VAPRO: Performance Variance Detection and Diagnosis for Production-Run Parallel Applications

Liyan Zheng
Tsinghua University
zhengly20@mails.tsinghua.edu.cn

Jidong Zhai
Tsinghua University
zhaijidong@tsinghua.edu.cn

Xiongchao Tang
Tsinghua University and Sangfor Technologies Inc.
tomxice@gmail.com

Haojie Wang
Tsinghua University
wanghaojie@tsinghua.edu.cn

Teng Yu
Tsinghua University
yuteng@tsinghua.edu.cn

Yuyang Jin
Tsinghua University
jyy17@mails.tsinghua.edu.cn

Shuaiwen Leon Song
University of Sydney
leonangel991@gmail.com

Wenguang Chen
Tsinghua University and BNRist
cwg@tsinghua.edu.cn

Abstract

Performance variance is a serious problem for parallel applications, which can cause performance degradation and make applications' behavior hard to understand. Therefore, detecting and diagnosing performance variance are of crucial importance for users and application developers. However, previous detection approaches either bring too large overhead and hurt applications' performance, or rely on nontrivial source code analysis that is impractical for production-run parallel applications.

In this work, we propose VAPRO, a performance variance detection and diagnosis framework for production-run parallel applications. Our approach is based on an important observation that most parallel applications contain code snippets that are repeatedly executed with fixed workload, which can be used for performance variance detection. To effectively identify these snippets at runtime even without program source code, we introduce State Transition Graph (STG) to track program execution and then conduct lightweight workload analysis on STG to locate variance. To diagnose the detected variance, VAPRO leverages a progressive diagnosis method based on a hybrid model leveraging variance breakdown and statistical analysis. Results show that the performance overhead of VAPRO is only 1.38% on average. VAPRO can detect the variance in real applications caused by hardware bugs, memory, and IO. After fixing the detected variance, the standard deviation of the execution time is reduced by up to 73.5%. Compared with the state-of-the-art variance detection tool based on source code analysis, VAPRO achieves 30.0% higher detection coverage.

CCS Concepts: • **Computing methodologies → Parallel algorithms**; • **Software and its engineering → Software performance**.

Keywords: Performance Variance; Anomaly Detection; System Noise

1 Introduction

Performance variance has been confirmed as a serious problem when running parallel programs on data centers [10], supercomputers [22, 36], and cloud platforms [32, 41, 42], which happens in different processes or threads within one execution and between executions. As the execution time of a parallel program is mostly determined by the slowest process or thread, performance variance may slow down the whole program even when only one process or thread is affected. As shown in Figure 1, the time spent on the same task with fixed nodes varies greatly. Variance not only leads to performance degradation or resource waste, but also makes applications' behavior unstable and hard to understand.

Performance variance comes from various sources, including OS interruption [11, 26], memory errors [48], cache conflicts [34], network interference [23], and many other hardware or software faults [25]. The varying symptoms of performance variance make detection and diagnosis extremely difficult [25]. General approaches like rerunning, tracing applications, and executing benchmarks during the execution of applications can help detect variance. However, such intrusive approaches introduce large overhead and cannot be deployed in a production environment, which is a serious limitation due to the poor reproducibility of performance variance. Therefore, a lightweight online detection and diagnosis approach is necessary to find out whether and

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508411

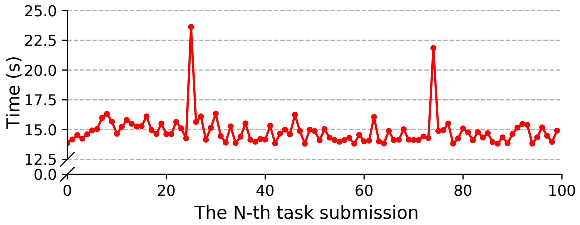

Figure 1. 100 repeated executions of 256-process NPB-CG on the same group of nodes on the Tianhe-2A supercomputer.

why performance variance happens during the execution of a running program.

To address this problem, we leverage an important observation that many parallel applications contain code snippets that are repeatedly executed with *fixed workload* [44, 45, 48]. For example, applications, such as neural networks and image processing, repeatedly execute certain math kernels to perform the same computation (with different data) in each iteration. These fixed-workload code snippets can be used as *benchmarks inside programs* to detect and diagnose performance variance at runtime since it is expected to take unchanged execution time for all executions.

Although other works have tried compiler techniques to identify fixed-workload snippets for variance detection, they have major limitations regarding variance detection, diagnosis, and usability. For the state of the art, vSensor [48], 1) it misses many fixed-workload snippets that cannot be determined at compilation, and fails to handle complex alias analysis [6]; 2) it cannot diagnose variance since it neglects the crucial properties of fixed workload for variance diagnosis; 3) it is impractical for closed-source applications and libraries, which are common in the production environment. Therefore, how to detect variance without source code and diagnose performance variance remains an **open problem**.

To overcome the limitations of existing approaches, we have to solve **two main challenges**. *1) How to identify.* An application generates a continuous instruction flow at runtime. We need to split the instruction flow into a set of fragments (i.e., an execution of a code snippet) and identify fragments with fixed workload at runtime. It is challenging because only limited runtime information is available for identification. *2) How to diagnose.* The runtime of programs does not contain much semantic information and the causes of variance are numerous. In addition, although various runtime performance data provides rich information, we should keep a small overhead for production environments, which limits the amount of collected data.

In this work, we propose Vapro, a light-weight performance variance detection and diagnosis tool without requiring source code, which is practical for production-run parallel applications. Vapro is based on two important observations missed by previous works. First, many code snippets have *de facto* fixed workload or only a few classes of workload, which are usable benchmarks inside programs but can only be identified at runtime. Second, the comparability of fixed workload makes it ideal for variance diagnosis. By comparing various performance information of fixed workload, the differences among them can effectively expose the causes of variance.

Based on the observations, we propose a series of novel approaches. There are three main contributions in our work.

- We propose a new data structure, called State Transition Graph (STG), to track program execution and reorganize the collected dynamic fragments. With a fixed-workload fragment identification algorithm executed on STG, we perform a light-weight online analysis algorithm to detect performance variance and quantify their influence.
- To diagnose variance without source code, we propose a progressive diagnosis method based on a hybrid model with a combination of variance breakdown and statistical analysis. It takes both software and hardware into consideration and is able to progressively locate fine-grained reasons with a small overhead, which can effectively guide variance diagnosis.
- We evaluate Vapro on real applications with up to 2,048 processes to verify its efficacy on large-scale parallel applications. Vapro only introduces 1.38% performance overhead on average and has a 30.0% higher detection coverage than the state-of-the-art tool. Vapro detects variances resulting from a hardware problem on Intel processors, distributed filesystem, memory, and computing resource competition. Experimental results show that optimizations based on the crucial reports from Vapro reduce the standard deviations of executions by up to 73.5% and bring a speedup up to 24.0%.

In this work, we focus on detecting and diagnosing the performance variance caused by external environment, such as the variance caused by hardware, OS, and communication functions implemented in shared libraries. Vapro helps users and system maintainers identify whether applications are running with performance slowdowns caused by environment. For the detected variance, Vapro provides the most possible reasons causing variance to help fix such problems.

2 Overview

Vapro is packaged as a dynamic library to perform data collection and analysis. It requires no re-compilation or re-link of applications. Figure 2 illustrates the workflow of Vapro. Each step is described in detail:

1. Intercepting Vapro splits the running progress of an application into a number of *fragments* (i.e., an execution of a code snippet) by intercepting the *external functions* provided by dynamic libraries. For a repeatedly executed code snippet, it generates many fragments at runtime.

2. Building STG (§3.2) Vapro generates an STG as a representation of the running progress of a program.

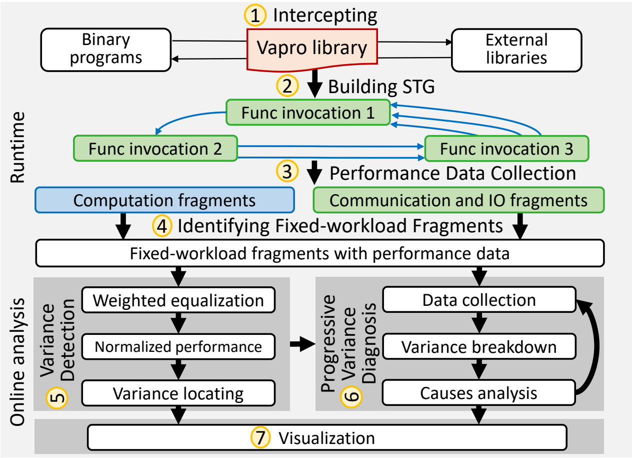

Figure 2. Vapro overview.

3. Performance Data Collection (§3.3) Vapro records runtime information for fragments, including elapsed time, function parameters, and performance counters.
4. Identifying Fixed-workload Fragments (§3.4) Vapro identifies fragments with fixed workload by clustering for each STG edge and vertex.
5. Variance Detection (§3.5) Vapro automatically locates performance variance by analyzing the clustering result.
6. Progressive Variance Diagnosis (§4) For detected variance, Vapro leverages a breakdown model and a statistical method to progressively pinpoint the potential causes.
7. Visualization For variance detection, Vapro plots a heat map to illustrate the normalized performance and reports the region of variance and the quantified performance loss. For variance diagnosis, Vapro breaks down the variance and shows the impact and time duration for each factor.

3 Performance Variance Detection

Vapro is based on the observation of fixed workload. In this section, we will introduce how Vapro locates performance variance by analyzing fixed-workload fragments.

3.1 Fixed-workload Fragments

```
MPI_Comm_size();
... // Omitted loops with similar characteristics
for (int i = 0; i < num_cols * num_vectors; i++)
    y_data[i] *= alpha;
...
MPI_Waitall();
```

Figure 3. A code snippet with fixed workloads in AMG [53].

A code snippet can generate several sets of fixed-workload fragments. Figure 3 shows an example of a code snippet between two MPI invocations. It is not a code snippet with a fixed workload at compilation time, since the loop termination condition is determined by two non-constant variables. However, although this snippet is executed hundreds of times in a program execution, there are only 7 different workloads. By distinguishing these different workloads at runtime and dividing them into separate sets of fixed-workload fragments, Vapro exploits code snippets that cannot be identified in static analysis-based tools, such as vSensor [48].

3.2 State Transition Graph

To identify potential code snippets with fixed workload for a parallel application, we first split the running progress of an application into a set of fragments. We propose a new data structure, named State Transition Graph (STG), to organize these fragments. We give a formal definition of STG below.
Definition 1. *State Transition Graph (STG) is a representation of the running progress of a parallel application. In an STG, vertices record a program's running states, while edges represent their transitions between different states. An application's running progress is partitioned into a set of fragments. From one fragment to another, the program has a state transition.*

STG is built during program executions. Vapro creates a vertex for each *running state* and an edge if the program transfers from one state to another. A key point of building STG is to attach fragments to STG according to their running states. In Vapro, we have two alternative approaches to record a running state. They are based on *call-site* and *call-path* information respectively. Using call-site information as running states generates a *context-free* STG, while using call-path generates a *context-aware* STG.

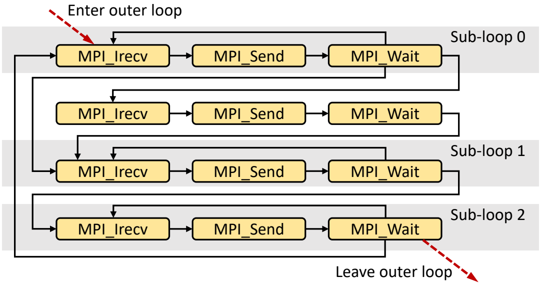

Figure 4. A context-free STG.

Context-free STG In a context-free STG, the state of a fragment is only determined by the call-site of the corresponding invocation. We use the CG program of NPB benchmark [9] as a running example to show how we build a state transition graph. Figure 4 shows the context-free STG for a nested loop in CG[1]. A vertex in Figure 4 represents a communication call-site in the source code. Edges in Figure 4 represent the transition between communication call-sites, i.e., the computation code snippets among them. With a context-free STG, all the communication and IO invocation fragments from the same call-site are attached to the same vertex, and all the computation fragments from the same computation block are attached to the same edge.

[1] It is the `cgitmax` loop in `cg.f:1170-1360`.

Context-aware STG Different from a context-free STG, a context-aware STG takes the entire call stack of external invocations into consideration. Invocations from the same call-site may have different call-paths. For example, each vertex or edge in Figure 4 corresponds to two vertices or edges in a context-aware STG, since the code is executed in both warm-up and real test stages with different call-paths.

3.3 Performance Data Collection

Performance Counters Performance counters, including software counters such as the number of page faults and context switches, and hardware counters like performance events provided by performance monitor unit (PMU), are valuable information for understanding performance. Vapro collects runtime data through performance counters for fixed workload identification and variance diagnosis. It adopts different methods for computation, communication, and IO workloads. We elaborate each on type as below.

Computation Workload The ideal way to classify the workload of two computation fragments is by comparing their instruction flows. However, its enormous overhead makes it impossible for light-weight online analysis. We have to find proxy metrics that are able to represent the workload and remain stable even under performance variance.

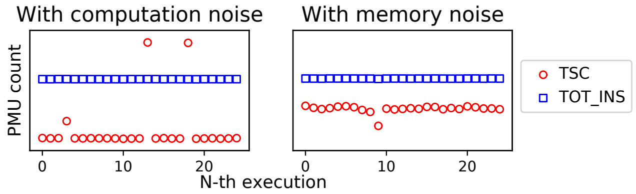

Figure 5. Performance data of fixed-workload computation fragments in 16-process B-scale CG under the computation and memory noises.

Figure 5 shows the values of TOT_INS (total number of instructions) and TSC (timestamp counter, a high-precision clock in CPU) for fixed-workload fragments in CG. We inject CPU and memory noises [2] while CG is executing. The results show that the TOT_INS is stable and insensitive to the noises, while the TSC, i.e., the execution time, is affected. Thus, Vapro takes TOT_INS as a crucial proxy metric for computation workload analysis by default. Users are able to specify other PMU metrics for analysis in Vapro as well, e.g., the number of load and store instructions, or cache miss rate. Collecting more performance metrics improves the precision of workload representation but introduces extra overhead.

Communication Workload Different from computation workload, PMU metrics of CPU cannot directly reflect communication workload. For example, if a receiving process is waiting for its sending process via busy-waiting, it will generate lots of memory access instructions. As a result, the number of memory access instructions is proportional to the waiting time rather than actual communication time. To address this problem, Vapro uses communication invocation arguments, including message size, the source and destinations processes, and other invocation-specific information, such as the scope of broadcast communication, instead of PMU values to approximate communication workload.

Vapro records the elapsed time of each communication invocation to analyze its performance. Although the elapsed time can be affected by load imbalance and some other factors, we take them as a whole into account since they demonstrate communication performance in some degree. For more precise timing on non-blocking communication, users can also choose the communication libraries exposing underlying communication time, such as the MPI library with an enhanced profiling layer [49].

IO Workload Similar to communication workload, Vapro collects function parameters to identify IO workload. Parameters that have an influence on IO performance are recorded, such as sizes of data, file descriptors, and IO modes.

3.4 Identifying Fixed-workload Fragments

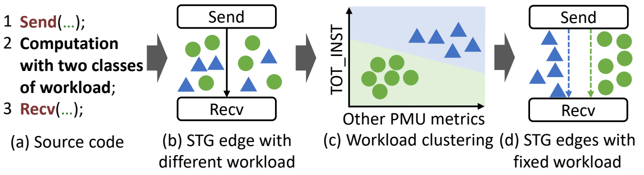
```
Send(...);
Computation
with two classes
of workload;
Recv(...);
```

Figure 6. Clustering fragments by their workload. The workload of fragments is represented with different shapes.

So far, we have attached fragments with runtime information to STG. However, as shown in Figure 6b, fragments on an STG edge or vertex can have various workload patterns, which cannot be directly used by Vapro for variance analysis. To identify fragments with fixed workload, we propose a light-weight approach based on workload clustering. Although Figure 6 only shows the clustering of computation fragments on edges, we similarly identify fixed-workload communication and IO fragments on STG vertices.

We represent all kinds of workload with a **workload vector** which contains normalized performance metrics and/or invocation arguments, and then cluster these workload vectors. Vapro has to cluster millions of fragments collected at runtime without any priori knowledge, such as the number of clusters (i.e., the number of different workloads). A large number of algorithms have been studied to decide the optimal number of clusters during clustering, such as hierarchical clustering and minimum radius based automatic k-means clustering [55]. However, they have high time complexities of at least $O(n^2)$ [12] and $O(n^{dk+1})$ [27], where k, d, n are the

[2] In this work, the computation noise is generated by executing stress [5] on the same CPU core of applications and the memory noise is generated by executing stream [33] on the idle cores.

number of clusters, dimensions, and vectors to be clustered. Most of these algorithms require complex computation (non-linear time complexity in the number of vectors) which is not suitable for light-weight performance analysis, especially for production-run parallel applications.

Clustering Algorithm To address this problem, we present an *ad-hoc clustering algorithm* (Algorithm 1) leveraging properties of the performance metrics. The Euclidean norm, i.e., the length of vectors, is used to classify different workload vectors. This is because a smaller norm of workload vectors, such as a smaller number of cache misses, usually means better performance. Performance variance usually enlarges these metrics rather than decreasing them. For metrics that are the larger the better, we convert them into the opposite metrics. Then, for fixed-workload fragments, their norms have a concentrated distribution near the smallest norm of all data, which indicates the stable performance. After selecting the least norm of unprocessed fragments, we find all fragments whose distance from the fragment with the least norm is smaller than a predefined threshold (5% in our implementation). For example, computation fragments within 1000-1050 instructions and 200-210 load and store instructions are put into the same cluster. The computational complexity of this algorithm is linear with respect to the number of workload vectors without regard to the sorting, so it introduces a small overhead. This will be shown in the evaluation in §6.2.

Algorithm 1: Clustering algorithm for identifying fixed workload snippets.

```
1 foreach edge/vertex in an STG do
2     Sort all fragments attached to this edge/vertex
        according to the norms of workload vectors
3     while unprocessed fragments exist do
4         Select the fragment with the smallest norm
5         Find similar fragments whose distance from
            the selected fragment is less than a
            predefined threshold
6         Move them into a new cluster
7     end
8     Report clusters with too few fragments
9 end
```

In Vapro, we do not strictly require the workload in the same cluster to be identical and tolerate a small difference. The main reason is the inherent error of PMU mechanism [51]. Additionally, Vapro aims to detect performance variance that has a significant performance impact, so the small difference of workload does not prevent detection for severe performance variance. During the post-processing (Line 8), clusters with too few fragments (less than 5 in our current implementation) but long execution time will be reported, which means that the corresponding execution path is not executed repeatedly. Users need to pay attention to whether these fragments represent abnormal performance.

3.5 Performance Variance Detection

After workload clustering, Vapro uses these fixed-workload fragments to detect performance variance. For a parallel application, Vapro detects performance variance both within a single process (in the temporal dimension) and across multiple processes (in the spatial dimension).

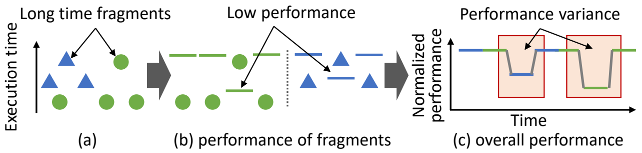

Figure 7. Detecting variance from multiple fragment clusters. Circles and triangles mean fragments with different workload and lines mean their normalized performance.

Intra-process Detection For fragments with the same workload, Vapro calculates the *normalized performance* for every fragment. As shown in Figure 7b, fragments with different workloads are analyzed separately. For each fragment in a cluster, Vapro normalizes their performance according to their time consumption. The performance of the fastest fragment is normalized to 1, and the others are between 0 and 1. Then, the normalized performance of both clusters is merged to produce an overall performance report. To report the performance of profiled programs concisely, Vapro merges the normalized performance from all clusters for computation, network, and IO, respectively.

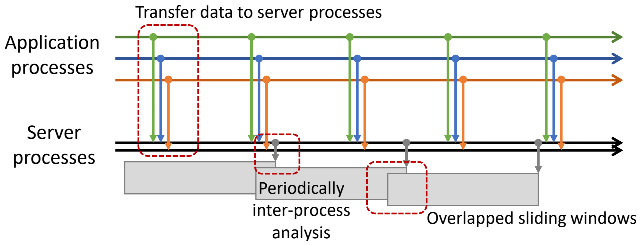
Transfer data to server processes
Application processes
Server processes
Periodically inter-process analysis
Overlapped sliding windows

Figure 8. Periodic analysis for multiple processes.

Inter-process Detection Since different processes and threads often have similar tasks for parallel applications, Vapro detects inter-process variance by analyzing fixed-workload fragments from multiple processes or threads. As shown in Figure 8, Vapro uses dedicated server processes for inter-process analysis. The server processes collect performance data from clients periodically. Each time, the server processes analyze the data for the last time window. The periods of analysis windows are overlapped so that the analyzed results from different periods can be concatenated together.

Vapro servers report normalized performance as a heat map, where performance variance is represented by light-colored blocks. Figure 9 shows an example of multi-threaded

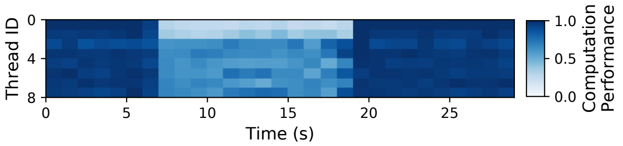

Figure 9. 8-thread PageRank under a memory noise.

PageRank under injected memory noise, where the vertical axis denotes different processes or threads and the horizontal axis denotes time progress.

Variance Locating Vapro automatically pinpoints the variance by the region growing method. It regards a contiguous region with normalized performance below a threshold (0.85 in our implementation) in the heat map as a possible variance. All possible variance is reported to users according to their impact on performance, which is calculated by the normalized performance. Users are able to select regions of interest on the heat map for diagnosis as well.

Sampling Similar to most performance monitoring tools, sampling is an optional approach for Vapro to trade off between overhead and accuracy. By skipping recording part of external invocations, Vapro dynamically achieves a desired balance between overhead and the detection ability. Heuristic sampling policies can be adopted by Vapro, such as skipping short fragments instead of long ones, to maintain high detection coverage with low overhead.

4 Performance Variance Diagnosis

In this section, we describe how Vapro automatically diagnoses the detected variance. Based on a variance breakdown model (§4.1), the execution time is broken down into several factors(§4.2). Vapro adopts a progressive analysis method to effectively locate the causes of variance (§4.3).

4.1 Variance Breakdown Model

Vapro leverages the crucial comparability of fixed workload to diagnose variance. Since fixed-workload fragments without performance variance should have the same execution time and similar results of performance counters, differentiating performance counters can reveal the reasons for variance. In this work, we only use performance counters inside processors and OS to illustrate our approach. Even though, the sources of variance vary and hundreds of counters exist. However, only a small number of counters can be simultaneously collected due to overhead constraints.

To diagnose the variance with a small overhead, we propose a variance breakdown model to guide the direction of diagnosis. As shown in Figure 10, it covers both hardware and software variance. A node in Figure 10 represents a *factor* accounting for partial execution time, which corresponds to certain hardware or software performance counters. Nodes are organized hierarchically according to the inclusion relation of their execution and form several stages for variance diagnosis. The model first divides computation time into five stage-one (S1) factors. For the time when processes are running on CPUs, the variance breakdown model divides it into four S1 factors according to the top-down structure of PMU events [54]. For example, the S1 factor backend bound represents the time spent on computation and memory access, i.e., the S2 factors core bound and memory bound in Figure 10. These S2 factors can be further broken down into S3 factors. For the process suspension caused by OS, its time is included in the S1 factor of suspension. Similarly, suspension can be further divided into fine-grained factors, such as page faults and other common OS events, which is extendable for covering more factors in diagnosis. By differentiating the time on each factor for fixed-workload fragments, Vapro quantifies the variance caused by each factor.

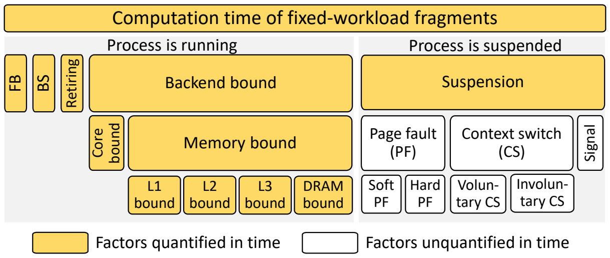

Figure 10. Variance breakdown model. Nodes with vertical text indicate that the underlying fine-grained factors are omitted. FE and BS mean frontend bound and bad speculation.

4.2 Quantifying Time of Factors

To compare the impact of different factors on variance diagnosis, Vapro quantifies the time cost for each factor by collecting corresponding performance counters. Performance counters have different units and Vapro classifies them according to whether they can be directly quantified in time. In Figure 10, the factors with background color are directly quantifiable in time, such as how long a process spends on CPU frontend bound [3]. With the help of well-designed hardware PMU events, a top-down time breakdown is feasible for factors in CPUs [54]. Since this breakdown relies on formulas according to the meaning of PMU events, we call this a *formula-based method.*

However, there are still many factors that cannot be directly quantified in time. For example, OS provides users with the count of page faults. But we cannot directly calculate the time of page faults according to this data. We propose an OLS-based (ordinary least squares) *statistical method* to

[3] On the Intel Ivy Bridge CPUs, the time fraction of frontend bound is equal to IDQ_UOPS_NOT_DELIVERED.CORE / (4 * CPU_CLK_UNHALTED.THREAD)

estimate the time of unquantified factors for variance diagnosis. Vapro separately processes fixed-workload fragments to leverage their comparability. For each cluster, all factors are normalized to the range of 0 to 1. Then, Vapro checks the multicollinearity of factors by the Farrar-Glauber test [21]. In the multivariate OLS, multicollinearity means that one explanatory variable can be linearly correlated with others. This makes the estimated coefficients unstable and possibly reduces the precision of results. Since some factors are related to each other, such as that a page fault in user space is also a context switch, multicollinearity tends to occur in our analysis. Vapro removes the multicorrelated factors one-by-one until multicollinearity does not exist in OLS.

Vapro takes execution time as the explained variable and factors as explanatory variables for OLS. For the OLS results, only factors with a significant influence ($p < 0.05$) on the time are considered in the following diagnosis. After scaling the coefficients to recover the normalization, we obtain the estimated time impact of each factor. For factors excluded from OLS due to multicollinearity, their coefficients are estimated by their multicollinear relationship. Thus, Vapro calculates the time of each factor to facilitate the performance diagnosis in §4.3.

To verify this OLS-based statistical method, we compare it with the formula-based method. For the injected noise, which will be shown in Figure 11, the impact of backend bound and suspension estimated by the formula-based method (89.4% and 4.9%) is consistent with the statistical method (86.6% and 3.1%).

4.3 Progressive Variance Diagnosis

Vapro adopts a progressive diagnosis method based on the above variance breakdown model, which progressively locates *major factors* in the current stage and diagnoses its fine-grained factors. The major factors are decided according to their *contribution* to variance, which means how much slowdown a factor causes. To calculate the contribution, fragments costing more than k_a times of the fastest fragment are regarded as abnormal fragments (1.2 is used in our implementation) and the others are normal ones. Vapro takes the average time of each factor in normal fragments as a reference value. Thus, the contribution of a factor is the difference between the time of this factor in abnormal fragments and the reference value. By summing up the contribution of all abnormal fragments, we obtain the contribution of a factor during a period of execution.

Figure 11 shows that fixed-workload fragments are injected with computing noise and memory contention using the same method in Figure 5. Since the noises mainly increase two S1 factors, suspension and backend bound, we take them as axes and omit the other three S1 factors. The average of the normal fragments is the origin in Figure 11. Thus, the coordinates of fragments mean the contribution of factors.

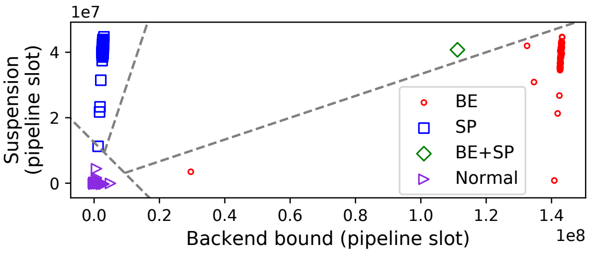

Figure 11. Variance breakdown of the fixed-workload fragments of 16-process CG under concurrent computing noise and memory contention. Each point represents a fragment and its marker indicates the major factor resulting in variance for fragments under variance. BE and SP mean backend bound and suspension. The dashed line shows the region boundary of different major factors.

Vapro selects the factors contributing more than a threshold (0.25 in our implementation) of overall variance as *major factors* for further diagnosis. Then, the server notifies clients to collect data for fine-grained factors. This diagnosing process repeats until the most fine-grained factors are determined. In such a progressive way, Vapro requires only a small number of concurrently active performance counters and thus imposes low overhead. As a trade-off, this method costs n client-server data transferring periods and n server analysis latencies to locate an Sn factor in the variance breakdown model. Compared with the long execution time of applications in the production environment, this diagnosis method reacts efficiently.

Vapro reports the impact and duration of each factor to users. The impact of a factor is calculated by summing up its contribution from all abnormal fragments. The duration of a factor is the total time of abnormal fragments whose major factors include it. For example, in the case of Figure 11, the process suspension accounts for 60.3% of the slowdown and influences 24.2% of the execution time. Previous noise detection tools cannot break down variance for analysis due to the lack of the precondition of fixed workload.

5 Implementation

In this section, we discuss the implementation details about how Vapro records performance data into STG and processes performance data for a parallel application.

Intercepting External Functions We leverage runtime symbol look-up interfaces on Linux, i.e., `dlsym`, and an environment variable of dynamic linkers, `LD_PRELOAD`, to transparently intercept these functions. Currently, Vapro supports the following external functions.

- Communication: MPI communications functions.
- IO: POSIX IO interfaces and MPI-IO functions.
- Multithreading: main POSIX `pthread` interfaces.
- User-defined explicit invocations.

Although most parallel applications heavily rely on external libraries, some of them execute with very few external function invocations for a long period. For these programs, Vapro inserts user-defined invocations into programs with the support of Dyninst [13], a binary rewriting tool. We insert explicit invocations at some key points, such as the entry and exit of functions. The binary exponential backoff strategy [24] is applied to adapt the frequency of profiling data collection and limit the overhead of Vapro.

Performance Data Analysis Vapro provides a multi-threaded server for online variance detection and diagnosis. For large-scale parallel applications, Vapro supports concurrent data collection with multiple servers to improve throughput. By equally assigning parallel processes to different servers, servers can achieve load balance. Further optimizations are feasible with data collection frameworks such as MRNet [15], which organizes servers into a tree-like structure.

6 Evaluation

6.1 Evaluation Setup

Vapro is implemented as a dynamic library and should be preloaded at program executions. We collect TOT_INS for workload clustering and evaluate it with real cases.

Platform We conduct the multi-process evaluation on the Tianhe-2A supercomputer, whose nodes have dual 12-core Intel Xeon E5-2692 v2 processors and 50 Gbps networks. The multi-threaded applications are evaluated on a server with dual 12-core Intel E5-2670 v3 processors.

Applications We evaluate (1) BERT [1], an efficient inference framework for the popular natural language processing model BERT, (2) PageRank [4], a multi-threaded graph computing application, (3) WordCount [2], a MapReduce-style program, (4) AMG [53], a parallel algebraic multigrid program solver, (5) CESM [30], the state-of-the-art climate simulator with more than 500,000 lines of code, (6) 6 programs from the PARSEC suite [14] which covers the applications of image processing, finance, and hardware design, and (7) 7 programs from the NPB [9] benchmarks with E-class problem size. These programs are from diverse fields and cover both multi-threading and multi-processing.

6.2 Overhead and Detection Coverage

Overhead Table 1 shows the performance overhead and detection coverage of Vapro and the state-of-the-art vSensor [48]. Since vSensor does not support multi-threaded applications, we present the evaluating results of multi-process applications in detail. The overhead of Vapro is small since Vapro is triggered only when external functions are invoked, which are time-consuming communication and IO operations. Therefore, the overhead of Vapro is bounded by a small ratio of the cost of external function invocations. Vapro with context-aware STG has higher overhead (3.81%) than that with context-free STG (1.80%) due to the costly call stack backtracing operation for the call-path information. Although both Vapro and vSensor introduce low performance overhead, vSensor, as a tool relying on source code analysis, fails to work on complex applications with large codebases, such as CESM.

Vapro is configured with a 15-second reporting period and one server process serves 256 application processes. The overhead of Vapro servers is only 0.4% (1/256) of the resources used by applications. For the storage overhead, Vapro generates 12.8 or 47.4 KB data per second for one thread or process on average. Since detailed performance data can be periodically analyzed and merged as normalized performance, Vapro has a small storage requirement.

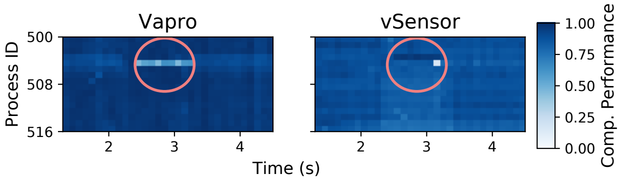

Figure 12. 1024-process SP under a computing noise.

Detection Coverage Since only the variance during the execution of fixed-workload fragments is detected, we define *detection coverage* as the ratio of time on repeated fixed-workload fragments to total execution time. To exemplify the importance of coverage, we compare different tools for the C-scale NPB SP program under the computing noise that lasts one second. As shown in the red circles in Figure 12, Vapro accurately detects the 50% performance loss caused by OS process scheduling, which equally divides CPU time for the application and the noise process. However, vSensor incorrectly reports a 90% performance loss lasting 1/10 second, since its detection coverage (8.7%) is significantly lower than Vapro (36.4%). The low coverage causes that vSensor cannot collect enough fragments to correctly show the impact of context switch caused by OS scheduling.

The detection coverage of Vapro exceeds 70.0% and outperforms vSensor by 30.0% on average, which is critical for the precision of detection. More importantly, Vapro works on programs with runtime fixed workload, such as AMG and EP, which static analysis-based vSensor cannot handle.

In Table 1, context-free STG outperforms context-aware STG with 10.8% higher average coverage and smaller overhead. This is because workload clustering overcomes the disadvantage of context-free STG. According to these evaluating results, context-free STG is more favorable, and we use it in the following experiments.

6.3 Verification of Fixed Workload Identification

To verify the fixed workload identification algorithm, we record the exact execution paths and compare them with the clustering results of Vapro. We evaluate four applications with medium codebases. All loops and branches in their hot

Table 1. Performance overhead and detection coverage (with 2048 and 1024 processes for CESM and the other multi-process applications, and 16 threads for multi-threaded programs). CA and CF mean Vapro with context-aware and context-free STG.

	Multi-process	Overhead (%)			Coverage (%)				Multi-threaded	Overhead (%)	Coverage (%)
	Applications	vSensor	CA	CF	vSensor	CA	CF		Applications	CF	CF
	AMG	2.02	1.34	0.37	0.00	57.5	66.4		BERT	0.75	72.8
	CESM	N/A	8.06	0.02	N/A	33.8	47.7		PageRank	2.70	47.3
NPB	BT	0.22	2.07	2.00	80.1	83.7	86.2		WordCount	0.41	74.1
	CG	0.00	0.29	0.72	19.5	78.3	78.2	PARSEC	FFT	0.16	66.9
	EP	0.00	1.04	1.04	0.0	87.5	87.5		blackscholes	0.00	84.9
	FT	2.21	4.73	5.13	93.2	72.0	72.2		canneal	2.63	81.3
	LU	1.12	8.56	2.88	65.9	97.4	97.7		ferret	0.02	79.0
	MG	1.03	6.99	2.86	76.2	5.1	77.7		swaptions	0.00	92.4
	SP	1.22	1.19	1.23	29.4	66.6	66.3		vips	1.85	96.7
	Mean	0.98	3.81	1.80	45.5	64.7	75.5		Mean	0.95	74.1

Table 2. Verification of fixed workload identification. C, H, and V mean completeness, homogeneity, and V-Measure (the harmonic mean of C and H) scores. Programs are executed with 16 processes or threads.

Applications	Number of fragments	C	H	V
CG	3801	1.00	1.00	1.00
FT	640	1.00	1.00	1.00
EP	16	1.00	1.00	1.00
PageRank	672	1.00	0.74	0.85

spots, which cover more than 80% of the execution time, are instrumented for recording execution paths.

Table 2 shows the clustering results by the homogeneity, completeness, and V-Measure scores [39]. All the completeness scores are equal to one, which means that fragments with the same workload are in the same cluster. For PageRank, the homogeneity score (0.74) indicates that some fragments with different workloads are clustered together. By inspecting its source code, we find that some fragments with approximately equal workload (e.g., a common 100000s-iteration loop with only less than twenty different arithmetic operations) are put into one cluster. Since these workload differences are small, such mixed clusters do not hinder Vapro from detecting variance which significantly impacts performance.

6.4 Comparing with Profiling Tools

We generate parallel computational noises to interfere with processes. After 2048-process E-scale CG is started, computing noises are injected into two different computing nodes. Figure 13 shows that Vapro accurately locates the performance variance (two white boxes) and reports a 42.8% computation performance loss. With the regression based on the variance breakdown model, Vapro reports that involuntary context switches have a significant negative influence ($p < 0.001$) on performance.

The execution time breakdown provided by profiling tools is often misleading. We take mpiP [50] as an example, whose result summarizes the computation time and communication time. The result of mpiP in Figure 14 shows that the communication time increases and the computation time remains the same, which indicates a network problem. mpiP highlights the significant increase in communication time caused by dependence, but omits the relatively small changes in computation time. However, with the help of fixed-workload fragments, Vapro catches the nuanced change and diagnoses it. For vSensor, it cannot pinpoint the source of variance although the variance is detected.

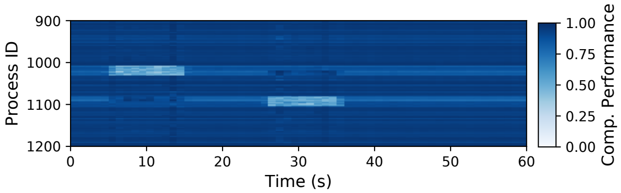

Figure 13. Detection results of 2048-process CG under software noises by Vapro.

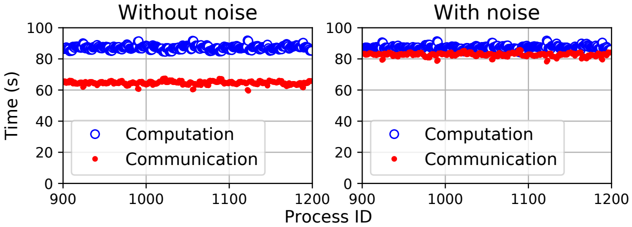

Figure 14. Results of 2048-process CG by mpiP.

6.5 Case Studies

In this subsection, we present three case studies covering variance caused by hardware cache variance, memory problem, and IO variance.

6.5.1 Detection of a Hardware Bug. In this case, we evaluate High Performance LINPACK (HPL) [20] with 36 processes on a computing node with dual 18-core Intel Xeon

Gold 6140 processors. All processes are bound to the dedicated processor cores to mitigate the interference of OS scheduling. In our test, HPL usually has a stable performance with performance variation of less than 2%, since it has relatively little communication. However, Vapro captures an abnormal execution with 22.2% longer execution time than the normal run.

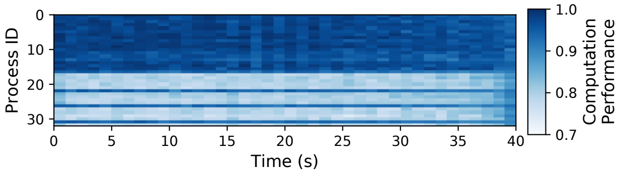

Figure 15. Detection results of an HPL execution under hardware variance detected by Vapro.

Figure 15 shows the normalized performance reported by Vapro. From the figure, we can find that there is a large performance variance among processes, especially for processes on the second processor socket, whose process IDs are between 16 and 31. With the progressive variance diagnosis, Vapro reports that 96.6% of the slowdown results from the backend bound in the CPU pipeline. The fine-grained breakdown shows that the L2 and DRAM bound (48.2% and 38.0% of the slowdown) are mainly responsible for this variance. This result implies that the extra cache misses and memory accesses impair the performance. By recording several low-level micro-architecture PMU events related to cache and repeating the execution, we verify that this variance is correlated with a PMU metric counting the number of CPU cycles stalling on L2 cache miss[4]. This phenomenon of variance matches a severe Intel processor hardware bug related to the L2 cache [28, 34], which makes data in the L2 cache evicted and randomly generates significant slowdowns.

To mitigate this problem, we leverage the huge page mechanism to decrease the frequency of problematic L2 cache evictions. Figure 16 shows the cumulative distribution function of the HPL performance. With the original page size of 2 MB, significant performance degradation is shown on the left side of the figure. After using 1 GB pages, the standard deviation of the execution time is reduced by 51.3%.

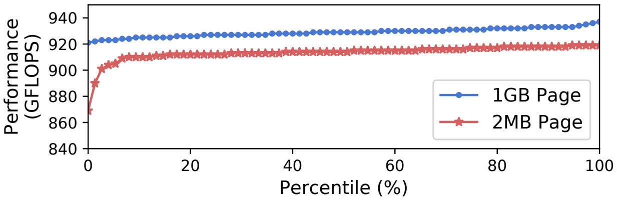

Figure 16. Distribution of HPL performance.

For this problem, which influences all programs on the problematic processors, Vapro provides an online detection and diagnosis approach based on comparing fixed-workload fragments from different processes. Since the inter-process comparison fails without the presupposition of fixed workload, other profiling tools, such as perf [17], cannot achieve it. vSensor fails in this case as well since it is a closed-source application provided by Intel. Vapro not only facilitates an early stop for the affected programs, but also avoids time-consuming re-executions for diagnosing this non-deterministic problem.

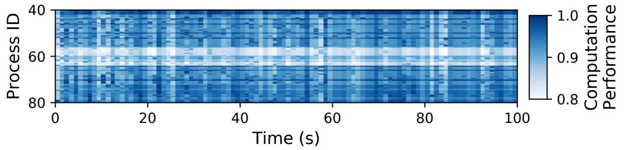

Figure 17. Detection results of computation results for 128-process Nekbone by Vapro.

6.5.2 Detection of Memory Problem. In this case, we execute the 128-process Nekbone [3], a computational fluid dynamics problem solver, on Tianhe-2A. As shown in Figure 17, Vapro locates processes on a node which is slower than others. By breaking down the variance, 97.2% of the slowdown is caused by backend bound, and nearly all of it is contributed by the memory bound. With memory tests, we find that the memory bandwidth of this node is 15.5% lower than others. By replacing this problematic node, it yields a speedup of 1.24×. We have reported this finding to the system administrator.

One could argue that this variance can be detected by benchmarks in advance, but as shown in the previous case and Figure 1, variance happens even when programs are executed on the same nodes.

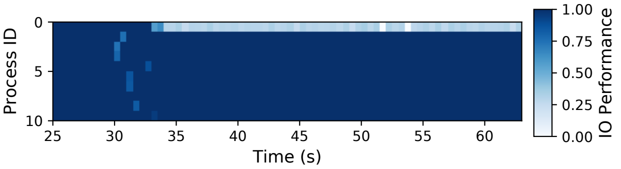

Figure 18. Detection results of IO performance for 512-process RAxML by Vapro.

6.5.3 Detection of IO Performance Variance. The third case study focuses on RAxML [46], a popular phylogenetic analysis application. We execute this application with 512 processes and observe a significant execution time variance, which ranges from 41.1 to 68.0 seconds for 10 consecutive executions. Vapro suggests that both computation and communication performance are stable. However, as shown in Figure 18, the IO performance variance is reported by Vapro

[4] The event name is CYCLE_ACTIVITY.STALLS_L2_MISS

for the first process, which has significantly lower performance than the others. Vapro identifies the most varied fixed-workload IO fragments and plots their execution time in Figure 19.

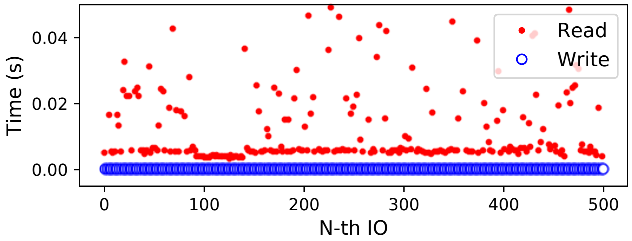

Figure 19. IO performance of consecutive read and write operations with fixed workload in RAxML.

Following this important hint, we further investigate RAxML and find that it merges data from multiple small files. Thus, its performance is vulnerable to the variance of the shared distributed file system. To reduce the distributed file system access, we implement a simple file buffer for these files. This optimization yields a 17.5% speedup and a 73.5% reduction in the standard deviation of overall execution time. In this case, although we cannot collect performance counters from the distributed file system due to security reasons, Vapro still efficiently filters out irrelevant factors to provide crucial hints for the solution.

7 Related Work

Detecting variance by general approaches There have been several general approaches for performance variance detection and diagnosis. Micro-benchmark is a classic approach to detect system variance [26, 32]. But running benchmarks is intrusive, it interferes with other applications and is not suitable for online production detection. The major drawback of tracing is its prohibitive data volume and performance overhead [29, 52, 56]. Program profiling [50] often discards time sequence information so it is difficult to detect performance variance in the time dimension. Although variance can be detected by performance modeling [38], building accurate models is extremely difficult.

Detecting variance caused by environment On detecting performance variance, an effective approach is *differentiation*, which means determining the processes or periods with different behavior. For works based on fragment-level fixed workload differentiation, vSensor [48], identifies such fixed-workload code snippets with static analysis to detect variance. However, relying on source code analysis, vSensor is impractical for proprietary programs and misses snippets with *de facto* fixed workload which cannot be determined at compilation. Shah [43] estimated the impact of external interference on bulk-synchronous MPI applications by comparing fixed-workload segments. However, neither of this work nor vSensor can diagnose the causes of variance. In contrast, Vapro diagnoses variance and provides crucial guides to solve it, which is not supported by these works.

Many research works are able to detect or diagnose the performance variance with differentiation in other methods. IASO [37] detects fail-slow, i.e., extremely severe performance variance, by monitoring the response time of requests. X-ray [8] leverages performance summarization and deterministic replay to locate basic-block-level causes of performance anomalies. UBL [18] predicts performance anomalies in the cloud by unsupervised learning. VarCatcher [57] detects and analyzes variance patterns by the parallel characteristics vector. Compared with these works, Vapro is able to locate, quantify and diagnose the variance online, which cannot be covered by a single one of the above works.

Detecting variance caused by applications Software bugs lead to performance variance as well. STAT [7] detects the root cause of the program hanging problem by finding out the processes with a different call-stack. AutomaDeD [31] uses a Markov model to find bugs by comparing their control-flow behavior history and finding the least-progress process. Su *et al.* [47] identifies several performance bugs by recording function-level variance. PerfScope [19] analyzes system call invocations to locate candidate buggy functions. Sahoo *et al.* [40] finds software bugs by monitoring program invariants. Other tools such as Hytrace [16] and PRODOMETER[35] focus on this topic as well. Orthogonal to these works, Vapro focuses on the diagnosis of performance variance caused by the external environment instead of functional and performance bugs inside applications.

8 Conclusion

We present Vapro, an online light-weight performance variance detection and diagnosis tool for production-run parallel applications. We combine a novel data structure to dynamically identify fixed workload for variance detection. Based on the variance breakdown model, Vapro diagnoses variance and reports the most possible reasons that the previous tools cannot realize.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work is supported by National Key R&D Program of China (2021YFB0300300), National Natural Science Foundation of China (U20A20226), Beijing Natural Science Foundation (4202031), SOAR fellowship, University of Sydney faculty startup funding, Australia Research Council (ARC) Discovery Project (DP210101984), China Postdoctoral Science Foundation (2020TQ0169), ShuiMu Tsinghua Scholar fellowship (2019SM131), Tsinghua University Initiative Scientific Research Program (20191080594). Jidong Zhai is the corresponding author of this paper.

References

[1] [n.d.]. *The cuBERT framework*. https://github.com/zhihu/cuBERT.

[2] [n.d.]. *The MapReduce framework*. https://github.com/sysprog21/mapreduce.

[3] [n.d.]. *The Nekbone program*. https://github.com/Nek5000/Nekbone.

[4] [n.d.]. *The parallel PageRank program*. https://github.com/nikos912000/parallel-pagerank.

[5] [n.d.]. stress. https://packages.debian.org/buster/stress

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. [n.d.]. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc.

[7] Dorian C Arnold, Dong H Ahn, BR De Supinski, Gregory Lee, BP Miller, and Martin Schulz. 2007. Stack trace analysis for large scale applications. In *21st IEEE International Parallel & Distributed Processing Symposium (IPDPS'07), Long Beach, CA*.

[8] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 307–320.

[9] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow. 1995. *The NAS Parallel Benchmarks 2.0.* NAS Systems Division, NASA Ames Research Center, Moffett Field, CA.

[10] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*. 242–253.

[11] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, and Susan Coghlan. 2006. The influence of operating systems on the performance of collective operations at extreme scale. In *2006 IEEE International Conference on Cluster Computing*. IEEE, 1–12.

[12] P. Berkhin. 2006. A Survey of Clustering Data Mining Techniques. In *Grouping Multidimensional Data: Recent Advances in Clustering*, Jacob Kogan, Charles Nicholas, and Marc Teboulle (Eds.). Springer.

[13] Andrew R Bernat and Barton P Miller. 2011. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. 9–16.

[14] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT'08)*. 72–81.

[15] Michael J Brim, Luiz DeRose, Barton P Miller, Ramya Olichandran, and Philip C Roth. 2010. MRNet: A scalable infrastructure for the development of parallel tools and applications. *Cray User Group* (2010).

[16] Ting Dai, Daniel Dean, Peipei Wang, Xiaohui Gu, and Shan Lu. 2018. Hytrace: a hybrid approach to performance bug diagnosis in production cloud infrastructures. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 30, 1 (2018), 107–118.

[17] Arnaldo Carvalho De Melo. 2010. The new linux perf tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[18] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. 2012. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th international conference on Autonomic computing*. 191–200.

[19] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. 2014. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*. 1–13.

[20] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.

[21] Donald E Farrar and Robert R Glauber. 1967. Multicollinearity in regression analysis: the problem revisited. *The Review of Economic and Statistics* (1967), 92–107.

[22] Kurt B Ferreira, Patrick G Bridges, Ron Brightwell, and Kevin T Pedretti. 2013. The impact of system design parameters on application noise sensitivity. *2010 IEEE International Conference on Cluster Computing* 16, 1 (2013), 117–129.

[23] Yifan Gong, Bingsheng He, and Dan Li. 2014. Finding constant from change: Revisiting network performance aware optimizations on iaas clouds. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 982–993.

[24] Jonathan Goodman, Albert G Greenberg, Neal Madras, and Peter March. 1988. Stability of binary exponential backoff. *Journal of the ACM (JACM)* 35, 3 (1988), 579–602.

[25] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. 2018. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)* 14, 3 (2018), 23.

[26] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. 1–11.

[27] Mary Inaba, Naoki Katoh, and Hiroshi Imai. 1994. Applications of weighted Voronoi diagrams and randomization to variance-based k-clustering. In *Proceedings of the tenth annual symposium on Computational geometry*. 332–339.

[28] Intel. 2018. *Addressing Potential DGEMM/HPL Perf Variability on 24-Core Intel Xeon Processor Scalable Family.* White paper, number 606269, revision 1.0.

[29] TR Jones, LB Brenner, and JM Fier. 2003. Impacts of operating systems on the scalability of parallel applications. *Lawrence Livermore National Laboratory, Tech. Rep. UCRL-MI-202629* (2003).

[30] JE Kay, C Deser, A Phillips, A Mai, C Hannay, G Strand, JM Arblaster, SC Bates, G Danabasoglu, J Edwards, et al. 2015. The Community Earth System Model (CESM) large ensemble project: A community resource for studying climate change in the presence of internal climate variability. *Bulletin of the American Meteorological Society* 96, 8 (2015), 1333–1349.

[31] Ignacio Laguna, Dong H Ahn, Bronis R de Supinski, Saurabh Bagchi, and Todd Gamblin. 2015. Diagnosis of Performance Faults in LargeScale MPI Applications via Probabilistic Progress-Dependence Inference. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 26, 5 (2015), 1280–1289.

[32] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 409–425.

[33] John. McCalpin. 2018. Memory Bandwidth: STREAM Benchmark Performance Results. https://www.cs.virginia.edu/stream/

[34] John D McCalpin. 2018. HPL and DGEMM performance variability on the Xeon Platinum 8160 processor. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 225–237.

[35] Subrata Mitra, Ignacio Laguna, Dong H Ahn, Saurabh Bagchi, Martin Schulz, and Todd Gamblin. 2014. Accurate application progress analysis for large-scale parallel debugging. In *ACM SIGPLAN Notices (PLDI'14)*, Vol. 49. ACM, 193–203.

[36] Oscar H Mondragon, Patrick G Bridges, Scott Levy, Kurt B Ferreira, and Patrick Widener. 2016. Understanding performance interference in next-generation HPC systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 384–395.

[37] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S Gunawi. 2019. IASO: a fail-slow detection and mitigation framework for distributed storage services. In *2019*

USENIX Annual Technical Conference (USENIX ATC'19). 47–62.

[38] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. 2003. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE, 55–55.

[39] Andrew Rosenberg and Julia Hirschberg. 2007. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*. 410–420.

[40] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. 2013. Using likely invariants for automated software fault localization. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. 139–152.

[41] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 460–471.

[42] Malte Schwarzkopf, Derek G Murray, and Steven Hand. 2012. The seven deadly sins of cloud computing research. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'12)*.

[43] Aamer Shah, Matthias Müller, and Felix Wolf. 2018. Estimating the impact of external interference on application performance. In *European Conference on Parallel Processing*. Springer, 46–58.

[44] Timothy Sherwood, Erez Perelman, and Brad Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*. IEEE, 3–14.

[45] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase tracking and prediction. In *ACM SIGARCH Computer Architecture News*, Vol. 31. ACM, 336–349.

[46] Alexandros Stamatakis. 2006. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics* 22, 21 (2006), 2688–2690.

[47] Pengfei Su, Shuyin Jiao, Milind Chabbi, and Xu Liu. 2019. Pinpointing performance inefficiencies via lightweight variance profiling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*. 1–19.

[48] Xiongchao Tang, Jidong Zhai, Xuehai Qian, Bingsheng He, Wei Xue, and Wenguang Chen. 2018. vSensor: leveraging fixed-workload snippets of programs for performance variance detection. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP'18)*. 124–136.

[49] Jeffrey Vetter. 2002. Dynamic statistical profiling of communication activity in distributed applications. *ACM SIGMETRICS Performance Evaluation Review* 30, 1 (2002), 240–250.

[50] Jeffrey Vetter and Chris Chambreau. 2005. mpip: Lightweight, scalable mpi profiling. (2005).

[51] Vincent M Weaver, Dan Terpstra, and Shirley Moore. 2013. Non-determinism and overcount on modern hardware performance counter implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. IEEE, 215–224.

[52] Brian J. N. Wylie, Markus Geimer, and Felix Wolf. 2008. Performance measurement and analysis of large-scale parallel applications on leadership computing systems. *Scientific programming* 16, 2-3 (April 2008), 167–181.

[53] Ulrike Meier Yang et al. 2002. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics* 41, 1 (2002), 155–177.

[54] Ahmad Yasin. 2014. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. IEEE, 35–44.

[55] Teng Yu, Wenlai Zhao, Pan Liu, Vladimir Janjic, Xiaohan Yan, Shicai Wang, Haohuan Fu, Guangwen Yang, and John Thomson. 2019. Large-Scale Automatic K-Means Clustering for Heterogeneous Many-Core Supercomputer. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2019).

[56] Jidong Zhai, Jianfei Hu, Xiongchao Tang, Xiaosong Ma, and Wenguang Chen. 2014. Cypress: Combining static and dynamic analysis for top-down communication trace compression. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 143–153.

[57] Weihua Zhang, Xiaofeng Ji, Bo Song, Shiqiang Yu, Haibo Chen, Tao Li, Pen-Chung Yew, and Wenyun Zhao. 2016. Varcatcher: A framework for tackling performance variability of parallel workloads on multi-core. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 28, 4 (2016), 1215–1228.