HPAS: An HPC Performance Anomaly Suite for Reproducing Performance Variations

Emre Ates
Boston University
Dept. of Electrical and Computer Eng.
Boston, MA
ates@bu.edu

Yijia Zhang
Boston University
Dept. of Electrical and Computer Eng.
Boston, MA
zhangyj@bu.edu

Burak Aksar
Boston University
Dept. of Electrical and Computer Eng.
Boston, MA
baksar@bu.edu

Jim Brandt
Sandia National Laboratories
Albuquerque, NM
brandt@sandia.gov

Vitus J. Leung
Sandia National Laboratories
Albuquerque, NM
vjleung@sandia.gov

Manuel Egele
Boston University
Dept. of Electrical and Computer Eng.
Boston, MA
megele@bu.edu

Ayse K. Coskun
Boston University
Dept. of Electrical and Computer Eng.
Boston, MA
acoskun@bu.edu

ABSTRACT

Modern High Performance Computing (HPC) systems, including supercomputers, routinely suffer from performance variations. The same application with the same input can have more than 100% performance variation, and such variations cause reduced efficiency and wasted resources. There have been recent studies on performance variability and on designing automated methods for diagnosing "anomalies" that cause performance variability. These studies either observe data collected from clusters, or, as in most cases, they rely on synthetic reproduction of performance variability scenarios. However, there is no standardized way of creating synthetic anomalies; so, researchers rely on designing ad-hoc methods for reproducing performance variability.

This paper addresses this lack of a common method for creating relevant performance anomalies by introducing HPAS, an *HPC Performance Anomaly Suite*, consisting of anomaly generators for the major subsystems in HPC systems. The easy-to-use synthetic anomaly generators would facilitate effortless evaluation and comparison of various analytics methods as well as performance or resilience of applications, middleware, or systems under realistic performance variability. The paper also provides an analysis on the behavior of the anomaly generators and demonstrates several use cases: (1) performance anomaly diagnosis using HPAS, (2) evaluation of resource management policies under performance variations, and (3) designing applications that are resilient to performance variability.

ICPP '19, August 05–08, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

CCS CONCEPTS

• **General and reference** → **Experimentation**; • **Networks** → *Network experimentation*; • **Computer systems organization** → *Grid computing*;

KEYWORDS

HPC, Performance variability, Anomaly, Benchmark

ACM Reference Format:
Emre Ates, Yijia Zhang, Burak Aksar, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. 2019. HPAS: An HPC Performance Anomaly Suite for Reproducing Performance Variations. In *Proceedings of ICPP '19: 48th International Conference on Parallel Processing (ICPP '19).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Modern high performance computing (HPC) systems routinely encounter performance variability as a result of hardware manufacturing variability, software problems, or resource contention among or within jobs [4, 22, 28, 44]. Performance variability results in sub-optimal scheduling, wasted compute cycles (and therefore, loss of efficiency and higher cost), and user dissatisfaction. Detecting performance variability and designing methods to minimize the unwanted variations are among the major challenges in production HPC systems.

One factor contributing to these challenges is the lack of an open-source, widely-applicable method for reproducing realistic scenarios that create performance variability. As a result, researchers and system administrators have to either operate with real performance variation data collected from production systems or create their own models for performance variations. Real systems data is often not available to researchers, and the available data is limited to a few known cases where the expected and/or observed performance for the majority of the applications are unknown [14, 26]. Performance variation models include simulations of systems under contention [3, 19] and synthetic reproduction of performance variations (i.e., anomaly injection) [23, 31, 48, 49]. Due to the lack of a common experimentation mechanism to generate performance

anomalies, researchers create their own methods, which results in a fragmented research space, difficulty in repeatability/comparison of results across different research teams, and loss of valuable research and system time.

In this paper, we introduce HPAS, a new *HPC Performance Anomaly Suite*, with the goal of enabling researchers, engineers and administrators to repeatably and systematically study performance variability in HPC systems. We follow the scientific intuition that standardized benchmarks play an important role in the development of computer hardware and software as well as in the evaluation of middleware and policies. Such benchmarks relieve engineers and scientists from the burden of developing representative workloads. In addition, as discussed above, coming up with realistic examples or use cases may often be difficult—or impossible—for many researchers. Using benchmarks, researchers can compare different approaches in computer systems in a fair manner and help advance science. It is our intent to advance the state-of-the-art in reproducible HPC research with this anomaly suite contribution.

Our anomaly suite, HPAS, consists of a set of synthetic "anomalies" that reproduce common root causes of performance variations in supercomputers: CPU contention, cache evictions, memory bandwidth interference, memory intensive processes, memory leaks, network contention, and contention in the shared filesystem metadata servers and storage servers. We design these synthetic anomalies using processes that run in userspace; thus, our suite does not require modifications to hardware, any other applications, or the kernel. Our anomalies can be configured for various intensities at runtime using several knobs. Furthermore, each anomaly is designed to minimize its interference in the subsystems that it is not targeting. Specific contributions of this paper are as follows:

- A rich suite of anomaly generators that can be used to study performance variability in HPC systems, evaluate the performance variability of new systems, and/or develop software resilient against future performance variations[1];
- an analysis of the characteristics of each of these anomalies by themselves and their impact on various colocated applications; and
- demonstrations of the uses of these anomalies, including generating synthetic data for the evaluation of anomaly diagnosis methods and comparing the effects of performance variability on load balancing policies and system management policies.

2 BACKGROUND ON PERFORMANCE VARIABILITY

Realistic and systematic reproduction of performance variability is a prerequisite for most research on the elimination of it, and the first step in reproduction is an analysis of the causes and consequences of performance variability. This section presents a general overview of performance variability based on previous research.

It is important to note the difference between performance variability and faults. *Faults* include behaviors in the computer system that result in errors in the correctness of the executed program. We focus on *performance variability*, which results in sub-optimal execution time while still obtaining correct results. For example, our focus on performance variability includes anomalies such as network contention which do not result in wrong results but may lead to reduced performance in systems where different applications share network resources.

Performance variability adversely affects supercomputers in many ways. Users request more time than required for their jobs because they cannot reliably predict job running time, which in turn harms scheduling. Researchers measuring performance need to take a large number of measurements since results may be invalid if insufficient measurements are taken [18, 34]. Programmers are unable to decide whether a code change improves or degrades performance due to high run-to-run performance variability.

Several researchers have investigated the root causes of performance variability, which can be grouped by the subsystem in which the performance anomaly manifests. This grouping is useful for us because the method of replication depends highly on the subsystem. The major subsystems affecting performance in a supercomputer are the CPU, the cache hierarchy, the memory, the high speed network, and the storage system. We provide examples of performance variability caused in each subsystem below:

CPU: Several examples of performance variations stem from the CPU. Orphan processes left from previous jobs that are still consuming CPU cycles are one of the anomalies that clog the CPU [6, 7]. System processes may also use a high amount of CPU because of software errors or miscalibration [8], or OS scheduling may result in unpredictable execution times—also known as "OS jitter". Manufacturing variability of CPUs have also been reported to affect the performance of HPC jobs [22, 33].

Cache Hierarchy: Modern CPUs' cache hierarchies consist of several levels of cache, and the cache hierarchy is a typical source of performance variation for both distributed HPC applications and single-server or single-CPU applications. Some of the cache-related performance variation is unavoidable (e.g., the cold-start effect), while some of it is a result of software problems (e.g., false sharing) or a combination of software and hardware problems [9, 36].

Memory: In systems where the nodes are shared between different applications, memory is shared as well, causing contention on the memory. In systems without node-sharing, the placement of application data into different memory banks may affect performance [37]. Other examples are memory intensive orphan processes or memory leaks.

Network: The high speed network is typically shared between many applications, and certain usage patterns may cause network contention, adversely affecting other applications [3, 12, 15]. The location and severity of contention in a network depend on the network topology, whether nodes are shared between different jobs, the number of NICs per node, the number of links between two nodes in the network, and other factors.

Shared Storage: Performance variations can also be caused by interference or other problems in shared filesystems [11]. In most cases, MPI requires all of the communicating nodes to have the same binaries in the same paths, and shared filesystems are a common way to accomplish this. These filesystems are also used for checkpointing data and other inputs/outputs. Both the speed of

[1]We will release the source code and additional documentation of the anomalies described here with the publication of this paper.

Table 1: A list of HPAS anomalies and their details. Every anomaly has configurable start/end times as well.

Anomaly type	Anomaly name	Anomaly behavior	Runtime configuration options
CPU intensive process	cpuoccupy	Arithmetic operations	utilization%
Cache contention	cachecopy	Cache read & write	cache (L1/L2/L3), multiplier, rate
Memory bandwidth contention	membw	Not-cached memory write	buffer size, rate
Memory intensive process	memeater	Allocate, fill, & release memory	buffer size, rate
Memory leak	memleak	Increasingly allocate & fill memory	buffer size, rate
Network contention	netoccupy	Send messages between two nodes	message size, rate, ntasks
I/O metadata server contention	iometadata	File creation & deletion	rate, ntasks
I/O bandwidth contention	iobandwidth	File read & write	file size, ntasks

checkpointing and the speed of I/O depends on the performance of the shared filesystem, which can vary significantly over time.

Our anomaly suite, HPAS, implements eight performance anomalies, shown in Table 1, in order to replicate the types of performance variability mentioned above in the subsystems described.

3 SYNTHETIC ANOMALIES

Our goal when designing HPAS is to accurately reproduce performance variations that are commonly encountered in HPC systems. This section first describes our design constraints for the synthetic anomalies and then provides the details of each synthetic anomaly. We select the sources of interference and design the anomalies based on discussions with experts in addition to a review of the literature (Section 2). Four of the anomalies we present are based on those used in our previous work [48, 49], which have also been used by Netti et al. [38] (i.e., cpuoccupy, cachecopy, memleak, memeater in our suite).

When designing the anomalies, we aim to balance the usability of the anomalies with realistic reproduction. We implement all anomalies such that no modifications to the benchmark applications, shared libraries, operating system kernel, and drivers or supercomputer hardware are required. For example, even though a memory leak could be more realistically reproduced by modifications to application code, such an approach would require separate modifications to each benchmark and would reduce the reusability of the anomaly suite. Instead, we use a separate userspace process that has a similar impact on the system.

We also design the anomalies such that the intensity of the anomaly can be adjusted using command line options. For example, for anomalies that require memory allocation, the amount and rate of memory allocated is adjustable; or, for some anomalies, a variable amount of sleep is inserted between periods of activity to reduce the intensity of the anomaly. This configurability also enables composing more complicated variability patterns (e.g., [30]) by using multiple anomaly instances.

We consider each of the major subsystems in an HPC system, i.e., the CPU, the cache hierarchy, the memory, the high speed network, and the storage system. For each subsystem, we design synthetic anomalies that replicate causes of performance variation that stem from that subsystem. Table 1 provides a summary of our anomalies that will be elaborated upon in the following subsections.

3.1 CPU

We model CPU-based performance variability with the cpuoccupy anomaly. This anomaly performs arithmetic operations on random values in a loop and sleeps for a given percentage of the time, using setitimer(). In this way, the activity of the anomaly has negligible impact on the cache or memory, and the utilization of the CPU can be adjusted to a given percentage. The CPU consists of many components which may independently affect performance; however, the contention in HPC systems is typically between different processes, and different processes contend for CPU time, which can be adequately reproduced using cpuoccupy.

The cpuoccupy anomaly can be used to emulate CPU-intensive orphan processes by running it on the same node with the application, or it can emulate OS jitter by setting the consumed CPU time to a low value, which impacts the scheduling behavior of the OS.

3.2 Cache Hierarchy

We model cache-related performance variations with the cachecopy anomaly that intensively uses the cache. The anomaly generator allocates two arrays, each of which are half the size of the L1, L2 or L3 caches, based on user-chosen parameters and repeatedly copies the contents of one array to the other one. The two arrays are contiguous in memory and are allocated using posix_memalign(). In this way, the specific level of the cache is effectively utilized by the anomaly, and the cache lines belonging to applications that share the same level of cache as the anomaly are expected to be frequently evicted.

cachecopy can be used to emulate cache contention, other hardware or software problems that may cause cache lines to be unexpectedly evicted, or running on a machine with a smaller cache.

3.3 Memory

We create three synthetic anomalies to model memory-related performance variability: Memory-intensive orphan processes, memory leak, and memory bandwidth contention. These anomalies can be used to mimic different types of memory contention or dead memory regions.

3.3.1 Memory Intensive Process. The memeater anomaly allocates an array of a given size (35MB by default, but adjustable) and fills it with random values. Later, it uses realloc() to increase the array's size by the same amount, fills the remaining area with random

```c
#include <xmminitrin.h>
void temporal_copy(double **orig, double **swap) {
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      _mm_stream_pi((__m64 *) (&swap[j][i]),
        *(__m64 *) (&orig[i][j])); // MOVNTQ
      _mm_empty(); // EMMS
    }
  }
}
```

Figure 1: A C code sample for creating memory bandwidth contention.

values, and repeats until the time or size limit given by the user is reached.

3.3.2 Memory Leaks. We model memory leaks using the MEMLEAK anomaly, which allocates an array of characters of a given size (20 MB by default) and fills it with random characters in each iteration. The addresses of the arrays are not stored and are not freed at each iteration, causing a memory leak.

3.3.3 Memory Bandwidth. The MEMEATER anomaly uses a large amount of cache as well, so we also design MEMBW to create contention only in the memory bandwidth. We model memory bandwidth contention by using the x86 SSE non-temporal memory instructions such as MOVNT*. These instructions are accessible from intrinsic functions which are supported by most compilers. When the data is marked with the non-temporal hint, i.e., that it will only be used once, it is not loaded into the cache. Our anomaly, MEMBW, first allocates two 2D matrices in the stack and fills one of them with random values. Then, it writes the transpose of the first matrix into the second matrix using the non-temporal hint, as shown in Figure 1. The transpose operation repeats for the duration of the anomaly.

3.4 Network

There are several methods for implementing inter-node communication in supercomputers, and when designing the NETOCCUPY anomaly we choose the method that introduces the least software overhead and the most emphasis on the network. MPI is the dominant parallel programming model, and many MPI implementations offer OS-bypass and other optimizations, allowing for faster communication compared to using raw sockets. However, we choose the SHMEM API since it has been shown to have a lower latency, thus higher load on the network, compared to MPI on the Cray Aries network [2], and it also offers similar optimizations as MPI. We focus on the Cray Aries because it is one of the most commonly used networks in the top 10 computers in the Top500 list (tied with Mellanox EDR Infiniband).

Our network interference generator can be executed in any two nodes provided that the link or router to be congested lies in the main path between the nodes. The anomaly then pairs the ranks on either node, such that the ranks on one node send messages to their corresponding rank on the other node using SHMEM_PUTMEM(). We use 100 MB messages because we observe that using messages smaller than 100 MB results in less contention, while messages larger than 100 MB do not noticeably increase bandwidth usage of the anomaly.

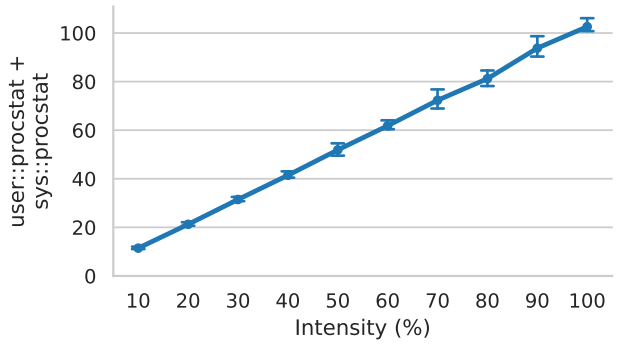

Figure 2: CPUOCCUPY intensity vs. CPU utilization in Voltrino. CPUOCCUPY uses the given percentage of the CPU.

The main usage of the NETOCCUPY anomaly is to emulate network contention. The bandwidth consumption of the anomaly can be tuned to emulate different levels of network contention.

3.5 Shared Storage

We target a common shared filesystem architecture, where there are one or a few metadata servers that manage the creation/deletion of files and other metadata such as locks and the locations and permissions of the files. Each metadata operation first passes through these metadata servers, and the actual contents of the files are located in storage nodes. The communication between the filesystem and the compute nodes is performed using either a separate network or the same interconnect that is used for inter compute-node communication.

Using the POSIX API, we can stress both the metadata servers and the storage servers separately; therefore, we design two anomalies. The metadata server is stressed using the IOMETADATA anomaly that creates and opens files, writes one character to each in a loop, closes all open files, and deletes them after 10 iterations. The IOBANDWIDTH anomaly uses `dd` [21] to copy random data into a file. It then copies that file to another file and so on. This anomaly causes contention in the disks of the storage servers, as well as the interconnect between the filesystem and compute nodes. Both of these anomalies can be used standalone, or they can be started using MPI to achieve higher contention, in which case they use separate files for each rank.

4 EVALUATION

To evaluate the proposed anomaly suite, we inspect the effect of each anomaly on target subsystems and different applications. We run our experiments on two systems: Voltrino, a Cray XC40m supercomputer located at Sandia National Laboratories, and Chameleon Cloud [24], a cluster of bare-metal servers. Voltrino has 24 nodes with two Intel Xeon E5-2698 v3 processors with 16 cores per socket and 24 nodes with one Intel Xeon Phi 7250 processor with 68 cores. We run all of the experiment on Voltrino using the nodes with Haswell Xeon E5-2698. Chameleon Cloud (CC) has two 12-core Intel Xeon E5-2670 v3 processors per node. Both systems have 125GB memory per node.

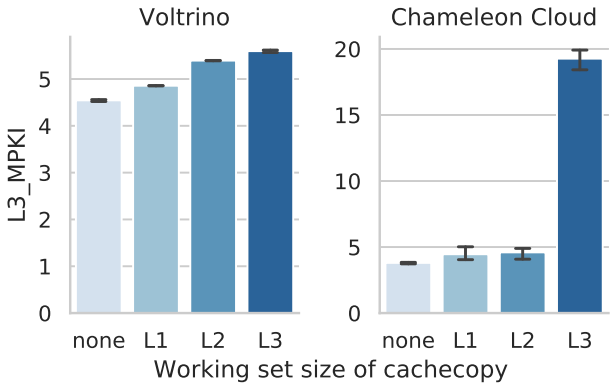

Figure 3: CACHECOPY vs. L3 misses per 1000 instructions (MPKI) in miniGhost in Voltrino and Chameleon Cloud.

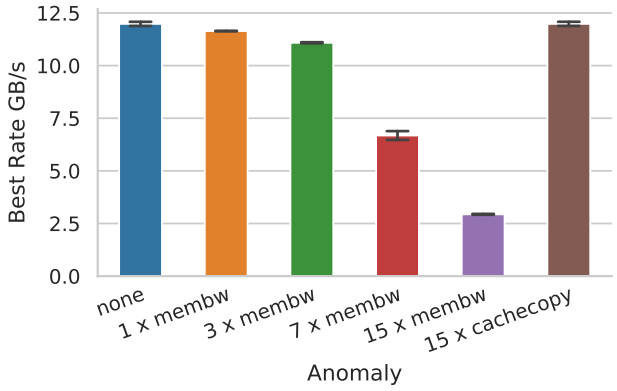

Figure 4: MEMBW and CACHECOPY effects on memory bandwidth on Voltrino. As expected, CACHECOPY has no significant impact on memory bandwidth while MEMBW significantly reduces memory bandwidth available to the application.

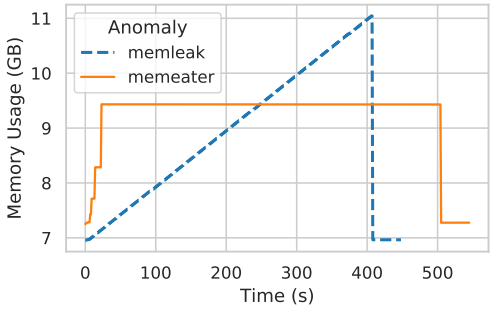

Figure 5: Memory usage over time for MEMLEAK and MEMEATER on Voltrino.

We collect monitoring data using the open-source Lightweight Distributed Metric Service (LDMS) [1]. LDMS on Voltrino uses several samplers: PROCSTAT collects processor metrics from /PROC/STAT; MEMINFO and VMSTAT collects memory metrics from /PROC/MEMINFO and /PROC/VMSTAT, respectively; ARIES_NIC_MMR collects hardware counters from the Aries Network Interface Cards (NICs); CRAY_ARIES_R collects Cray-specific hardware counters; and SPAPIHASW collects hardware counters using PAPI [46]. In the rest of the paper, we indicate the sampler for each metric using '::'. For example, USER::PROCSTAT indicates the metric USER from /PROC/STAT. In Voltrino, LDMS is configured to collect 2121 metrics per second from each node in our experiments.

4.1 Effects of the Anomalies on Their Respective Subsystems

We evaluate the effectiveness of each anomaly on its target subsystems in this section. Figure 2 shows the total CPU utilization in one node (i.e., USER::PROCSTAT + SYS::PROCSTAT) against the chosen intensity for CPUOCCUPY. Aside from the variability caused by the operating system, CPUOCCUPY can accurately use the given percentage of the CPU, and can be used to model CPU contention. The results from CC agree with Voltrino results.

The effect of CACHECOPY is demonstrated in Figure 3. For this experiment, a single-rank instance of miniGhost [17] and CACHECOPY is placed on the same physical core, but two different logical cores using hyperthreading, causing them to share L1, L2 and L3 caches. We increase the working set size of the anomaly from the size of L1 cache to the size of L3. As the working set size is increased, more last level cache misses are observed for miniGhost. As CC has a smaller L3 cache than Voltrino, it suffers from more L3 cache misses with the anomaly.

Figure 4 shows the memory bandwidth as measured by the STREAM benchmark [35] in presence of the MEMBW or CACHECOPY anomalies. We place STREAM on core 0 and place the anomalies on cores other than 0 until we use all the other 15 cores of the socket for the anomaly. We also report results for CACHECOPY, which has negligible effect on memory bandwidth as expected, even though it uses 15 cores. The results from CC agree with those from Voltrino.

We show the memory behavior over time for MEMLEAK and MEMEATER in Figure 5. While MEMEATER behaves like a memory-intensive application and allocates a large amount of memory at initialization, it does not increase the total memory footprint. On the other hand, MEMLEAK displays the typical pathological memory allocation pattern that keeps increasing. Both anomalies terminate after the given duration. The amount of memory allocated and the behavior over time can be tuned in our anomaly generators. The results from CC agree with those from Voltrino.

To quantify the effectiveness of the NETOCCUPY anomaly, we measure the bandwidth between two nodes in two different switches in Voltrino using the OSU benchmark [39], as shown in Figure 6. The Aries interconnect has 4 nodes connected to each switch; thus, we allocate the remaining 6 nodes for the network anomalies. We use 1, 2, and 3 pairs of nodes for the anomaly (corresponding to 2, 4, 6 nodes in the figure). The anomalies reduce the effective bandwidth of the OSU benchmark. Note that we use Cray MPI's MPICH_GNI_GET_MAXSIZE parameter to observe the effect of network congestion for smaller message sizes. The reduction of bandwidth is limited because of the topology of Voltrino, which has many redundant links and uses adaptive routing to avoid congested links. Another consequence of this adaptive routing is the possible congestion caused in links not directly targeted by the anomaly. Runtime control for adaptive routing should resolve such issues;

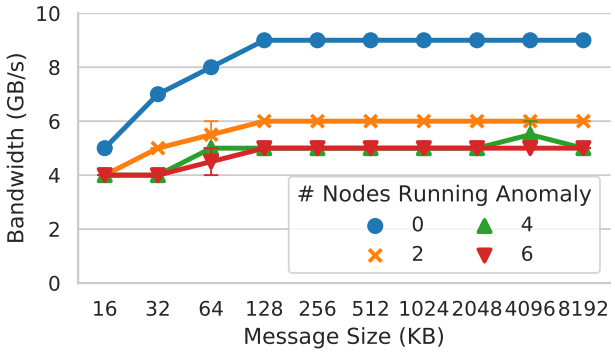

Figure 6: Message size of the OSU benchmark vs. network bandwidth in presence of NETOCCUPY anomaly in Voltrino.

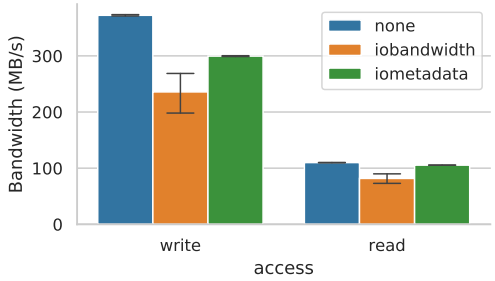

Figure 7: Impact of I/O anomalies when run on Chameleon Cloud.

however such control is not typically available. We cannot evaluate the network anomaly in CC because of its simple star network topology, which means that the network links are only between the single router and the nodes.

For evaluation of the I/O anomalies, we only use Chameleon Cloud (CC) because the filesystem in Voltrino is shared by many systems other than Voltrino, resulting in inherent performance variability even when there are no applications running on Voltrino. Furthermore, our initial experiments using 256 instances of IOMETADATA and IOBANDWIDTH anomalies caused outages in Voltrino's Lustre filesystem. On CC, we use the "Network File System (NFS) share" complex appliance of CC[2] to set up one NFS server and five clients. The storage server has one 250 GB ST9250610NS disk, and has the same CPU as the compute nodes. We run the IOMETADATA or IOBANDWIDTH anomalies on four nodes (48 instances per node) while measuring filesystem performance by running the IOR application [32] on the remaining node. Figure 7 demonstrates that IOBANDWIDTH reduces the effective bandwidth of IOR placed in the other node by clogging the disk on the storage node. The IOMETADATA anomaly also affects the bandwidth, since the CC filesystem does not have a separate metadata server. The impact of IOBANDWIDTH is higher in our case because the NFS server is using a single disk and 24 threads for metadata operations.

[2]https://www.chameleoncloud.org/appliances/25/

4.2 Effect of Anomalies on HPC Applications

We analyze the impact of our synthetic anomalies on a diverse set of eight benchmark applications shown in Table 2. Among them, Cloverleaf, CoMD, miniAMR, miniGhost, and miniMD are from the Mantevo Benchmark Suite [17], which are proxy applications mimicking different scientific computation kernels. Kripke is a proxy application for particle transport simulation developed to study the performance characteristics of data layouts and sweep algorithms [29]. MILC represents part of the codes written by the MIMD Lattice Computation collaboration to study quantum chromodynamics [47]. SW4lite is also a proxy application containing the computational kernels of SW4 which solves an elastic wave equation for seismic simulations [43].

To first understand how intensively the benchmark applications use certain system resources, we analyze the characteristics of the selected benchmark applications based on collected performance metrics (without any anomalies). We evaluate CPU-intensiveness by instruction per second (IPS) through the metric INST_RETIRED:ANY::SPAPIHASW; we evaluate memory-intensiveness by observing cache misses through the metric L2_RQSTS:-MISS::SPAPIHASW. We evaluate network-intensiveness by a network traffic counter through the metric AR_NIC_NETMON_ORB_-EVENT_CNTR_REQ_FLITS::ARIES_NIC_MMR. Based on our analysis, we summarize the characteristics of the benchmark applications in Table 2.

Figure 8 shows the running time of applications when they are run with the anomalies. Each anomaly affects application performance in different ways. The anomalies that affect performance the most are CACHECOPY, CPUOCCUPY and MEMBW. For example, CPU-intensive applications, including CoMD, miniMD, and SW4lite, are all heavily affected by CACHECOPY and CPUOCCUPY. The memory-intensive applications, including Cloverleaf, MILC, miniAMR, and miniGhost, are more impacted by MEMBW than other anomalies. None of the applications are affected significantly by the network anomaly because of the highly connected network of Voltrino, that is designed for much larger supercomputers with adaptive routing. Also, memory anomalies such as MEMLEAK and MEMEATER do not visibly affect performance because Voltrino does not use swap and applications are killed when they run out of memory. Indeed, if the size of the memory anomalies are set too large, they result in application crashes.

5 USE CASES FOR HPAS

In this section, we show, using experiments on Voltrino, that our anomaly suite can be used in the following three example cases: (1) Evaluate tools that diagnose performance deviation on HPC systems, (2) systematically evaluate the performance of system management policies under the conditions of resource contentions, and (3) develop applications and systems resilient to performance variability. We envision that the usage of HPAS will be advantageous in many other performance or resilience studies as well.

5.1 Evaluating Anomaly Diagnosis Tools

Anomaly detection and diagnosis are widely researched areas [23, 26, 31, 48, 49]. Some of these methods are based on machine learning algorithms and require a considerable amount of training data to

Table 2: Characteristics of the benchmark applications.

	Cloverleaf	CoMD	Kripke	MILC	miniAMR	miniGhost	miniMD	SW4lite
CPU-intensive		✓					✓	✓
Memory-intensive	✓			✓	✓	✓		
Network-intensive	✓		✓	✓	✓	✓		

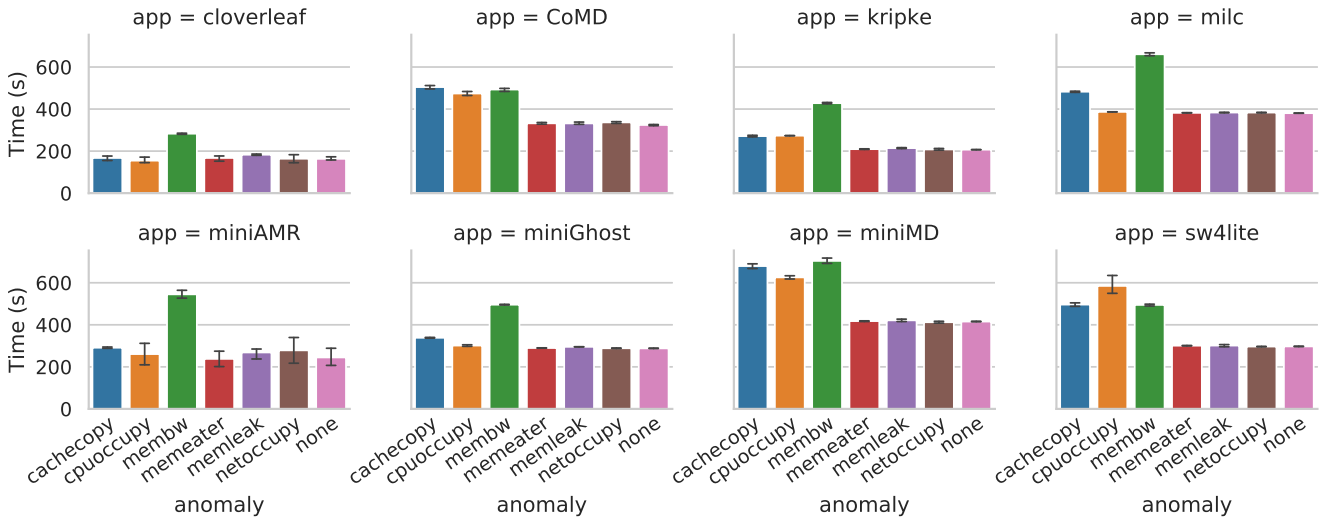

Figure 8: Execution time of each application with each anomaly on Voltrino.

use and evaluate. Since collecting ground truth anomaly data on HPC systems is not an easy task, data generated using our anomaly suite can be used instead. Furthermore, using the same methods for anomaly generation can make it easier for researchers to compare anomaly diagnosis methods.

To demonstrate the use of our anomaly suite in evaluating anomaly diagnosis tools, we use our previous work on anomaly detection [48, 49]. Our framework contains an offline training phase and a runtime diagnosis phase. In the offline training phase, we first use resource usage and performance counter data from known healthy and anomalous runs to extract useful statistical features calculated from time series. Then, these features are used to train the tree-based machine learning algorithms. At runtime, we generate statistical features from resource usage and performance counter data. Using these features, the machine learning model predicts the root cause (e.g., CPU contention, memory leak, network contention) of performance variations occurring at certain times. In a very similar manner to our previous work [48], we collect similar metrics using LDMS and generate the statistical features mentioned in the paper. We use decision tree, AdaBoost, and random forest algorithms for training and prediction.

We run eight benchmark applications with and without our anomalies and use the data generated to evaluate the diagnosis framework using 3-fold cross-validation. The F1-scores for individual anomalies are reported in Figure 9 and the confusion matrix which shows accuracy for each class is reported in Figure 10. While the framework is good at identifying whether there is an anomaly or not, the CACHECOPY, CPUOCCUPY, and MEMBW anomalies are

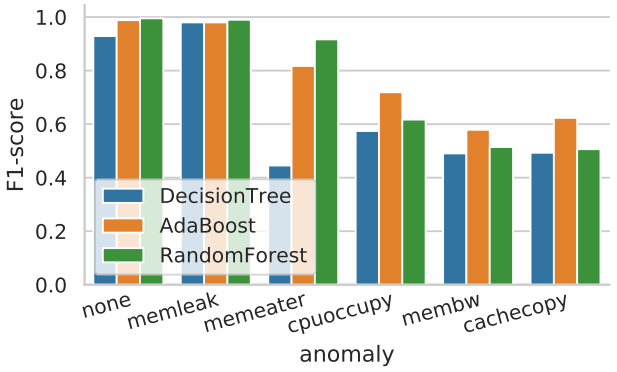

Figure 9: Results for classification of the anomalies. The overall F1-score using Random Forest algorithm is 0.94.

sometimes mistaken for each other. This could be due to the lack of metrics representing memory bandwidth in the monitoring data. In general, the results are compatible with our earlier results, demonstrating the usability of our suite in the evaluation of anomaly diagnosis methods. Our new anomalies also demonstrate room for improvement for better diagnosis of cache and CPU anomalies in Figure 10.

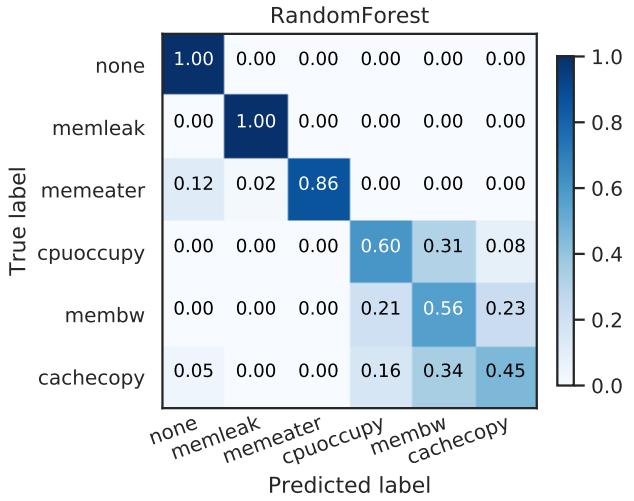

Figure 10: Confusion matrix for anomaly diagnosis using Random Forest.

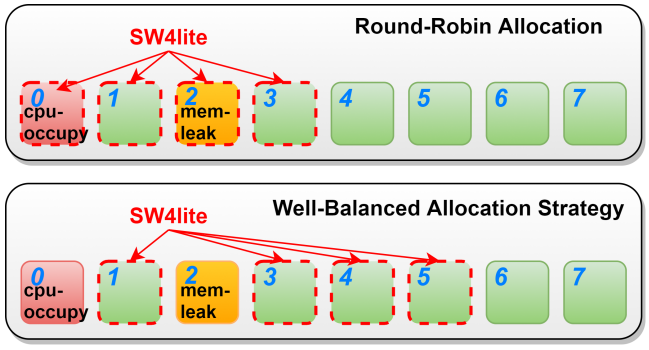

Figure 11: Allocation of SW4lite with two policies.

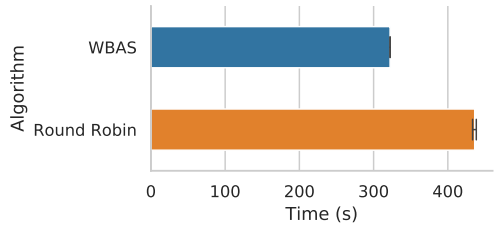

Figure 12: Evaluating the impact of anomalies on two different job allocation policies.

5.2 Evaluating System Management Policies

System management policies on HPC systems, such as job scheduling, job allocation, or task mapping, play a vital role in efficient usage of system resources [50, 51]. Anomalies in a system may affect the behavior of a system management policy. Knowing how the scheduling/allocation of jobs changes when there are performance anomalies helps us evaluate system management policies in a more realistic manner and select a policy that is resilient to anomalies.

In the following, we demonstrate how two job allocation policies react differently under presence of CPU and memory anomalies. The two policies are the Round-Robin (RR) policy and the Well-Balanced Allocation Strategy (WBAS) by Yang et al. [51]. The RR policy simply allocates a job to the available nodes in the system following the label order. The WBAS policy prioritizes selecting the nodes with lower CPU load and larger free memory. To accomplish this, the WBAS policy calculates a computing capacity (CP) value for each node by $CP = (1 - Load\%) \times Mem_{free}$. Here, the CPU load $Load\%$ is derived from both current load and the average load of the recent several minutes according to the formula $Load = \frac{5}{6}Load_{current} + \frac{1}{6}Load_{5minAvg}$. In our system, we collect the current CPU load of the node using the metric USER::PROCSTAT, and we monitor the free memory (Mem_{free}) by the metric MEMFREE::MEMINFO.

In our case study, we run the SW4lite application on 4 nodes of the system out of 8 available nodes (Nodes [0..7]), as shown in Figure 11. To create an anomaly, we run CPUOCCUPY on Node 0, and run MEMLEAK on Node 2. CPUOCCUPY can be used to change the CPU load to any given value between 0% and 100% for each core, we set it to 100% for one core. MEMLEAK can be used to reduce free memory on Node 1 to any given value, we set it to 1GB. The WBAS policy avoids using the two nodes with the anomalies and allocates the job to Nodes [1, 3..5] instead. Meanwhile, the RR policy allocates the job to Nodes [0..3].

With each of the two allocation policies, we run the SW4lite application 3 times and report the execution time in Figure 12. On average, the job execution time is 322 s with the WBAS policy, and it is 436 s with the RR policy. These results show that compared to the RR policy, the WBAS policy reduces the execution time by 26% on average through actively avoiding the anomalous nodes. This experiment provides an example of how our synthetic anomaly suite can be utilized to evaluate and compare different job allocation policies. HPAS brings the ability to independently change the $Load\%$ and Mem_{free} components of the CP equation, enabling a very systematic evaluation of the equation and allowing for more complicated models perhaps with cache or network components as well. Without the usage of our suite, it is much more difficult to systematically test and compare different system management techniques under controlled but different anomaly situations.

5.3 Developing Applications that are Resilient to Performance Variability

One way of developing applications resilient to performance variations is to be aware of how much a given application is affected by anomalies in different subsystems. As an example, we show the use of HPAS in demonstrating the effect of a load balancing algorithm by using the Charm++ runtime system.

We use a simple 3D stencil application given in Charm++ examples and execute it on one node while changing the intensity of the CPUOCCUPY anomaly from zero to 100% of 32 CPUs. Figure 13 shows the performance of two load balancers: LBOBJONLY that only uses object properties and GREEDYREFINELB load balancer that measures CPU capacity before scheduling tasks. The two load

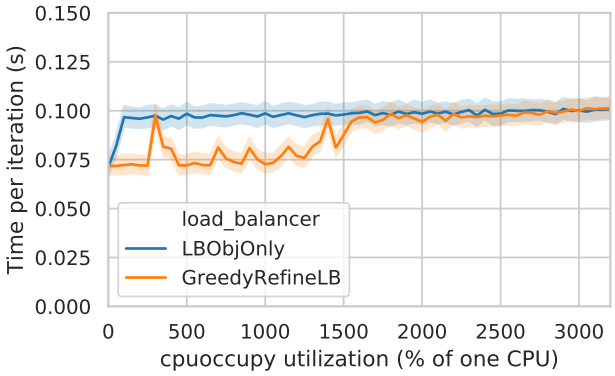

Figure 13: Performance of 3D stencil with different load balancers with increasing CPUOCCUPY intensity on Voltrino.

balancers perform similarly when there are no anomalies (utilization = 0), and when more than 16 CPUs are used by the anomaly. However, in most cases where the anomaly uses fewer than 16 CPUs, GREEDYREFINELB, which measures CPU capacity, outperforms the other one. Notably, a degradation of performance where the anomaly uses 4 CPUs may indicate room for improvement in the load balancing strategy, as anomaly intensities at 5 or 6 CPUs can be mitigated successfully.

This example use case illustrates how our anomaly suite can be used to inform the choice of the load balancer, the development of new load balancers, or the decision to use a load balancer or not.

6 RELATED WORK

Performance variations on HPC systems have been studied by many researchers. Skinner et al. report more than 100% slowdown compared to the average in production supercomputers [44]. They examine several different systems and report that cache contention, network contention, file system contention, kernel process scheduling, and system activity are the main causes of performance variations. Bhatele et al. show that on a Cray XE system, the execution time of communication-intensive applications ranges from 28% faster to 41% slower than the average performance [4]. They find investigate various causes for this performance variability such as operating system activity and job allocation strategy, and concur that interference on network links is the principal cause.

Several recent approaches detect and analyze the cause of performance variations. Varbench is a tool for measuring the performance variation experienced by applications [27]. Our previous work introduced a framework that relies on tree-based machine learning algorithms to diagnose the causes of performance variations given that there are available training data [48, 49]. Similarly, Klinkenberg et al. use descriptive statistics and machine learning to predict node failures, relying on monitoring data [26]. Kasick et al. diagnose performance variations in parallel file systems by comparing the probability distribution functions of various performance counters [23]. To evaluate their tools, researchers generate their own synthetic anomalies [16, 23, 31, 48, 49]. However, since synthetic anomaly generation is not the focus of these studies, the method for performance variability generation is not explained in sufficient detail to replicate the results and they have not released their codes for generating these synthetic anomalies.

There are various existing tools for creating performance variability on computer systems. Delimitrou et al. build a workload suite for data centers called iBench that induces interference in various shared resources, mostly architectural CPU components [10]. Their tool helps quantify the contention created by applications as well as the contention that can be tolerated by the applications. However, the released version is substantially limited compared to the tool described in the publication. Specifically for networked systems, Sato et al. build a tool called NINJA that mimics network noise by injecting sleep before MPI calls and, thus, creates a message race for MPI applications [41]. They demonstrate their tool can manifest subtle message races in MPI applications more frequently; however, their approach is not applicable for most forms of anomaly diagnosis since no actual network contention occurs. Netti et al. introduce a framework called FINJ that enables injecting anomalies into HPC systems [38], but their focus is on the mechanism for injecting anomalies, not the anomalies themselves; thus, they do not analyze the behavior and effect of the anomalies. Gremlins is a suite for emulating future HPC systems, e.g., power constrained systems, on current hardware [42]; their methodology is not explicitly targeting performance variability, thus they miss significant components such as network and I/O contention.

Another topic related to performance anomalies is fault injection [20]. Some approaches propose creating faults by flipping bits in registers or memory [25, 40, 45]. For networked systems, one way to create failures is to disable some nodes, links, or blades. Formicola et al. inject faults in this way and demonstrate the analysis of failure events using log data, Cray network performance counters, and benchmark application performance [14]. Another way to inject faults into networks is by creating timing delays, message omission, or message corruption. Some works propose tools for injecting these kinds of faults into MPI applications [5, 13]. As we have clarified in Section 2, faults affect the correct execution of a program and they are not the focus of this work. Meanwhile, our focus is on performance variation that does not affect the correctness of a program but affects its execution time.

7 CONCLUSION

We have presented HPAS, a suite of anomaly replication tools that realistically replicate performance variability in specific subsystems such as the CPU, cache, memory, network, or storage. We demonstrated compelling use cases for HPAS, including performance variation diagnosis and evaluation of system management policies and applications. In many of the use cases, HPAS has shown that there is room for improvement in the state-of-the-art. We believe that the adoption of this suite will have a positive impact on research and development on resolving performance variability in HPC systems.

ACKNOWLEDGMENTS

This work has been partially funded by Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering

Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under Contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

REFERENCES

[1] Anthony Agelastos et al. 2014. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 154–165.

[2] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. 2012. Cray XC series network. https://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf

[3] A. Bhatele, N. Jain, Y. Livnat, V. Pascucci, and P. Bremer. 2016. Analyzing Network Health and Congestion in Dragonfly-Based Supercomputers. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 93–102.

[4] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs. 2013. There goes the neighborhood: Performance degradation due to nearby jobs. In *SC*. 1–12.

[5] D. M. Blough and Peng Liu. 2000. FIMD-MPI: a tool for injecting faults into MPI application. In *IPDPS*. 241–247.

[6] Jim Brandt et al. 2009. Methodologies for Advance Warning of Compute Cluster Problems via Statistical Analysis: A Case Study. In *Proceedings of the 2009 Workshop on Resiliency in High Performance*. 8.

[7] J. Brandt et al. 2010. Quantifying effectiveness of failure prediction and response in HPC systems: Methodology and example. In *International Conference on Dependable Systems and Networks Workshops*. 2–7.

[8] Cisco. 2017. Cisco Bug: CSCtf52095 - Manually Flushing OS Cache during load impacts server. https://quickview.cloudapps.cisco.com/quickview/bug/CSCtf52095.

[9] Brandon Cook et al. 2017. Performance Variability on Xeon Phi. In *High Performance Computing*. Springer International Publishing, Cham, 419–429.

[10] C. Delimitrou and C. Kozyrakis. 2013. iBench: Quantifying interference for datacenter applications. In *IEEE International Symposium on Workload Characterization (IISWC)*. 23–33.

[11] M. Dorier et al. 2014. CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination. In *IPDPS*. 155–164.

[12] J. J. Evans, W. D. Groop, and C. S. Hood. 2003. Exploring the relationship between parallel application run-time and network performance in clusters. In *Annual IEEE International Conference on Local Computer Networks*. 538–547.

[13] K. Feng, M. G. Venkata, D. Li, and X. Sun. 2015. Fast Fault Injection and Sensitivity Analysis for Collective Communications. In *International Conference on Cluster Computing (CLUSTER)*. 148–157.

[14] Valerio Formicola et al. 2017. Understanding Fault Scenarios and Impacts through Fault Injection Experiments in Cielo. *Cray User Group* (2017).

[15] Ryan E. Grant, Kevin T. Pedretti, and Ann Gentile. 2015. Overtime: A Tool for Analyzing Performance Variation Due to Network Interference. In *Proceedings of the 3rd Workshop on Exascale MPI*. Article 4, 10 pages.

[16] Q. Guan, S. Fu, N. DeBardeleben, and S. Blanchard. 2013. Exploring Time and Frequency Domains for Accurate and Automated Anomaly Detection in Cloud Computing Systems. In *IEEE 19th Pacific Rim International Symposium on Dependable Computing*. 196–205.

[17] Michael A Heroux et al. 2009. *Improving Performance via Mini-applications*. Technical Report SAND2009-5574. Sandia National Laboratories.

[18] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. In *SC*. Article 73, 12 pages.

[19] T. Hoefler, T. Schneider, and A. Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[20] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4 (April 1997), 75–82.

[21] IEEE and The Open Group. 2018. POSIX Standard: dd. http://pubs.opengroup.org/onlinepubs/9699919799/utilities/dd.html.

[22] Y. Inadomi et al. 2015. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *SC'15*. 1–12.

[23] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. 2010. Black-box Problem Diagnosis in Parallel File Systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*. 4–4.

[24] Kate Keahey et al. 2018. Chameleon: a Scalable Production Testbed for Computer Science Research. In *Contemporary High Performance Computing: From Petascale toward Exascale*. Chapman & Hall/CRC Computational Science, Vol. 3. Chapter 5.

[25] G. Kestor, I. B. Peng, R. Gioiosa, and S. Krishnamoorthy. 2018. Understanding scale-Dependent soft-Error Behavior of Scientific Applications. In *18th International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 482–491.

[26] J. Klinkenberg, C. Terboven, S. Lankes, and M. S. Müller. 2017. Data Mining-Based Analysis of HPC Center Operations. In *CLUSTER*. 766–773.

[27] Brian Kocoloski and John Lange. 2018. Varbench: An Experimental Framework to Measure and Characterize Performance Variability. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018)*. ACM, New York, NY, USA, Article 18, 10 pages.

[28] William T. C. Kramer et al. 2003. Performance Variability of Highly Parallel Architectures. In *Computational Science — ICCS 2003*. 560–569.

[29] AJ Kunen, TS Bailey, and PN Brown. 2015. *KRIPKE-a massively parallel transport mini-app*. Technical Report. Lawrence Livermore National Laboratory.

[30] C. Kuo, A. Shah, A. Nomura, S. Matsuoka, and F. Wolf. 2014. How file access patterns influence interference among cluster applications. In *CLUSTER*. 185–193.

[31] Zhiling Lan, Ziming Zheng, and Yawei Li. 2010. Toward Automated Anomaly Identification in Large-Scale Systems. *IEEE Trans. Parallel Distrib. Syst.* 21, 2 (Feb. 2010), 174–187.

[32] Lawrence Livermore National Laboratory. 2018. IOR benchmark application. https://github.com/hpc/ior.

[33] Aniruddha Marathe et al. 2017. An Empirical Survey of Performance and Energy Efficiency Variation on Intel Processors. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing (E2SC)*. Article 9, 8 pages.

[34] Aleksander Maricq et al. 2018. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 409–425.

[35] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[36] John D. McCalpin. 2018. HPL and DGEMM Performance Variability on the Xeon Platinum 8160 Processor. In *SC'18*. Article 18, 13 pages.

[37] C. McCurdy and J. Vetter. 2010. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 87–96.

[38] Alessio Netti et al. 2018. FINJ: A Fault Injection Tool for HPC Systems. https://arxiv.org/abs/1807.10056. arXiv:cs.DC/1807.10056

[39] Dhabaleswar K. Panda et al. 2018. OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[40] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer. 2008. SymPLFIED: Symbolic program-level fault injection and error detection framework. In *IEEE International Conference on Dependable Systems and Networks (DSN)*. 472–481.

[41] Kento Sato et al. 2017. Noise Injection Techniques to Expose Subtle and Unintended Message Races. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 13.

[42] M Schulz et al. 2014. Performance analysis techniques for the exascale co-design process. *Advances in Parallel Computing* 25 (01 2014), 19–32.

[43] Bjorn Sjogreen. 2018. *SW4 final report for iCOE*. Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

[44] D. Skinner and W. Kramer. 2005. Understanding the causes of performance variability in HPC workloads. In *IEEE Workload Characterization Symposium*. 137–149.

[45] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer. 2000. NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings IEEE International Computer Performance and Dependability Symposium*. 91–100.

[46] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*. 157–173.

[47] The MIMD Lattice Computation (MILC) Collaboration. 2016. MILC benchmark application. http://www.physics.utah.edu/~detar/milc/.

[48] Ozan Tuncer et al. 2017. Diagnosing Performance Variations in HPC Applications using Machine Learning. In *International Supercomputing Conference (ISC-HPC)*. 355–373.

[49] O. Tuncer et al. 2019. Online Diagnosis of Performance Variation in HPC Systems Using Machine Learning. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (April 2019), 883–896.

[50] Qingqing Xiong, Emre Ates, Martin C. Herbordt, and Ayse Kivilcim Coskun. 2018. Tangram: Colocating HPC Applications with Oversubscription. In *IEEE High Performance Extreme Computing Conference*. 1–7.

[51] Chao-Tung Yang, Kuan-Chou Lai, and Hao-Yu Tung. 2011. On construction of a well-balanced allocation strategy for heterogeneous multi-cluster computing environments. *The Journal of Supercomputing* 56, 3 (01 Jun 2011), 270–299.