vSensor: Leveraging Fixed-Workload Snippets of Programs for Performance Variance Detection

Xiongchao Tang
Tsinghua University
txc13@mails.tsinghua.edu.cn

Jidong Zhai
Tsinghua University
zhaijidong@tsinghua.edu.cn

Xuehai Qian
University of Southern California
xuehai.qian@usc.edu

Bingsheng He
National University of Singapore
hebs@comp.nus.edu.sg

Wei Xue
Tsinghua University
xuewei@tsinghua.edu.cn

Wenguang Chen
Tsinghua University
cwg@tsinghua.edu.cn

Abstract

Performance variance becomes increasingly challenging on current large-scale HPC systems. Even using a fixed number of computing nodes, the execution time of several runs can vary significantly. Many parallel programs executing on supercomputers suffer from such variance. Performance variance not only causes unpredictable performance requirement violations, but also makes it unintuitive to understand the program behavior. Despite prior efforts, efficient on-line detection of performance variance remains an open problem.

In this paper, we propose vSensor, a novel approach for light-weight and on-line performance variance detection. The key insight is that, instead of solely relying on an external detector, the source code of a program itself could reveal the runtime performance characteristics. Specifically, many parallel programs contain code snippets that are executed repeatedly with an invariant quantity of work. Based on this observation, we use compiler techniques to automatically identify these fixed-workload snippets and use them as performance variance sensors (v-sensors) that enable effective detection. We evaluate vSensor with a variety of parallel programs on the Tianhe-2 system. Results show that vSensor can effectively detect performance variance on HPC systems. The performance overhead is smaller than 4% with up to 16,384 processes. In particular, with vSensor, we found a bad node with slow memory that slowed a program's performance by 21%. As a showcase, we also detected a severe network performance problem that caused a 3.37× slowdown for an HPC kernel program on the Tianhe-2 system.

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178497

CCS Concepts • **Computing methodologies** → **Parallel computing methodologies**; • **Computer systems organization** → *Availability*; • **Software and its engineering** → *Software maintenance tools*;

Keywords Performance Variance; Anomaly Detection; Compiler Analysis; System Noise

ACM Reference Format:
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 PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Vienna, Austria, February 24–28, 2018 (PPoPP '18),* 13 pages.
https://doi.org/10.1145/3178487.3178497

1 Introduction

Performance variance is a serious problem on current High-Performance Computing (HPC) systems [9, 21, 23]. The execution times of different runs for the same program on current large-scale HPC systems could vary significantly due to the performance variance of a system [34]. This phenomenon is quite common on large-scale super-computing centers. Figure 1 shows an example of performance variance on a real HPC system. A program (NPB-FT, CLASS=D) with 1024 processes is repeatedly executed on a fixed set of nodes of Tianhe-2 supercomputer [3]. Figure 1 shows a contiguous piece of a much longer completed log. The background noise was probably caused by the system itself or by other jobs. We find that the performance variance among different runs is severe, — the maximum execution time is more than *three times* of the minimum.

The prevalent performance variance can impact both normal system users and program developers in negative ways. For normal users, it may cause unpredictable performance for a running program, leading to more performance requirement violations and more resource consumption. Moreover, measuring and comparing the performance of different programs become more difficult with unstable performance. For program developers, the benefit of a new optimization can be hidden by the background performance variance. Based

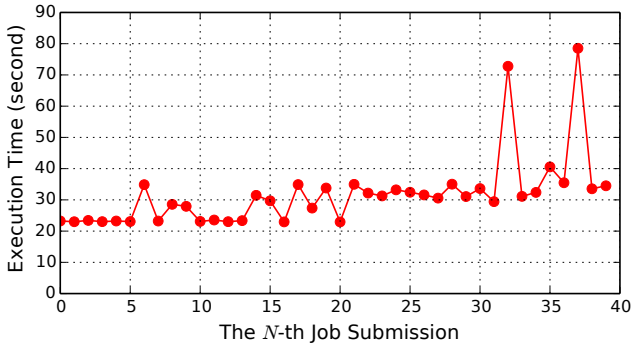

Figure 1. Performance variance on fixed nodes. The execution time of NPB-FT with 1024 processes on fixed nodes of Tianhe-2 system varies considerably among different runs.

on previous studies [12, 21, 26], performance variance is due to a number of reasons, e.g., network contention, OS schedule, zombie processes, hardware faults, etc. Depending on the cause, certain kinds of performance variance can be effectively avoided while others are inevitable. For example, if the performance variance is caused by a bad node, users can replace it with a good one and then resubmit the job. On the contrary, users have few options to avoid variance due to network contention, since the network is normally shared by many users. Therefore, before blaming the system or resubmitting a job, we need to understand two main questions: 1) the amount of the performance variance and 2) its root cause.

So far, there have been four major approaches to detect and handle performance variance but none of them are sufficient in answering the two above-mentioned questions. *1) Rerun.* A straight-forward method to detect performance variance is to run a program for multiple times and compare the execution times of different runs. Obviously, this approach is both time and system resource-consuming for long-running programs. *2) Performance Model* [11, 23]. An accurate performance model can predict the execution time of a program, based on which performance variance can be estimated by comparing predicted performance and measured performance. Unfortunately, most performance models can only predict the amount of overall performance variance, but cannot identify *where* such variance comes from. Moreover, a performance model is not portable: a model built for one application may not achieve good prediction results for a different application. *3) Profiling and Tracing* [14, 28]. While widely used, they bring key drawbacks: profiling-based methods cannot detect the performance variance in the time dimension because time sequence information is omitted; trace-based methods usually generate large volume of traces, especially as applications scale up in problem size and job scale [36]. Moreover, the tracing overhead prevents its application in on-line variance detection. Even with trace compression [10, 39], knowledge of applications is required to analyze trace data, which is quite difficult for non-expert users. *4) Benchmark* [12]. Performance variance can be detected with fixed-work quanta (FWQ) benchmarks. When a fixed quantity of work is executed repeatedly, a performance variance is detected if execution time for the same work changes. For example, repeatedly performing and observing the same communication can detect the performance variance of network. The key problem is that this approach is *intrusive*: due to the resource contention among the benchmarks and the original program, they can introduce *extra* performance variance. Therefore, it is not suitable for production runs.

Despite the significant prior work on detecting performance variance and identifying the root cause for different variances, effective on-line variance detection for a large-scale parallel program is still an **open problem**.

To overcome the drawbacks of existing approaches, this paper proposes vSensor, a novel approach for light-weight and on-line performance variance detection. The key insight is that many parallel programs contain code snippets with the same behavior of FWQ benchmarks. For example, some code snippets inside a loop have the same quantity of work among different iterations. Those *fixed-workload* snippets can be considered as *FWQ benchmarks embedded in a program*. We call such a snippet a **v-sensor**, which can sense performance variance at runtime *without* introducing additional performance overhead. Leveraging v-sensors inside programs to detect performance variance brings three benefits. *1)* No performance model is required, which by itself can be quite complex for a real application. *2)* Low overhead and interference. vSensor requires no external programs launched during the program execution, which avoids the performance overhead for monitoring daemon or resource contention due to external benchmarks. *3)* v-sensors makes it easier to locate the root causes of performance variance.

However, to realize the idea of v-sensors in a real performance variance detection tool, we face **two challenges**. *1) v-sensors identification.* It is non-trivial to pinpoint v-sensors from a large amount of source code, e.g., branches may take different paths, functions may be invoked with different arguments, and importantly, many snippets may have different workloads over iterations. Although developers have the best understanding of program semantics and can potentially annotate the fixed-workload snippets as v-sensors, doing it manually is not realistic for complex programs. *2) minimizing overhead for on-line detection.* While v-sensors are parts of original programs, additional instrumentation and analysis are still required to detect performance variance. The overhead must be small enough to avoid slowing down production run or introducing additional variance.

To overcome the challenges, we adopt a hybrid approach with a combination of static and dynamic analysis. At compile-time, to automate v-sensors identification, we propose a dependency propagation algorithm to analyze a program

then generate and select appropriate v-sensors for instrumentation. At runtime, we propose a lightweight on-line analysis algorithm to detect performance variance based on the instrumented v-sensors. We leverage the intrinsic properties of v-sensors to detect performance variance through comparing the current performance of v-sensors with history. For a large-scale parallel program, performance variance across processes is detected by comparing the performance of the same v-sensor on different processes.

To evaluate the effectiveness of the proposed approach, we implement vSensor as a complete tool chain with LLVM compiler [17]. It currently supports MPI [1] parallel programs, which are widely used in the HPC domain. We evaluate vSensor on the Tianhe-2 system with *up to 16,384 MPI processes*. Experimental results show that vSensor can identify v-sensors accurately, and the on-line variance detection method only introduces less than 4% performance overhead. Moreover, the effectiveness of vSensor is demonstrated with *real usage cases*: it identified a bad node in the Tianhe-2, which slowed the performance of the program CG by 21%. It also detected a severe network performance problem that caused a 3.37× slowdown for the program FT.

The remainder of this paper is organized as follows. Section 2 gives a high-level architecture and workflow of our approach. Section 3 describes the method of searching for v-sensors at compile time. Section 4 discusses the rules of choosing v-sensors for instrumentation. Section 5 presents our on-line detection algorithm. Section 6 shows the experimental results. Section 7 discusses related work. Section 8 concludes the paper.

2 vSensor Overview

The core idea of vSensor is to identify fixed-workload snippets, i.e., v-sensors, in programs, then analyze their actual execution time at runtime. Due to the fixed workload of a v-sensor, its time variance must be caused by performance variance, instead of workload variance. With various v-sensors inside a program, we can estimate the performance variance during the program execution through the time variance of v-sensors. vSensor is implemented as a complete tool chain for large-scale parallel programs on HPC systems.

vSensor consists of a static module and a dynamic one. The static module applies compiler techniques to automatically identify v-sensors and instrument programs. In current implementation, we use the LLVM [17] compiler to search for v-sensors. This process is done by analyzing LLVM-IR, thus programs of different languages (C/C++/Fortran) can be handled in a uniform manner. We notice that most HPC programs do not use LLVM, instead they use vendor compilers for special performance optimization. Therefore, vSensor maps v-sensors from LLVM-IR to source code, in which instrumentation is performed, allowing programs to use their original compilers.

The dynamic module collects and analyses performance data at runtime, giving a final performance report at the end. During program execution, the performance report is updated periodically, thus users can notice performance variance without waiting for a program to finish.

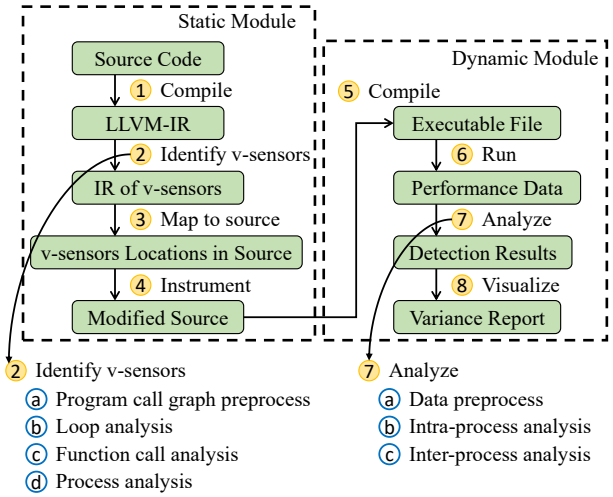

Figure 2. The Workflow of vSensor.

The detailed workflow of our approach is shown in Figure 2. We describe each step of the workflow as follows.

1. Compile. Compiling the source code of a program into intermediate representation (LLVM-IR) with LLVM.

2. Identify v-sensors. In LLVM-IR, a program is represented with basic blocks and instructions. Our v-sensors identification algorithm is performed on this level (Section 3). First, it processes the program call graph to handle special cases like recursive invocations. Then, loop analysis and function call analysis are performed. Since vSensor aims at multi-process parallel programs, we also analyze the behavior of v-sensors in different processes.

3. Map to source. The v-sensors obtained in step 2 are represented in LLVM-IR. In this step, we pinpoint v-sensors locations in source code.

4. Instrument. Source code instrumentation is performed based on the information in step 3. Several rules are used to guide the appropriate selection of v-sensors for instrumentation (Secton 4).

5. Compile. We use *original compilers* to compile the instrumented source code and generate the modified program, thus special compiler optimization options can be preserved.

6. Run. Run the modified programs on a real HPC system. With the customized instrumented functions, performance data will be generated at runtime.

7. Analyze. The performance data is preprocessed before being used for variance detection. For each process, an on-line algorithm detects performance variance by comparing current performance data with history (Section 5). The performance of the same v-sensors of different processes is used to detect variance across different processes.

8. Visualize. Finally, figures are produced to demonstrate the distribution of performance variance. Since new data is generated while a program is running, and the variance detection is performed on-line, figures can be updated periodically.

Step 2 (Identifying v-sensors) and Step 7 (Analyze) are the two most challenging steps in vSensor. We elaborate them in the following sections.

3 Identifying v-sensors

In this section, we concretely define v-sensors and then present v-sensor identification using our dependency propagation algorithm. Although the algorithm works on LLVM-IR, we explain the key ideas with C-style pseudo code for clarity. Since MPI [1] is widely used in HPC, we use MPI code in our examples, though our approach can be applied to other message passing parallel programs.

3.1 Definition of v-sensors

In our approach, v-sensors are used to denote performance benchmarks embedded inside a program. A v-sensor must be a snippet of code inside a loop, so that it can be executed repeatedly. There can be many snippets inside a loop. A **v-sensor** of a loop is defined as a snippet with fixed quantity of work over loop iterations. For example, snippet-2 in Figure 3 is a v-sensor of the `for` loop.

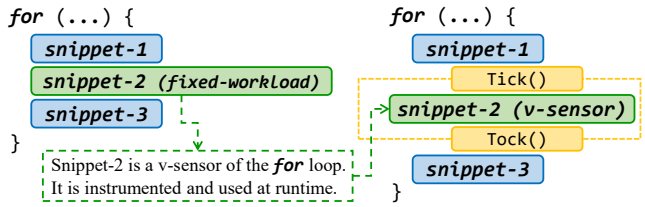

Figure 3. Example of v-sensors and snippets.

Based on the definition, a snippet can be as small as a statement, but the granularity of v-sensors has to be carefully chosen to ensure an acceptable instrumentation overhead for on-line detection. In vSensor, only loops and function calls are considered as v-sensor candidates.

```
int main() {                          int GLBV = 40;
  int n, k, value = 0;                int foo(int x, int y) {
L1 for (n=0; n<100; ++n) {              int i, j, value = 0;
L2  for (k=0; k<10; ++k) {            L4 for (i=0; i<x; ++i) {
C1    foo(n, k);                          value += y;
C2    foo(k, n);                      L5  for (j=0; j<10; ++j)
    }                                       value -= 1;
L3  for (k=0; k<10; ++k)                }
      count ++;                         if (x > GLBV)
C3  MPI_Barrier(MPI_COMM_WORLD);          value -= x*y;
  }                                     return value;
}                                     }
```

Figure 4. Example code illustrating vSensor ideas.

We use the sample code in Figure 4 to explain the ideas of vSensor. In Figure 4, loops and function calls are labeled with ID numbers. Global variables, loop variables, and function arguments are highlighted. In this example, Call-1 and Call-2 are v-sensor candidates of Loop-2. Loop-5 is a v-sensor candidate of Loop-4. However, the statement `count++` is not a v-sensor candidate of Loop-3 because it is not a loop or a call.

Next, we elaborate the notion of **quantity of work**. We classify code snippets into three types based on their purposes: **computation**, **network**, and **IO**. Loop-4 and Call-3 in Figure 4 are computation and network snippet, respectively. Each type of snippet has a different standard of fixed-workload. Even for snippets of the same type, the standard to measure the quantity of work depends on users expectation. In the following, we list our default decision rules, and vSensor allows users to add more constraints.

- **Computation.** If a computation snippet corresponds to the same sequence of instructions over iterations, then it is a computation v-sensor.
- **Network.** If the message size and message type of a communication operation do not change over iterations, then it is a network v-sensor.
- **IO.** If the input/output size of an IO operation is unchanged, then it is an IO v-sensor.

To ensure flexibility, vSensor allows users to specify other factors to be used in determining v-sensor. For example, the same instruction sequences with different cache misses could have different performance. Users can decide whether a constant cache miss rate should be used as an additional requirement for v-sensors. For network and IO snippets, there are more factors such as communication destination, network distance, IO frequency, etc. The factors can be either *static* rules, which can be known at compile time, or *dynamic* rules, which depend on runtime information.

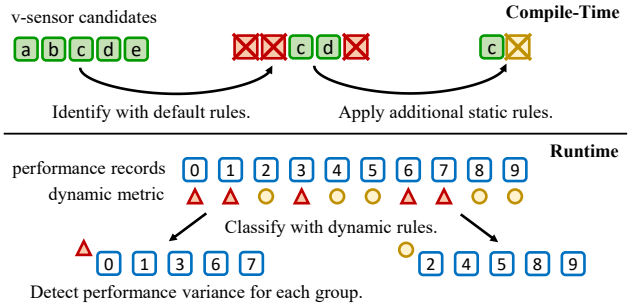

Figure 5. Static and Dynamic Rules.

Figure 5 shows how additional rules could affect v-sensor selection. At compile-time, static rules are used to identify v-sensors from candidates. Intuitively, more strict static rules produce less v-sensors. Dynamic rules allow further classification of v-sensors based on runtime information. As illustrated in Figure 5, at compile-time vSensor ignores all

dynamic rules and identifies v-sensors according to static rules; at runtime, v-sensors are further classified into two groups according to performance data of a v-sensor that can be only known during execution. With dynamic rules, it is possible to detect performance variance for each group.

For real-world MPI programs, network destination in MPI communications can be used in static rules since it is known at compile time. On the contrary, cache miss rate is a factor that needs to be considered in dynamic rules. At runtime, the performance records can be classified into different groups based on the range of cache miss rates, e.g., 0-10%, 10%-20%.

3.2 Intra-Procedural Analysis

In this section, we describe the intra-procedural analysis of snippets inside a procedure which has no function invocations. Since network and IO operations are performed by calling functions in most programs, all v-sensor candidates considered in this analysis are loops with pure computation. According to the rules described in Section 3.1, if the instruction sequence of a snippet is not changed over iterations, then the snippet has a fixed-workload. In other words, the quantity of work is determined by the control expressions of the loop and branch statements. If a candidate has control expressions that are affected by some variables between two executions, it is not a v-sensor. The dependency between variables is analyzed using a compiler technique – *use-define chain* analysis.

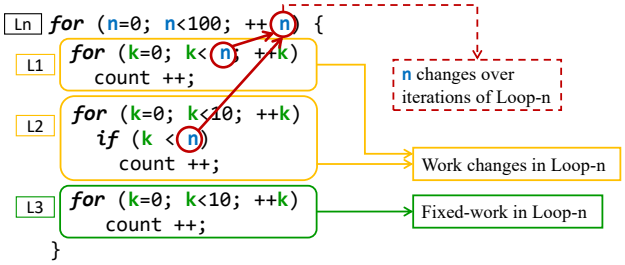

Figure 6. Intra-Procedural Analysis Example.

Figure 6 shows an example of pure-computation snippets. The outermost loop (Loop-n) has three subloops (Loop-1,2,3). We can see that the index variable n of the outermost loop is used in some control statements of Loop-1 and Loop-2. Since variable n changes between multiple executions of Loop-1 and Loop-2, these two subloops are not v-sensors of Loop-n. On the other hand, the control expression of Loop-3 is independent of n, so the work quantity of Loop-3 is invariant over iterations of Loop-n. Therefore, Loop-3 is a v-sensor of Loop-n.

Based on this intra-procedural analysis, we can identify two v-sensors in Figure 4: Loop-3 is a v-sensor of Loop-1, and Loop-5 is a v-sensor of Loop-4.

3.3 Inter-Procedural Analysis

Figure 7 explains the principles of our proposed inter-procedural analysis. Suppose there are three calls *C1*, *C2*, *C3* to function

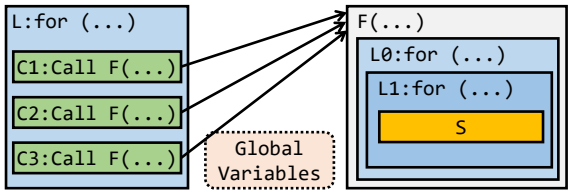

Figure 7. The Principle of Inter-Procedural Analysis.

F inside a loop *L*, and there is a snippet *S* in *F*. Snippet *S* is a v-sensor of *L* if the following two conditions are satisfied.

- Snippet *S* is a v-sensor for all its parent loops inside function *F*. This condition ensures that the work quantity of *S* does not change during an execution of *F*.
- If the work quantity of *S* is affected by some arguments of *F* or global variables, their values must not change over iterations of *L*, for all invocations of *F* inside *L* (*C1*, *C2*, *C3*). Therefore, the work quantity of *S* is the same in different invocations of *F*.

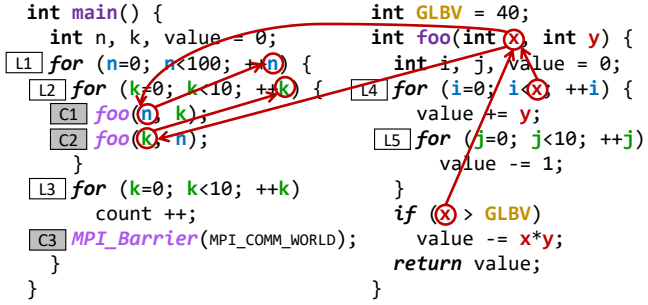

Figure 8. Inter-Procedural Analysis Example.

We use the sample code in Figure 8 as a concrete example of our inter-procedural dependency propagation analysis. In this example we analyze whether a function call is a v-sensor. An inter-procedural dependency chain is shown in Figure 8. For each function, we need to analyze the relationship between its arguments and workload. In Figure 8, the the workload of function `foo` is determined by its argument `x` and a global variable `GLBV`. If the values of `x` and `GLBV` do not change between invocations, then function `foo` have the same quantity of work. We can see that Call-1 is a v-sensor of Loop-2, because the changing value of `k` does not affect any control statements of function `foo`. However, since the value of `n` changes over iterations of Loop-1, Call-1 is not a v-sensor of Loop-1. Similarly, we can see that Call-2 is not a v-sensor of Loop-1 or Loop-2.

Next, we consider Loop-5 in Figure 8, which is a v-sensor of Loop-4. Furthermore, because it does not depend on any function arguments or global variables, Loop-5 is also a v-sensor of Loop-1 and Loop-2. According to our definition, Loop-4 is not a v-sensor of Loop-2 because the argument `x` will change between different invocations of Call-2.

3.4 Analysis for Multiple Processes

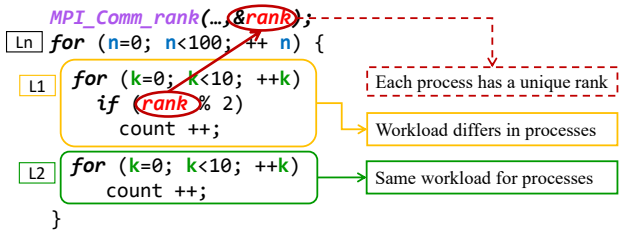

Figure 9. Analysis for Different Processes.

For parallel programs running with multiple processes (e.g., MPI programs), it is necessary to analyze the workload for different processes. Figure 9 shows an example for which the workload of a snippet is invariant over iterations but differs between processes. The function `MPI_Comm_rank` gives a unique rank (i.e., ID) to each process. For Loop-1, the workload is affected by the process rank. Although Loop-1's workload is fixed over iterations with a given rank, it is not the same for different processes. In particular, processes with odd ranks calculate the `count++` statement while others with even ranks do not. In vSensor, snippets with fixed-workload across processes will be used for runtime inter-processes variance detection.

We use a similar mechanism to analysis the dependency between workload and process ID. Firstly functions that generate process identifications, e.g., `MPI_Comm_rank, gethostname`, are specially handled. Then we can analyze whether the quantity of work is related to the process ID variables, e.g., rank and host names. If workload is not affected by process ID variables, then a snippet is considered fixed-workload across processes.

3.5 Analysis for a Whole Program

To efficiently analyze each procedure, we make a topological sort on a program call graph to determine the analysis order. In essence, we perform a bottom-up analysis over the program call graph. A callee is analyzed before its callers so that we can efficiently propagate the information of the callee to its callers and perform inter-procedural v-sensor detection.

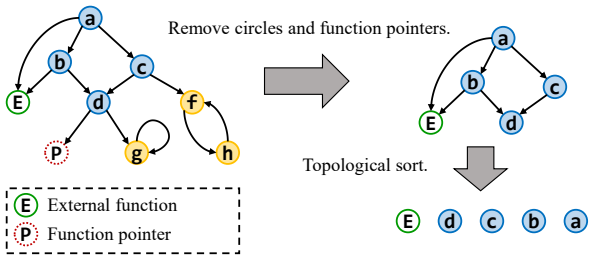

Figure 10. Whole Program Analysis.

Several special cases need to be handled properly. Recursive invocations generate cycles in a program call graph, which prevent a topological sort. Some programs have function pointers, whose call targets are difficult to identify at compile time. So recursive invocations and invocations to function pointers are removed from a program call graph. This process is shown in Figure 10.

Many programs have invocations of external functions whose source code is not available. For example, `printf`, `fopen`, and MPI functions are used in many programs. Without the source code, we cannot analyze their behavior at compile-time. We use a conservative strategy by default. An external function without any user-defined description will be treated as *never-fixed workload*. It means that all snippets containing calls to those never-fixed-workload functions are never considered as v-sensors. This strategy may miss some potential v-sensors, but it avoids false positives, which is more harmful. Optionally, users can describe the behavior of external functions, i.e., what arguments affect the workload. vSensor provides default descriptions for common functions in Lib-C and MPI library.

4 Instrumentation

With a set of v-sensors identified in Section 3, we next discuss how to select appropriate v-sensors for instrumentation. Our discussion is based on the example in Figure 11.

- **Scope.** The scope indicates the loops that a snippet is considered as a v-sensor. For example, there are two loops (L1, L2) and two v-sensors (S1, S2) in Figure 11. S2 has a bigger scope (L1) than S1 (L2). The workload of S1 is fixed over iterations of L2, but is not fixed over iterations of L1. In each iteration of L1, v-sensor S1 cannot use the data of previous L1 iterations, since the data is no longer valid. As a result, the data collected by a v-sensor with a bigger scope are more durable, thus can be used to detect performance variance in a longer period. If a snippet is a v-sensor in an out-most loop, then it has a whole program scope, or *global scope*. We call those snippets *global v-sensors*. In current vSensor implementation, only global v-sensors will be chosen for instrumentation.

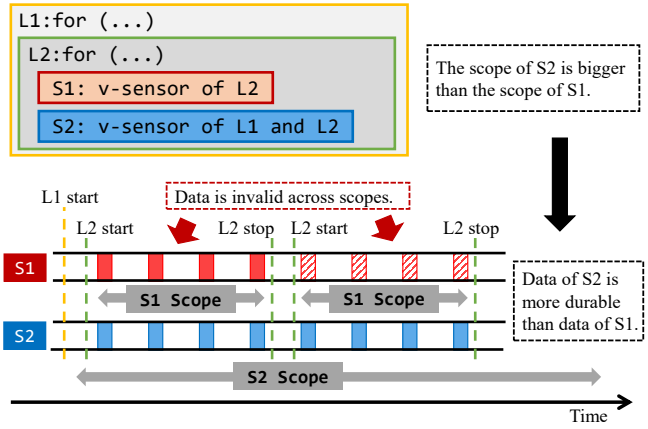

Figure 11. Scope of v-sensors.

- **Granularity.** We need to consider the trade-off between detection capability and overhead. Small snippets can detect fine-grained performance variance, but bring large overhead. On the contrary, big snippets may miss high-frequency performance variance. In vSensor implementation, we allow users to specify a *max-depth* parameter. An out-most loop is *depth-0*, and its direct subloops are *depth-1*, and so on. Only the v-sensors with a depth less than *max-depth* will be chosen for instrumentation. However, this compile-time strategy is only an estimation, the actual execution time of a v-sensor is determined at runtime. Therefore, we also apply runtime optimizations to reduce the overhead brought by fine-grained v-sensors (see details in Section 5).
- **Nested v-sensors.** For nested v-sensors, if the outside one is selected for instrumentation, inside v-sensors will not be used, and vice versa. This is because our instrumentation functions themselves are not fixed-workload snippets. If we instrument v-sensors inside, then the outside v-sensor will contain our instrumentation function, thus is no longer a v-sensor. In our implementation, we prefer to choose the outermost v-sensors for instrumentation.

To measure execution time of v-sensors, customized functions are instrumented at compile time (Function `Tick` and `Tock` in Figure 3) before and after each v-sensor. The performance variance detection algorithm is triggered inside these Tick/Tock functions.

5 Runtime Detection

In this section, we describe how the performance data generated from instrumented v-sensors are used for performance variance detection.

5.1 Data Smoothing

Most computer systems have high frequency but short duration noise usually caused by OS interruptions. The system noise is typically periodical and is inevitable for common users since it comes from the kernel [12]. In this work we regard system noise as a characteristic of systems rather than performance variance. vSensor focuses on more durable and repairable performance variance. However, some v-sensors have very short execution time so they will be affected by system noise then generate false alerts of performance variance. To avoid false positive results, we aggregate and average performance data during a small time slice (1000us by default) to smooth data.

Figure 12 shows background noise under different time resolutions. A v-sensor with approximately 10us workload is executed repeatedly on Tianhe-2 system, and we record the wall time for each execution. With a high resolution (10us), its performance data appear to be chaotic. However, if we plot the average of longer intervals (1000 us), the curve becomes smoother. The smoothed data can filter out a lot of

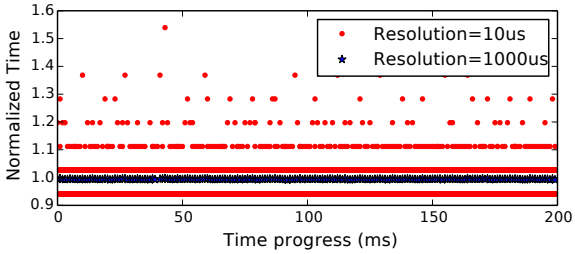

Figure 12. Filtering out background noise.

background system noise and let us focus on more durable and more severe performance variance. Also, data analysis algorithm is triggered only once for a time slice, this reduces the overhead introduced by fine-grained v-sensors.

5.2 Performance Normalization

In Subsection 3.1 we mention that v-sensors have types of Computation, Network, or IO. The type of the v-sensor helps us to locate the root cause of the detected performance variance. Actually, different v-sensors of the same type represent the performance of the same system component, so their performance data can be merged to improve detection accuracy. For example, there are 10 network v-sensors each executes per 1000us, then we can analyze their performance per 1000us. After data merging, we can analyze the network performance per 100us, thus it is easier to catch a network performance variance. Since different v-sensors have different workloads, we cannot compare their execution time directly, instead we use *normalized performance*. Each v-sensors compare their records to the fastest record. The fastest record will be normalized to 1.00, and a record with double execution time has a normalized performance of 0.50. The normalized performance of v-sensors with the same type represents the performance of a certain component of a system. Low normalized performance indicates that a component has performance degradation.

5.3 Comparing with History

To ensure low storage overhead of analysis, vSensor records the historical performance data (e.g., wall-time) and detects variance by comparing current performance data with history. For a given v-sensor, its work quantity should never change. Instead of saving a long list, only a scalar value of *standard time* needs to be saved for each v-sensor. As discussed in Subsection 5.1, there is a record for each time slice, which represents the average execution time during the past time slice. The standard time of each v-sensor is dynamically updated to the execution time of the fastest record. Besides the elapsed time, more metrics such as cache miss and memory access can be obtained through processors' performance monitor unit (PMU [22]).

To further reduce the overhead due to fine-grained v-sensors, vSensor will turn off the analysis for v-sensors

that are too short at runtime. In other words, the Tick/Tock functions wrapped those v-sensors will not trigger analysis.

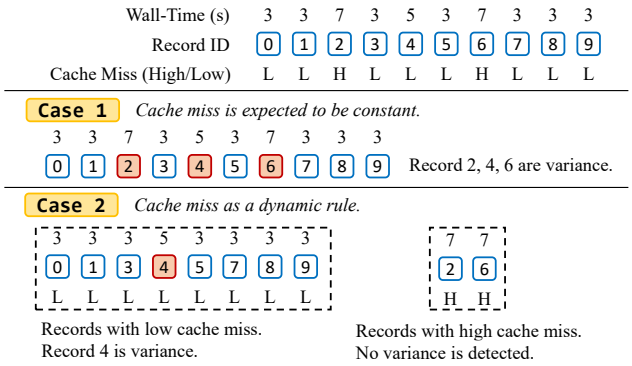

Figure 13. Online Detection Example.

Figure 13 shows an example of online detection. A v-sensor is executed ten times, and we record its wall-time and cache miss rate. For simplicity, cache miss rate has two values: high and low. If cache miss rate is expected to be constant, then records 2,4,6 will be detected as variance since their execution time is much longer than other records. However, when cache miss rate is used as a dynamic rule, the records can be clustered into two groups, for high and low cache miss, respectively. After this refinement, record 4 is a variance in low-cache-miss group, and no variance is detected in high-cache-miss group.

5.4 Analysis for Multiple Processes

In our implementation, vSensor uses a dedicated process (called *analysis-server*) for inter-process analysis. It collects performance data from all processes to detect inter-process performance variance by comparing the performance of the same v-sensor on different processes. This can be done by processes sending messages to analysis-server or by updating shared files. Experimental results in Section 6 show that the data transferred to server is quite small and will not cause severe network or IO congestion. To reduce overhead, each process buffers its data locally and periodically transfers them in batch to analysis-server. Instead of sending many small messages, it generates smaller number of network-friendly batched messages.

5.5 Reporting Performance Variance

A visualizer is used to make performance detection results easier for developers to understand. In our implementation, a performance matrix can be generated for each type and represent the performance of each component (Computation, Network, IO). Figure 14 is a visualized performance matrix generated by vSensor with v-sensors of computation type. It shows the computational performance of 128 processes during 100 seconds.

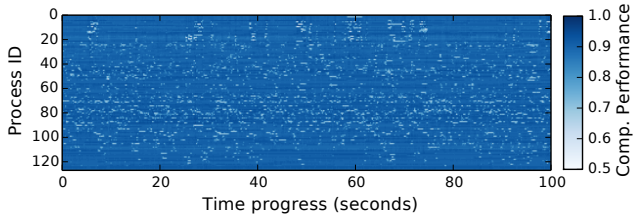

Figure 14. An example of performance matrix.

The horizontal axis represents time, with a resolution of 200ms. The vertical axis represents the MPI rank, i.e., process ID. We use different colors to represent the relative performance, deep blue means the best performance and white means the performance is only half of the best. As a result, performance variance will be shown as white blocks in a performance matrix figure. Although there are scattered white dots in Figure 14, the whole program has a good performance in total. We will see figures for severe performance variance from real cases in Section 6.

The position of white blocks indicates the time and location of performance variance, and the v-sensor type tells which component caused the variance. For example, if vSensor reports that some processes are slower than others, that may imply potentially buggy hardware in those nodes. vSensor is designed to detect performance variance with low performance overhead and low manual intervention. After vSensor identifies the source of performance variance and points out the corresponding time, processes, and component in a coarse-grain fashion; it is the users' choice to repair the system, or to resubmit the job, or to launch more accurate but larger overhead diagnosis tools.

6 Evaluation

6.1 Methodology

We evaluate vSensor with eight typical HPC programs: five benchmark programs (BT, CG, FT, LU, and SP from NPB benchmark suite [6]) and three applications (LULESH [15], AMG [37], and RAXML [24]). We perform all experiments on the Tianhe-2 supercomputer, with up to 16,384 processes. Each node of the Tianhe-2 has two Xeon E5-2692(v2) processors (24 cores in total) and 64GB memory. The Tianhe-2 uses a customized high-speed interconnection network.

The current implementation uses LLVM-3.5.0 for v-sensors identification; and uses Clang-3.5.0, Dragonegg-3.5.0, and ROSE for source code instrumentation. The framework of vSensor and the visualizer are built on Python scripts. Although the current implementation does not yet include all features described in previous sections, it can detect various forms of performance variance, as we will see in the following subsections.

In the experiments, we firstly provide a detailed analysis of the identified v-sensors with vSensor, validate their correctness, and show the performance overhead of on-line

Program	Code (KLoc)	Number of snippets	Number of v-sensors	Instrumentation number and type	Workload max error	Performance overhead	Sense-time coverage	Frequency (MHz)
BT	11.3	476	190	87Comp	4.78%	2.31%	87.08%	5.759
CG	2.0	83	25	7Comp+5Net	0.07%	2.37%	14.52%	0.107
FT	2.5	162	49	17Comp+3Net	3.91%	3.73%	42.64%	11.369
LU	7.7	328	168	83Comp	3.82%	2.08%	64.03%	0.484
SP	6.3	554	85	61Comp+6Net	3.76%	0.22%	45.32%	5.346
AMG	75.0	4695	555	143Comp+3Net	0.86%	1.62%	0.18%	0.004
LULESH	5.3	1401	333	21Comp+3Net	3.14%	0.21%	15.88%	1.197
RAXML	36.2	2742	677	277Comp+24Net	4.84%	3.46%	17.23%	7.077

Table 1. Results for vSensor validation (with 16,384 MPI processes and 15,625 processes for LULESH).

detection algorithms. Then, we discuss vSensor's ability to detect performance variance. Finally, we give several case studies using vSensor to detect real performance variance on the Tianhe-2.

6.2 Validation and Overhead

Table 1 lists several basic metrics of compile-time analysis and runtime detection. Columns on the left are compile time analysis results. Columns on the right are runtime results with 16,384 MPI processes (15,625 processes for LULESH).

The results includes the lines of source code, the number of snippets, the number of identified v-sensors, and the number and type of instrumented v-sensors. In our experiments, most of the instrumented v-sensors are of type Computation. As discussed in Section 3.1, loops and calls are candidate snippets of v-sensors. For example, AMG has 75K lines of source code and 4695 candidates. After compile-time analysis, 555 snippets in AMG are identified as v-sensors. Among them, only 146 v-sensors are finally selected to instrument for on-line detection. This result show that vSensor can filter out many snippets with not-fixed workload, thus avoid unnecessary instrumentation.

Next we validate the correctness of the identified v-sensors, i.e. to check if their workloads are really fixed. We validate the correctness of network v-sensors by recording their message sizes, and the experimental results show those arguments are unchanged. However, it is more complicated to validate the correctness of identified v-sensors, we use the hardware Performance Monitor Unit (PMU) to record the instruction counts of computation v-sensors and check whether the workload is fixed over iterations. For each v-sensor, we get a sequence of instruction count for each execution: $v_0, v_1, \ldots, v_n$. Let $P_s = MAX(v_i)/MIN(v_i)$. Theoretically, all values for a v-sensor should be the same and $P_s = 1$. Since PMU measurement is not perfectly accurate [32], P_s is approximately equal to 1 in our experiments. Let $P_a = MAX(P_s)$ denote the maximum difference of all v-sensors. And let $P_m = MAX(P_a)$ denote the maximum difference for all processes. We list the values of $P_m - 1$ in Column *Workload max error* of Table 1, which denotes the error of vSensor. We can see that the average error is less than 5%. Considering the PMU precision, these results indicate that our compile-time analysis algorithm works well.

The performance overhead is measured by comparing the execution time of the original with the instrumented programs. Since the Tianhe-2 has a significant performance variance, we run them repeatedly and use the shortest time for comparison. Table 1 also shows that the performance overhead for instrumented programs are less than 4%. It indicates that our instrumentation strategy is effective and the on-line detection algorithm is light-weight.

6.3 Distribution of v-sensors

vSensor's ability to detect performance variance highly depends on the distribution of v-sensors. Figure 15 illustrates key notions related to v-sensors distribution. The total execution time of a program is represented horizontally. A red color block means a v-sensor is executing, and we call it a *sense*. The length of a single red block is called the duration of a sense. *Sense-time* is the sum of all senses' duration and *interval* is the length between two senses. We define the *coverage* of v-sensors as the ratio between sense-time and total-time. The average *frequency* of v-sensors is defined as the ratio between sense-count and total-time.

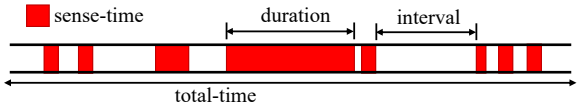

Figure 15. The distribution of v-sensors.

The right-most two columns in Table 1 list coverage and frequency for each program. We see that most programs have a coverage over 10% and frequency higher than 100KHz. For example, the sense frequency for CG is 107KHz, which means there is a sense for each 10us.

To analyze the duration and interval of senses, we cluster them into four groups according to their length: (1) less than 100us; (2) 100us to 10ms;(3) 1ms to 1s; (4) more than 1s. Figure 16 and Figure 17 show the length of duration and interval for each program.

We see that the duration time of most senses are shorter than 100us, none of them is longer than 1s. It means most

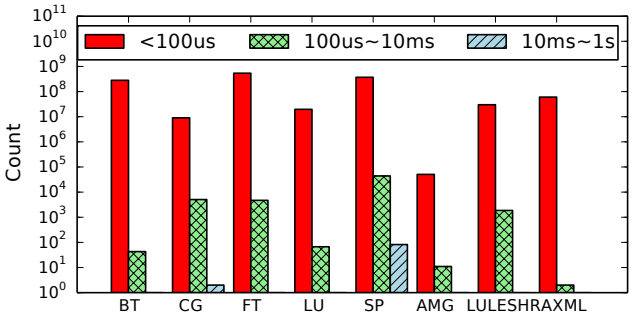

Figure 16. The duration of senses.

v-sensors are fine-grained snippets, which implies the necessity of aggregation (Section 5.1).

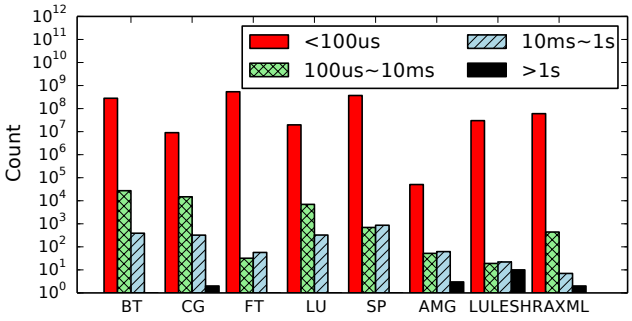

Figure 17. The interval between senses.

For most programs, intervals are shorter than 1s, thus vSensor would not miss performance variance longer than 1s. Some programs have several long intervals (longer than 1s). For LULESH, its long intervals are caused by a big non-fixed snippet in its main loop. However, there are enough v-sensors spanning across the whole program, so vSensor can still detect performance variance. On the contrary, v-sensors in AMG only appear in a portion of lifetime, and there is no v-sensor for almost half of lifetime. We can also see AMG's low coverage and frequency in Table 1. This behavior is due to the adaptive mesh refinement algorithm used in AMG, which leads to runtime workload changes, so there are only few fixed workload snippets. Nevertheless, many HPC programs have static work partition and vSensor works well for these programs.

6.4 Noise Injection

We use vSensor and a profiler (mpiP [31]) to study a manual noise injection example. The program is cg.D.128, with 128 MPI processes. Due to the difficulty of controlling the Tianhe-2 system, this injection experiment is done on a local cluster with dual Xeon E5-2670(v3) and 100Gbps 4×EDR Infiniband.

Figure 18 shows the mpiP profile result for a normal run without noise injection. All processes spend about 50 seconds

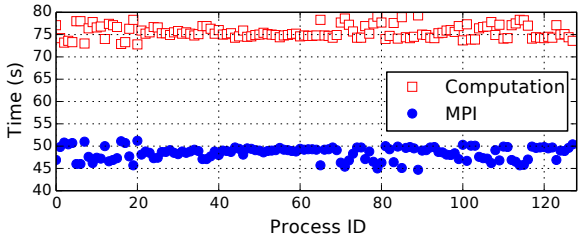

Figure 18. mpiP profile result for a normal run.

on MPI communication and about 75 seconds on computation. The slight time difference between processes may be due to different workloads or different node performance, which can not be distinguished with this profile result only. Then we re-run the program with noise injection. The noise injection is done as following. While the program is running, we launch another program (*noiser*) on some nodes, thus the program will compete with *noiser* for CPU and memory resource. We inject noise twice and it lasts for 10 seconds each time. With noise injection, the program becomes slower.

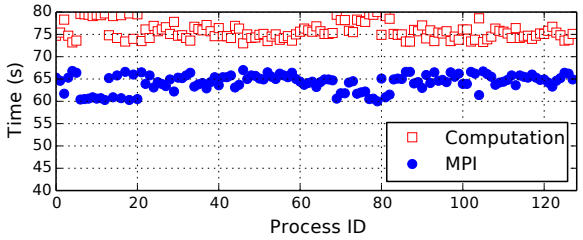

Figure 19. mpiP profile result for a noise-injected run.

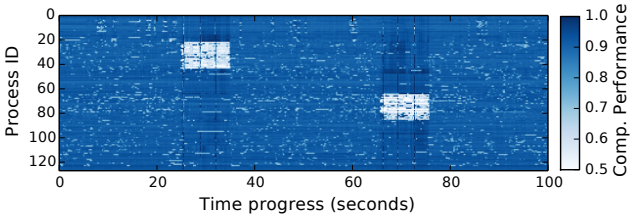

Figure 20. vSensor result for a noise-injected run.

Figure 19 is the mpiP profile result for the noise injected run. Comparing Figure 19 with Figure 18, the MPI time increases from about 50 seconds to about 65 seconds, while the computation time is almost the same. If we only look at Figure 19, we cannot tell whether the program is affected by performance variance. Even worse, we are mislead to suspect the network since the MPI communication becomes longer. After carefully reading the mpiP output and the source code of the program CG, we try to give an explanation for those counter-intuitive results: The injected workload is likely to be scheduled inside MPI function calls, since cpu cores are more idle during communication. As a result, the

computation is only stretched a little, but the communication on some processes is delayed a lot. Then the slowed processes cause communication waiting between processes, and this waiting time is calculated as MPI time by the mpiP profiler. Nevertheless, this is a non-trivial analysis especially for users who are not MPI experts.

While mpiP profile results fail to point out what and where the noise is injected, vSensor can detect and locate the injected noise. The computation performance matrix of a normal run has been shown in Figure 14. Injected noise is illustrated clearly as two white blocks in Figure 20. Figure 20 indicates that the noise is injected to process 24-47 at 34 second, and to process 72-96 at 66 second. As a conclusion, comparing with a profiler, vSensor has an advantage on detecting and locating performance variance.

We also use an MPI tracer (ITAC [2]) to collect trace for analysis. Despite the requirement of expert knowledge to analyze trace, ITAC generates much more data than vSensor (501.5 MB vs. 8.8 MB). Since trace-based analysis tools generate much more trace data, which can incur large overhead in terms of space and time, vSensor has better scalability than traditional tracing tools. In this example run, the data generation rate for each process is 0.5 KB/s (8.8 MB for 128 processes and 140 seconds). Based on this data generation rate, even 16,384 processes will only generate data at 8 MB/s, which is a small network bandwidth consumption.

6.5 Case Studies

Next, we share our experience of using vSensor to detect and locate performance variance on the Tianhe-2 system. Figure 21 shows the computation performance of CG running with 256 processes.

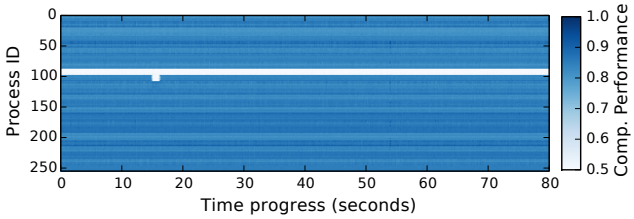

Figure 21. vSensor detection result for a CG run.

We notice that there is a white line near process 100, which means these processes suffer from low computation performance for the whole execution. vSensor reveals that these slow processes are all running on the same node, therefore, the problem is probably caused by a bad node. To confirm it, we run computation benchmarks to test the performance of CPU and memory on that node. Benchmark results show that, while the node has common CPU performance, one of its processor has low memory access performance, which is only 55% of other processors. We report this finding to the administrator and use other nodes to resubmit our job. After removing this bad node, the time of running CG reduces from 80.04s to 66.05s, — a 21% performance improvement.

One could argue that a performance test before launching programs (*pre-test*) could avoid this kind of performance variance. However, as shown in Figure 1, performance variance exists even if programs run on fixed nodes. In our experiments, we keep running FT using 1024 processes on a fixed set of nodes, a performance variance occurs in the middle of execution, which cannot be detected by pre-test.

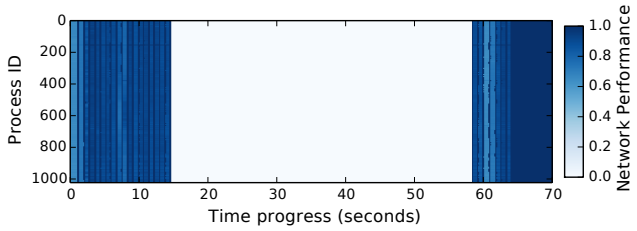

Figure 22. vSensor detection result for an FT run.

As shown in Figure 1, a normal run of FT takes 23.31s. However, an abnormal run may take up to 78.66s, which is 3.37× slower. The performance variance is clearly identified with vSensor, as illustrated by the network performance matrix in Figure 22. We can see that there is a clear network performance degradation in the period between 16s and 67s. After reading the source code of FT, we notice that FT calls `MPI_Alltoall` for exchanging data among processes. The function `MPI_Alltoall` is a complex communication operation that involves all processes, thus is vulnerable to network problems. We indeed found that network system in the Tianhe-2 has occasional performance issue, which is identified as the root cause of the performance variance we observed in FT. Unfortunately, this network performance degradation is probably caused by network congestion, and is difficult to avoid. In this situation, vSensor can timely detect the performance variance and notify users. Then it is users' choice to continue or re-submit the job.

7 Related Work

Performance variance is a major problem for long-running programs on HPC systems [21]. Skinner *et al.* [27] examined variance on large-scale systems and demonstrated performance gains by reducing variance. Hoefler *et al.* [12] quantify the impact of OS variance on large-scale parallel applications. They found that application performance is mostly determined by variance pattern rather than serial variance intensity. Tsafrir *et al.* [30] found that periodic OS clock interrupt was a major source of variance on general-purpose OS. Co-scheduling of operating system is suggested as a solution to avoid system variance [13].

Agarwal *et al.* [4] studied the impact of different performance anomaly distributions on the scalability of parallel applications. They found that a heavy-tailed or a Bernoulli

noise can cause significantly performance degradation. Ferreira *et al.* [9] used a kernel-based noise injection method to inject various levels of noise into applications running on a supercomputer and quantified the interference of noise on real applications. Beckman *et al.* [7] reported that synchronizing the system interrupts can largely reduce its influence. During the optimization of application NAMD on a large supercomputer, Phillips *et al.* [25] found that service daemon processes prevents using all processors on each node for useful computation effectively.

Models have already been used for performance variance detection. Petrini *et al.* [23] used analytic models to identify the source of a performance problem for SAGE running on ASCI Q machine. Their method can quantify the total impact of system noise on application performance. However, their method is both time- and resource-consuming. For more generally purpose, Lo *et al.* [19] build a toolkit for performance analysis on a variety of architectures based on roofline model [33]. Although Calotoiu *et al.* [8], Jae-Seung Yeom *et al.* [38], Lee *et al.* [18] and many other researchers made efforts to accelerate model building, it is still challenging to build an accurate performance model for a complex parallel application.

Jones *et al.* [14] utilized the postmortem analysis of detailed system traces to discover the root cause of the source of performance variance. However, the traces will become very huge as applications scale up in problem size and job scale [36]. Collecting huge volume of traces also introduces large interference with user applications [35, 39]. As a result, it is impractical to accurately identify system noise through trace-based methods on a large-scale system. Profilers [31] and statistical tracers [29] are proposed for less-accurate but light-weight analysis.

While vSensor in this paper focuses on the performance variance of systems, there are other performance tools such as STAT [5], AutomaDeD [16] and PRODOMETER [20] aim at performance faults or functional bugs of programs.

In summary, previous studies are not sufficient to perform light-weight on-line performance variance detection. vSensor leverages the intrinsic characteristics within applications and allows efficient variance detection with the combined static and dynamic techniques.

8 Conclusion

This paper proposes vSensor, a novel approach for light-weight and on-line performance variance detection. Instead of relying on an external detector, we show that the source code of a program itself could reveal the runtime performance characteristics. Specifically, many parallel programs contain snippets that are executed repeatedly with an invariant quantity of work. We use compiler techniques to automatically identify these fixed-workload snippets and use them as performance variance sensors (v-sensors) that enable effective detection. The evaluation results show that vSensor can effectively detect performance variance on the Tianhe-2 system. The performance overhead is less than 4% with up to 16,384 processes. In particular, with vSensor, we found a bad node with slow memory, which slowed a program's performance by 21%. We also detected a severe network performance problem that causes a 3.37× slowdown for a program.

Acknowledgment

We would like to thank our shepherd Prof. Bernhard Egger and the anonymous reviewers for their insightful comments and suggestions. We also thank Yifan Gong, Zhen Zheng, Yuyang Jin, and Bowen Yu for their valuable feedback and suggestions. This work is partially supported by National Key R&D Program of China (Grant No. 2016YFB0200100, 2016YFA0602100), National Natural Science Foundation of China (Grant No. 91530323, 41776010, 61722208, 61472201), Science and Technology Plan of Beijing Municipality (Grant No. Z161100000216147), Tsinghua University Initiative Scientific Research Program (20151080407). Xuehai's work is supported by NSF-CCF-1657333, NSF-CCF-1717754, NSF-CNS-1717984, NSF-CCF-1750656. Bingsheng's work is supported by an NUS startup grant in Singapore and a collaborative grant from Microsoft Research Asia. Jidong Zhai is the corresponding author of this paper (Email: zhaijidong@tsinghua.edu.cn).

References

[1] 2016. MPI Documents. (2016). http://mpi-forum.org/docs/
[2] 2017. Intel Trace Analyzer and Collector. (2017). https://software.intel.com/en-us/intel-trace-analyzer
[3] 2017. top500 website. (2017). http://top500.org/
[4] Saurabh Agarwal, Rahul Garg, and Nisheeth K Vishnoi. 2005. The impact of noise on the scaling of collectives: A theoretical approach. In *High Performance Computing–HiPC 2005*. Springer, 280–289.
[5] 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 (IPDPSâĂŹ07), Long Beach, CA*.
[6] 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.
[7] 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.
[8] Alexandru Calotoiu, David Beckinsale, Christopher W Earl, Torsten Hoefler, Ian Karlin, Martin Schulz, and Felix Wolf. 2016. Fast Multi-Parameter Performance Modeling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 172–181.
[9] 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.
[10] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*

22, 6 (2010), 702–719.

[11] 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.

[12] 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.

[13] Terry Jones. 2012. Linux kernel co-scheduling and bulk synchronous parallelism. *International Journal of High Performance Computing Applications* (2012), 1094342011433523.

[14] TR Jones, LB Brenner, and JM Fier. 2003. Impacts of operating systems on the scalability of parallel applications. *Lawrence Livermore National Laboratory, Technical Report* (2003).

[15] Ian Karlin, Abhinav. Bhatele, Bradford L.. Chamberlain, Jonathan. Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, Edward Luke, Scott Lloyd, Jim McGraw, Rob Neely, David Richards, Martin Schulz, Charle H. Still, Felix Wang, and Daniel Wong. 2012. *LULESH Programming Model and Performance Ports Overview*. Technical Report LLNL-TR-608824. 1–17 pages.

[16] 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* 26, 5 (2015), 1280–1289.

[17] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[18] Seyong Lee, Jeremy S Meredith, and Jeffrey S Vetter. 2015. Compass: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 405–414.

[19] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J Ligocki, Matthew J Cordery, Nicholas J Wright, Mary W Hall, and Leonid Oliker. 2014. Roofline model toolkit: A practical tool for architectural and program analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 129–148.

[20] 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*, Vol. 49. ACM, 193–203.

[21] Oscar H Mondragon, Patrick G Bridges, Scott Levy, Kurt B Ferreira, and Patrick Widener. 2016. Understanding performance interference in next-generation HPC systems. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 384–395.

[22] Philip Mucci, Jack Dongarra, Shirley Moore, Fengguang Song, Felix Wolf, and Rick Kufrin. 2004. Automating the Large-Scale Collection and Analysis of Performance Data on Linux Clusters1. In *Proceedings of the 5th LCI International Conference on Linux Clusters: The HPC Revolution*.

[23] 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 *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC'03)*. ACM.

[24] Wayne Pfeiffer and Alexandros Stamatakis. 2010. Hybrid MPI/Pthreads parallelization of the RAxML phylogenetics code. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 1–8.

[25] J. C. Phillips, Gengbin Zheng, S. Kumar, and L. V. Kale. [n. d.]. NAMD: Biomolecular Simulation on Thousands of Processors. In *Supercomputing, ACM/IEEE 2002 Conference*. 36–36.

[26] David Skinner and William Kramer. 2005. Understanding the causes of performance variability in HPC workloads. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*. IEEE, 137–149.

[27] David Skinner and William Kramer. 2005. Understanding the causes of performance variability in HPC workloads. In *Proceedings of the IEEE International Workload Characterization Symposium, 2005.* IEEE, 137–149.

[28] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. 2010. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11.

[29] Nathan R Tallent, John Mellor-Crummey, Michael Franco, Reed Landrum, and Laksono Adhianto. 2011. Scalable fine-grained call path tracing. In *Proceedings of the international conference on Supercomputing*. ACM, 63–74.

[30] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. 2005. System Noise, OS Clock Ticks, and Fine-grained Parallel Applications. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. ACM, New York, NY, USA, 303–312.

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

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

[33] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[34] Nicholas J Wright, Shava Smallen, Catherine Mills Olschanowsky, Jim Hayes, and Allan Snavely. 2009. Measuring and understanding variation in benchmark performance. In *DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2009*. IEEE, 438–443.

[35] Xing Wu and Frank Mueller. 2013. Elastic and scalable tracing and accurate replay of non-deterministic events. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 59–68.

[36] 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.

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

[38] Jae-Seung Yeom, Jayaraman J Thiagarajan, Abhinav Bhatele, Greg Bronevetsky, and Tzanio Kolev. 2016. Data-driven performance modeling of linear solvers for sparse matrices. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 32–42.

[39] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 143–153.