.

3 The Framework

In VarCatcher, we first need to collect relevant features during a parallel execution as described in Section 3.1. It allows us to characterize different execution patterns of a parallel application. We then need to distinguish different execution patterns based on the collected runtime features, and group similar execution patterns through clustering as described in Section 3.2. An overview of VarCatcher's workflow is shown in Fig. 2.

3.1 Characterization of Parallel Execution Patterns

There have been many features, such as basic block vector (BBV) [8], [16], [17] or memory reuse distance [18], which are used to characterize the execution behavior of applications running on single-core machines. However, they are insufficient to characterize the execution patterns of parallel workloads running on multi-core platforms. Here, we first discuss why these features for sequential applications (se-feature) cannot be used to represent the execution behavior of parallel workloads. Then, we give out the feature design for parallel workloads, called parallel characteristics vector.

3.1.1 Limitation of Traditional Features to Parallel Execution

To characterize a parallel execution, one intuitive solution is to collect the se-features for each core and then concatenate

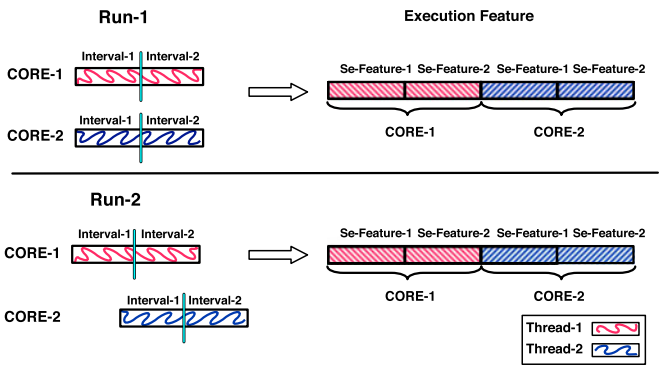


Fig. 3. Sefeatures in each core alone cannot reflect the relative execution speed among different cores.

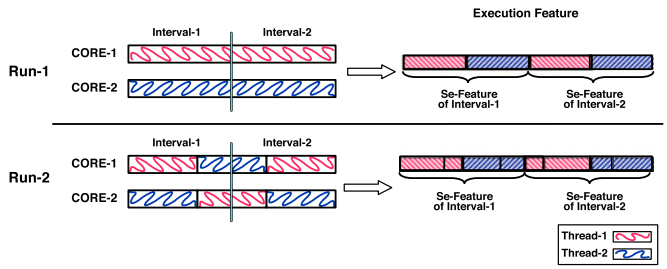


Fig. 4. Each core's execution paths are not reflected in a global interval.

all of them with a fixed order in the interval. The se-features for each core can be collected independently. Although such a method is simple, unfortunately it is not enough to accurately represent the behavior of parallel execution. This is because, besides the execution path of each core, relative execution speed among different cores also leads to performance variability of a parallel application. Fig. 3 shows such a case. In the figure, the figures on the left-hand side are the different execution patterns and the figures on the right-hand side are their execution features. se-feature-1 stands for the se-feature captured from Interval-1 while se-feature-2 stand for the se-feature captured from Interval-2. Run-1 on the upper one side and Run-2 on the bottom one are apparently different due to the difference in the relative execution speed between core-1 and core-2.[2] However, since the se-features taken separately from each core are similar, such a method (i.e., concatenation of se-features) may lead to a wrong conclusion that the two runs are similar. We thus need to capture the relative execution speed among different cores in order to reflect real execution patterns of parallel execution.

To achieve such a goal, one solution is to define an interval globally across all cores, and collect the se-features in the global interval. This approach could account for relative execution speed among cores. But, it is insufficient to capture the execution paths (control flow) of each core, which is critical to performance variability as mentioned earlier. As shown in Fig. 4, there is no thread migration in Run-1 and there are two thread migrations in Run-2. Since we use a global feature for each interval without distinguishing the behavior in each core, the migration behavior between two threads in run-2 will be mixed up together. In such a scenario, even though the corresponding global se-features in the two runs are similar and the relative execution speed of the two cores are kept the same, it is obvious that the two runs are different because the execution paths of the corresponding cores are different.

3.1.2 Parallel Characteristic Vector

To account for both execution paths and relative execution speed of each core at the same time, we propose Parallel Characteristics Vector as shown in Fig. 5. It defines intervals globally but collects each core's se-features independently. A global interval is defined in term of a constant number of

2. This is caused by thread scheduling. For example, there are two threads in a program. They are created at the same time. However, due to thread threading, one of the threads may begin to execute after the other one completes. We observed such a condition when we analyzed the factors influencing the variability.

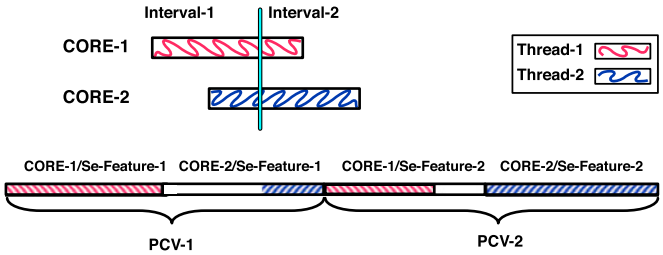


Fig. 5. PCV includes concatenated se-features information from each core in globally partitioned intervals.

clock cycles on each core across all cores, e.g., 100 K cycles on each core. It requires a PCV to be a two-dimensional vector. The first dimension identifies the core number. The second dimension records the se-feature information of each core. They are concatenated into a PCV for the corresponding interval.

After the program finishes its execution, PCVs for all global intervals are collected. All PCVs are then concatenated to characterize the execution pattern of that particular run. Such a PCV will include all execution paths (control flows) within each core, and the relative execution speed among different cores.

It is worth noting that in the original se-feature definition, such as BBV [8], [16], [17] or memory reuse distance [18], an interval is defined in term of the total number of instructions executed (e.g., per 100 K instructions). However, such a definition will have several problems in defining a global interval. One is that the elapsed time for such a global interval may vary significantly depending on the composition of the instructions in each core within the interval. Each core may have a different number of instructions included in the interval, which will make it difficult to track and account for the relative execution speed among different cores. Another is that we will need a global instruction counter to keep track of the total number of instructions executed from all cores, which requires a lot of synchronization and communication as the core number increases. If we divide the total number of instructions evenly among cores in each global interval, the varying execution time of those instructions on each core will make it very difficult to align them in terms of global time across all cores. Such misalignment will accumulate and become worse, and eventually be untrackable if there is a large number of cores and a large number of intervals.

3.2 Clustering

After characterizing parallel executions using PCVs, we can apply clustering techniques on the collected PCVs to identify similar execution patterns among different runs. However, before applying clustering techniques on the collected raw PCVs data, we can sharpen the similarity among different runs by eliminating some misalignment during the data collection process.

There are two main causes of misalignment. In Section 3.2.1, we present a preprocessing step for interval alignment, and in Section 3.2.2, we present the other preprocessing step for core alignment. The number of execution patterns can be substantially reduced after interval and core alignment. We then cluster the preprocessed PCVs data to identify similar execution patterns as presented in Section 3.2.3.

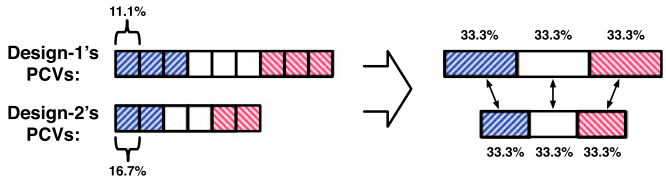


Fig. 6. Re-partitioning for interval alignment using the ratio of total execution time.

3.2.1 Interval Alignment

In clustering algorithms such as K-means [7], [8]), the dimension of all input vectors must be the same. The dimension in our case is the total number of PCVs collected in each run. Because the global intervals used in PCVs are defined in clock cycles, the total number of PCVs for each run may be different due to different execution time caused by different architectural designs or simply parallel execution variability.

Using the two designs presented in Section 2.1 as an example, the number of intervals in runs with Design-1 (i.e., 128 KB L2) may be 34 percent larger than that with Design-2 (i.e., 1 MB L2) using the same interval size of 100 K clock cycles because the execution time for Design-1 is longer.

Fig. 6 shows a simplified example to compare two architectural designs by assuming they have the same execution paths and the same relative execution speed among the cores during the execution, i.e., both runs have the same execution patterns. The performance of design-2 is better than that of Design-1. Thus, Design-2 has a shorter execution time. Using the same interval size on both runs to obtain PCVs, we have nine intervals for Design-1 while only six intervals for Design-2.

To use clustering algorithms such as K-means, we need them to have the same number of intervals. One naive solution is to pad three zero PCVs to Design-1 or trim three PCVs from Design-2. It is obvious that neither approach will lead to an accurate conclusion after clustering. Actually, the two will be considered as two completely different execution patterns, even though they should have the same pattern as mentioned earlier.

The essence of such a problem is that when comparing the execution patterns of two different designs, the length of intervals should reflect and be adjusted according to the difference in their execution time.

In the example shown in Fig. 6, each interval in Design-1 represents about 11.1 percent of the total execution time while each interval in Design-2 is about 16.6 percent. To determine if the execution patterns of the two runs are similar or not, the size of intervals should first be adjusted accordingly. It allows clustering algorithms such as K-means to give a more accurate classification of the execution patterns that truly reflect the execution paths (i.e., control flow) and the relative execution speed among cores in each run.

Therefore, in interval alignment, VarCatcher re-partitions intervals based on the total execution time of each run. After interval alignment, each interval will be equal to the same percentage of the total execution time in each run as shown in the right part of Fig. 6. After interval alignment, each run will have the same number of intervals, i.e., the same number of PCVs, and thus K-means could give a more accurate clustering to identify similar execution patterns.

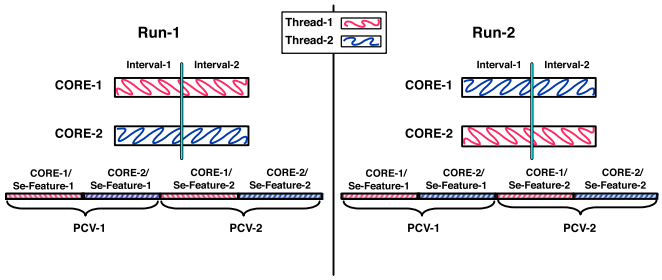


Fig. 7. Core misalignment problem. Two runs have similar execution patterns, but their thread mappings are different.

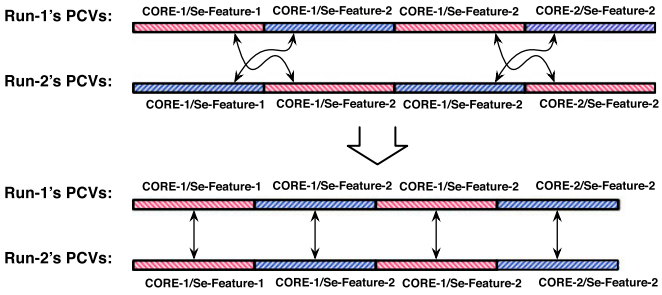


Fig. 8. Core alignment. Adjust the BBV sequence in PCVs to account for different thread mappings.

To facilitate interval alignment, our approach is to first use very fine-grained intervals to collect PCVs for all runs, e.g., instead of using a 10-million cycle interval size we use a 100 K-cycle interval size. However, using a smaller interval size could incur a larger storage overhead for storing such information. Hence, the granularity of the initial interval size should be chosen to balance the precision and the space cost. During the interval alignment, we then merge these fine-grained PCVs into a larger interval that is proportional to the execution time.

3.2.2 Core Alignment

After interval alignment, runs with the same execution patterns may still not be clustered as similar because of varied thread mappings to cores. Fig. 7 shows such an example. In Run-1 (shown on the left) and Run-2 (shown on the right), the execution paths of Thread-1 and Thread-2 are the same in both runs, and there is no thread migration during the run. Moreover, the relative execution speeds are also the same. Therefore, these two runs should be classified as the same execution pattern after clustering. However, their PCVs are distinct because the first dimension of PCVs specifies the core ID, and the second dimension records the se-features of the thread running on that core.

The essence of this misalignment problem is due to different thread mappings to cores. To account for such thread mappings, we present a core alignment scheme to identify runs with similar execution patterns but with different thread mappings as shown in Fig. 8. It consists of four steps.

1. *Forming gse-features for Each Core.* As we are trying to identify thread mapping on each core in each run, we first characterize the execution path of each core by grouping all BBVs on each core to form a global se-feature (gse-feature) for that core as shown in Fig. 9. In a sense, such gse-features represent a particular thread mapping.
2. *Grouping Similar gse-features Into Clusters.* To identify similar gse-features (i.e., potentially same threads, but mapped on different cores), we use K-means [7], [8] to group these gse-features into clusters. The input parameter K is selected by Bayesian Information Criterion (BIC) method [19] since it can easily be automated and has fewer errors in practice. The cores mapped into the same cluster have similar gse-features, and thus similar execution paths from similar threads.
3. *Generate an Alignment Matrix.* Based on the clustering results, we generate an alignment matrix as shown in Fig. 10. Each row of the matrix represents a run of a particular design. Each column represents a cluster identified by K-means. If a core's gse-feature in a run belongs to a particular cluster, the corresponding entry in the matrix is marked with the core ID. Otherwise, that entry is NULL. Using the previous example, we run *WC* six times for Design-1 marked as (a), and Design-2 marked as (b) on a 3-core platform. There are eighteen gse-features, and they are grouped into five clusters by K-means. The alignment matrix are formed as shown in Fig. 10.
4. *Identifying Similar Execution Patterns by Factoring Out Thread Mapping.* If rows have identically filled positions, such as Run-2 and Run-4 in the alignment matrix, then they have similar execution patterns despite different thread mapping on core-2 and core-3, according to cluster 3 (C-3) and cluster 4 (C-4) in the alignment matrix. Based on such observations, we can realign their core IDs in PCVs using the information from the alignment matrix. For example, in Fig. 11, we adjust the PCV positions of core-2 and core-3 in Run-4 to make the core alignment between Run-2 and Run-4.

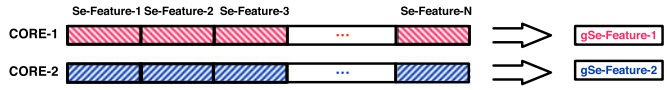


Fig. 9. Generate gse-feature vectors of each core through merging each core's BBVs in every interval together.

$$\begin{array}{c} \\ Run\text{-}1(a) \\ Run\text{-}2(a) \\ Run\text{-}3(a) \\ Run\text{-}4(b) \\ Run\text{-}5(b) \\ Run\text{-}6(b) \end{array} \begin{array}{c} \begin{array}{ccccc} C\text{-}1 & C\text{-}2 & C\text{-}3 & C\text{-}4 & C\text{-}5 \end{array} \\ \begin{pmatrix} core\text{-}2 & core\text{-}1 & core\text{-}3 & & \\ & core\text{-}1 & core\text{-}2 & core\text{-}3 & \\ core\text{-}1 & & core\text{-}3 & core\text{-}2 & \\ & core\text{-}1 & core\text{-}3 & core\text{-}2 & \\ core\text{-}1 & core\text{-}2 & core\text{-}3 & & \\ & & core\text{-}3 & core\text{-}2 & core\text{-}1 \end{pmatrix} \end{array}$$

Fig. 10. Alignment Matrix.

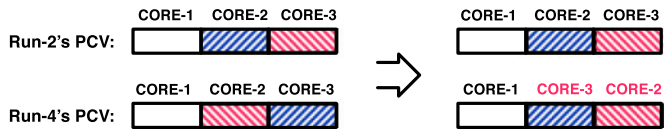


Fig. 11. Adjust core IDs in PCV to achieve core alignment based on the alignment matrix. After exchanging the gse-feature positions of core-2 and core-3 in Run-4, Run-2 and Run-4 have the same execution patterns.

After the core alignment, similar execution patterns with different thread mapping could be correctly identified and adjusted.

3.2.3 Final Clustering

After the interval and core alignment, we are ready to cluster all runs to identify different execution patterns by applying K-means again using BIC method to select K [19]. The K-means algorithm is applied to all aligned PCVs (as opposed to gse-features in the core alignment step) of all runs to identify similar execution patterns among them.

For asymmetric multi-cores (i.e., cores with different clock frequencies) or multi-cores that allow dynamic frequency and voltage scaling, the clock frequencies of different cores can be different, and thus the clock cycle time is different on different cores. If two runs follow the same execution pattern but their intervals are partitioned using different clocks on different cores, they may not be grouped into the same cluster.

To solve this problem, cycle alignment needs to be done before the interval alignment is applied. Based on the frequencies of different cores, we can choose a base frequency (the maximum or minimum frequency) and calculate the ratio of the base frequency to the frequency of other cores, or other periods of intervals in the same core. We can then decouple each core's PCVs and re-partition each core's intervals according to these ratios. For example, if the frequency ratio of two cores is 3:4, we can combine three PCV intervals on the first core while combine four of them on the second core. We can then apply interval and core alignment as before after the cycle alignment.

There are some architectures that can dynamically change the CPU frequencies according to different system loads, such as DVFS. The information about DVFS frequency change can be collected through system instrumentation or architecture extension. Therefore we can then apply interval alignment with the corresponding frequency information of each interval. VarCatcher can also work in similar scenarios as long as we can get the information about the real-time CPU frequency changes.

Effectiveness of Interval and Core Alignment. We measure the number of clusters with and without interval and core alignment based on a 200-run of the Phoenix-2 benchmark (experimental setup is described in Section 2.1). There are 18.14 and 43.28 clusters on average with and without the two alignments, respectively. Specifically, with only interval alignment, there are on average 20 clusters (a 52.48 percent reduction, with a maximum of 61.36 percent); with only core alignment, there are 42 clusters (a 2.64 percent reduction on average, with a maximum of 9.1 percent). Hence, with interval and core alignment, clustering algorithm could be more effective in identifying different execution patterns.

4 Implementation & Evaluation

In this section, we will first introduce our BBV-based PCV and some optimizations. Then, we will discuss the implementation of VarCatcher.

4.1 BBV-Based PCV

4.1.1 BBV-Based PCV

It is convenient for PCV to combine with these traditional se-features, such as BBV and memory reuse distance. We have tested the efficiency of PCV through combining it with different se-features, including BBV and memory reuse distance, they can achieve the similar accuracy. Because it is easier to collect BBV information through hardware performance counter, we choose BBV as the se-feature in our following implementation.

Basic block vector [8], [16], [17] is a metric, which is used to record how many times the basic blocks (BBs) in a program interval have been executed. To obtain BBVs, the execution of a program is divided into intervals by a fixed number of instructions. Then, the BB information in each interval is collected and recorded into the corresponding BBV. The Manhattan distance [8] of the two normalized BBVs is used to decide whether two intervals are similar. If the distance is smaller than a threshold, the intervals are similar. Due to its efficiency to represent the dynamic behavior of a program execution, it has been widely used for dynamic analysis.

4.1.2 Optimizations for PCV

Since PCV needs to record all se-feature information in an interval, a PCV is essentially a high-dimensional vector. Actually, even for a small program such as *matrix multiply*, the dimension of each core's se-feature, such as BBV, can be over 4,000. The larger the program is, the more dimensions the PCV could have. Storing such information is expensive, and processing such high-dimensional data with K-means is time-consuming. Here, we use two optimizations: dimension reduction and phase table to reduce the overheads.

Dimension Reduction. We use the linear projection technique [8] to reduce the dimension of PCV, and thus decrease the space requirement to store PCVs and accelerate the clustering process. Before each core's BBVs in an interval are concatenated into a PCV, each BBV is randomly projected onto a 15-dimension vector. Such projections reduce computation complexity and storage requirements for the trace file. Yet, they still preserve most of the pattern information for later analysis or for reproducibility. Through this optimization, VarCatcher can reduce 70 percent of the dimensions while maintaining 95 percent variance of the origin data. Furthermore, the dimension reduction also improves processing speed and cut down 68 percent of the clustering time on average for the Phoenix-2 benchmarks.

Using Phase Behavior to Reduce Storage Overhead. In the origin design, we collect PCVs in every interval. Although we have reduced the dimension of PCVs, the PCV is still a high-dimensional vector (15 dimensions for each core). It still has a large storage requirement for PCVs when a program has a long execution time. To further reduce such overheads, we leverage the phase behavior in most application programs. While a program is being executed, it generally has many execution intervals with repetitive patterns due to loop structures and multiple call sites in the program. Phase analysis has been well established as a standard technique that characterizes the set of execution intervals with similar behavior in the same phase [16], [17]. Moreover, even if a program has a long execution time, the

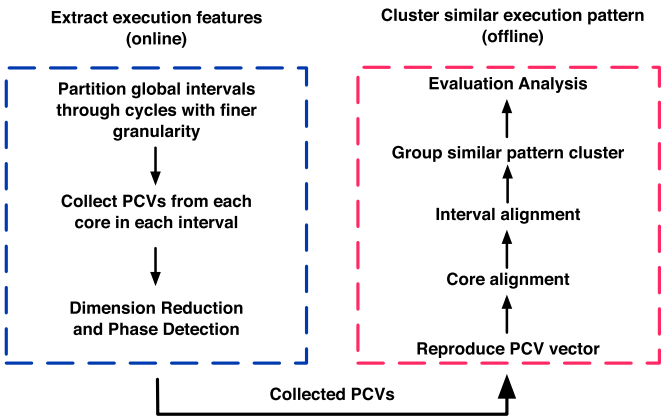


Fig. 12. The workflow of VarCatcher.

number of phases is usually small. For example, the average number of phases for programs in Phoenix-2 [10], [11] is only 16, and the maximum number is only 28. Therefore, we only need to store a phase ID using less than 3-byte for each interval, instead of 60 bytes for each BBV to include all of the basic blocks in the program, which can greatly reduce the space overhead.

4.2 Implementation

The workflow of VarCatcher is summarized Fig. 12. VarCatcher first collects PCVs during program execution. It partitions the global intervals (measured in clock cycles) with a fine granularity for later interval alignment.

It then collects se-features, i.e., BBVs, from each core in each interval with reduced dimension, and leverages the program's phase behavior to reduce storage requirements. After concatenating those se-features from each core with core IDs to form PCVs, it performs core alignment by grouping gse-features from each core into clusters and generating an alignment matrix based on these clusters. It also performs interval alignment through re-partitioning of the intervals so that the number of intervals in different runs becomes the same. K-means is finally applied on the aligned PCVs to group similar runs into clusters.

Implementing VarCatcher as a System Tool. We have implemented a software tool using Intel Processor Trace (IPT) [9], a performance monitoring tool offered on Intel processors, to non-intrusively capture runtime traces. The trace includes each core's branch information. After the trace is collected, the instruction count of each BB can be calculated based on each BB's entry and exit instructions. The PCV can then be constructed by combining BBVs with core IDs. Dimension reduction and program phases are used to reduce processing and storage overheads as described earlier.

Hardware Support for VarCatcher. To reduce storage and processing overhead, we also provide some hardware support for VarCatcher based on traditional phase detection architecture [16], [17]. The main differences are the interval is partitioned by clock cycles instead of instruction counts and there is a phase sequence buffer to record the feature of the whole execution. Fig. 13 shows such hardware support on each core. When a program is being executed, the BBV information on each core is collected through the BBV accumulator, which counts the number of instructions of each BB. At the end of each cycle interval, its BBV is used to

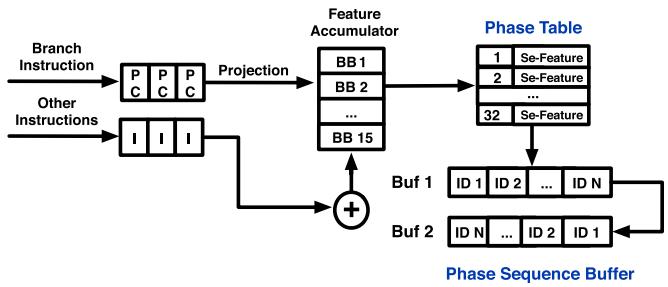


Fig. 13. The online instrumentation structure in each core. It generates phase ID based on the BBV of each interval and stores the phase information in phase sequence buffer.

search the phase table. Each entry of the phase table records a distinctive BBV with a phase ID. If a similar phase table entry (i.e., the Manhattan distance between the two BBVs is smaller than a threshold) is not found, a new phase table entry with a new phase ID is created and entered in the phase table. If a phase ID is obtained (i.e., a hit on the phase table), it is entered in the phase sequence buffer. To avoid the execution stall, the sequence buffer is implemented as a rotation buffer (Buf-1 and Buf-2). If the buffer is full, its contents are dumped to the memory, and the other buffer continues to collect the execution information.

In our current design, the interval length is 100 K cycles, and each buffer contains 100 entries. It means a phase sequence is dumped to memory every 10 million cycles, which is at the level of a context switch [17]. In memory, we allocate 8 KB space (two 4 KB pages) for each core to store the dumped information. If the page is full, it is dumped to disk, and the other page continues to store the information.

4.3 Evaluation

To study the efficiency and the accuracy of VarCatcher, we have tested applications in PARSEC-2.1 [20] and Phoenix-2 [10], [11] on the environment in Section 5.

For system tool implementation, it incurs about 3 and 3.15 percent runtime overhead on average, which should be low enough for many cases. The trace storage and the offline processing overhead are noticeable. The average size of IPT trace is about 652.85 MB (ranging from 71 to 965 MB) for Phoenix-2, and 865.63 MB (ranging from 189 MB to 1.9 GB) for PARSEC-2.1. It needs extra offline processing time to convert IPT trace to PCVs, which takes on average 158.32s and 193.90s respectively.

We have implemented our architectural design using Transformer [12] to evaluate the effectiveness of such a hardware support. The hardware runtime overhead is less than 0.01 percent based on our results. The total space overhead is less than 2 KB for each core. The average space overhead of the final PCVs is about 2.36 MB, and the offline processing time for 400 runs is about 44s. Our simulation results from Transformer also show the average runtime overhead for the architecture extension is only about 0.45 percent on average.

We have also conducted evaluations on different varied metrics, such as different INT ALU latencies, memory access latencies and numbers of ALUs. VarCatcher works well under all these settings and we got similar conclusions. Due to space constraints, we only use the varied L2 size as an example and demonstrate the effectiveness of VarCatcher.

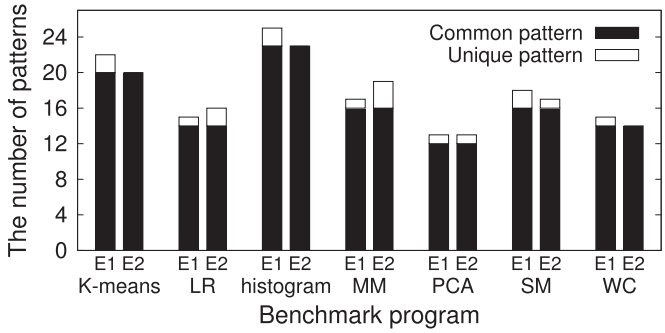


Fig. 14. The number of common patterns across two 100-run experiments (E1 and E2) for Phoenix-2 on two designs.

Validation of Clustering. To validate the runs in each cluster are actually similar, we compare the branch traces collected with IPT. The traces are basically matched for the runs clustered in the same group. Besides, the CoV of the performance metrics (IPC, L1 cache miss rate) from the runs in the same cluster under the same configuration is 0.65 percent on average, indicating they are indeed very similar.

5 Case Studies

With the capability to capture similar execution patterns in parallel execution, VarCatcher can be used in many scenarios that require better analyzability and reproducibility for parallel workloads. This section studies the effectiveness of VarCatcher through five case studies, including (1) in-depth result analysis, (2) customized evaluation strategies, (3) a reproducibility study, (4) a feature impact study, and (5) performance diagnosis. We used both Intel processor trace (IPT) and Transformer [12] to evaluate their effectiveness and overheads.

We collected performance metrics, such as Instruction per cycle (IPC), throughput, execution time, and cache miss rate using VarCatcher. Their results are all very similar. Hence, we only provide IPC data here for brevity. We tested applications in Phoenix-2 [10], [11] and PARSEC-2.1 [20] with the reference input. Due to the long simulation time for PARSEC-2.1 with the reference input, we only use Phoenix-2 in simulation studies.

The baseline hardware for simulation has 4 cores with 64KB private L1 data cache, 64 KB private L1 instruction cache and 1 MB shared L2 cache. The memory latency is 200 cycles. Other configurations are similar to those widely-used in other architecture evaluations [12], [21], [22], [23], [24]. The IPT-based VarCatcher runs on Intel Processor 5Y70 CPU at 1.10 GHz, It has two cores, four hardware threads and 8 GB main memory.

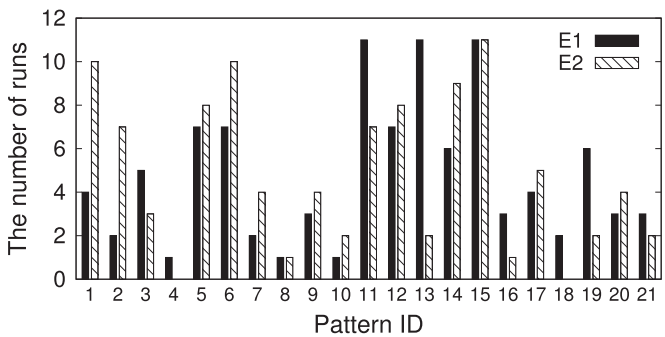


Fig. 15. The distribution of the 21 distinct patterns in the two 100-run experiments (E1 and E2) for *K-means*.

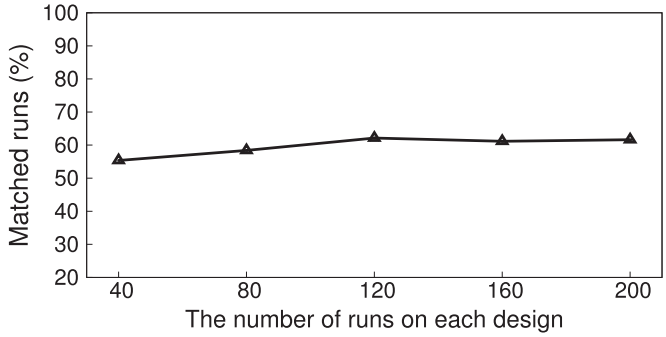


Fig. 16. The percentage trend of matched runs. Each application runs from 40 times to 200 times on two designs. Matched run means a run has a one-to-one corresponding run on the other design with similar execution pattern.

5.1 In-Depth Result Analysis

With VarCatcher, we can have a more in-depth analysis on the experimental results. Here, we present a comprehensive analysis on execution patterns and their distribution across multiple runs to gain some insights to the variability of their performance.

Different run sets share many common execution patterns. We run two 100-run sets on two L2 cache designs in Section 2.1 as experiments E1 (for Design-1) and E2 (for Design-2) using all applications in Phoenix-2 (including *linear regression* (*LR*), *matrix multiply* (*MM*), *PCA*, *word count* (*WC*), *histogram*, *string match* (*SM*), and *K-means*). We collect the number of common patterns in each 100-run set as shown in Fig. 14. The common patterns are the execution patterns appeared in both two experiments, and the "unique pattern" are the remaining patterns that appear only in one experiment. Only around 7.06 percent of patterns are unique in both 100-run sets. Therefore, most of the execution patterns appear in both run sets if the number of runs is large enough, as shown in Fig. 14.

The distribution of execution patterns in different run sets is quite random. There is a total of 21 distinct execution patterns in the benchmark program *K-means* in the above two 100-run sets E1 and E2. We plot the distribution of the execution patterns in each set with *Pattern ID* marked on the *x*-axis in Fig. 15 for it. The characteristics of the distribution are very similar in all other benchmarks.

Each bar shows the number of times that each execution pattern shows up in each run set (in E1 and E2). Most execution patterns appear in both run sets, but the distribution of execution patterns in each run set is very random.

The percentage of matched runs in two run sets does not increase notably even if we increase the number of runs in each set. The matched runs in two run sets are the runs in one run set having a one-to-one corresponding run with similar execution pattern on the other execution set. For example, there are four matched runs in each design on Pattern-1 in Fig. 15. To see whether the number of matched runs will increase with the number of runs increasing, we increase the number of runs from 40 to 200 times on two designs. As shown in Fig. 16, the percentage of matched runs remains pretty flat, and reaches around 61.6 percent when we increase from 40 runs to 200

TABLE 2
A Summary of Performance Improvement of Design-2 Over Design-1 Using Three Customized Strategies and a Statistical Normality Approach

benchmark	4-core				8-core			
	M-Run	**C-Pattern**	**A-Pattern**	**Normality**	**M-Run**	**C-Pattern**	**A-Pattern**	**Normality**
K-means	1.00%	0.93%	0.89%	-0.87%	0.81%	2.21%	2.23%	5.38%
LR	6.37%	6.52%	6.93%	7.41%	6.94%	7.04%	4.58%	1.04%
histogram	2.07%	2.01%	1.77%	-0.47%	7.18%	6.45%	6.03%	4.52%
MM	1.68%	1.49%	3.26%	13.06%	4.87%	4.66%	2.81%	-1.15%
PCA	8.08%	6.88%	4.45%	3.37%	2.28%	2.32%	3.02%	5.13%
SM	4.94%	5.12%	5.66%	8.22%	2.35%	2.73%	2.44%	9.58%
WC	2.19%	1.98%	4.96%	8.18%	5.26%	6.90%	8.75%	13.08%

runs in each run set. This study confirms that the distribution of execution patterns in a large number of runs does not necessary follow a normal distribution [3].

5.2 Customized Evaluation Strategy

Using VarCatcher, users can customize evaluation strategies to summarize the performance results. We present three possible strategies to summarize the performance results based on matched execution runs, common execution patterns, and all execution patterns, respectively.

In the first strategy, we summarize the performance results based on the execution runs with matched execution patterns to minimize potential performance variability. In the second strategy, since different sets of runs usually share many common execution patterns as shown in Fig. 14, we could summarize the performance results based on all common execution patterns instead of all execution runs. In the third strategy, since all distinct execution patterns have different performance characteristics, we can summarize performance results based on all distinct execution patterns.

We use shared L2 cache design in Section 2.1 again as our example. Table 2 lists summarized results using different strategies on the improvement of Design-2 over Design-1 on 4-core and 8-core platforms. All applications in Phoenix-2 are executed 200 times. Note that the number of runs to achieve a 95 percent confidence level is 121 using the statistical normality approaches [1] as mentioned in Section 2.1.

The second and the sixth columns marked under (M-Run) show the performance results using the first strategy to summarize the results of matched runs. The third and seventh columns marked under (C-Pattern) show the performance results using the second strategy to summarize the results based on common patterns. The fourth and eighth columns marked under (A-Pattern) show the performance results using the third strategy to summarize the results based on all patterns. The fifth and ninth columns marked under (Normality) show the performance results using the statistical normality approach that calculates the arithmetic mean using all runs.

Based on these results, we can see that performance results using statistical normality approach can differ significantly from those of three customized strategies, especially in *K-means*, *histogram*, and *MM*, while the performance results among the three customized strategies appear to be more consistent. The first reason is that the performance variability caused by different distinct execution patterns are better controlled and more stable in the three customized strategies than the statistical normality method in which there is no control of the distribution of the distinct execution patterns. The second reason is that, due to unmatched runs, there are only about 61.6 percent matched runs on average for all applications as shown in Fig. 16. Therefore, 38.4 percent runs are unmatched, which could cause more performance variability to the summarized results. More importantly, the performance of these unmatched runs is different from that of matched ones, which significantly affects the results of the statistical approach.

As opposed to statistical inference, VarCatcher has no constraint on the number of runs as it focuses more on the distinct execution patterns than the number of runs. The number of runs depends on a user's choice, and on how many distinct execution patterns the user intends to capture. As most distinct execution patterns can be captured in a small number of runs, it is more efficient for users to summarize the performance results based on distinct execution patterns than on a large number of runs as in statistical inference. We run 200 times in our experiments just for a fair comparison with statistical normality approaches, which needs a large number of runs to achieve a certain confidence level.

VarCatcher tackles performance variability through managing distinct execution patterns, which allows for more detailed exploration on performance evaluation. It will be our future work to combine more rigorous statistical approaches with the execution patterns obtained by VarCatcher.

5.3 Systematic Approach to Reproducibility

As mentioned earlier, it is neither very effective nor efficient to reproduce the results of parallel execution using statistical inference for validation. Many deterministic execution [25], [26], [27], [28], [29] and deterministic record-and-replay [30], [31], [32], [33], [34], [35] have been proposed to counter the performance variability for reproducibility. However, most of those techniques can only produce a specific execution pattern with significant overheads.

In VarCatcher, we run a parallel workload a limited number of times and record all different execution patterns of those runs with their corresponding performance results. All these recorded execution patterns and performance results can then be used for validation of future runs if the same execution patterns are reproduced. It is a systematic way of capturing all execution patterns, as opposed to a single particular execution pattern as in deterministic record-and-replay and deterministic execution approaches.

TABLE 3
The Cycle Results of Original Runs and the Repeated Runs

	Pattern-1	Pattern-2	Pattern-3	Pattern-4
O1 Evaluation	3,092,990	2,469,376	3,250,440	2,637,824
O1 Repeat	3,080,096	2,457,596	3,323,520	2,612,323
	Pattern-5	**Pattern-6**	**Pattern-7**	-
O3 Evaluation	1,750,112	2,100,952	2,019,312	-
O3 Repeat	1,797,044	2,173,944	2,022,360	-

Their performance results are consistent.

As shown in Section 5.1, most of the parallel workloads do not have too many execution patterns. It also does not take many runs to capture most of those execution patterns. New execution patterns can be added to such a database when they appear.

Here we use a performance comparison between GCC *O1* option and *O3* option as an example. *O3* is a higher optimization level than *O1*. We run 5 times for each option, and simply use their averages to compute the speedup of O3 over O1. The PCVs of all those runs are collected using IPT.

Here, we use matrix multiply (MM) in Phoenix-2 to show how it works. There are 4 distinct execution patterns for the *O1* option. It requires 13 runs to get all 4 patterns. There are 3 distinct patterns for *O3* option, and it takes 7 runs to capture all of them. Table 3 shows the execution time in cycles for all distinct execution patterns for *O1* and *O3* in the first and the fourth row, respectively. When repeated runs with similar execution patterns are identified, their execution time is also very similar to the recorded results, thus the results of the two runs are validated.

We also study all applications in Phoenix-2 and several applications in PARSEC-2.1 (including *blackscholes* (*BS*), *bodytrack* (*BT*), *fluidanimate* (*FA*), *freqmine* (*FQ*), *swaptions* (*SWAP*), *x264* and *vips*). Fig. 17 shows the average speedups of *O3* over *O1* with 5 runs. The left bar marked as "Evaluation" shows the speedup values of the original 5 runs. The middle bar marked as "Repeat" shows the speedup values of 5 repeated runs with the same corresponding execution patterns to the original 5 runs using VarCatcher. The right bar marked with "Stat. Repeat" shows the speedup values based on the statistical nonparametric testing approach [14]. As the results show, statistical testing approaches can hardly match the reproducibility of VarCatcher to reproduce the original speedup values as closely as VarCatcher can.

5.4 Feature Impact Study

Designers often need to determine the performance impact of changing a specific feature of the system. However, it is hard to analyze whether such effect appears and how they impact on the results on the multi-core platform, because the effect may be mixed with parallel variability. With VarCatcher, we can evaluate the influence of these factors to realize *casual analysis* [36] or *before-and-after* comparison technique by judging comparing the performance of each distinct execution pattern before and after the feature is added or changed.

Here, we use the link order during the compilation of a program with multiple modules as an example. We would like to see how the change of the link order impacts the performance of the generated code. To do this, we can compare the performance of the generated code under different link orders on the same execution pattern, and see if there is any significant performance change.

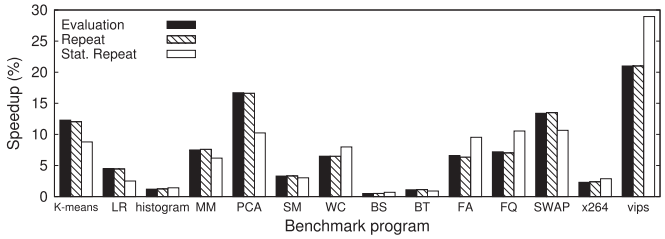


Fig. 17. Reproducibility comparison on the same platform.

We first run the target applications with the original link order. We then change the link order and compare the performance changes on each distinct execution pattern. If the performance of many execution patterns changes, then the performance will be affected by link order. For example, we first collect the performance data of *x264* in PARSEC-2.1 25 times in its original link order. Using VarCatcher, we found there were 15 distinct execution patterns. The CoV of the performance metric (e.g., IPC) in the same cluster is only about 0.5 percent on average. We then run 25 times of 9 other different link orders. Among the 15 execution patterns identified, 8 patterns show performance variability among 9 different link orders ranging from 3.53 to 26.19 percent. The change of link orders also affects 3 out of 8 execution patterns for *WC* in Phoenix-2, 4 out of 9 patterns for *freqmine*, and 3 out of 12 patterns for *swaptions* in PARSEC-2.1 ranging from 1.56 to 21.89 percent. From such an analysis, we can confirm that the link order has a significant influence on the performance for parallel applications.

5.5 Support for Performance Diagnosis

With significant performance variability during parallel executions, designers or programmers may want to find out what cause such performance discrepancies and how they influence the performance. However, prior methods diagnose performance under a constrained variability condition or a replay condition, which limit their capability to fix as many as performance bugs as possible. Since VarCatcher can distinguish different execution patterns, it can uncover how many and which execution patterns would have performance problem after an optimization or a design adjustment. Therefore, with the help of VarCatcher, traditional performance tools can diagnose performance for the fixed execution patterns.

Using shared L2 cache design example in Section 2.1, the average performance improvement in the first 10 runs is 12.62 percent while the average improvement in the second 10 runs is only 1.12 percent. Designers may want to carry out performance diagnosis on how such low performance comes about. Using VarCatcher to obtain execution patterns and their distribution, we can more easily diagnose where the performance improvement is coming from by identifying the best and the worst performed execution patterns.

Here, we use *LR* in Phoenix-2 as an example. We run *LR* 200 times for two shared L2 designs, respectively. As shown in Table 4, there are 15 matched clusters for the two designs, and the performance difference ranges from 0.92 to 15.85 percent. Based on such results, designers could identify the

TABLE 4
The Performance Improvement of *LR* in Two Designs

Pattern-1	Pattern-2	Pattern-3	Pattern-4	Pattern-5	Pattern-6
1.66%	15.85%	9.24%	6.90%	10.93%	4.41%
Pattern-7	Pattern-8	Pattern-9	Pattern-10	Pattern-11	Pattern-12
11.36%	4.33%	3.37%	3.90%	2.41%	8.66%
Pattern-13	Pattern-14	Pattern-15	Pattern-16	Unmatch	
0.92%	8.56%	2.72%	4.12%	10.38%	

The result of each pattern is calculated by comparing the runs from two designs in the same cluster. The pattern marked "unmatch" is the performance improvement by comparing the patterns that are not common to both designs.

patterns with the worst performance improvement (e.g., Pattern-13 achieves only 0.92 percent improvement) or the pattern with the best performance improvement (e.g., the Pattern-2 achieves 15.85 percent performance improvement) during the performance diagnosis, and look into more details on why such discrepancies occur based on their execution patterns.

If the number of runs with Pattern-13 appears frequently in the experiment, the improvement would be below expectation. In such a case, designers may want to find ways to avoid certain patterns or promote certain patterns to occur using some stable scheduling tools [37].

6 Related Work

Feature Extraction and Sampling. Sherwood et al. [8] introduced basic block vector to characterize the program behavior on singe-core platforms. Kambadur et al. [38] presented Harmony, which collects parallel block vector (PBV) in the compiler to extract parallel features under thread granularity for hardware or software optimization and acceleration. Due to non-uniformly partitioned intervals, PBV cannot reflect the execution speed of different threads and thus cannot be used to characterize the execution patterns defined in VarCatcher. Based on the repetitive execution behavior of programs, sampling techniques, such as Co-Phase Matrix [39] and multi-threaded sampling techniques [40], are proposed to reduce simulation time through only simulating the representative portion of the applications.

VarCatcher is also based on collecting online execution characteristics (PCV), but with a completely different goal from the above work. Further, PCV not only characterizes the execution paths within a core, but also the relative speeds among cores.

Statistics-Based Analysis. Such approaches can be mainly divided into three categories: Normality [1], [2], [13], Non-parametric testing [3], [14] and Visual test [4], which try to summarize the varied performance of multiple runs.

Normality approaches [1], [2] are based on the Central Limit Theorem, which indicates the average of a sufficiently large number of independent random variables tends to follow the normal distribution. They calculate the necessary number of runs to fulfill the preset confidence level using the parametric estimate based on this theorem. However, As analyzed in [3], the runs of multiple applications do not always follow the normal distribution even with many execution data collected unless there is a mechanism to ensure the normality of sample runs [13].

Non-parametric testing (hypothesis testing), such as Mann-Whitney U Test or Wilcoxon Rank-Sum Test [3], do not make any assumption on any distribution. These approaches determine whether two sets of samples are significantly different or not, and can be extended to calculate performance speedup [3]. Such results provide little information for analysis and are also vulnerable to the number of runs [2].

Visual tests first calculate the confidence interval of the performance for each design and then check the overlapping of two intervals. Such results provide very limited information for analysis. Georges et al. [4] summarizes the comparison results if the width of the interval is less than a threshold with an increasing number of sample runs. The conclusions are also considered to be too conservative [2].

In addition, some randomization techniques are often combined with statistical methods to summarize the results with variability, such as environmental randomization [13], [36], [41] and workload randomization [42].

While such methods are useful to summarize results among multiple sample runs, they are not sufficient to resolve the analyzability and reproducibility issues of the parallel workloads. VarCatcher can provide analyzability and reproducibility for performance evaluation with variability compared to these methods.

Other Approaches. There are some other approaches that try to tackle performance variability through minimizing variability. Lepak et al. [15] and Pusukuri et al. [43] reduce variability in evaluation or optimization through modifying simulators or scheduling policy on threads. Vera et al. [44] executes a representative trace of every test benchmark to reduce experimental bias.

Such approaches can mitigate performance variability by only dealing with a specific execution pattern, but they cannot handle large performance variability in different execution patterns observed in many parallel workloads, while VarCatcher can capture various patterns from different runs with variability.

7 Conclusion

To tackle performance variability in parallel workloads, we propose an evaluation framework called VarCatcher. It uses PCVs to characterize the execution patterns of parallel execution, and use a clustering method to group similar execution patterns for further analysis. We provide several use scenarios to show such an approach can be quite useful and effective on various aspects of the performance evaluation on parallel workloads and machine designs.

Acknowledgments

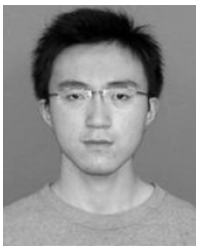
We are grateful to supports from the National Key Research and Development Program of China (No. 2016YFB0800104), the National Natural Science Foundation of China (No. 61672160) and Shanghai Science and Technology Development Funds (16JC1400801). We would like to thank all our anonymous reviewers for valuable feedback on the paper. Weihua Zhang is the corresponding author.

References

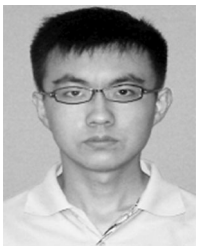
[1] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulators of multi-threaded workloads," in *Proc. 9th Int. Symp. High-Perform. Comput. Archit.*, 2003, pp. 7–18.

[2] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," *ACM SIGPLAN Notices*, vol. 48, no. 11, pp. 63–74, 2013.
[3] T. Chen, Y. Chen, Q. Guo, O. Temam, Y. Wu, and W. Hu, "Statistical performance comparisons of computers," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2012, pp. 1–12.
[4] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 57–76, 2007.
[5] D. J. Lilja, *Measuring Computer Performance, A Practitioner's Guide*. Cambridge, U.K.: Cambridge Univ. Press, 2004.
[6] S. Krishnamurthi and J. Vitek, "The real software crisis: Repeatability as a core value," *Commun. ACM*, vol. 58, no. 3, pp. 34–36, 2015.
[7] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Math. Statist. Probability*, 1967, pp. 281–297.
[8] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. 10th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2002, pp. 45–57.
[9] Intel, "Intel processor tracing," 2013. [Online]. Available: https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing
[10] K. Taura, K. Kaneda, T. Endo, and A. Yonezawa, "Phoenix : A parallel programming model for accommodating dynamically joining/leaving resources," in *Proc. 9th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2003, pp. 216–229.
[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2007, pp. 13–24.
[12] Z. Fang, et al., "Transformer: A functional-driven cycle-accurate multicore simulator," in *Proc. 49th Annu. Des. Autom. Conf.*, 2012, pp. 106–114.
[13] C. Curtsinger and E. D. Berger, "Stabilizer: Statistically sound performance evaluation," in *Proc. 18th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2013, pp. 219–228.
[14] E. D. B. Charlie Curtsinger, "Coz: Finding code that counts with causal profiling," in *Proc. 25th Symp. Operating Syst. Principles*, 2015, pp. 184–197.
[15] K. M. Lepak, H. W. Cain, and M. H. Lipasti, "Redeeming IPC as a performance metric for multithreaded programs," in *Proc. 12th Int. Conf. Parallel Architectures Compilation Techn.*, 2003, pp. 232–243.
[16] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proc. 30th Int. Symp. Comput. Archit.*, 2003, pp. 336–349.
[17] J. Lau, S. Schoenmackers, and B. Calder, "Transition phase classification and prediction," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2005, pp. 278–289.
[18] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," in *Proc. 11th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2004, pp. 165–176.
[19] R. E. Kass and L. Wasserman, "A reference Bayesian test for nested hypotheses and its relationship to the schwarz criterion," *J. Amer. Statistical Assoc.*, vol. 90, no. 431, pp. 928–934, 1995.
[20] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Architectures Compilation Techn.*, 2008, pp. 72–81.
[21] H. Asadi, V. Sridharan, M. B. Tahoori, and D. Kaeli, "Vulnerability analysis of l2 cache elements to single event upsets," in *Proc. Des., Autom. Test Eur.*, 2006, pp. 1–6.
[22] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proc. 41st Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2008, pp. 222–233.
[23] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal block placement and replication in distributed caches," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 184–195.
[24] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan, "SPATL: Honey, I shrunk the coherence directory," in *Proc. Int. Conf. Parallel Architectures Compilation Techn.*, 2011, pp. 33–44.
[25] C. Pereira, H. Patil, and B. Calder, "Reproducible simulation of multi-threaded workloads for architecture design exploration," in *Proc. IEEE Int. Symp. Workload Characterization*, 2008, pp. 173–182.
[26] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic shared-memory multiprocessing," *IEEE Micro*, vol. 30, no. 1, pp. 40–49, Jan./Feb. 2010.
[27] S. R. Sarangi, B. Greskamp, and J. Torrellas, "Cadre: Cycle-accurate deterministic replay for hardware debugging," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2006, pp. 301–312.
[28] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "CoreDet: A compiler and runtime system for deterministic multithreaded execution," in *Proc. Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2010, pp. 53–64.
[29] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu, "DrDebug: Deterministic replay based cyclic debugging with dynamic slicing," in *Proc. Annu. IEEE/ACM Int. Symp. Code Generation Optimization*, 2014, Art. no. 98.
[30] Y. Chen and H. Chen, "Scalable deterministic replay in parallel full-system emulator," in *Proc. 18th ACM SIGPLAN Symp. Pinciples Practice Parallel Program.*, 2013, pp. 207–218.
[31] Y. Chen, W. Hu, T. Chen, and R. Wu, "LReplay: A pending period based deterministic replay scheme," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2010, pp. 187–197.
[32] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution effciently," in *Proc. 35th Annu. Int. Symp. Comput. Archit.*, 2008, pp. 289–300.
[33] K. Veeraraghavan, et al., "DoublePlay: Parallelizing sequential logging and replay," in *Proc. 16th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2011, pp. 15–26.
[34] M. Xu, R. Bodik, and M. D. Hill, "A hardware memory race recorder for deterministic replay," *IEEE Micro*, vol. 27, no. 1, pp. 48–55, Jan. 2007.
[35] D. Ernst, A. Hamel, and T. Austin, "Cyclone: A broadcast-free dynamic instruction scheduler with selective replay," in *Proc. 30th Annu. Int. Symp. Comput. Archit.*, 2003, pp. 253–262.
[36] T. Mytkowicz, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proc. 14th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2009, pp. 265–276.
[37] H. Cui, et al., "Parrot: A practical runtime for deterministic, stable, and reliable threads," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 388–405.
[38] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: Collection and analysis of parallel block vectors," in *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 452–463.
[39] M. Van Biesbrouck, T. Sherwood, and B. Calder, "A co-phase matrix to guide simultaneous multithreading simulation," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2004, pp. 45–56.
[40] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2013, pp. 2–12.
[41] A. R. Alameldeen, et al., "Evaluating non-deterministic multithreaded commercial workloads," in *Proc. 5th Workshop Comput. Archit. Evaluation Using Commercial Workloads*, 2002, pp. 30–38.
[42] D. Tsafrir, K. Ouaknine, and D. G. Feitelson, "Reducing performance evaluation sensitivity and variability by input shaking," in *Proc. 15th Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst.*, 2007, pp. 231–237.
[43] K. K. Pusukuri, R. Gupta, and A. N. Bhuyan, "Thread tranquilizer: Dynamically reducing performance variation," *ACM Trans. Archit. Code Optimization*, vol. 8, no. 4, pp. 46–66, 2012.
[44] J. Vera, F. J. Cazorla, and A. Pajuelo, "Fame: Fairly measuring multithreaded architectures," in *Proc. 16th Int. Conf. Parallel Archit. Compilation Techn.*, 2007, pp. 305–316.

Weihua Zhang received the PhD degree in computer science from Fudan University, in 2007. He is currently an associate professor in Parallel Processing Institute, Fudan University. His research interests include compilers, computer architecture, parallelization, and systems software.



Xiaofeng Ji is currently working toward the graduate degree in Software School, Fudan University and working with Parallel Processing Institute. His work is related to CUDA programming, simulation, parallel optimization and so on.



Bo Song is working toward the graduate degree in Software School, Fudan University. He is working in the architecture team of Parallel Processing Institute, Fudan University. His recent area is image retrieval algorithms. His work is also related to computer architecture, simulation, execution variability and so on.

Shiqiang Yu is currently working toward the graduate degree in Software School, Fudan University and working with Parallel Processing Institute. His work is related to CUDA programming, computer architecture, parallel optimization and so on.

Haibo Chen received the BSc and PhD degrees in computer science from Fudan University, in 2004 and 2009, respectively. He is currently a professor in the School of Software, Shanghai Jiao Tong University, doing research that improves the performance and dependability of computer systems. He is a senior member of the IEEE and the IEEE Computer Society.

Tao Li received the PhD degree in computer engineering from the University of Texas at Austin. He is a full professor in the Department of Electrical and Computer Engineering, University of Florida. His research interests include computer architecture, microprocessor/memory/storage system design, virtualization technologies, energy-efficient/sustainable/dependable data center, cloud/big data computing platforms, the impacts of emerging technologies/applications on computing, and evaluation of computer systems. He received 2009 National Science Foundation Faculty Early CAREER Award, 2008, 2007, 2006 IBM Faculty Awards, 2008 Microsoft Research Safe and Scalable Multi-core Computing Award, and 2006 Microsoft Research Trustworthy Computing Curriculum Award. He co-authored a paper that won the Best Paper Award in HPCA 2011 and five papers that were nominated for the Best Paper Awards in ICPP 2015, CGO 2014, DSN 2011, MICRO 2008, and MASCOTS 2006. He is one of the College of Engineering winners, University of Florida Doctor Dissertation Advisor/Mentoring Award for 2013-2014 and 2011-2012.

Pen-Chung Yew has been a professor in the Department of Computer Science and Engineering, University of Minnesota since 1994, and was the head of the Department and the holder of the William-Norris Land-Grant chair professor between 2000 and 2005. He also served as the director of the Institute of Information Science (IIS), Academia Sinica, Taiwan, between 2008 and 2011. Before joining the University of Minnesota, he was an associate director of the Center for Supercomputing Research and Development (CSRD), University of Illinois at Urbana-Champaign. From 1991 to 1992, he served as the program director of the Microelectronic Systems Architecture Program in the Division of Microelectronic Information Processing Systems, US National Science Foundation, Washington, DC. He served as the editor-in-chief of the *IEEE Transactions on Parallel and Distributed Systems* between 2000 and 2005. He has also served on the organizing and program committees of many major conferences. His current research interests include system virtualization and compilers and architectural issues related multi-core/many-core systems. He is a fellow of the IEEE.

Wenyun Zhao received the master's degree from Fudan University, in 1989. He is a full professor in the School of Computer Science, Fudan University. His current research interests include software reuse, software product line, software component, and architecture.