Varbench: an Experimental Framework to Measure and Characterize Performance Variability

Brian Kocoloski
Department of Computer Science & Engineering
Washington University in St. Louis
brian.kocoloski@wustl.edu

John Lange
Department of Computer Science
University of Pittsburgh
jacklange@cs.pitt.edu

ABSTRACT

Performance variability is a major problem for extreme scale parallel computing applications that rely on bulk synchronization and collective communication. While this problem is most prominent in the context of exascale systems, it is increasingly impacting other communities such as machine learning and graph analytics. In this paper, we present an experimental performance analysis framework called *varbench* that is designed to precisely measure the prevalence of performance variability in a system, as well as to support workload characterization with respect to how and when a workload generates variability. We demonstrate several of varbench's capabilities as they pertain to exascale-class systems, including its utility for discovering architectural trends, for performing cross-architectural comparisons, and for understanding key statistical properties of performance distributions that have implications for how system software should be designed to mitigate variability.

CCS CONCEPTS

• **General and reference** → **Measurement**; **Performance**; • **Computer systems organization** → *Multicore architectures*;

ACM Reference Format:
Brian Kocoloski and John Lange. 2018. Varbench: an Experimental Framework to Measure and Characterize Performance Variability. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225125

1 INTRODUCTION

Bulk Synchronous Parallelism (BSP) still dominates the application sets targeted by exascale system architectures. BSP applications execute in "lockstep," alternating between concurrent distributed computation and collective synchronization across the system. While BSP has long been the primary parallel programming model used in the High Performance Computing (HPC) community, recent years have seen increasing adoption of BSP in other communities as well, such as large scale graph processing [23] and machine learning [2]. It is thus likely that problems addressed in exascale systems will be relevant to other environments beyond those targeting HPC.

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225125

One of the main challenges faced by BSP workloads is performance variability. When different parallel tasks require different amounts of time between global synchronization phases, workload imbalance arises, and runtime and energy efficiency are determined by the "worst" performing processors or tasks in the machine. Despite the fact that variability is a well-known problem in the HPC community, with a plethora of different sources of variability identified at the application [3], system software [17], and hardware levels [9, 10], it remains a major issue. As an example, in a set of NERSC Petascale machines, up to 75% of the aggregate processing time across all processors can be spent "waiting" for global communication and synchronization [15].

While variability has long been a challenge for HPC systems, there are indications it will be *harder* to manage in the future. With the focus at exascale shifting towards designing more energy efficient machines, the manner in which HPC systems will be built, programmed and utilized could change in substantial ways that will likely cause more variability. At the hardware level, many-core node architectures are becoming more complex, heterogeneous, and interconnected than ever, with hundreds to thousands of cores per node, deep memory hierarchies, more heterogeneous processing elements, and complex on-chip interconnect topologies. At the application level, workloads are becoming more data driven [19] with workflows [6] that have a strong dependence on I/O performance in addition to computation and network based communication. Finally, at the broader system level, it is likely that systems will not be solely batch-scheduled entities serving one user at a time, but rather will schedule workflows from multiple users at a time that more appropriately reflect the data-driven nature of future applications, as well as to more efficiently utilize energy.

The problem we address in this paper is that there is currently no common performance evaluation framework with which to measure the presence of variability on a system. While many techniques offer solutions to specific problems in specific applications, it is hard to generalize solutions because it is hard to determine how variability compares across hardware, workloads, and system configurations. This paper addresses this issue by introducing the *varbench* framework. Varbench is a modular, extensible, experimental performance analysis framework that allows users to measure the extent to which performance variability arises on a given platform. Varbench utilizes a BSP-style computation/synchronization model to facilitate the execution, in parallel, of a user-configurable, iterative computational kernel across each processor in the system, and performs global synchronization between successive iterations in order to measure variability across periods of parallel execution.

Varbench brings two main capabilities to users. First, it provides a common performance evaluation framework to measure

variability across different architectures, or with a different configuration of system resources within an architecture (e.g., use of hyperthreads, TurboBoost, power cap levels, etc.). In comparison to mini-applications that project performance and/or scalability for real world applications, varbench is designed to focus solely on variability. Secondly, varbench provides fine-grained measurements of variability that are useful beyond determining whether one system has "more" or "less" variability than another. These measurements allow users to, for example, determine whether variability manifests as infrequent but extreme outliers, frequent but less extreme outliers, or other statistical profiles. Such distinctions are important because they suggest whether a given mitigating approach – such as predicting future variability based on past observations – is appropriate or not for a system.

In this paper we demonstrate varbench's utility by performing a detailed experimental analysis of low-level hardware variability. We summarize several of our findings, including that hardware performance is increasingly *temporally* variable, as opposed to simply spatially variable as assumed in several HPC middlewares/runtimes that adapt to performance imbalance. We also find that variability continues to increase both spatially and temporally in more recent version of common server architectures. Finally, we demonstrate how varbench can shed light on the impact of tunable system criteria, such as power capping mechanisms. To summarize, the primary contributions of this paper are:

(1) We identify concise, expressive statistical criteria that reflect key characteristics of variability in BSP-style workloads.
(2) We present a new performance evaluation framework, *varbench*, that provides a common set of low level workloads and has an easily extensible design with which to incorporate new workloads. Varbench provides precise measurements to characterize spatial and temporal variability.
(3) We perform case studies using varbench to illustrate its utility. Our evaluation shows (i) the prevalence of temporal variability in hardware performance, (ii) the exacerbating impacts of heterogeneous resources on spatial variability, and (iii) the influence of tunable system parameters in the case of node-level power caps.

2 RELATED WORK

Performance variability is a well-studied issue in the HPC community, with many different sources identified at different levels of the system stack. Applications themselves can be sources, as it can be a challenge to evenly balance workload across all tasks [3]. The operating system can generate variability through untimely preemption or interrupt handling, resulting in OS interference or "noise" [17]. At the broader cluster/system level, variability arises from from network congestion [24], heterogeneity [28] or workload contention from nearby jobs [4]. Variability also arises from hardware, either due to manufacturing variation [9], contention for architectural resources [10] or even intrinsically due to non-uniform resource performance in heterogeneous systems [5]. Finally, variability arises from I/O subsystems [21].

Due to the many diverse sources of variability, there have been several different proposals and techniques to mitigate it. Task-based runtimes [13, 14] attempt to load balance in response to application-induced imbalance. Lightweight operating systems [11, 26] eliminate unnecessary system services from commodity OSes and thus reduce OS-level interference. Other approaches model specific sources of variability and apply techniques such as speed scaling [16, 31, 33] to optimize program behavior in its presence. Though there has been much work to address specific issues, there has not been much convergence towards a single or small set of techniques, and it is still an open question as to which of these techniques should be the prevalent strategy in the future. With exascale computing further driving significant changes in how machines will be programmed and utilized, including via techniques such as power capping [27] and *in situ* co-scheduling of applications [20], assumptions underlying existing approaches may be changing, so it is even less clear what the correct strategies are moving forward.

	Software Induced	**Hardware or External Induced**
Spatially Variant	Application-level Imbalance [3]	Process Variation [9] Network Heterogeneity [28] Many-Core Architectures [5]
Temporally Variant	Resource Contention [4] OS noise [17]	Network Congestion [24] Power Heterogeneity [27] Intrinsic HW Variability [10] IO Variability [21]

Table 1: A Taxonomy of Variability Characteristics with Sources Expected in Exascale Systems

We believe that by characterizing performance variability, it will be possible to better understand how different mitigating techniques relate to each other, and to project whether or not a technique may be able to address a particular class of variability in the future. Table 1 illustrates one of the key criteria that distinguishes sources of variability from each other, which is the extent to which imbalance varies only spatially across tasks, meaning some are consistently higher/lower performing than others, versus the extent to which performance varies temporally within the same task over time. As the table shows, it is possible to categorize many sources of variability on today's machines with this characteristic. One example of why this is valuable is that, if variability on a new platform exhibits some characteristics that place it in one quadrant of the table, it indicates that some mitigating techniques will be unlikely to address it. For example, lightweight kernels, which are designed to eliminate rare temporal outliers resulting from transient OS interruptions, are unlikely to address sources that exhibit similar behavior to those on the top half of the table.

2.1 Beyond BSP

While the high-level performance implications of variability are well understood for HPC workloads, we note that variability is an issue for applications outside of HPC. Many applications value consistency from a computer system as much as, if not more than, maximal but inconsistent performance. This is true for applications in distributed cloud and data-center environments, particularly for latency sensitive and/or real-time applications, where inconsistencies make it challenging to provide guaranteed levels of service [18].

While we leverage BSP-style workloads to measure and characterize variability, we note the benefits provided by characterization are not limited to only BSP applications.

3 A CASE FOR CHARACTERIZATION

We claim that there is significant value in characterizing performance variability on a particular system. In this section, we motivate why characterizing variability is important, and why characterization can be a useful tool for researchers designing hardware and software for exascale machines. Sections 4 and 5 will then introduce the workloads our framework, varbench, provides as well as the statistical methods we use to perform characterizations.

3.1 Identifying Trends and Revisiting Assumptions

Having a common framework to measure variability will be useful for identifying trends and tracking the progression of key workloads over time in different environments. In this paper we discuss several trends we have identified with this approach, including that (1) hardware performance can be highly temporal in nature, varying over the lifetime of a workload, particularly when stressing shared architectural resources; (2) newer generations of architectures generally exhibit more variability than past generations when executing the same workload; and (3) that heterogeneous resources tend to further exacerbate the issue. In addition, we have found that previously identified issues such as chronically "slow" nodes do exist in some cases, but that they are workload dependent and more likely to occur for workloads that exhibit high temporal variability. These results are discussed in Section 6.

3.2 Considering Candidate Solutions

Mitigating performance variability has long been a primary goal of HPC system software environments, both by identifying and removing sources that can be eliminated [26] as well as by adapting application behaviors to account for it. In the latter case, several parallel runtimes have incorporated mechanisms that either balance load [13, 14] based on emergent imbalances or apply energy optimization techniques such as speed scaling [31, 33] in order to save energy on the faster processors that are likely to be idle as they wait for the slower ones to catch up.

In the application space, assumptions are often made to maximize the possible energy or performance gain. For example, Adagio [31] reduces energy consumption by monitoring each recurring MPI collective call in a program, and during runtime observes performance imbalance at these collectives. The system assumes that past imbalances are predictive of the future state of the system, and so configures non-uniform CPU frequencies across different processors to balance execution based on the predicted imbalances. Similar assumptions are made in parallel runtime systems such as Charm++ [14], which periodically observes workload imbalance to migrate tasks across a distributed system based on the assumption that whatever generated the imbalance will persist in the future.

While these approaches are able to address load imbalances resulting from consistent behaviors, as we observe in this work, they are not appropriate solutions for all forms of performance variability. These and other systems are primarily designed to measure *application induced* workload imbalance that results from intrinsic challenges in balancing application workloads [3], which tend to naturally drift into unbalanced states. However, when variability arises from forces external to the application, such as the operating system [17], or intrinsically from the hardware itself, it is less clear that imbalances observed in the past will provide predictability for future iterations. A key benefit of characterizing variability is that distinctions such as the presence of temporal variability can be concisely quantified, and thus it can be determined whether or not a particular technique is likely to be successful at mitigating variability on a new architecture or platform.

3.3 Understanding Architectural Differences

The exascale-driven emphasis on power and energy efficiency has led to interest in more energy efficient architectures, including those based on ARM [29], and SPARC [25]. While HPC systems have been predominantly composed of x86 processors for the past two decades [1], there are several factors converging to threaten its dominance, including the advent of heterogeneous computing combined with more energy efficient processors. Previous work has shown that some processors generate more variability than others [34]; it is important to further understand how to characterize variability on these architectures, and whether or not different processors lead to different performance profiles.

In addition to ISA diversity, on-chip resource heterogeneity is increasingly being adopted in large scale HPC systems, with architectures such as Intel Xeon Phi [32] now deployed on a significant fraction of the Top500 [1], including 4 of the current top 10 systems. As systems increasingly incorporate heterogeneous processors such as GPUs and FPGAs, multiple memory technologies [12], and more complex interconnects (PCIe, QPI/EPYC, NVLink, etc.) to move data between components, it is likely that workloads exercising these resources will experience more variability than the same workloads leveraging more homogeneous legacy architectures with simpler and more direct access modalities. While we do note focus on cross-ISA differences in this paper, we characterize hardware variability in both Xeon Phi and Xeon server architectures.

3.4 Determining Impact of Configurations

The push towards exascale computing has been sufficiently challenging to drive innovations across many layers of the system stack. One of the primary reasons has been the need for better energy efficiency compared to recent generations of HPC systems. To this end, a significant focus has been placed on solutions that limit power consumption. Some techniques throttle resources such as CPU frequency at opportune times [31, 33], usually in response to observed workload imbalance. Other techniques incorporate power budgets for applications [27] and enforce them via node-level tuning mechanisms, such as Intel's RAPL [8]. Beyond directly controlling power consumption, others have proposed ways to improve energy efficiency by more intelligently scheduling workloads to share resources and limit data movement, e.g., by co-running components of a job *in situ* [20] on the same nodes.

Characterizing performance variability across multiple system configurations will allow the analysis of the impact of these options on program behavior. We note that such comparisons are not

only useful for understanding whether a particular configuration increases or decreases variability, but also whether other statistical characteristics – such as the prevalence of temporal variability – change as a result of the configuration. Because it is computationally expensive to perform large scale application runs, we believe characterizing variability at small scale to project large scale impact is a key capability of our framework.

4 VARBENCH

In this section we introduce a new performance analysis framework, *varbench*. Varbench's design was guided by three high level goals: (1) to provide a set of computational kernels to measure performance variability on a platform, and to be easily extensible to incorporate new kernels as required by users; (2) to collect precise measurements of variability as needed to perform detailed characterization - e.g., to characterize spatial as well as temporal variability; and (3) to provide a common set of performance measurements with which to compare the prevalence of variability across different workloads and platforms. We will first discuss the core methodology used to design and implement varbench, and then illustrate the computational kernels currently provided and used for the analysis in this paper.

4.1 Methodology

Varbench is designed to behave in a similar manner to Bulk Synchronous Parallel (BSP) applications; that is, each workload alternates between (i) concurrent computation across all parallel processes and (ii) global synchronization operations. In general, during concurrent computational periods, each processor in the system performs the same set of operations on a different piece of data, and thus there are no synchronization or message passing operations in between global synchronization points, except for workloads that are designed to measure variability of message passing.

Varbench applications have three logical components: *kernels*, *instances*, and *iterations*. A *kernel* is the singular workload running during the course of the application. Section 4.2 will discuss the kernels used for this evaluation. An *instance* can be thought of as a "rank" in traditional MPI terminology. We thus define concepts such as spatial and temporal variability based on the degree to which performance varies across instances (spatial) at a single point in time, or within the same instance over time (temporal). Finally, *iterations* are recurring invocations of a kernel across all instances in the machine. Each iteration performs the exact same set of operations as every other iteration. As discussed above, within an iteration there is no cross-instance communication, and there are no cross-instance synchronization methods.[1] Iterations thus allow us to study the manifestation of temporal variability.

4.2 Kernels

A varbench *kernel* is the singular workload running on the machine. Our vision is that each kernel in the framework will stress a particular component of an HPC platform. These components may take the form of specific architectural resources (e.g., the last level cache), or more broadly represent important workload requirements (e.g., message passing over an interconnect, local I/O, etc.). We envision that users can contribute workload(s) to the framework that reflect the characteristics of the application(s) they care about, and thus gain the characterization benefits of the framework.

[1]Or, more precisely, there are no **explicit** communications or synchronizations. The underlying architecture may impose them implicitly (e.g., cache coherence operations)

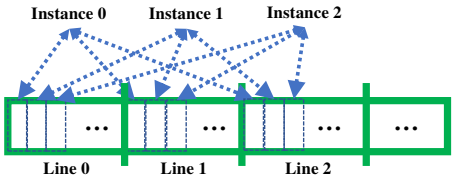


Figure 1: The Cache False Sharing kernel

In this paper we chose to focus specifically on a set of kernels that stress low level architectural resources. As discussed previously, architectural diversity is growing in HPC machines, both as a result of resource heterogeneity as well as the focus on power and energy efficiency. Our analysis is thus to be understood as an illustration of how varbench can measure and characterize a specific type of variability, that which is intrinsically hardware induced, and to demonstrate interesting lessons learned from that analysis, rather than a comprehensive analysis of all forms of variability.

Each of the kernels we designed stresses a different low level architectural feature. We focus our attention on last level caches (LLC), memory subsystems, and on-chip interconnect networks, as these are the resources most commonly shared by threads on modern architectures and thus illustrate key differences across architectures. We now describe the five kernels used for our evaluation.

Cache False Sharing is designed to determine the impact of cache coherence traffic required to share cache lines among instances. At a high-level, this kernel measures the impact of frequent coherence traffic across inter-processor interconnects. The kernel is illustrated in Figure 1. One instance allocates an array that fits entirely in the LLC, and every other instance maps this array into its address space. For each cache line that stores the array's contents, every instance "owns" a particular byte with that line. For each byte that it owns, an instance walks through the array and, with equal probability, either reads or writes a value from/to that particular byte.

Cache Capacity is designed to measure the impact of parallel memory requests bringing data from different cores into the LLC. There is no sharing of cache lines between instances. Instead, each instance allocates its own private array such that the sum of all array sizes is equal to two times the LLC capacity. Then, each instance iterates through its array and alternates between reading and writing each consecutive byte to generate parallel traffic to the memory system via capacity misses.

Random Access is designed similarly to the Random Access benchmark from the HPCC suite [22]. In our system, each instance executes its own private version of the HPCC Random Access algorithm with no explicit sharing of data. This kernel is a common indicator of scalability for HPC workloads.

Stream is designed similarly to the STREAM benchmark from the HPCC suite, a benchmark that measures sustainable memory bandwidth in an architecture [22]. As in the case of Random Access, each instance executes its own private version of Stream without explicit

RV Statistic	Interpretation
$2 < \|RV(S)\| < 4$	Mesokurtic, with single mode
$0 < \|RV(S)\| < 2$	Platykurtic, with broad/multiple modes
$RV(S) > 4$	Leptokurtic, with "slow" outliers
$RV(S) < -4$	Leptokurtic, with "fast" outliers

Table 2: Interpreting the RV statistic of a sample

sharing of data. This kernel is designed to measure the impact of contention for memory controllers among many processors.

Dgemm is designed to measure the performance of matrix-matrix multiplication, a common HPC workload. Dgemm is also provided by the HPCC suite, and as in the Random Access and Stream cases, our implementation consists of private Dgemm executions in each instance with no explicit sharing. This kernel is designed to be a more realistic HPC workload which stresses resources in several resources and subsystems.

5 STATISTICAL METHODOLOGY

This section describes the statistical methodology that supports characterization of variability. A major focus of our methodology is to distinguish between spatial and temporal variability in a kernel. While both classes of variability are damaging to application performance and energy efficiency, the implications that each has for system software and applications on exascale systems are very different. For example, workloads that generate spatial variability – different processors have difference performance – but for which performance within a processor is consistent over time can, in principle, be statically characterized and mitigated in a more straightforward fashion than kernels that generate greater temporal variability. This section will describe our statistical formulations for these classes of variability.

5.1 Quantifying Variability

Our approach is to understand performance measurements as samples drawn from an underlying probability distribution. Depending on the class of variability in question, the sample is selected in different ways. When analyzing temporal variability, we select one sample per instance that consists of performance measurements for that instance across all iterations. To analyze spatial variability, we select one sample per iteration that consists of performance measurements during that iteration across all instances.

With data organized in this manner, we consider the problem of characterizing distributions to provide answers to our questions. In order to quantify the shape of a distribution, we utilize the moments of the distribution, namely the skewness and kurtosis, to create a single metric with which to reason about the shape of a sample. We call this metric the `Resource Variability (RV)` statistic, and define it in Equation 1:

$$RV(S) = \left\{ \begin{array}{ll} -1 * Kurtosis(S), & \text{if } Skewness(S) < 0 \\ Kurtosis(S), & \text{if } Skewness(S) \geq 0 \end{array} \right\} \quad (1)$$

Kurtosis is a measure of tail extremity, and thus indicates the degree to which sample variance is driven by the presence of outliers [35]. A sample with "large" kurtosis has infrequent but extreme outliers, while a sample with "small" kurtosis does not produce such outliers. "Large" and "small" are defined in the context of the normal distribution which has kurtosis of 3. Samples with kurtosis less than, greater than, or equal to 3 are called *platykurtic*, *leptokurtic*, and *mesokurtic*, respectively. Skewness, on the other hand, is a measure of the (a)symmetry of a distribution, and can be used to determine whether the left hand side tail (shorter runtime) is longer, shorter, or similar in length to the right hand tail (longer runtime).

With this understanding, Table 2 demonstrates how to interpret the RV statistic. Samples with $2 < |RV(S)| < 4$ have similar tail extremity to the normal distribution; this does not mean the samples are normally distributed, but rather that they have a normal distribution of extreme outliers. Samples with $0 < |RV(S)| < 2$ have few outliers, while those with $|RV(S)| > 4$ have infrequent but extreme outliers. We note that these interpretations do not represent specific inflection points in a rigid mathematical sense, but rather are simple guidelines for interpreting the statistic.

This metric is valuable because it gives a simple way to shape a relevant sample in order to answer the questions we are interested in. For example, to determine the extent to which "slow" temporal outliers exist in a system - that is, individual instances that have rare slow iterations - we can examine RV ratings for all instances in a run and look for those with values greater than 4. We can also use the statistic to determine the prevalence of slow processors by calculating the RV stat over an iteration rather than an instance.

5.2 Selecting Representative Data Samples

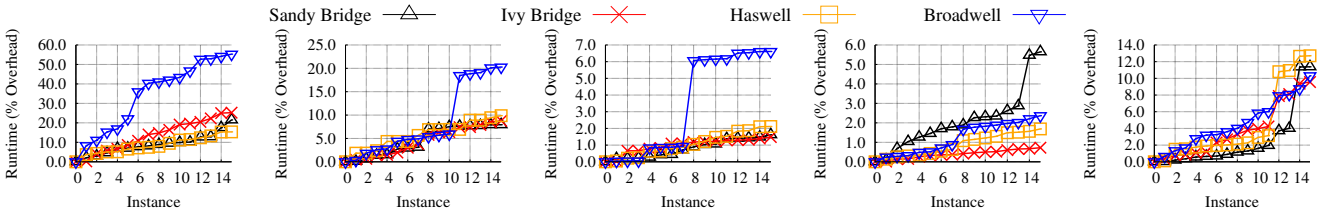
Finally, when selecting samples to study spatial variability, there is the challenge of selecting data from a single iteration of the kernel. Because we do not want to capture temporal effects in this sample, we cannot simply select all data across all processors and iterations. To select a single iteration to analyze, we formulate the notion of the *most representative iteration*, which in a statistical sense is the iteration whose performance distribution is closest (has the least distance) from the remaining unselected iterations.

To select this iteration, we calculate the Kolmogorov-Smirnov statistic [7] for each iteration in the kernel, and then perform pairwise invocations of the Kolmogorov-Smirnov test to determine whether or not a pair of iterations varies at the 95% confidence interval. To select the most representative iteration, we select the iteration that varies from the fewest number of remaining iterations.

6 PERFORMANCE ANALYSIS

The primary goal of our analysis is to demonstrate that our approach to characterizing variability can produce interesting insights and provides some of the benefits discussed in Section 3. To this end, we performed experimental analyses in several different architectures and system configurations. In each case, we provide statistical characterizations and describe how the results give useful information for hardware, system software, and applications developers on future platforms. We note that these experiments are not designed to be an exhaustive list of our framework's capabilities, but to demonstrate some examples of how it can be used and lessons we learned in the process.

Our first set of experiments were designed to study the prevalence of variability across different generations of Intel Xeon server architectures. These experiments, discussed in Section 6.1, lead to two interesting observations. The first is that cross-core spatial



(a) Cache Capacity (b) Cache False Sharing (c) Random Access (d) Stream (e) Dgemm

Figure 2: Spatial Variability of Hardware Resource Performance in Different Xeon Architectures

Processor Codename	Node Characteristics	Year Released
Intel "Sandy Bridge"	Dual socket; 6 cores (12 HT) @ 2.2 GHz 12 GB RAM per socket 32 KB L1(i+d), 256 KB L2, 15 MB shared L3	2009
Intel "Ivy Bridge"	Dual socket; 6 cores (12 HT) @ 2.1 GHz 16 GB RAM per socket 32 KB L1(i+d), 256 KB L2, 15 MB shared L3	2013
Intel "Haswell"	Dual socket; 12 cores (24 HT) @ 2.3 GHz 64 GB RAM per socket 32 KB L1(i+d), 256 KB L2, 30 MB shared L3	2013
Intel "Broadwell"	Dual socket; 18 cores (36 HT) @ 2.1 GHz 64 GB RAM per socket 32 KB L1(i+d), 256 KB L2, 45 MB shared L3	2015

Table 3: Characteristics of Xeon Server Architectures

variability has generally become more pronounced in newer, more recent architectures. The second is that in many cases hardware performance is temporally variant, meaning that past optimizations based on static characterizations of processor variation are not likely to mitigate this class of variability. Our second set of experiments were designed in a similar fashion to understand how these findings relate to many-core heterogeneous architectures. These experiments, presented in Section 6.2, illustrate that spatial variability is even more pronounced than in homogeneous architectures, and again that for some classes of workloads performance can vary greatly over the lifetime of an application.

Finally, our last set of experiments were designed to demonstrate the usefulness of characterization for studying the impact of tunable system configurations. As discussed, at exascale we expect a larger set of possible configuration options, and in general uncertainty regarding how to best administer the platform to meet global system objectives. In this section, we focus on one example in the form of power capping, showing that different configurations can greatly influence the prevalence and shape of variability, and thus the scalability of the system environment. Our analysis echoes some results found previously [30], but further demonstrates non-determinism in the power capping mechanism itself, a finding enabled by our detailed characterization of temporal resource variability.

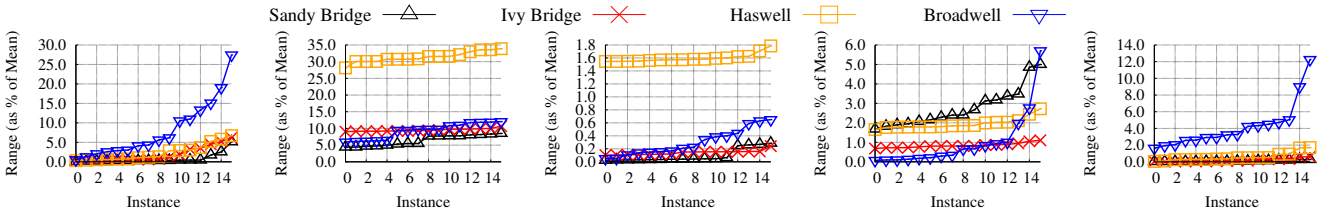
6.1 Variability In Different Xeon Generations

Our first set of experiments was designed to demonstrate the prevalence of variability on a set of recent Xeon server architectures. For these experiments we executed each varbench kernel across the four processor architectures listed in Table 3. Each kernel was executed for a period of 100 iterations using 16 instances. Each instance was pinned to a specific hardware thread on the system, with both hyperthreads of each core utilized by a separate instance. In all architectures, 8 instances are executed on each of the two sockets on the machine, and TurboBoost is disabled to limit the impact of frequency throttling.

We first analyze the presence of spatial variability across each instance. Figure 2 illustrates spatial variability for each kernel and architecture, showing performance during the most representative iteration (see Section 5.2) from each experiment. Each figure plots the runtime degradation as a percentage for each instance compared to the "fastest" instance on the machine. For cache intensive and random access workloads, spatial variability is most pronounced in the most recent architecture we tested ("Broadwell"). For both cache-intensive kernels, variability is significantly greater (30% and 15%, respectively) than the worst performing instances on any other architecture. Furthermore, in both cases there is a noticeable drop in performance for about half of the instances on the machine. In the Cache Capacity kernel, this indicates a lack of consistency in LLC eviction operations, while in the Cache False Sharing case this reflects variability in the QPI layer which forwards cache coherence traffic between sockets. Random Access exhibits similar degradation, showing that half of the instances experience about 6% degradation compared to the faster instances. Degradation is similar across all architectures for Dgemm, which does not stress shared architectural resources outside of the core to the same degree as the other kernels, indicating that spatial variability is largely a function of parallel access to shared resources.

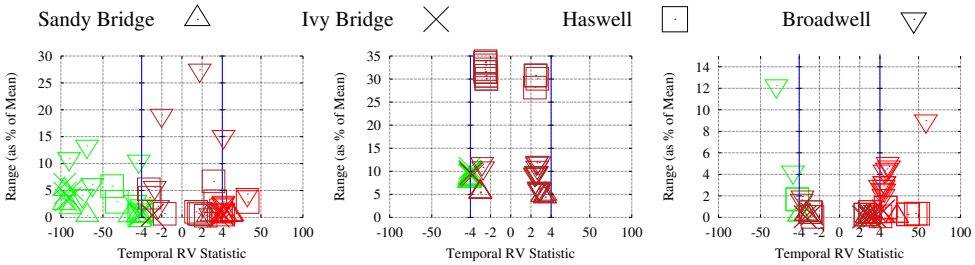
On the other hand, Figure 3 illustrates temporal variability on these architectures by plotting each instance's performance range (max minus min runtime) normalized to its mean across all iterations of the kernel. First, this figure again shows that more recent architectures[2] tend to exhibit much greater temporal variability than their earlier counterparts. This is true for all kernels, with Stream the only case of similar performance across all workloads. To further understand the nature of temporal variability on these workloads, Figure 4 plots the resource variance (RV, see Section 5.1) statistic for each instance against its range, showing the three kernels that exhibited the greatest propensity for temporal variability. Each architecture is represented with a different point type, while colors reflect different ranges of the RV statistic according to the interpretations in Table 2: green: $-100 < RV < -4$; maroon: $-4 \leq RV \geq 4$; red: $4 < RV < 100$. This illustration is designed to indicate

[2] "Haswell" and "Broadwell" are similar processor models, with the latter mainly a die-shrink of the former



(a) Cache Capacity (b) Cache False Sharing (c) Random Access (d) Stream (e) Dgemm

Figure 3: Temporal Variability of Hardware Resource Performance in Different Xeon Architectures



(a) Cache Capacity (b) Cache False Sharing (c) Dgemm

Figure 4: Distribution of the Temporal RV Statistic in Different Xeon Architectures

whether large instance ranges are a result of rare or transient "outliers" that extend the performance range, or whether performance is more broadly distributed across the range.

There are two general takeaways from Figure 4. The first is that Dgemm is the only kernel where large instance ranges can be attributed to "slow" outliers. Interestingly, this kernel shows that two instances actually exhibit "fast" outliers with occasional iterations that execute more quickly than most. In combination with the presence of "slow" outliers this indicates some instances are usually, but not always, given preferential access to some resource that is critical to the workload. In neither of the cache intensive kernels did either process exhibit slow outliers. These results strongly suggest that, when sharing resources of the architecture, variability does not generate "slow" outliers in a fashion similar to, e.g., OS interference, but rather has a more consistent impact on the performance of an instance that can probably not be eliminated by the system software. The second key result of this figure is that the Sandy Bridge and Ivy Bridge architectures generated only modest amounts of temporal variability in these workloads. While this is not an exhaustive set of kernels, this result does indicate that temporal variability may be a fairly recent development that has not been as big of an issue on past systems.

Finally, in order to understand the implications of small scale variability on actual performance, we measured the runtime of each of these kernels on the same Broadwell architecture using a 512-node cluster located at Sandia National Laboratories. Figure 5 demonstrates the results. As the Figure shows, the kernels that exhibit the worst performance at scale are those that experienced the highest degree of temporal variability on a single node (Cache Capacity and Dgemm; note that high temporal variability was only found for Cache False Sharing on Haswell). This indicates that high

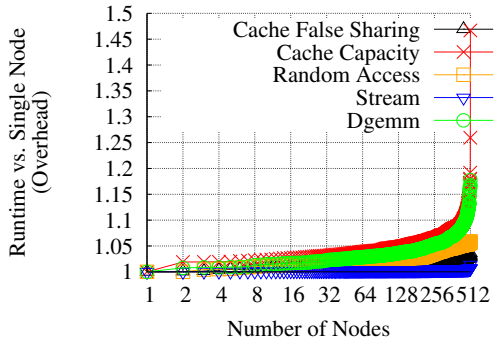
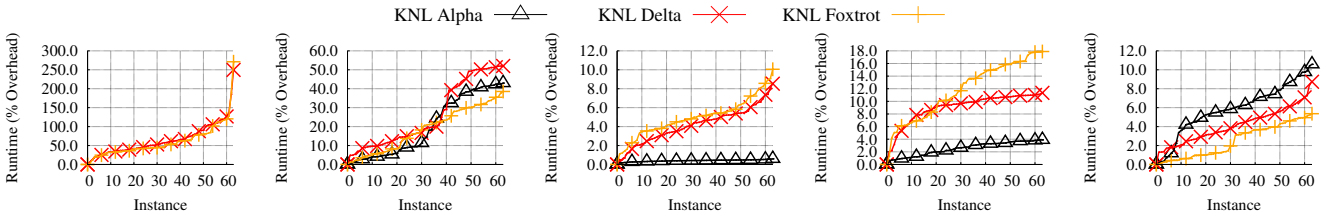


Figure 5: Scalability of Computational Kernels

temporal variability is a better indicator, for these workloads, of scalability than spatial variability. This illustrates that our framework is able to provide an indication of performance scaling via precise single node variability characterization.

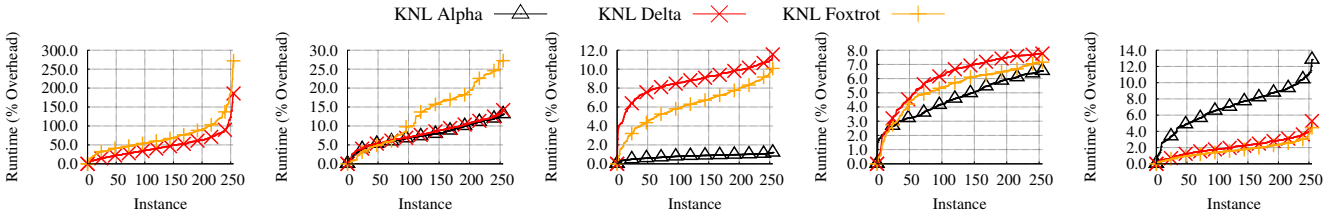
6.2 Variability in Different KNL Configurations

Our next set of experiments demonstrate the prevalence of hardware induced variability on a heterogeneous many-core architecture, the Intel Xeon Phi [32]. We are interested in understanding the nature of both spatial and temporal variability on this architecture, particularly as they compare to the more homogeneous dual-socket architectures studied in the previous section. However, in addition to understanding how variability evolves when moving to a many-core architecture, this section also sheds light on how different configurations of the systems resources, including configuration of



(a) Cache Capacity (b) Cache False Sharing (c) Random Access (d) Stream (e) Dgemm

Figure 6: Spatial Variability of Performance in Different KNL Configurations Using 64 Instances (2 per Tile)



(a) Cache Capacity (b) Cache False Sharing (c) Random Access (d) Stream (e) Dgemm

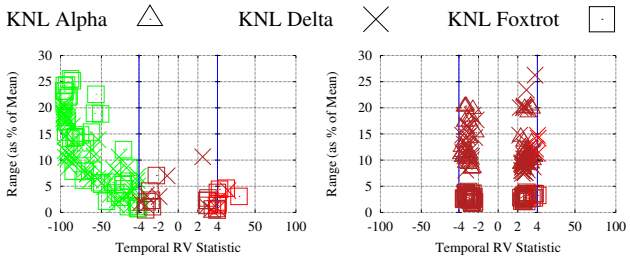
Figure 7: Spatial Variability of Performance in Different KNL Configurations Using 256 Instances (8 per Tile)

# Instances	# Tiles	# Cores / Tile	# Threads / Core
64	32	2	1
256	32	2	4

Table 4: Mapping of Instances to the KNL Architecture

Alias	Clustering Mode	MCDRAM Configuration
"KNL Alpha"	Quadrant Mode	Flat (Only uses DDR4)
"KNL Delta"	Quadrant Mode	100% Cache
"KNL Foxtrot"	Sub NUMA Clustering (SNC=4)	100% Cache

Table 5: KNL Configurations Analyzed



(a) Cache Capacity (b) Cache False Sharing

Figure 8: Distribution of the Temporal RV Statistic in the Cache Sensitive Kernels in Different KNL Configurations

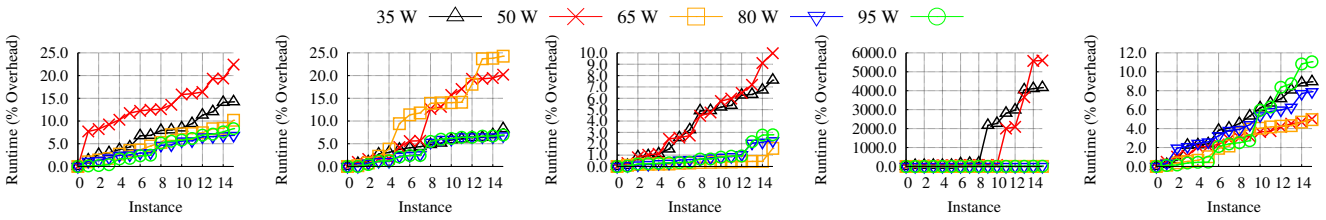
the MCDRAM memory, as well as different strategies for mapping instances to the architecture's hardware threads, impact variability.

Tables 4 and 5 show the configurations we evaluated.[3] As in the previous experiments, we first analyze the spatial variability exhibited by the architecture. Figure 6 shows the results when with 64 instances on the architecture, using 64 different cores and only 1 SMT thread per core as shown in Table 4, again plotting the most representative iteration from each kernel and demonstrating the runtime degradation experienced by all instances compared to the fastest instance. In comparison to the Xeon results (Figure 2), all kernels exhibit a greater degree of spatial variability in the worst case, but the cache sensitive kernels in particular demonstrate very significant degradation, ranging from only a few percent in some instances to over 40% in Cache False Sharing, and over 200% in Cache Capacity on the slowest instance. While Random Access, Stream, and Dgemm all exhibit some spatial variability, the cache sensitive kernels exhibit much more. This evidence indicates that coherence traffic and routing of memory requests across the KNL 2d-mesh interconnect is a major source of variability.

Figure 7 illustrates the spatial variability results in the configuration where, instead of only using 1 SMT thread per core, all 4 SMT threads are utilized and 256 instances in total execute in parallel on the architecture. The most interesting result from this figure, in comparison to the 64 instance experiments, is that spatial variability actually trends at about the same level, and in some cases reduces. This behavior is most pronounced in both of the "KNL Alpha" and "KNL Delta" configurations in the Cache False Sharing kernel. This result is consistent with the conclusion that coherence traffic across the distributed cross-tile architecture is the primary driver of variability, as the more dense packing of the architecture likely creates tile-local resource contention that, to a degree, masks the impact of variability on the interconnect.

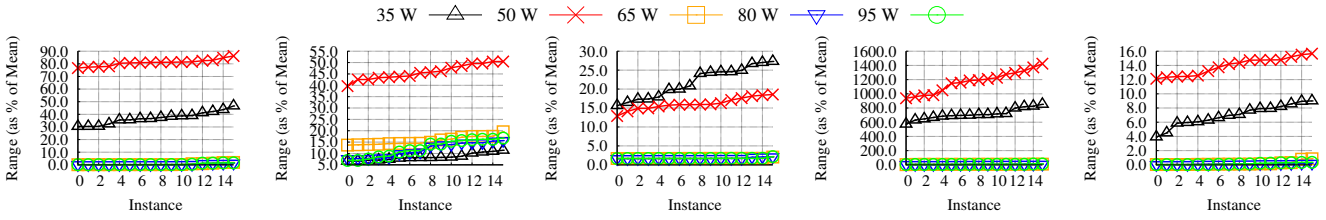
Finally, Figure 8 illustrates the distribution of the per instance temporal resource variance statistic in the 64 instance (2 per tile) KNL experiments. For space reasons, this figure only illustrates the

[3]In this section, we do not have "KNL Alpha" results for Cache Capacity, as the MCDRAM memory is not configured as cache and thus the LLC is actually the per-tile L2 cache, making a direct comparison with "Delta" and 'Foxtrot" challenging. In all KNL configurations, we obtain Cache False Sharing results by configuring the kernel to use the per-core L2 caches, rather than the MCDRAM cache.



(a) Cache Capacity (b) Cache False Sharing (c) Random Access (d) Stream (e) Dgemm

Figure 9: Spatial Variability of Hardware Resource Performance as a Function of Different RAPL Power Caps



(a) Cache Capacity (b) Cache False Sharing (c) Random Access (d) Stream (e) Dgemm

Figure 10: Temporal Variability of Hardware Resource Performance as a Function of Different RAPL Power Caps

distribution for the cache sensitive kernels, which exhibited the highest temporal variance of all five kernels. As in the Xeon dual-socket experiments, we again see that temporal variability is almost never characterized as being driven by slow outliers, which suggests the variability is a result of intrinsic hardware characteristics and not transient software effects. However, in comparison to the large increase of spatial variability compared to the Xeon platforms, the degree of temporal variability is more modest, particularly in the Cache Capacity example, where no instance's performance varies by more than 30%, but for which the difference between the fastest and slowest ranks is nearly 300%. This is consistent with our finding that, in comparison to the Xeon architectures, heterogeneity primarily exacerbates spatial variability.

6.3 Impact of Power Capping

Our last set of experiments were performed to demonstrate an additional benefit of characterization beyond performing architectural comparisons and understanding trends, which is to understand how different tunable system criteria impact variability and influence the scalability of the platform. In this section, we provide one example of this by studying the impact of power capping via the Intel RAPL [8] interface. Rather than provide cross-architectural comparisons, we study the prevalence of variability under different power caps and highlight the key differences as they pertain to the likely scalability of the system.

These experiments were performed on the Intel "Ivy Bridge" architecture shown in Table 3. We ran each computational kernel with differing package level power caps of 95, 80, 65, 50, and 35 W. First considering spatial variability, as shown in Figure 9, for all kernels there is no significant difference between the 95 W and 80 W power caps. However, the 65 W cap leads to a larger degree of variability in the Cache False Sharing kernel, while in all kernels but Dgemm the 50 and 35 W power caps exhibit more pronounced variability. This result reflects the fact that manufacturing variation, which without power caps is generally imperceptible from a performance standpoint but which causes different levels of power consumption across processors, becomes inverted under a power cap, with each processor performing differently at the same power level.

However, by also focusing on temporal variability, we are able to understand an additional result that has different implications for scalability and potential mitigation than simply spatial variation. As Figure 10 shows, in all cases where there is significant spatial variability there is also significant temporal variability. In fact, under the 35 and 50 W power caps, performance within a specific instance actually varies more over time than performance across processors. This demonstrates the interesting fact that non-determinism within the RAPL logic actually creates more variability than the cross-core processor variation that generates spatial imbalance.

6.4 Lessons Learned

As discussed at the beginning of our evaluation, the purpose of these experiments was to demonstrate that characterization provides interesting insights and to enable architectural comparisons, help identify trends, and in general provide a common framework with which to evaluate tunable system criteria that can provide indications of large scale system behavior without requiring large scale performance results. We conclude with our primary findings:

(1) Due to the presence of intrinsically shared resources such as caches, buses, and on chip interconnects, architectures often generate variability even for singular workloads, without interference from co-running applications.
(2) Heterogeneity creates more pronounced "hot spots" that exacerbate cross-core spatial variability. Temporal variability is not exacerbated to the same degree as spatial variability.

(3) In all kernels except for Dgemm, temporal performance variability is not characterized as having "slow outliers." While we cannot fully preclude the OS as a source of overhead, it is unlikely that software creates this variability.
(4) Power capping can have subtle but important implications for scalability. While runtime may not vary drastically from, e.g., 50 W to 65 W, the former executes much more variably.

7 CONCLUSION

Variability is a major challenge for large scale HPC systems due the prevalence of BSP applications. In this paper, we argued that characterizing variability can provide numerous benefits for hardware and software design on future machines. To this end, we designed and implemented the varbench experimental performance analysis framework. We demonstrated several of varbench's capabilities, including its utility for quantifying architectural trends and enabling cross-architectural comparisons. Finally, by studying key statistical properties of performance distributions, we showed how varbench sheds light on the impact of system parameters such as power caps.

REFERENCES

[1] 2018. Top500: The List. https://www.top500.org. (2018). Online, Accessed: 2018-01-24.
[2] Martin Abadi et al. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. In *Proc. of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.
[3] Michael Adams, Phillip Colella, Dan Graves, Hans Johansen, N.D Keen, Terry Ligocki, Dan Martin, Peter McCorquodale, D. Modiano, and Peter Schwartz. 2013. *Chombo Software Package for AMR Applications - Design Document.*
[4] Abhinav Bhatele, Kathryn Mohror, Steven Langer, and Katherine Isaacs. 2013. There Goes the Neighborhood: Performance Degradation due to Nearby Jobs. In *Proc. of the 25th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*.
[5] Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. 2017. Run-to-run Variability on Xeon Phi based Cray XC Systems. In *Proc. of the 29th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*.
[6] Rafael da Silva, Rosa Filgueira, Ilia Pietri, Ming Jiang, Rizos Sakellariou, and Ewa Deelman. 2017. A Characterization of Workflow Management Systems for Extreme-scale Applications. *Future Generation Computer Systems* 75 (2017), 228–238.
[7] Donald Darling. 1957. The Kolmogorov-Smirnov, Cramer-von Mises Tests. *The Annals of Mathematical Statistics* 28, 4 (1957), 823 – 838.
[8] Howard David, Eugene Gorbatov, Ulf Hanebutte, Rahul Knanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *Proc. of the ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED '10)*.
[9] Saurabh Dighe, Sriram Vangal, Paolo Aseron, Shasi Kumar, Tiju Jacob, Keith Bowman, Jason Howard, James Tschanz, Vasantha Erraguntla, Nitin Borkar, Vivek De, and Shekhar Borkar. 2011. Within-Die Variation-Aware Dynamic-Voltage-Frequency-Scaling With Optimal Core Allocation and Thread Hopping for the 80-Core TeraFLOPS Processor. *IEEE Journal of Solid-State Circuits* 46, 1 (2011), 184–193.
[10] Kristof Du Bois, Stijn Eyerman, Jennifer Sartor, and Lieven Eeckhout. 2013. Criticality Stacks: Identifying Critical Threads in Parallel Programs using Synchronization Behavior. In *Proc. of the 40th International Symposium on Computer Architecture (ISCA '13)*.
[11] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert Wisniewski. 2010. Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK. In *Proc. of the 23rd ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*.
[12] Joe Jeddeloh and Brent Keeth. 2012. Hybrid Memory Cube: New DRAM Architecture Increases Density and Performance. In *Proc. of the 2012 Symposium on VLSI Technology (VLSIT '12)*.
[13] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. 2009. ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *Proc. of the International Conference on Parallel Processing Workshops (ICPPW '09)*.
[14] Laxmikant Kale and Gengbin Zheng. 2009. *Advanced Computational Infrastructures for Parallel and Distributed Applications.* Wiley, Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects.
[15] Brian Kocoloski, Leonardo Piga, Wei Huang, Indrani Paul, and John Lange. 2016. A Case for Criticality Models in Exascale Systems. In *Proc. of the 18th IEEE International Conference on Cluster Computing (CLUSTER '16)*.
[16] Wim Lavrijsen, Costin Iancu, Wibe de Jong, Xin Chen, and Karsten Schwan. 2016. Exploiting Variability for Energy Optimization in Parallel Programs. In *Proc. of the Eleventh European Conference on Computer Systems (EuroSys '16)*.
[17] Edgar Leon, Ian Karlin, and Adam Moody. 2016. System Noise Revisited: Enabling Application Scalability and Reproducibility with SMT. In *Proc. of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS '16)*.
[18] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proc. of the 9th European Conference on Computer System (EuroSys '14)*.
[19] Chee Liew, Malcolm Atkinson, Michelle Galea, Tan Ang, Paul Martin, and Jano Van Hemert. 2017. Scientific Workflows: Moving Across Paradigms. *Comput. Surveys* 49, 4 (2017).
[20] Jiaqi Liu and Gagan Agrawal. 2017. Supporting Fault-Tolerance in Presence of In-Situ Analytics. In *Proc. of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '17)*.
[21] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. 2010. Managing Variability in the IO Performance of Petascale Storage Systems. In *Proc. of the 22nd Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*.
[22] Piotr Luszczek, Jack Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. 2005. *Introduction to the HPCChallenge Benchmark Suite.* Technical Report. University of Tennessee.
[23] Grzegorz Malewicz, Matthew Austern, Aart Bik, James Denhert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a System for Large-Scale Graph Processing. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*.
[24] Maxime Martinasso and Jean-Francois Mehaut. 2-11. A Contention-Aware Performance Model for HPC-Based Networks: A Case Study of the Infiniband Network. *Lecture Notes in Computer Science* 6852 (2-11), 91–102.
[25] Hiroyuki Miyazaki, Yoshihiro Kusano, Naoki Shinjou, Fumiyoshi Shoji, Mitsuo Yokokawa, and Tadashi Watanabe. 2012. Overview of the K computer System. *Scitech* 48, 3 (2012), 255–265.
[26] Jiannan Ouyang, Brian Kocoloski, John Lange, and Kevin Pedretti. 2015. Achieving Performance Isolation with Lightweight Co-kernels. In *Proc. of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*.
[27] Tapasya Patki, David Lowenthal, Anjana Sasidharan, Matthias Maiterth, Barry L. Rountree, Martin Schulz, and Bronis de Supinski. 2015. Practical Resource Management in Power-Constrained, High Performance Computing. In *Proc. of the 24th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*.
[28] Bogdan Prisacari, German Rodriguez, Philip Hiedelberger, Dong Chen, Cyriel Minkenberg, and Torsten Hoefler. 2014. Efficient Task Placement and Routing of Nearest Neighbor Exchanges in Dragonfly Networks. In *Proc. of 23rd ACM International Symposium on High Performance Parallel and Distributed Computing (HPDC '14)*.
[29] Nikola Rajovic, Paul Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. 2013. Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC?. In *Proc. of the 26th ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*.
[30] Barry Rountree, Dong Ahn, Bronis de Supinski, David Lowenthal, and Martin Schulz. 2012. Beyond DVFS: A First Look at Performance Under a Hardware-Enforced Power Bound. In *Proc. of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '12)*.
[31] Barry Rountree, David Lowenthal, Bronis de Supinski, Martin Schulz, Vincent Freeh, and T. Bletsch. 2009. Adagio: Making DVS Practical for Complex HPC Applications. In *Proc. of the 23rd ACM International Conference on Supercomputing (ICS '09)*.
[32] Avinash Sodani. 2015. Knight's Landing KNL: 2nd Generation Intel Xeon Phi Processor. In *Proc. of the IEEE Symposium on High Performance Chips (HC27)*.
[33] Akshay Venkatesh, Abhinav Vishnu, Khaled Hamidouche, Nathan Tallent, Dhabaleswar Panda, Darren Kerbyson, and Adolfy Hoisie. 2015. A Case for Application-oblivious Energy-efficient MPI Runtime. In *Proc. of the 27th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*.
[34] Hannes Weisbach, Balazs Gerofi, Brian Kocoloski, Hermann Härtig, and Yutaka Ishikawa. 2018. Hardware Performance Variation: A Comparative Study using Lightweight Kernels. In *Proc. of the International Conference, ISC High Performance (ISC HPC '18)*.
[35] Peter Westfall. 2014. Kurtosis as Peakedness, 1905 - 2014. R.I.P. *The American Statistician* 68, 3 (2014), 191–195.