SCALABILITY IN THE PRESENCE OF VARIABILITY

by

Brian Kocoloski

Bachelor of Science, University of Dayton, 2011

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of Arts and Sciences in partial

fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2017

UNIVERSITY OF PITTSBURGH

KENNETH P. DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Brian Kocoloski

It was defended on

September 22nd 2017

and approved by

Dr. John Lange, Department of Computer Science, University of Pittsburgh

Dr. Bruce Childers, Department of Computer Science, University of Pittsburgh

Dr. Rami Melhem, Department of Computer Science, University of Pittsburgh

Dr. Peter Dinda, Department of Electrical Engineering and Computer Science, Northwestern University

Dissertation Director: Dr. John Lange, Department of Computer Science, University of Pittsburgh

SCALABILITY IN THE PRESENCE OF VARIABILITY

Brian Kocoloski, PhD

University of Pittsburgh, 2017

Supercomputers are used to solve some of the world's most computationally demanding problems. Exascale systems, to be comprised of over one million cores and capable of 10^{18} floating point operations per second, will probably exist by the early 2020s, and will provide unprecedented computational power for parallel computing workloads. Unfortunately, while these machines hold tremendous promise and opportunity for applications in High Performance Computing (HPC), graph processing, and machine learning, it will be a major challenge to fully realize their potential, because to do so requires balanced execution across the entire system and its millions of processing elements. When different processors take different amounts of time to perform the same amount of work, performance imbalance arises, large portions of the system sit idle, and time and energy are wasted. Larger systems incorporate more processors and thus greater opportunity for imbalance to arise, as well as larger performance/energy penalties when it does. This phenomenon is referred to as *performance variability* and is the focus of this dissertation.

In this dissertation, we explain how to design system software to mitigate variability on large scale parallel machines. Our approaches span (1) the design, implementation, and evaluation of a new high performance operating system to reduce some classes of performance variability, (2) a new performance evaluation framework to holistically characterize key features of variability on new and emerging architectures, and (3) a distributed modeling framework that derives predictions of how and where imbalance is manifesting in order to drive reactive operations such as load balancing and speed scaling. Collectively, these efforts provide a holistic set of tools to promote scalability through the mitigation of variability.

TABLE OF CONTENTS

1.0 INTRODUCTION 1

1.1 Performance Variability in Bulk Synchronous Parallel Workloads 2

1.2 Performance Variability at Exascale 6

1.3 Overview: Prevention vs. Detection 7

1.3.1 Preventing Software Induced Variability 9

1.3.2 Detecting and Modeling Temporally Induced Variability 10

1.3.3 Thesis Statement 11

1.4 Research Contributions 12

2.0 PREVENTING SOFTWARE INDUCED VARIABILITY 13

2.1 Hobbes Enclaves 15

2.1.1 Hobbes Building Blocks 16

2.1.1.1 Kitten and Palacios 16

2.1.1.2 Pisces 17

2.2 XEMEM: Efficient Shared Memory for Composed Applications on Multi-OS/R Exascale Systems 17

2.2.1 Multi-OS/R Shared Memory 19

2.2.1.1 Common Global Name Space 19

2.2.1.2 Arbitrary Enclave Topologies 20

2.2.1.3 Dynamic Sharing of Memory Regions 22

2.2.1.4 Localized Address Space Management 23

2.2.2 Implementation 23

2.2.2.1 Isolated Data Planes 24

2.2.2.2 XPMEM . . . 25
2.2.2.3 Shared Memory Protocol . . . 25
2.2.2.4 OS Memory Mapping Routines . . . 26
2.2.2.5 Palacios Host/Guest Memory Sharing . . . 27
2.2.2.6 Cross-Enclave Communication Channels . . . 29
2.2.3 Shared Memory Evaluation . . . 30
2.2.3.1 System Configuration . . . 31
2.2.3.2 Shared Memory Attachment vs. RDMA Throughput . . . 31
2.2.3.3 Scalability of Multi-OS/R Shared Memory . . . 32
2.2.3.4 Performance of Shared Memory Using Virtual Machines . . . 34
2.2.3.5 Operating System Noise . . . 35
2.2.4 Single Node Benchmark Evaluation . . . 37
2.2.4.1 Sample *In Situ* Workload . . . 37
2.2.4.2 Execution and Memory Registration Models . . . 38
2.2.4.3 System Configuration . . . 39
2.2.4.4 Results . . . 39
2.2.5 Multi-Node Benchmark Evaluation . . . 42
2.2.5.1 Workload and System Configuration . . . 42
2.2.5.2 Results . . . 43
2.2.6 XEMEM Summary . . . 44
2.3 Composing Applications across Enclaves via XEMEM . . . 44
2.3.1 Application Composition Use Cases . . . 45
2.3.2 Example Applications . . . 46
2.3.2.1 Crack Detection in Molecular Dynamics Simulations . . . 46
2.3.2.2 Plasma Microturbulence . . . 47
2.3.2.3 Neutronics Energy Spectrum Analysis . . . 49
2.3.3 Evaluation . . . 49
2.3.3.1 LAMMPS . . . 50
2.3.3.2 Gyrokinetic Toroidal Code (GTC) . . . 51
2.4 Related Work . . . 53

2.4.1 Overview of HPC OS/R Architectures . . . 53
2.4.2 Overview of Composition Techniques . . . 53
2.5 Summary . . . 54
3.0 CHARACTERIZING VARIABILITY ON EXASCALE MACHINES . 56
3.1 Variability in Exascale Machines . . . 58
3.1.1 Complex, Heterogeneous and Distributed Node Architectures . . . 59
3.1.2 System Objectives and Compositional Applications . . . 60
3.1.3 Shortcomings of Existing Approaches to Variability . . . 61
3.2 Varbench . . . 62
3.2.1 Core Methodology . . . 63
3.2.2 Quantifying Resource Variability . . . 64
3.2.3 Kernels . . . 66
3.3 Performance Analysis . . . 68
3.3.1 Variability Across Generations of Server Architectures . . . 68
3.3.1.1 Spatial Variability . . . 70
3.3.1.2 Temporal Variability . . . 71
3.3.2 Variability in Many-core Architectures . . . 75
3.3.2.1 Cache Kernels . . . 76
3.3.2.2 Memory Kernels . . . 78
3.3.2.3 Dgemm . . . 80
3.3.3 Variability at Scale . . . 82
3.3.3.1 Infrastructure . . . 82
3.3.3.2 Variability Across Nodes . . . 83
3.3.3.3 Impact of Variability on Runtime . . . 83
3.4 Summary . . . 84
4.0 MODELING VARIABILITY WITH CRITICALITY MODELS . . . 85
4.1 Criticality Models . . . 87
4.1.1 A Case for Criticality Models . . . 87
4.1.2 Statistical Model Training . . . 88
4.1.3 Distributed Model Generation . . . 89

4.1.4 Parallel and Autonomous Model Execution 90
4.2 Proof of Concept: Criticality Modeling on a Small Cluster 91
4.2.1 Performance Measurements and Profiling 91
4.2.1.1 Instruction Based Sampling 91
4.2.2 Constructing the Models 92
4.2.3 Evaluating the Models 94
4.2.3.1 Classification Accuracy 94
4.2.3.2 Generating Rank-Level Predictions 96
4.2.3.3 Comparison with Simpler Spatial Modeling 98
4.2.3.4 Predictive Performance Counters 100
4.3 Related Work 100
4.4 Summary 101
5.0 CONCLUSION 103
BIBLIOGRAPHY 106

LIST OF TABLES

1 Likely sources of performance variability at exascale 8

2 The XPMEM user-level API 24

3 Cross-enclave throughput using shared memory between a Linux process and a native Kitten process executing in a co-kernel enclave. Each value represents the throughput of at least 500 attachments to a 1 GB memory region 34

4 Enclave configurations for the sample *in situ* workload in the single node experiments 39

5 LAMMPS multi-enclave runtimes (s) 51

6 GTC multi-enclave runtimes (s) 52

7 Interpreting the RV statistic of a sample 64

8 Characteristics of examined server architectures 68

9 Spatial resource variability (RV) statistic 71

10 KNL configurations analyzed 75

11 Mapping of Varbench instances to the Knight's Landing architecture 76

12 Prediction frequency based on criticality threshold 96

LIST OF FIGURES

1 Variability in Bulk Synchronous Parallel (BSP) workloads 3
2 BSP in Google's Pregel graph processing library (image reproduced [82]) . . . 4
3 Impact of variability on application runtime in a petascale supercomputer . . 5
4 The Hobbes OS/R supporting an application composed in two enclaves . . . 15
5 Exascale enclave partitioning 18
6 Exascale enclave topology 21
7 Shared memory protocol 26
8 Palacios memory translations for shared memory attachments 28
9 Cross-enclave throughput using shared memory and RDMA Verbs over Infiniband. Each XEMEM data point represents the throughput of 500 attachments for a given size 30
10 Cross-enclave throughput using shared memory between native Linux processes and native Kitten processes in co-kernel enclaves. Each data point represents the throughput of at least 500 attachments for a given size 33
11 Noise profile of a Kitten enclave serving XEMEM attachment requests on a single core 36
12 Performance of a sample *in situ* benchmark on a single node using various communication and execution behaviors 40
13 Performance of a sample *in situ* benchmark on a multi-node cluster using an asynchronous execution model 41
14 Illustration of PreDatA operations on GTC particle data 48
15 Memory routing in a Knight's Landing chip (image reproduced [95]) 59

16 The Cache False Sharing (CFS) kernel 66
17 Spatial variability in various server architectures 69
18 Temporal variability in various server architectures 72
19 Temporal resource variability (RV) statistic 74
20 Variability in cache kernels on Knight's Landing 77
21 Variability in memory subsystem kernels on Knight's Landing 79
22 Variability in Dgemm on Knight's Landing 81
23 Distribution of tail instances in a 512 node machine 82
24 Scalability of Varbench kernels 84
25 A high-level view of criticality models. 86
26 Cumulative sum of MPI slack by imbalance observed in the collective 92
27 Classification accuracies of criticality prediction models built from node-level performance counters. Missing bars indicate that no periods with the specified imbalance occurred in the application. 95
28 Accuracy generating rank-level predictions with criticality models (CM), compared to a simplified model that considers only spatial variability (Spat.). . . 97
29 Information gain from IBS performance events. Larger values better model performance variability. 99

1.0 INTRODUCTION

Supercomputers are used to solve some of the world's most computationally intensive problems. The first exascale supercomputer (capable of 10^{18} floating point operations per second) will probably exist in the early 2020s, and machines of this sort will provide unprecedented potential for computationally intensive workloads.

The only way to utilize machines of this stature is through parallel programming, and the predominant parallel programming model for today's large scale machines is Bulk Synchronous Parallelism (BSP). While this model has long been embraced in the High Performance Computing (HPC) community and its wide array of computational science applications, recent years have seen adoption of BSP in other communities. Several large scale graph processing libraries have adopted BSP models in part because bulk synchronization makes it much simpler to reason about characteristics of parallel algorithms at scale [108, 69]. Furthermore, the machine learning community has begun adopting BSP approaches because some iterative algorithms, including stochastic gradient descent, a key deep learning algorithm, converge more quickly when equal progress across all parallel processors can be achieved [5, 26]. This guarantee is difficult to meet in other more loosely synchronized programming models.

While exascale class systems hold tremendous promise and opportunity for BSP workloads, it will be a major challenge to realize this full potential. These applications perform frequent global communication and synchronization across the system, and thus performance and energy efficiency are largely dictated by the slowest components of the system. When different processors take different amounts of time to perform the same amount of work, performance imbalance arises, large portions of the system sit idle, and time and energy are wasted. This phenomenon is referred to as *performance variability*. While variability

is already a major concern on today's petascale machines - some applications spend over 50% of their execution time "waiting" for slower processors to synchronize with the rest of the system [44] - there are good reasons to believe the situation will be more challenging at exascale.

In this dissertation, we explain how to design system software to mitigate variability on large scale parallel machines. Our approaches span (1) the design, implementation, and evaluation of a new high performance operating system to reduce performance variability induced by commodity system software, (2) a new performance evaluation framework to holistically characterize key features of variability on new and emerging architectures, and (3) a distributed modeling framework that derives predictions of how and where imbalance is manifesting in order to drive reactive operations such as load balancing and speed scaling. Collectively, these efforts provide a holistic set of tools to promote scalability in the presence of variability.

1.1 PERFORMANCE VARIABILITY IN BULK SYNCHRONOUS PARALLEL WORKLOADS

In this dissertation, we study the implications of performance variability for the runtime and energy efficiency of Bulk Synchronous Parallel workloads. In the context of large scale parallel computing workloads, BSP workloads have a defining characteristic: they perform frequent global communication and synchronization operations over all parallel processors in the computation. All processors alternate between two general modes of operation: concurrent computation on local data, and global communication and synchronization operations that exchange data between processors and/or ensure a single view of the global problem's state. A high-level view of these operations is shown in Figure 1. As the figure suggests, applications typically leverage BSP in an iterative fashion by repeatedly performing the same computational operations on a distributed problem state that is refined and exchanged across all processors during each iteration. Historically, such algorithms have been the domain of HPC systems which are primarily devoted to computational science problems [41]. However,

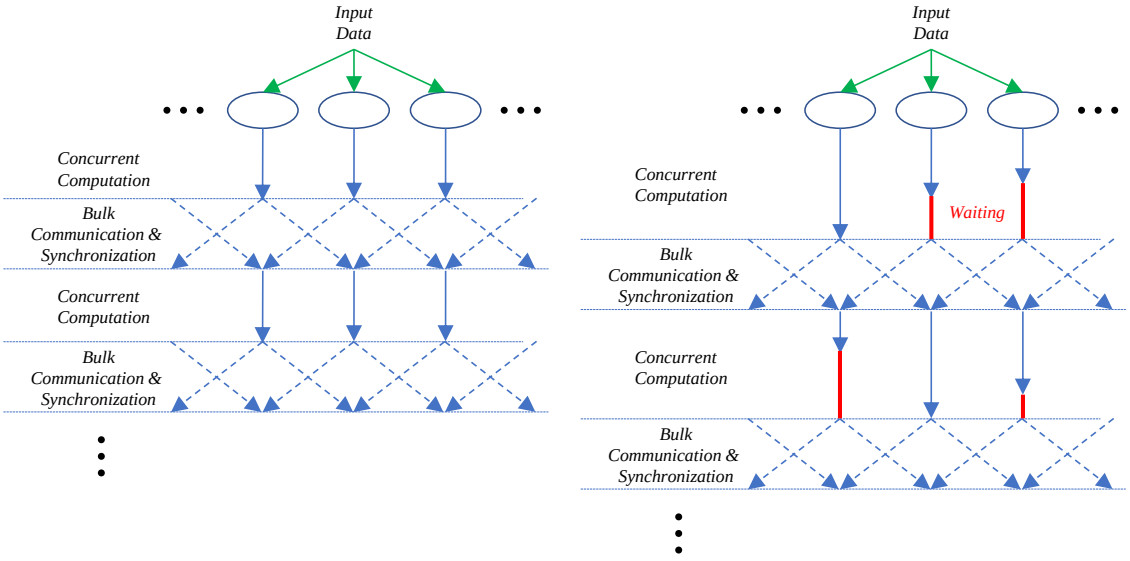


Figure 1: Variability in Bulk Synchronous Parallel (BSP) workloads

in the past couple of decades, a wider array of applications are leveraging BSP, including machine learning workloads, such as deep learning applications based on stochastic gradient descent [5, 26], as well graph processing engines [108, 69] that leverage BSP not in the context of specific algorithms but as central components of their parallel processing frameworks. For example, Google's Pregel library [69], illustrated in Figure 2, operates by mapping computational operations to each vertex in a graph, performs bulk communication by passing messages along the edges of the graph to communicate state between processors, and applies barrier synchronization to ensure message delivery at all vertices before continuing to the next iteration of an algorithm. It is worth noting that Pregel also underlies GraphX [113], the graph processing engine of Apache Spark [114].

With BSP seeing increased adoption in many parallel processing communities, performance variability is a critical issue. Referring back once more to Figure 1, we see a simplified view of what happens when variability impacts a BSP workload. On the left hand side of the figure, each parallel processor takes the same amount of time to perform its concurrent computation. As a result, the global synchronization points are reached at the same time by all processors and the next iteration can begin. In contrast, on the right hand side of the

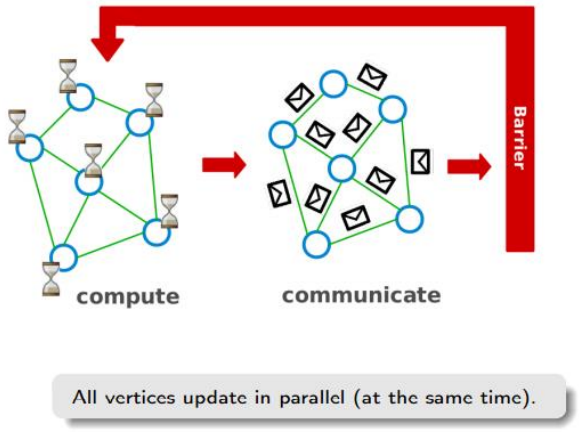


Figure 2: BSP in Google's Pregel graph processing library (image reproduced [82])

figure, one or more processes takes a longer amount of time to complete its computation. Thus, all other processes must wait to exchange messages and synchronize until the slowest processor completes its task. This process leads to wasted energy and prolongs the overall runtime of the application.

Even on today's systems, variability induces significant overhead for BSP workloads. Figure 3 illustrates the problem on a Petascale supercomputer at the National Energy Research Scientific Computing Center (NERSC) [44]. This figure breaks down runtime in a set of representative HPC workloads based on the prevalence of "slack," defined as the percentage of time during which an application's processes are delayed waiting for the slowest processor to synchronize. Each application was executed at three different problem sizes listed as "small," "medium," or "large." For these problem sizes, on the order of 100, 1,000, or 10,000 individual nodes, respectively, were used to run the application. As the Figure shows, when executing at the largest problem size, over 75% of runtime for some of these workloads consists of waiting for slow processors to synchronize, while for all applications slack constitutes at least 10% of total runtime. Given that exascale systems are expected to be comprised of one hundred to one thousand times more processors than this system, this level of performance loss is a major concern.

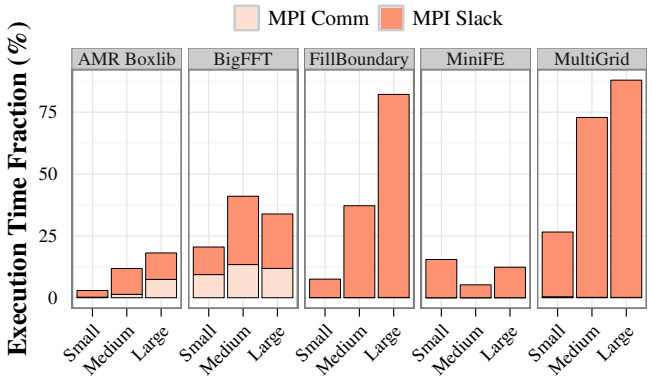


Figure 3: Impact of variability on application runtime in a petascale supercomputer

Given that variability is such a significant problem, many efforts to deal with it have been produced by different research communities. Google documented the extensive efforts it has taken to mitigate variability in its clusters [31], including duplicating workloads across several different machines, detecting and avoiding "slow" machines, and approximating results by proceeding when only a subset of processors have completed their computations. While these efforts were primarily focused on latency-sensitive *data parallel* workloads for which there is little dependence between tasks executing on different nodes, some similar techniques have been in applied in various HPC research communities to explicitly target variability in BSP workloads. New programming languages and parallel runtime systems have been designed to allow processors to efficiently steal work from slower processors that are delaying progress [13, 48, 49]. Lightweight operating systems have been built in order to attempt to reduce reduce the probability that a given processor will be slow by ensuring only the application's processes are scheduled on it [40, 61]. Furthermore, libraries have been designed to reduce energy consumption by lowering processor frequencies on processors that reach synchronization points before their slower counterparts [89, 103].

While these efforts have achieved varying degrees of success on today's machines, they are based on simplifying assumptions of how variability manifests across the processors of a parallel machine. These assumptions include that variability is primarily driven by imperfect distribution of workloads by applications themselves [25]; by "slow" or defective

nodes [31, 106]; by "slow" individual or defective processors [18]; or, by operating system jitter/interference [42, 93]. While these assumptions have been based on experiences with real large scale systems and have lead to practical solutions to dealing with variability in current and past systems, exascale machines are evolving in ways that challenge these assumptions and suggest that new approaches to understanding and managing variability will be needed.

1.2 PERFORMANCE VARIABILITY AT EXASCALE

In this context, there are two broad ways in which exascale systems are evolving. First, node hardware is becoming more heterogeneous, interconnected, and diverse than it has ever been, with nodes consisting of tens to hundreds of cores, multiple memory technologies, and heterogeneous accelerators/co-processors, all interconnected by shared resources such as buses and caches. The adoption of GPUs and many-core processors, such as the Intel Xeon Phi, are a significant departure from homogeneous, commodity architectures that have comprised recent generations of supercomputers, and thus suggests that characteristics such as "slow" processors/nodes may not be the correct way to understand hardware variability.

Secondly, exascale systems will be faced with a new set of constraints, driven in large part by the emergence of power and energy as scarce computational resources, that are changing the way system resources are administered as well as how applications are built to utilize the machine. Rather than simply run a single application end-to-end and attempt to minimize its runtime as HPC systems have historically done, exascale systems will be required to manage multiple workloads in an effort to minimize global system power consumption, and, in general, to prioritize energy efficiency at least as much as it does runtime. Examples of how these characteristics are emerging can be be seen in efforts to compose and cooperatively schedule *in situ* workflows [99, 87] on the same shared nodes, the incorporation of power budgets in jobs [80, 89], and runtimes that manage system resources to prioritize energy efficiency [103] or power capping [70] over runtime.

These developments may seem modest, but they are fairly disruptive in the context of HPC systems that have always been programmed to prioritize runtime of a single application.

Not only do these developments threaten to increase variability due to the reduced focus on explicitly optimizing a single application, they also question the validity of the assumptions outlined earlier about how variability impacts a system; e.g., that it is produced by application workload imbalance,"slow"/defective hardware, or from OS interference, etc. While the new characteristics we expect in exascale hardware, software, and system objectives are a direct result of the challenges in building larger and more energy-efficient machines, they are fundamentally changing the composition and orchestration of system resources in a manner that will have profound implications for BSP workloads.

1.3 OVERVIEW: PREVENTION VS. DETECTION

This dissertation presents a set of system software approaches to mitigate variability in the context of these emerging system characteristics. While performance variability is, generally speaking, the singular focus of this dissertation, there are many different sources of variability [101, 18, 32, 86, 16, 103, 25, 42, 93, 71, 91, 79, 84], and, importantly, *the characteristics of a particular source indicate to what extent different approaches to managing variability are appropriate.* Our efforts are broadly categorized based on (a) whether it is possible for system software to take some action to *prevent* a source from manifesting in the first place, or conversely (b) the extent to which the system can only hope to *detect and react* to the manifestation of a source because it is difficult or impossible to prevent.

This dichotomy provides the basis through which we present our efforts to mitigate variability. To motivate why this distinction is necessary, we consider a set of sources of performance variability in Table 1. This table presents a taxonomy of variability based on two distinct features: (1) is the event that creates variability driven by software or hardware/external system characteristics?; and (2) does the resulting imbalance the source creates vary over space (multiple processors have different performance) or over time (the same processor performs differently over time)? These distinctions frame our approach: software induced variability can be *prevented* by system software, while hardware/external system induced variability can be *detected and reacted to by system software.*

	Software Induced	Hardware/System Induced
Spatial	Application-level Workload Imbalance [25]	Process Variation [101, 18, 32] Network Heterogeneity [86] Modern Many-Core Architectures*
Temporal	Resource Contention [16, 53, 55]+ OS noise [42, 93]+	Network Contention [71] Power Heterogeneity [91, 79] Intrinsic Hardware Resource Sharing*

+ *Technique: Prevention*; topic of Chapter 2

* *Technique: Detection and Reaction*; topic of Chapters 3 and 4

Table 1: Likely sources of performance variability at exascale

The left hand side of Table 1 lists sources of software induced variability. As the name suggests, these are examples of sources of variability that arise from some layer of software in the system. The upper-left hand quadrant of the table, that which refers to spatially-variant software induced variability, shows sources generated when an application itself does not uniformly distribute its work to all processors; over time, the same processors have more work to perform than others. The bottom-left hand quadrant shows examples of temporally-variant, software induced variability. These sources differ from the first category in that the imbalance they generate is inconsistent over time - in other words, even if a single processor was "slow" to reach a synchronization point in a past iteration of an algorithm, there is no reason to expect it to be slow again in the future. OS noise and contention for software resources such as locks in the kernel generate this form of variability.

Finally, the right hand side of Table 1 lists sources of variability induced either by hardware features or by external characteristics such as system optimization criteria or scheduling decisions. Spatially-variant sources in this category include features such as process variation, whereby different processors have different behavior (e.g., execute at different peak CPU frequencies) that are due to manufacturing differences which are consistent over time. Finally, the bottom-right quadrant lists temporally-variant sources, including contention for shared resources such as interconnects or power budgets, or underlying node hardware resources such as caches and memory channels.

1.3.1 Preventing Software Induced Variability

We first consider the distinction between software and hardware induced variability, as *this is precisely the distinction that determines if system software is able to prevent performance variability.* Consider the sources on the left side of Table 1. For each of these sources, it is theoretically possible, if not practically simple, for software to eliminate variability through a careful management of system resources and application workloads. In the spatial case, work in the areas of graph partitioning [15] and coarse-grained workload repartitioning [49] addresses this type of variability, and we expect these approaches to remain active and relevant in exascale-class machines.

On the other hand, temporal variability that arises from contention for OS resources and OS noise/interference is posed to present a greater threat for future large scale machines. While this type of interference has existed for some time in supercomputers, one of the key approaches to eliminate it has been to use lightweight operating systems [40, 61] that are designed specifically to eliminate OS interference. The problem is that such operating systems alone are not considered to be reasonable solutions for exascale machines, in large part because applications and emerging *in situ* workflows are increasingly dependent on features only provided in fullweight, commodity operating systems such as Linux.

Chapter 2 addresses the intersection of these issues. We designed and implemented an operating system, Hobbes, that provides the features of a full-fledged, Linux-based operating system required by applications, but at the same time provides a stronger guarantees for performance isolation than possible in Linux through its use of *enclaves* that protect applications from interference endemic to Linux based systems. Hobbes is thus designed to *prevent* software induced variability by provisioning enclaves that are isolated from low-level sources of operating system variability. This chapter will briefly discuss how Hobbes facilitates the creation of these enclaves, but will focus primarily on how applications can communicate across enclave boundaries to support complex higher-level workflows without sacrificing enclave-level performance isolation.

1.3.2 Detecting and Modeling Temporally Induced Variability

The distinction between software and hardware induced variability determines whether it is possible to eliminate variability with system software. The remaining distinction, that of spatial versus temporal variability, highlights the need for a better understanding of performance variability than is currently available in the literature.

At a high level, this distinction describes how imbalance from variability will be distributed across an application's processors. In general, spatial variability can be statically characterized throughout the duration of an application (e.g., "processor x is slower than processor y because of manufacturing defects or network heterogeneity"). However, for temporal variability, such static classifications are not appropriate because a node's behavior is a function of characteristics that change over time.

The reason this distinction is interesting is that current approaches to mitigate variability, such as asynchronous many-task runtimes [13, 48] or application-level middlewares based on MPI [49, 90], assume that imbalance is predominantly spatially variant, the result of imbalanced application workload distribution, or of "slow" processors and nodes that consistently delay progress over the lifetime of an application. Historically, of course, these have been the significant sources of variability at scale, and approaches such as these thus reflect the characteristics of previous generations of supercomputers. However, these assumptions are being called into question by the significant changes in how large scale systems are being built and managed, as discussed in the previous section. With more complex system objectives leading to co-scheduled and/or power constrained applications as well as heterogeneous and complex node architectures, it is likely that temporal variability will become at least as much of a concern as spatial variability in these machines.

Chapter 3 revisits these assumptions in the context of recent high performance architectures. Not only do we expect existing temporal issues to be more prevalent (network contention, power heterogeneity), but we also find that temporal variability is increasingly arising due to the heterogeneous and distributed nature of resources within emerging server architectures. The analysis in this chapter leverages a new performance analysis framework, `varbench`, which provides a detailed characterization of how variability arises in the context

of various architectures. Our analysis is the first to explicitly quantify the degree to which spatial and temporal variability arise in architecture. By using varbench, we demonstrate that many existing assumptions of how variability occurs are no longer true, thus strongly suggesting the need for a more holistic approach to understanding, detecting and modeling variability on future systems.

In Chapter 4, the last component of this dissertation presents such an approach with a technique called *criticality modeling*. Criticality models are not based on simplifying assumptions about how performance variability impacts a system, but rather are built to reflect the propensity of different classes of variability to arise on a specific architecture in the context of a specific application. Criticality models observe the manifestation of imbalance within a given architecture, and use statistical modeling techniques to determine which low-level hardware characteristics correlate with observed imbalance. With criticality models, a higher-level service can make predictions about how imbalance will manifest without making assumptions, but rather based on relationships learned by monitoring its behavior over time.

1.3.3 Thesis Statement

This dissertation provides a set of tools to mitigate variability on future extreme scale platforms: (1) a new lightweight operating system, Hobbes, that prevents software induced variability by prioritizing performance performance isolation between applications; (2) a new performance analysis framework, varbench, that characterizes spatial and temporal variability on a given node architecture; and (3) a framework called criticality modeling that leverages architecture-specific knowledge of temporal and spatial variability to detect and react to performance variability.

1.4 RESEARCH CONTRIBUTIONS

We summarize the three primary contributions of this dissertation:

1. We demonstrate how to prevent software induced variability through the Hobbes Operating System and Runtime. Hobbes is designed to prevent software induced variability caused by contention for OS level resources between separate applications. We demonstrate how Hobbes supports emerging workload characteristics expected in exascale systems, including *in situ* workflows, without sacrificing performance isolation between separate application components. This contribution is discussed in Chapter 2.
2. We designed and implemented the varbench performance analysis framework to holistically characterize performance variability. Varbench provides a framework for measuring the degree to which spatially and temporally induced variability emerge on a given architecture. We demonstrate how varbench can be used to measure hardware induced variability, and show that variability from hardware resource sharing has emerged as a significant source of temporal variability on recent architectures. Our work is the first to explicitly quantify the prevalence of spatial and temporal variability in a node architecture.
3. We designed and implemented criticality models to detect variability as it manifests within a machine. Criticality models are based on knowledge of the existence of spatial and temporal variability in a given architecture. In contrast to alternative approaches in past HPC systems, criticality models do not make assumptions about the prevalence of spatial or temporal variability in a system, but instead measure it directly in the context of a particular workload and use these measurements to build models of performance variability. Criticality models are built to allow applications to detect performance variability at runtime and apply mitigating techniques in response to it.

2.0 PREVENTING SOFTWARE INDUCED VARIABILITY

This chapter introduces the Hobbes Operating System and Runtime (OS/R) [56, 20], providing details on its design, implementation, and evaluation in the context of key target applications. Hobbes is designed to prevent a particular class of software induced performance variability from occurring in future exascale platforms: that which arises from cross-workload contention for operating system resources.

Hobbes is based on a large body of existing research in lightweight OSes. Past generations of lightweight OSes, such as Kitten [61] from Sandia National Laboratories and IBM's CNK [40], were built to provide scalable execution environments for a single application at a time. In these OSes, the majority of features, system calls, and services provided by commodity OSes, such as Linux, are not present. Lightweight OSes eschew these features, which historically have not been needed for most large scale HPC workloads, and instead provide only a very thin management layer responsible for initializing raw hardware resources and assigning them to applications. Lightweight resource management policies are much simpler than those in commodity OSes, and in general they permit applications to directly manage the system as they see fit. A major benefit of lightweight OSes in this context is a reduction in OS interference, or noise, that results in Linux-based machines due to the scheduling of kernel threads and other system services on the same processors where the application executes [39, 73, 84, 42, 93].

Hobbes leverages principles from these OSes and strives to provide a consistent and predictable execution environment for large scale BSP applications. However, Hobbes is designed with the workload characteristics of exascale systems in mind, and these characteristics represent a significant departure from the way past generations of systems have been organized. In past systems, applications generally employed disaggregated models of

computation and orchestration. Tightly coupled BSP simulations executed on dedicated supercomputers, output from these simulations was moved to storage clusters, and analysis and visualization workloads required to process this data were executed on separate I/O nodes as part of an entirely separate job [23]. However, this approach is infeasible at extreme scale, because simulations are consuming and producing much larger amounts of data, creating bottlenecks in interconnects and I/O subsystems, and consuming nontrivial amounts of power and memory to move data between clusters [116, 97].

These issues have driven the emergence of composed workflows [68, 99, 105, 87] whereby the compute/memory intensive BSP simulation communicates directly with analysis and visualization routines that execute on the same physical node. However, while composed models have the potential to increase the throughput of an exascale system, the individual workload components present different resource management and isolation requirements to the OS. In this section, we will describe how Hobbes is able to support these workloads, maintaining a focus on how it explicitly provides performance isolation between all workload components to prevent the occurrence of software induced variability. Hobbes achieves this high level goal by (1) providing multiple specialized, isolated system software environments called *enclaves* that each host a different component of an application workflow, and (2) providing infrastructure to allow communication and orchestration of activities between workloads in different enclaves.

First, in Section 2.1, we provide a high level overview of the Hobbes OS/R. In this section, we discuss the core components of enclaves, including a brief overview of past projects upon which our efforts in this dissertation are based. This section will also describe a set of real-world *in situ* applications that motivate the Hobbes infrastructure. With this understanding of the high-level organization, Section 2.2 will provide a detailed view of how communication between enclaves is achieved *without sacrificing the performance isolation provided by the Hobbes OS*. This section introduces the XEMEM shared memory system [54]. XEMEM leverages a distributed message passing infrastructure to allow multiple independent enclaves to construct shared memory mappings between each other's processes. These mappings are constructed in a manner that removes each enclave's software environment from the critical path of data movement during application execution. As a result, application components

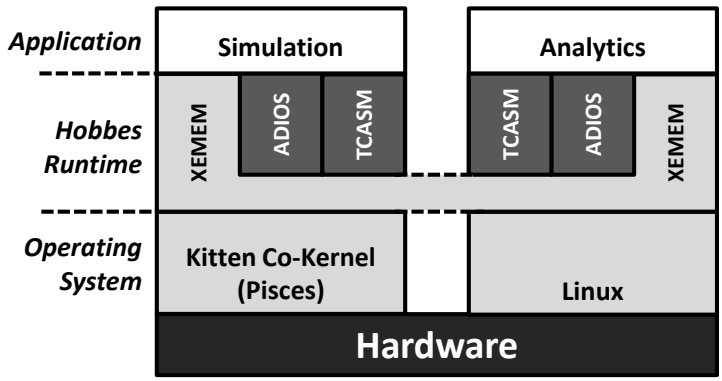


Figure 4: The Hobbes OS/R supporting an application composed in two enclaves

communicate directly through hardware, removing the potential for kernel-level interference to generate variability. Finally, we evaluate XEMEM, demonstrating the benefits of performance isolation for microbenchmarks as well as real-world composed applications.

2.1 HOBBES ENCLAVES

The need to provide diverse, customizable, and isolated system software environments form the basis for the Hobbes OS/R. At its core, Hobbes provides the infrastructure required to (1) deploy multiple diverse system software environments on a node, each specialized for a particular application workload (simulation, visualization, analytics, etc.); (2) support communication between applications executing in different software environments; and (3) ensure performance isolation between all workloads on the node.

Thus, the design of the Hobbes OS/R has been guided by three main considerations:

- The OS/R should allow dynamic (re-)configuration at runtime of hardware resource assignments and system software environments.
- The OS/R should provide a single, unified user space runtime API inside each enclave to support configuration flexibility and minimize application development effort.

- The OS/R should provide configurable isolation levels for each enclave in the system to meet the performance isolation requirements of each application.

We have designed the Hobbes OS/R architecture as a configurable runtime environment that allows dynamic configuration of isolated enclaves consisting of partitioned hardware resources and specialized system software stacks. The Hobbes environment provides a number of features to support composite applications including APIs for communication and control abstractions as well as languages describing the interactions, compositions, and deployment configurations of the target system environment.

The high-level overview of our design is illustrated in Figure 4. As the foundation for our architecture, we have based our work on the Kitten lightweight kernel [61] and the Pisces lightweight co-kernel architecture [77] along with the Palacios VMM [61, 60] to support Virtual Machine based enclaves. Our approach provides the runtime provisioning of isolated enclave instances that can be customized to support each application component. Additionally, our approach allows application composition through the use of cross enclave shared memory segments through the XEMEM shared memory system, whose application interface can be accessed using existing I/O mechanisms such as ADIOS [66] or TCASM [9].

2.1.1 Hobbes Building Blocks

2.1.1.1 Kitten and Palacios Kitten [61] is a special-purpose OS kernel designed to provide a simple, lightweight environment for executing massively parallel HPC applications. Like previous lightweight kernel OSes, such as Catamount [50] and CNK [40], Kitten uses simple resource management policies (e.g., physically contiguous memory layouts) and provides direct user-level access to network hardware (OS bypass). A key design goal of Kitten is to execute the target workload – highly-scalable parallel applications with non-trivial communication and synchronization requirements – with higher performance and more repeatable performance than is possible with general purpose operating systems. Kitten also supports virtualization capabilities through its integration with Palacios.

Palacios [61] is an open source VMM designed to be embeddable into diverse host OSes and currently fully supports integration with Linux and Kitten host environments. When

integrated with Kitten co-kernel hosts, Kitten and Palacios act as a lightweight hypervisor providing full system virtualization and isolation for unmodified guest OSes. The combination of Kitten and Palacios has been demonstrated to provide near native performance for large-scale HPC applications using Linux VMs running atop a Kitten host environment [60]. Palacios has also been shown to provide high performance guest environments that can outperform native environments for some workloads [51, 52], and to provide VMs that are well isolated from cross-workload interference [57].

2.1.1.2 Pisces Pisces [77] is a co-kernel architecture designed to allow multiple specialized OS/R instances to execute concurrently on the same local node. Pisces enables the decomposition of a node's hardware resources (CPU cores, memory blocks, and I/O devices) into partitions that are fully managed by independent system software stacks, including OS kernels, device drivers, and I/O management layers. Using Pisces, a local compute node can initialize multiple Kitten OS instances as co-kernel enclaves executing alongside an unmodified Linux host OS. Furthermore, by leveraging Palacios support, virtual machine instances can be created on top of these co-kernels as well. Pisces supports the dynamic assignment and revocation of resources between enclaves. Full co-kernel instances may be created and destroyed in response to workload requirements (e.g., application launch and termination), or individual resources may be revoked from or added to running instances. Specific details of these operations are presented elsewhere [77].

2.2 XEMEM: EFFICIENT SHARED MEMORY FOR COMPOSED APPLICATIONS ON MULTI-OS/R EXASCALE SYSTEMS

Thus far, we have discussed how the basic Hobbes building blocks, Kitten, Palacios, and Pisces, provide diverse system software environments via *enclaves* on a single node. In this section, we will discuss in detail our approach to support communication and coordination of workloads *across enclaves* through the XEMEM (Cross-Enclave Memory) shared memory architecture [54].

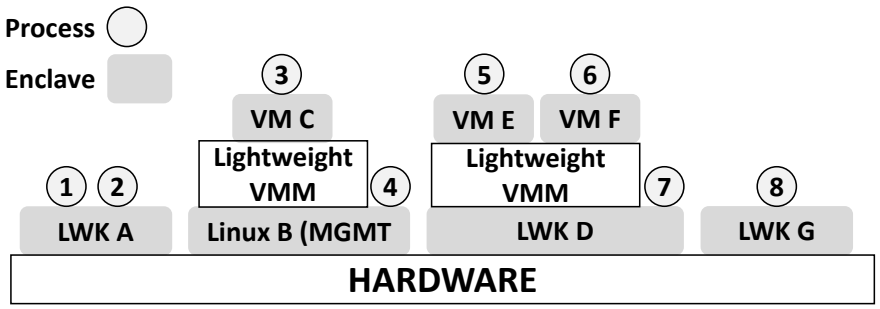


Figure 5: Exascale enclave partitioning

Figure 5 illustrate the desired capabilities of the XEMEM system. As the Figure shows, the individual OS/Rs of each enclave are capable of executing local processes, which by default are isolated from each other at both the system software and hardware layers. However, supporting composed workloads in these types of multi-enclave environments requires the ability to allow processes to communicate efficiently with other processes executing in separate enclaves. The key is that efforts to provide this capability must not sacrifice the performance isolation provided by Hobbes enclaves to the degree that variability arises from the system software architecture. This section presents the design, implementation, and evaluation of XEMEM, focusing on how it provides support for arbitrary process-level shared memory mappings that facilitate communication across enclave environments in a manner that remove OS kernels from the critical path of data movement.

We highlight the main contributions presented in this section:

- We present the design and implementation of XEMEM, a shared memory mechanism to enable composed workloads in a diverse set of multi-OS/R environments expected in exascale systems.
- We provide compatibility with existing applications by providing an implementation whose API is backwards compatible with the API exported by XPMEM [112], a shared memory implementation for large scale supercomputers. This allows unmodified applications to be deployed, without any knowledge of enclave topology or cross-enclave

communication mechanisms.

- We demonstrate the scalability of our implementation, with respect to both the sizes of shared memory regions, as well as the number of enclaves concurrently executing on the system. We
- By utilizing extensions to the Kitten lightweight kernel, Palacios virtual machine monitor, and a lightweight co-kernel architecture, we show that a multi-enclave system using shared memory allows a sample *in situ* workload to achieve superior and more consistent performance when compared to single OS/R configurations. Our results demonstrate that XEMEM-capable systems enjoy the benefits of cross-enclave communication while remaining resistant to OS induced performance variability.

2.2.1 Multi-OS/R Shared Memory

To support the types of environments likely to be seen in exascale systems requires that our system, XEMEM, support arbitrary enclave topologies while at the same time provide scalability as the number of co-located enclave OS/Rs increases. Furthermore, it is critical that the system scale to arbitrarily large shared memory regions as required by composed applications.

One of the key tenets of our approach is that, while meeting these requirements will require significant implementation effort in cross-enclave communication mechanisms and protocols, application programming for shared memory on an exascale system should not be more difficult than it is on current systems. Ideally, applications written for single OS/R systems should not need to be re-written or required to change at all to run in an multi-enclave environment. This section discusses how our system is built to maintain compatibility with existing shared memory applications while at the same time meeting the goals required for scalability and efficiency on future exascale architectures.

2.2.1.1 Common Global Name Space Our approach to maintaining the simplicity of shared memory application programming centers around the ability to provision a single common global name space for shared memory registrations that provides two key features:

unique naming and discoverability. In a single OS/R environment, shared memory regions can be readily named and tagged by a variety of basic mechanisms, such as using process IDs and virtual address ranges, which afford a simple way to maintain the unique addressability of each region. Furthermore, the OS/R has access to a plethora of shared IPC constructs, such as filesystems, to provide discoverability to processes.

In a multi-enclave environment, however, these operations are considerably more challenging. One approach to provide naming would be to eschew the requirement that the OS/R provide global uniqueness of identifiers, and instead force user applications to add an extra dimension to shared memory addresses by providing some form of unique enclave identifier. However, such an approach directly conflicts with our goal of maintaining application simplicity, as it would require applications to have knowledge of enclave configurations. Instead, our approach is to administer a common global name space by providing a centralized name server responsible for the allocation of segment identifiers for all shared address regions. This approach guarantees the uniqueness of all registered memory regions without requiring local OS/R environments to negotiate the availability of process IDs and virtual address regions, or adding complexity to application programming. This approach also allows our system to provide discoverability, as the name server can be queried for information regarding the existence and names of shared memory regions.

2.2.1.2 Arbitrary Enclave Topologies One of the key requirements for an exascale shared memory system is the ability to support the construction of arbitrary enclave topologies. It is our vision that not only will exascale environments incorporate a variety of different enclave architectures, but also that an individual node's partitions are likely to be dynamic and will change in response to the node's workload characteristics. At any point in time, we refer to an enclave's architectural partitioning, including the hardware-supported inter-enclave communication interfaces, as the *enclave topology*. Figure 6 shows the topology for the example enclave partitions shown in Figure 5, for the configuration in which the name server discussed in the previous section is configured to run in the single native Linux enclave. As the figure demonstrates, our system assumes that enclave topologies will be organized in a hierarchical fashion such that communication channels between enclaves can

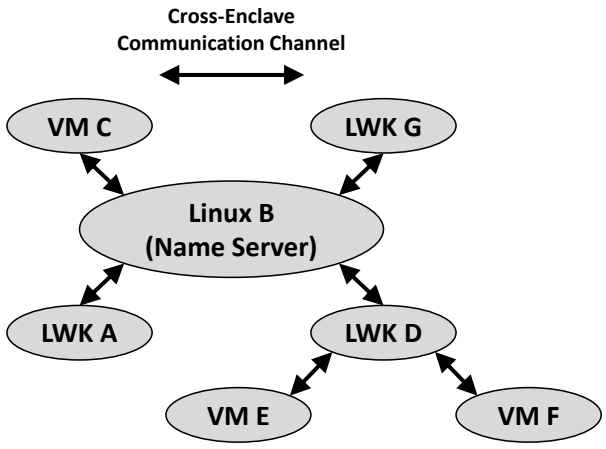


Figure 6: Exascale enclave topology

be restricted. This means that enclaves will not necessarily have the ability to communicate directly with all other enclaves in the system, but rather will be required to communicate via an alternative mechanism that supports the routing of information based on the hierarchical configuration.

Our system supports communication in any arbitrary topology by utilizing a hierarchical routing algorithm that associates each enclave in the system with a unique identifier called an *enclave ID*. In order to route messages through the system, the algorithm requires each enclave to perform three main operations: (1) determine the local communication channel through which it can communicate with the name server, (2) request an enclave ID via this channel, and (3) maintain a mapping of enclave IDs to local communication channels. Initially, an enclave queries the location of the name server by broadcasting a message on each of its communication channels. When another enclave receives a broadcast message, it creates a response if it knows a path to the name server through one of its own channels. Once a response is received, the enclave remembers the communication channel through which the response came, and then sends a request through the link to allocate an enclave ID, which gets forwarded to the name server.

To maintain a mapping of enclave IDs to communication channels, each enclave is re-

quired to keep track of enclave IDs as they are allocated by the name server and forwarded through the system. For example, again referring to Figure 6, assuming *VM F* has determined it can reach the name server through *LWK D*, it sends a request through *LWK D* to allocate an enclave ID. *LWK D*, which has previously identified the name server location and allocated an enclave ID for itself, forwards the request to the name server and remembers that the request came from *VM F*. The name server receives the request, allocates an enclave ID, and updates its enclave map to map the new enclave ID to *LWK D*. Upon receiving the new enclave ID, LWK D queries its outstanding request list and finds that a request previously came from *VM F*. Thus, it forwards the enclave ID to *VM F* and updates its internal map accordingly.

By maintaining enclave ID mappings in this way, enclaves are able to make routing decisions for shared memory registration messages by using a simple algorithm. When an enclave receives a message, it searches its map for the destination enclave ID. If it finds the enclave ID, it forwards the message along the associated communication channel for that enclave. Otherwise, it forwards the message through the channel used to reach the name server. With this hierarchical architecture, our system is able to support any arbitrary enclave topology. It should be noted that the name server can be deployed in any enclave on the system, though we envision that most exascale systems will likely place the name server in a management enclave.

2.2.1.3 Dynamic Sharing of Memory Regions Much of the previous work in high performance shared memory [21] has focused on efficiently creating large shared memory mappings between processes executing in single OS/R environments through the use of shared top-level page table entries. This approach is suitable for single-user environments such as the lightweight kernels it is currently deployed in.

Unfortunately, this approach is unsuitable in a multi-enclave environment for a variety of reasons. First, the sharing of page table entries across multiple heterogeneous processors may simply not be possible based on the hardware configurations in some exascale systems. Furthermore, heterogeneous software environments in different enclaves are likely to employ different address space management routines and operations. For example, full-featured

environments such as Linux will require mechanisms such as page protections and page faulting semantics in order to support higher-level application behavior. Coalescing this type of behavior with a static, distributed shared memory approach is likely to be difficult to implement efficiently and correctly.

Our approach is to dynamically support more fine-grained memory mapping requests according to the individual memory sharing requirements of composed application processes. As such, our system provides shared memory mappings as they are created/requested by processes executing in different enclave environments. While this approach adds a small amount of overhead in the creation and attachment of shared memory mappings, it allows for a more efficient use of virtual and physical address space, and increases the number of heterogeneous enclave environments likely to be supported by our system.

2.2.1.4 Localized Address Space Management

Given the significant scale and level of heterogeneity that is likely to be found in the hardware and software environments on exascale systems, our system requires the enclave operating systems to perform memory mapping operations locally using the techniques required by the enclave's hardware configuration. These operations include walking page tables to generate physical address regions that map to segment identifiers, as well as performing page table modifications to modify process address spaces. While it may be possible to perform shared memory mappings by modifying enclave address spaces remotely, it would likely be difficult to provide such an implementation correctly, and would at least require complicated and inefficient address space synchronization mechanisms.

2.2.2 Implementation

The goals of the implementation of XEMEM are threefold. First, we seek to preserve the benefits of performance isolation provided by Hobbes enclaves. That is, our implementation must not subject the application components to variability from any underlying OS operations involved in facilitating IPC. Second, we seek to provide scalability with respect to the number of concurrently executing enclaves, as well as the amount of memory being shared

Function	Operation
`xpmem_make`	Export address region as shared memory. Returns *segid*.
`xpmem_remove`	Remove an exported region associated with a *segid*.
`xpmem_get`	Request access to shared memory region associated with a *segid*. Returns permission grant.
`xpmem_release`	Release permission grant.
`xpmem_attach`	Map a region of shared memory associated with a *segid*.
`xpmem_detach`	Unmap a region of shared memory.

Table 2: The XPMEM user-level API

in the system at any point in time. Our third goal is to provide transparency to applications so that they are not required to have knowledge about the existence of enclaves or specific cross-enclave communication interfaces.

2.2.2.1 Isolated Data Planes The first goal is achieved through the high-level design of the XEMEM protocol. While there are extensive kernel-level operations required to facilitate memory mappings, we note that these are operations are in the control plane - all messages passed, interrupts delivered, etc. as part of XEMEM communication are only performed during the creation of memory mappings. Once mappings have been created, the OS kernels are completely out of the path of any data transfer between enclaves - the data plane - as these are done entirely via shared memory operations in hardware.

Our vision is that higher-level libraries will be used to orchestrate the transfer of data between application components on top of XEMEM shared memory regions. These libraries will have explicit knowledge of the communication and synchronization requirements of applications - which the OS kernels will not have - and thus can either create single persistent shared memory regions during the bootstrapping of workflow components, or can do so selectively during periods of execution when interference is acceptable (e.g., in between periods

of concurrent BSP computations).

2.2.2.2 XPMEM To provide transparency to user processes executing in heterogeneous enclave environments, we leveraged an API that is backwards compatible with the XPMEM API for Linux systems developed by SGI/Cray [112]. XPMEM provides zero-copy shared memory to areas of a process' address spaces by allowing processes to selectively export address space regions and associate publicly visible segment identifiers (*segids*) with them. Given a valid *segid*, a process can create shared memory mappings with the source process. The XPMEM user-level API is shown in Table 2.

XPMEM also provides a kernel module responsible for providing shared memory mappings between multiple native Linux processes. We developed a similar XEMEM kernel module which implements the same functionality as XPMEM for Linux systems, but also supports sending memory attachment requests to remote enclaves, as well as serving attachment requests from remote enclaves by performing page table walks to generate page lists. We implemented the name server as a component of our XEMEM kernel module, which is responsible for the creation of unique *segids* for all enclaves in the system. We also implemented the routing protocol discussed in Section 2.2.1.2 to support arbitrary enclave topologies. Finally, we implemented the XEMEM service in both the Kitten lightweight kernel and Palacios Virtual Machine Monitor.

2.2.2.3 Shared Memory Protocol While user-level processes are not required to have explicit knowledge of enclave topologies, they must still be able to discover exported memory regions created by external enclave processes. Our system provides a mechanism, illustrated in Figure 7, by which requests to create and attach to shared memory regions may be processed by the underlying enclave OS/R environments. As discussed in the previous section, XEMEM administers a common global name space for shared memory regions by utilizing a centralized name server responsible for assigning globally unique *segids* for all exported memory regions. Thus, to create a shared memory region, a local enclave's OS/R first sends a request to the name server to allocate a *segid*. The *segid* is then communicated to the user-level process according to the XEMEM API.

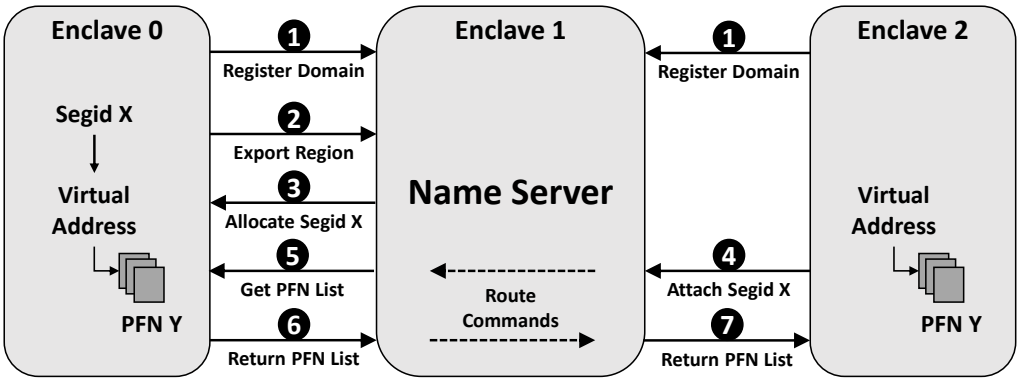


Figure 7: Shared memory protocol

In order to attach to a shared memory segment, a process wishing to map the region must first discover the unique *segid* from the source process in some way. While single OS/R environments would likely resort to the IPC constructs provided by the local OS, our system provides an alternative mechanism for querying the name server using kernel-level inter-enclave messages.

Given a valid *segid*, the local OS/R for the attaching process first determines if the segment was created by a local process or not. If so, the attachment proceeds using the conventions of the local OS shared memory facilities. Otherwise, an attachment request is sent to the name server using the command routing protocol described in Section 2.2.1.2. Upon receiving the attachment request, the name server, which maps *segids* to enclaves, then forwards the command to the destination enclave which owns the *segid*. The destination enclave creates a list of the physical page frames that have been allocated to map the *segid*. A response message containing the list of frames is then forwarded back through the name server to the attaching enclave using the same routing mechanisms. Upon receiving the list of page frames, the attaching enclave maps them to user memory using the memory mapping constructs provided by the local OS.

2.2.2.4 OS Memory Mapping Routines As discussed in the previous section, our system provides the ability to generate lists of page frames corresponding to *segids*, as well

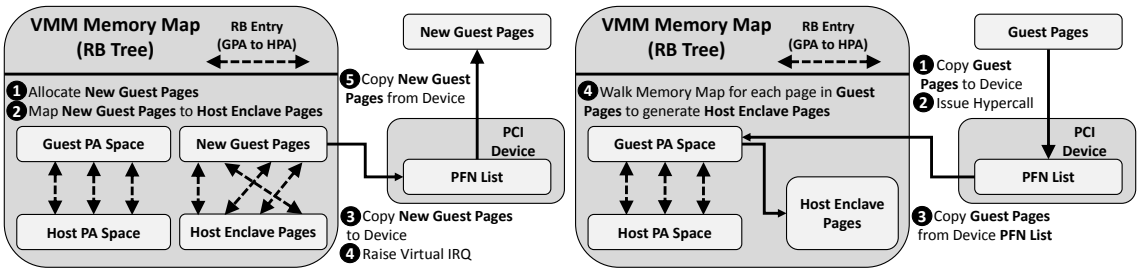
to map lists of page frames into destination enclave address spaces. Page frame lists are generated as enclaves receive remote requests for *segid* attachments. For Linux enclaves, memory must first be pinned in the user process' address space before a list can be generated. For this, we primarily rely on the `get_user_pages`[1] kernel function to allocate and pin physical memory for the process. Once the memory has been pinned, Linux provides a set of page table walking functions that allow the list to be easily generated. For mapping remote enclave page frame lists into Linux address spaces, we use the `vm_mmap` function to allocate an unused portion of virtual address space, and then use `remap_pfn_range` to map the page frames into the region.

For Kitten LWK enclaves, two existing design protocols complicate the dynamic creation of shared memory regions. First, the original shared memory mechanism in Kitten relies on the SMARTMAP [21] protocol, which uses page table sharing techniques to support local process shared memory in a way that would be difficult to extend to multi-enclave configurations. Furthermore, all virtual address space regions (heap, stack, etc.) for Kitten processes are mapped statically to physical memory as processes are created, and there was originally no support for dynamically expanding these regions. To address these issues, we added support for dynamic heap expansion, by which processes can map remote page frame lists in a way that does not sacrifice the ability to use SMARTMAP for shared memory with local processes, or negate the ability to map all other virtual regions with physical memory during process creation.

To implement page frame list generation for handling shared memory attachments from remote processes, we utilized existing page table walking functions already provided by the kernel.

2.2.2.5 Palacios Host/Guest Memory Sharing Providing support for shared memory between Palacios virtual machine enclaves and the corresponding host enclave presents two main challenges. First, both the guest and host need to be able to initiate shared memory operations, so our implementation must support efficient notifications of operations from

[1]Note that `get_user_pages` is a bit of a misnomer. Pages are generally already allocated when the function is invoked, with the main purpose being the pinning of memory to prevent paging out



(a) Guest Attachments to Host Memory (b) Host Attachments to Guest Memory

Figure 8: Palacios memory translations for shared memory attachments

host to the guest, as well as from the guest to the host. Additionally, Palacios should efficiently perform memory translations between host and guest page frame numbers for page lists that are passed during shared memory attachments.

To address these challenges, our approach, illustrated in Figure 8, consists of a virtual PCI device that allows efficient two-way notifications, as well as modifications to Palacios' internal memory management system to support the required memory translations. Figure 8(a) demonstrates the process by which a guest enclave completes an attachment to memory shared by the host. First, Palacios allocates a completely new region of guest physical address space that is equal in size to the amount of memory being shared. Then, Palacios updates its internal memory map to map this memory to the host page frames specified in the "Host Enclave Pages" list, which is supplied by the source enclave which exported the shared region. Once the memory map has been updated, Palacios copies the page frames corresponding to the new guest memory region to a list accessible through the virtual PCI device, and then issues an interrupt on the device. When the guest enters, it receives the interrupt, reads the page frame list from the device, and maps them into the attaching process' virtual address space.

It is important to note that Palacios' memory map is currently implemented as a red-black tree, where each entry in the tree maps a physically contiguous guest region to a physically contiguous host region. Palacios is usually configured to manage large blocks of

physically contiguous memory, and thus is able to map all guest memory with a relatively small number of entries in this tree. However, the host enclave page frames that are mapped as part of XEMEM attachments are not guaranteed to be contiguous, and thus the process of updating the memory map may require a new entry in the red-black tree for each host page frame. We study the performance implications of this process in detail in Section 2.2.3.4.

Figure 8(b) illustrates the process in the reverse direction, by which the host enclave can generate a host page frame list representing a memory region exported by the guest. To notify the host of the completion of an attachment operation, the guest copies the page frames to a list accessible through the PCI device, and then issues a hypercall to trigger an exit into the host. Using the pages frames specified by the guest, Palacios walks the memory map and generates the list of host page frames that correspond to the guest memory. The host enclave can then map the host page frames to the attaching process' virtual address space, or forward them according to the routing protocol if the attaching process exists in a separate enclave.

2.2.2.6 Cross-Enclave Communication Channels Facilitating the shared memory protocol described in Section 2.2.2.3 requires the ability to send messages between enclaves. Messages generally compose one of the commands shown in Table 2, but also exist to support the routing protocol outlined in Section 2.2.1.2, such as sending or responding to the broadcasts needed to query the name server location. Our current system provides two separate mechanisms for cross-enclave messages: a channel leveraging the Palacios virtual machine monitor for allowing communication between virtual machines and a native host enclave, and a channel based on the Pisces co-kernel architecture, which allows communication between two native enclaves.

The Palacios communication channel is implemented via a virtual PCI device, which is discussed at length in Section 2.2.2.5. In both transfer directions (host-to-guest or guest-to-host), if the message being transferred does not have an associated page frame list component (e.g., any of the operations in Table 2 except `xpmem_attach`), the mechanisms used are similar but simpler, as there is no need to translate page frame lists. For these operations, a simple command header located in the PCI device is set, and then either an interrupt into the guest

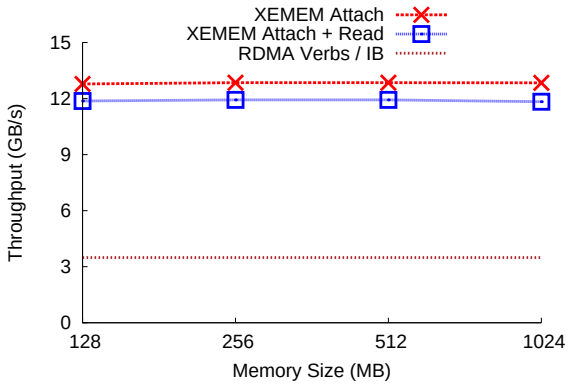


Figure 9: Cross-enclave throughput using shared memory and RDMA Verbs over Infiniband. Each XEMEM data point represents the throughput of 500 attachments for a given size

or a hypercall into the host is used to signal the message notification.

The other cross-enclave communication channel supported by our system is based on the Pisces co-kernel architecture. During the creation of a Kitten co-kernel enclave, the co-kernel creates a small shared memory region through which kernel-level messages can be transferred to/from the Linux management enclave. Both the Linux management enclave and the co-kernel enclave initialize special IPI (Inter-Processor Interrupt) vectors for negotiating access to this memory. To initiate a message transfer, an enclave sends an IPI to the destination enclave CPU using the indicated IPI vector. Upon receiving the IPI, the destination enclave sets a flag in the shared memory region indicating it is ready to receive data. The source enclave then copies the message into the shared memory region and waits for destination enclave to copy the message out into a separate locally allocated memory buffer.

2.2.3 Shared Memory Evaluation

In this section, we present an experimental evaluation of our cross-enclave shared memory implementation. This evaluation will focus on the scalability of our implementation, with respect to the amount of memory shared across enclave partitions, as well as with respect to the number of enclaves running simultaneously on the system. These experiments will focus

on performance measurements of shared memory attachment operations, as well as the performance implications for performing memory mappings in different enclave environments. We also compare our approach with an alternative approach to inter-enclave communication using RDMA.

2.2.3.1 System Configuration This evaluation was carried out on a Dell PowerEdge R420 server configured with dual-socket 6-core Intel Xeon processors running at 2.10 GHz, with hyperthreading enabled for a total of 24 threads of execution. The memory layout consisted of two NUMA sockets with 16 GB of memory each, with memory interleaving disabled for a total of 32 GB of RAM. The system was also configured with a dual-port QDR Mellanox ConnectX-3 Infiniband device with SR/IOV enabled in order to demonstrate the potential for cross-enclave communication using RDMA. The system ran a stock Fedora 19 operating system.

For each of our experiments, the XEMEM name server was configured to run in the native Linux control enclave. Based on the individual details of the experiments, we created a set of enclaves using the Kitten lightweight kernel and Palacios virtual machine monitor to study both native and virtualized environments. In some configurations, the Palacios VMs were configured to run on the native Linux management enclave, while in others the VMs were deployed in isolated co-kernel enclaves managed by Kitten. In either case, all VMs ran a stock Centos 7 guest operating system. Furthermore, in each experiment, enclaves were only allocated memory and CPUs from a single NUMA socket in order to avoid overhead resulting from cross-NUMA domain memory accesses.

2.2.3.2 Shared Memory Attachment vs. RDMA Throughput We first ran an experiment to demonstrate the potential throughput of cross-enclave communication using XEMEM compared to an alternative mechanism using RDMA. While these mechanisms are fundamentally different (byte-addressable memory operations versus block transfers over a peripheral bus), our goal was to demonstrate that our implementation does not add significant overhead, to the degree of diminishing throughout to the level of a simpler network-based approach. In this experiment, we created 1 Kitten enclave in addition to the Linux control

enclave. A process in the Kitten enclave exported a single memory region of varying sizes, ranging from 128 MB to 1 GB. On the Linux enclave, a process repeatedly attached to the exported memory region, measuring the time it took to complete the attachment, as well as the time to read out the memory contents. Each memory region attachment was repeated 500 times. For the RDMA experiment, we configured the system's dual-port Infiniband device with 2 virtual functions and assigned each to a separate KVM virtual machine. We then performed a simple RDMA write bandwidth test using the recommended MTU supported by the device.

The results of this experiment are shown in Figure 9. As can be seen, for each memory size the shared memory implementation achieves a sustained throughput of around 13 GB/s for attachment operations, with around 12 GB/s when including the time to read out the memory contents. In comparison, the RDMA bandwidth test demonstrated that slightly less than 3.5 GB/s can be transferred across the Infiniband device using RDMA, showing that the overhead of XEMEM operations does not significantly reduce shared memory throughput. The experiment also demonstrates good scalability as XEMEM memory sizes increase, which bodes well for the prospects of applications hoping to make use of large shared memory regions.

2.2.3.3 Scalability of Multi-OS/R Shared Memory Our next experiment was designed to demonstrate the ability of our implementation to scale to many enclave partitions running simultaneously on the same system, as well as the ability to support increasingly large shared memory regions in each enclave. In this experiment, we configured our system to create 1, 2, 4, or 8 Kitten enclaves using our co-kernel architecture. Each enclave was configured to run on a single core with 1.5 GB of memory, and to export memory regions ranging from 128 MB to 1 GB in size. Furthermore, for each enclave in the system a separate Linux process was created to attach to a single enclave's memory. In the case of 8 enclaves, this meant that up to 16 of the machine's 24 hardware threads were largely devoted to performing XEMEM operations. Note that although we chose a 1:1 communication model for this experiment, any arbitrary model is supported by our system.

The results of this experiment are illustrated in Figure 10. The figure demonstrates that

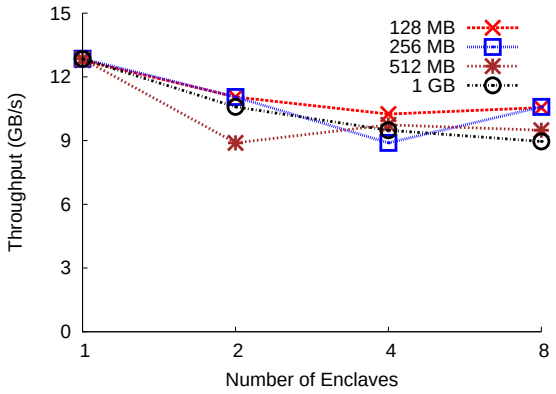


Figure 10: Cross-enclave throughput using shared memory between native Linux processes and native Kitten processes in co-kernel enclaves. Each data point represents the throughput of at least 500 attachments for a given size

the system scales well with respect to the amount of memory being shared at any point in time. This is a result of the fact that neither performing page table walks to generates page frame lists, nor subsequently communicating these lists across enclave communication channels, is a limit to the scalability of the memory sizes that can be shared. Furthermore, this figure also demonstrates that, beyond the initial configuration of only 1 enclave and 1 native Linux process, increasing the number of enclaves does not limit the scalability of the system. This indicates that our approach of using a centralized name server for allocating *segids*, as well as using a distributed routing protocol for performing shared memory attachments, are both suitable mechanisms for maintaining performance in the presence of high numbers of contending cores.

On the other hand, the figure does demonstrate slight performance degradation as the system scales from 1 to 2 enclaves. Given the good scaling demonstrated beyond 2 enclaves, we do not attribute this overhead to any scalability bottlenecks in our implementation, but rather attribute it to two main factors that are not fundamental limitations of our system. First, the co-kernel architecture used to perform cross-enclave message transfers (which are required for the transmission of XEMEM operations) currently restricts all IPI-based commu-

Exporting Enclave	Attaching Enclave	GB/s (w/o rb-tree inserts)
Kitten	Linux	12.841 (N/A)
Kitten	Linux (VM)	3.991 (8.79)
Linux (VM)	Kitten	12.606 (N/A)

Table 3: Cross-enclave throughput using shared memory between a Linux process and a native Kitten process executing in a co-kernel enclave. Each value represents the throughput of at least 500 attachments to a 1 GB memory region

nication with the Linux management enclave to core 0 of the system, and thus the presence of multiple enclaves may cause contention for interrupt handlers on this core. The result is that even though the separate Linux processes performing XEMEM attachments may be running on separate cores, the low-level messages facilitating their operation must be handled on a single core. Future work in the design of the co-kernel architecture will investigate more intelligent mechanisms for interrupt handling. Additionally, we also attribute this overhead to contention for Linux data structures that are accessed when multiple processes concurrently update memory maps.

2.2.3.4 Performance of Shared Memory Using Virtual Machines In addition to measuring the performance achievable in multi-enclave configurations with native OS/Rs, we also measured the throughput achievable when using the Palacios host/guest communication channel and memory translation mechanisms. We ran two separate experiments to measure both the host-to-guest and the guest-to-host performance as discussed in Section 2.2.2.5. In the first experiment, we launched a Linux VM running on the Linux management enclave, and we executed a single process in the VM that repeatedly attached to a 1 GB region of memory that was exported by a native Kitten process executing in a co-kernel enclave. In the second experiment, we deployed the same enclave configuration, but instead configured the Linux VM process to export a 1 GB region to be attached by the Kitten process.

The results of these experiments can be seen in Table 3. The top row of the table

shows the 1 GB result reported in Figure 9 for the native co-kernel configuration. The table demonstrates that moving the attaching process from the native Linux enclave to a virtualized Linux enclave causes a roughly 3x decrease in throughput when compared to the native configuration. In order to determine the source of this overhead, we measured the amount of time spent mapping remote enclave memory into the VM (those operations illustrated in Figure 8(a)), and we found that nearly 80% of the time was spent updating the guest's memory map. As discussed in Section 2.2.2.5, Palacios maintains a guest's memory map in the form of a red-black tree, and the process of mapping remote enclave memory generally requires as many updates to this tree as there are pages being shared. Thus, as the tree continues to grow, the cost for insertions and re-balancing operations increases, leading to performance degradation. Indeed, when removing the time spent updating the red-black tree, the throughput increases to 8.79 GB/s. In the future we intend to remove this overhead through the use of more intelligent radix tree based data structures that can more appropriately mimic a page table's organization.

Lastly, the bottom row of Table 3 shows that mechanisms for performing guest-to-host memory translations (illustrated in Figure 8(b)) are not nearly as costly, as the Kitten process is able to achieve over 12 GB/s throughput when attaching to the VM Linux process. This result indicates that performing page translations in Palacios does not add significant overhead to shared memory operations in the common case where the size of the Palacios memory map is limited.

2.2.3.5 Operating System Noise The final experiment in the first part of our evaluation measured the impact of performing page frame lookups in Kitten enclaves on the Kitten noise profile. OS noise, which has been identified as a significant source of overhead for HPC applications particularly at large scales [39, 73], is largely non-existent in Kitten as a result of its feature-limited design. Thus, while shared memory attachments may not necessarily qualify as noise, given that they are directly enabled by the HPC application and necessary in some sense for the progress of a composed workload, measuring the impact of attachment operations on the Kitten noise profile can provide some insight on the types of synchronization that may be needed to prevent attachments from perturbing simulations.

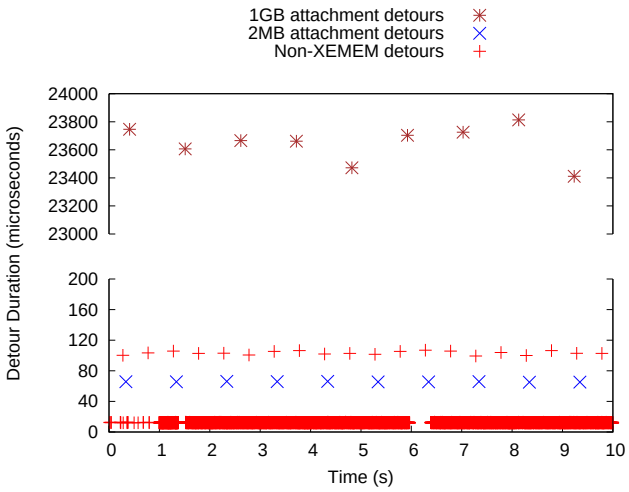


Figure 11: Noise profile of a Kitten enclave serving XEMEM attachment requests on a single core

In this experiment, we configured a Kitten enclave on a single core to export memory regions of sizes 4 KB, 2 MB, and 1 GB, and ran the Selfish Detour benchmark [14] from Argonne National Lab, a noise detection benchmark that is designed to measure the fraction of time the CPU spends executing instructions that are not part of the user's application. We ran a process on the Linux enclave to attach to each region, sleep for one second, and repeat for a period of 10 seconds. Figure 11 presents the results of the experiment. The figure shows that Kitten experiences a baseline level of frequent hardware noise around the 12 microsecond mark, as well as a set of less frequent interruptions likely caused by periodic hardware events such as SMIs around the 100 microsecond mark. Interestingly, detours caused by 4 KB attachments are not noticeable in the figure, as they cause detours similar in length to the frequent noise baseline. Detours caused by 2 MB attachments are more noticeable, but still cause less of a disturbance than the periodic hardware interrupts. Finally, the 1 GB attachments cause much larger detours, creating noise events that are 2 orders of magnitude longer than any other events. For these memory sizes, only the 1 GB attachments seem likely to cause problems for large-scale HPC workloads on Kitten,

and thus special care would be needed to synchronize their occurrence with respect to the application workflow.

2.2.4 Single Node Benchmark Evaluation

The second part of our evaluation uses a set of HPC benchmark workloads to demonstrate our system's ability to support sample *in situ* workloads in multi-enclave environments. In this section, our experiments will focus on the single-node performance implications of XEMEM in the presence of a wide range of enclave configurations and in situ workflow models. Then, in Section 2.2.5 we will evaluate a multi-node system configuration where each node employs XEMEM for local-node *in situ* execution, and demonstrate that the performance isolation benefits of multi-enclave configurations can lead to superior scaling behavior.

2.2.4.1 Sample *In Situ* Workload To present the performance characteristics of a composed *in situ* workload, we modified the HPCCG benchmark from the Mantevo suite [41] as well as the STREAM microbenchmark from the HPC Challenge suite [67] to synchronize their execution flows. Specifically, we modified the applications to use simple stop/go signals implemented on top of variables in shared memory. Throughout the evaluation sections, we refer to these modified benchmark components, respectively, as the HPC simulation and analytics program.

The HPC simulation executes an iterative conjugate gradient algorithm with collective operations in between iterations to gather intermediate results. We modified the benchmark to signal the analytics program at certain intervals during its execution to simulate an *in situ* workload. During these intervals, the simulation sends a signal to the analytics program by writing to a variable in shared memory. The simulation then waits for a return signal by polling on another variable in shared memory, which is written by the analytics programs when it determines to resume the simulation. In all, we configured the HPC simulation to execute 600 iterations of the conjugate gradient algorithm, and to communicate with the analytics program every 40 intervals for a total of 15 communication points. Upon receiving a signal from the HPC simulation, the analytics program may or may not attach to a new

exported region created by the simulation, a configuration option which we discuss below. In either case, the analytics program executes the STREAM benchmark over a 512 MB region specified by the simulation. The analytics program first copies the shared memory into a private array, and then executes STREAM over the array.

We note that currently the underlying enclave OS/Rs only support application communication through shared memory, and thus operations like event notifications must be supported via ad hoc techniques like polling on variables in memory. We plan to investigate techniques to support additional features in the OS/R environments as requirements of actual composed workflows become more evident.

2.2.4.2 Execution and Memory Registration Models The simulation and analytics programs can be configured based on two different parameters to mimic different *in situ* communication behaviors: synchronous/asynchronous execution models, and one time/recurring shared memory attachments.

When using a synchronous execution model, the simulation and analytics programs do not simultaneously execute. Rather, when the analytics program receives a signal from the simulation, it (optionally) attaches to simulation data via shared memory, and executes the STREAM benchmark. Once the benchmark completes, it sends a signal to the HPC simulation to continue. When using an asynchronous execution model, the simulation and analytics programs may execute simultaneously. When the analytics program receives a signal from the simulation, it (optionally) attaches to the simulation data via shared memory, and then immediately signals the HPC simulation to continue. Then, it executes STREAM as the HPC simulation resumes simultaneously.

When using a one time shared memory attachment model, the HPC simulation exports a single region of memory at the start of the simulation. During communication intervals, the analytics program does not attach to new memory regions, but rather attaches a single time to shared memory region which persists over the benchmark duration. Conversely, when using the recurring attachment model, the simulation exports a new shared memory region during each communication interval, which is subsequently attached by the analytics program. Thus, for the former case, the overhead of setting up shared memory attachments

Simulation Enclave	Analytics Enclave
Native Linux	Native Linux
Kitten Co-Kernel	Native Linux
Kitten Co-Kernel	Linux VM (Linux Host)
Kitten Co-Kernel	Linux VM (Kitten Host)

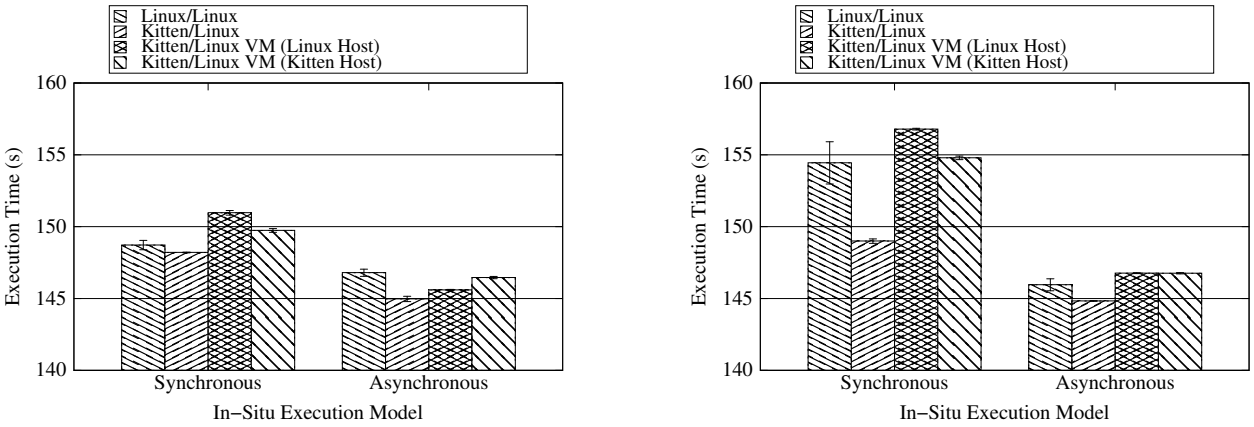
Table 4: Enclave configurations for the sample *in situ* workload in the single node experiments

is only experienced once at the beginning of the application; conversely, in the latter case, the overheads are experienced at each communication interval.

2.2.4.3 System Configuration The system used for this evaluation was a Dell OptiPlex server with a single-socket 4-core Intel i7 processor running at 3.40 GHz, with hyperthreading enabled for a total of 8 threads of execution. The memory layout consisted of a single memory zone with 8 GB of RAM. This system ran a stock Centos 7 operating system. As in the previous experiments, for each experiment in this section we configured the XEMEM name server to run in the native Linux control enclave.

We varied the execution environments for the HPC simulation and analytics applications to study a variety of configurations. For a baseline comparison we configured the simulation and analytics applications to execute in the native Linux control enclave, using the XEMEM shared memory facilities provided for single Linux environments. In the remaining configurations, the HPC simulation was configured to execute in a Kitten co-kernel enclave, while the analytics application was deployed in Palacios virtual machines executing on the host Linux control enclave as well as a separate Kitten co-kernel host. The configurations are shown in Table 4.

2.2.4.4 Results In addition to varying the enclave configurations, we executed the in sample *in situ* workload in four separate configurations based on the four combinations of the workflow parameters discussed in Section 2.2.4.2. The completion times of the HPC

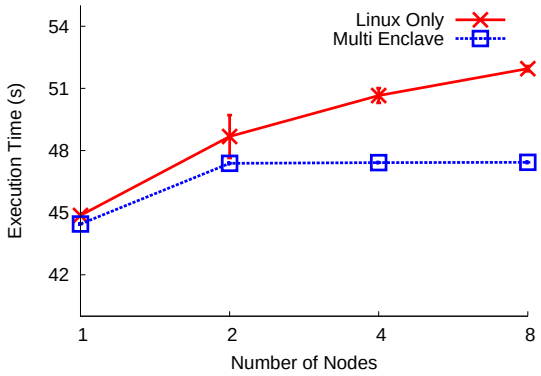


(a) One time shared memory attachment model

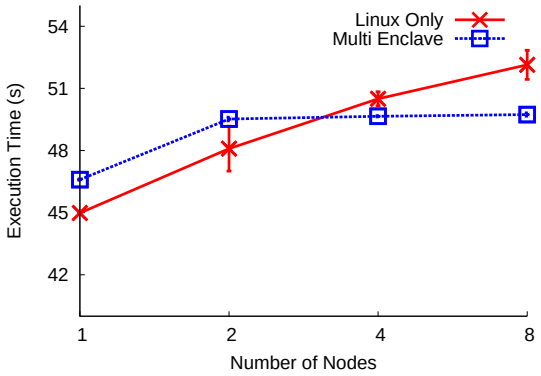
(b) Recurring shared memory attachment model

Figure 12: Performance of a sample *in situ* benchmark on a single node using various communication and execution behaviors

simulation using a one time shared memory attachment model are shown in Figure 12(a), where each bar reports the average and standard deviation of 10 runs of the application. As expected, for each environment the HPC simulation completes in less time when executing asynchronously with respect to the analytics program than executing synchronously. Under both execution models, the configuration deploying the HPC simulation in the Kitten enclave and the analytics program natively on the Linux enclave outperforms all other configurations. When using the asynchronous execution model, each environment utilizing the Kitten enclave for the simulation outperforms the Linux-only configuration. Conversely, when using the synchronous execution model, any overheads experienced in processing the analytics directly impact the runtime of the HPC simulation. Given the lack of shared memory communication overhead in these experiments, it is clear that the native analytics program slightly outperforms the same program running virtualized, particularly in the Palacios on Linux case. However, in each multi-enclave configuration the performance is more consistent than exhibited by the Linux-only environment, demonstrating that our approach does not add variable delays to the runtime of the simulation.



(a) One time shared memory attachment model



(b) Recurring shared memory attachment model

Figure 13: Performance of a sample *in situ* benchmark on a multi-node cluster using an asynchronous execution model

The results of the experiments using a recurring shared memory attachment model can be seen in Figure 12(b). Given the overheads from repeated memory map updates identified in Section 2.2.3.4, it is clear that performing recurring shared memory attachments in addition to using a synchronous execution model represents the worst case configuration for the virtualized enclaves. However, interestingly the Linux-only environment also suffers significant overhead as well as a marked increase in runtime variance in this case. We attribute this overhead mainly to the page faulting semantics with which single environment XEMEM attachment operations are performed in Linux. Indeed, when executing asynchronously, the overheads associated with both the virtualized and native Linux environments largely disappear. As in the previous cases, the performance in each multi-enclave configuration is very consistent, whereas the Linux-only configuration provides a lower degree of workload isolation leading to increased variance.

2.2.5 Multi-Node Benchmark Evaluation

The final experiments for our evaluation were designed to demonstrate the benefits multi-enclave configurations provide for composed *in situ* applications executing on multiple nodes. As the previous section demonstrated, even on a single node, the performance isolation that isolated enclaves provide leads to a more consistent runtime experience for the composed workload. In this section, we show that providing multi-enclave shared memory for in situ components running on each node of a multi-node system leads to superior scalability, demonstrating the value of workload isolation.

2.2.5.1 Workload and System Configuration For the experiments in this section, we used a local 8-node experimental research cluster. The nodes were each configured with dual-socket 6-core Intel Xeon processors running at 2.10 GHz, with hyperthreading enabled for a total of 24 threads of execution. The memory layout on each node consisted on two NUMA sockets with 16 GB of memory each, with memory interleaving disabled for a total of 32 GB of RAM. The nodes were interconnected with dual port QDR Mellanox ConnectX-3 Infiniband devices.

In these experiments we deployed two separate enclave environments in the system. In the default configuration, both *in situ* components were executed in the native Linux enclave with no other enclaves deployed in the system. The other configuration consisted of a Palacios VM enclave running on an isolated Kitten co-kernel host, in addition to the native Linux enclave. For this system composition, we ran the HPC simulation in the Palacios VM, while the analytics program executed in the native Linux enclave. For each individual experiment, every node ran the same enclave configuration.

Finally, we executed the same *in situ* workload used in the single node experiments, with the exception that the HPC simulation was compiled to use OpenMPI over the Infiniband interconnect for multiple node deployment. The HPC simulation used 8 cores of each node, while the analytics program was parallelized using OpenMP threads to use 8 additional cores on each node. The HPC simulation was configured to execute 300 iterations of the conjugate gradient algorithm, and to communicate with the analytics program every 30 intervals for

a total of 10 communication points on each node. The analytics program on each node then executes the STREAM benchmark over a 1 GB region specified by the simulation. As in the previous experiments, all workloads were explicitly pinned to NUMA domains in order to avoid the overheads on cross-domain contention. The benchmark was configured in weak-scaling mode, where the problem size of the HPC simulation scales with the number of nodes.

2.2.5.2 Results In order to evaluate the benefits of performance isolation that our multi-enclave system provides, we ran the *in situ* benchmark using an asynchronous execution workflow, meaning that during the application's communication intervals, the HPC simulation and the analytics program are executing concurrently. This workload deployment is representative of the types of applications that we envision will be present on exascale systems in that it requires high performance local-node communication between its individual components, but at the same time requires that the components be strictly isolated from interference caused by contention for the node's resources.

As in the previous experiments, we ran these experiments with two separate shared memory models: one time and recurring attachments. The results of the experiments can be seen in Figure 13, where each data point reports the average and standard deviation of five runs of the benchmark. As Figure 13(a) demonstrates, the scaling behavior of the multi-enclave configuration is superior to that provided by the baseline Linux only environment for the experiments using a one time memory attachment model. This result is particularly interesting because the HPC simulation is running in a virtualized environment, which demonstrates two key results. First, it shows that performance isolation is a requirement for these types of applications, to the degree that the same workload running virtualized can outperform itself running natively if it is better isolated. Furthermore, this result demonstrates that the XEMEM implementation yields a consistent execution environment on each node, which can be seen by the low standard deviation and very good scaling behavior.

Finally, the results of the experiments using a recurring shared memory attachment model can be seen in Figure 13(b). As the overhead of shared memory attachments is experienced multiple times during the application, the Linux only configuration is able to outperform the

multi-enclave configuration at a single node. However, both system configurations exhibit similar scaling behavior using this shared memory attachment model as they exhibited in Figure 13(a). Namely the lack of performance isolation in the single OS/R configuration leads to steady performance decline as each node has a different runtime experience, while the multi-enclave system shows almost no performance degradation past 2 nodes.

2.2.6 XEMEM Summary

Research trends in the HPC community indicate that exascale systems will be constructed from multiple heterogeneous hardware and system software configurations, called *enclaves*. We presented the design, implementation, and evaluation of XEMEM, a shared memory system capable of supporting a wide variety of enclave configurations likely to be seen in exascale systems. We demonstrated that XEMEM is able to scale to a high number of enclaves simultaneously executing on a system, even while each enclave creates increasingly large shared memory mappings. We further evaluated our system's ability to support a sample *in situ* workload utilizing a set of different of execution and memory registration models. Finally, we demonstrated XEMEM's ability to support a multi-enclave *in situ* application, which can outperform the same application executing in a single OS/R environment.

2.3 COMPOSING APPLICATIONS ACROSS ENCLAVES VIA XEMEM

Emerging HPC applications are increasingly composed of multiple communicating, cooperating components. This includes coupled simulation + analytics workflows, coupled multi-physics simulations and scalable performance analysis and debugging systems [111, 115, 76, 74, 92, 98, 12].

As a result, several projects have been undertaken in the context of the Hobbes OS/R in order to better support composition and coordination across cooperative application components [36, 38, 37], This section presents an analysis of how Hobbes, via XEMEM and these efforts, can be used to support composition of HPC applications from multiple cooperating

components. We show that Hobbes supports the composition of applications across multiple isolated enclaves with little to no performance overhead.

2.3.1 Application Composition Use Cases

While computational science and engineering applications embody a great deal of diversity, we have identified a number of composite application generic use case categories. These include:

- **Simulation + analytics** involves a simulation component communicating with one providing data analytics, visualization, or similar services. In most cases, data is presumed to flow from the simulation to the analytics component one way. However computational steering could introduce a reciprocal data flow, shared data, and/or a control interaction from the analytics to the simulation. While dynamic compositions are possible, we expect most applications of this type to involve static compositions where the components and their deployments are defined in advance and stay fixed throughout the execution of the application. Three examples are described in this paper.
- **Coupled multiphysics simulations** involve multiple simulation components, each modeling different physics, working cooperatively. Data may flow one-way, two-ways, or may be shared between components. There may also be control interactions between components. Coupled multiphysics simulations may be either static or dynamic. For example, there are an increasing number of examples in the literature of complex simulations in which new computational tasks (components, in this context) are instantiated on demand during a simulation and have a lifetime much shorter than the overall simulation. Examples of this category include [74, 92, 98, 12].
- **Application introspection** is a novel way of thinking about activities that need deep access to an application such as performance monitoring or debugging. Whereas today such applications usually involve complex combinations of simulation and "tool" processes, which must "attach" to them, in the Hobbes environment, such applications could be cast as co-located enclaves for which the communications can be defined as appropriate.

Composite applications can benefit from the Hobbes environment in several ways. Components may be designed with very different requirements from the underlying operating system. For example, a simulation component with few OS requirements might run well in a lightweight kernel (LWK) environment with minimal local performance overheads and inter-node jitter while an analytics component might require the richer services provided by a full Linux OS. Or two components in a coupled simulation might require different, incompatible runtime systems when used within the same executable or might not share resources well. Composition also allows increased component deployment flexibility in ways that can both accelerate computation and reduce resource requirements. For example, simulation and analytics components that require different runtime environments can be consolidated on the same node allowing communication to occur at memory speed rather than over a network.

2.3.2 Example Applications

To demonstrate the capabilities of the Hobbes OS/R to provide effective application composition we have focused on three example applications that represent common HPC compositions of the simulation + analytics type. These involve molecular dynamics, plasma microturbulence, and neutronics with corresponding analytics codes. These applications had previously been coupled using ADIOS (molecular dynamics and plasma microturbulence) and TCASM (neutronics), and no changes were required at the application level to deploy them in the Hobbes environment utilizing versions of ADIOS or TCASM which had been ported to use XEMEM.

2.3.2.1 Crack Detection in Molecular Dynamics Simulations LAMMPS (Large Scale Atomic/Molecular Massively Parallel Simulator) [85] is a molecular dynamics simulation used across a number of scientific domains, including materials science, biology, and physics. It is written with MPI (and also has options to use OpenMP and CUDA) and performs force and energy calculations on discrete atomic particles. After a number of user-defined epochs, LAMMPS outputs atomistic simulation data (positions, atom types, etc.), with the size of this data ranging from megabytes to terabytes depending on the experiment

being performed.

SmartPointer [111] is an associated analytics pipeline that ingests and analyzes LAMMPS output data to detect and then scientifically explore plastic deformation and crack genesis. Effectively, the LAMMPS simulation applies stress to the modeled material until it cracks and the goal of the SmartPointer analysis is to understand the geometry of the region around that initial break. The SmartPointer analytics toolkit implements these functions to determine where and when plastic deformation occurs and to generate relevant information as the material is cracked. The toolkit itself consists of a set of analysis codes that are decomposed as separately deployable applications that are chained together via data transfers identified by named channels, i.e., an ADIOS "file name." Further details of many of the SmartPointer functions can be found elsewhere [111]. For this set of experiments, we focus on the "Bonds" analytics code, which ingests LAMMPS atomistic data and performs a nearest neighbor calculation to output a bond adjacency list, which is a pair-wise representation indicating which atoms bonded together.

2.3.2.2 Plasma Microturbulence The Gyrokinetic Toroidal Code (GTC) [75] is a 3-Dimensional Particle-In-Cell code used to study micro-turbulence in magnetic confinement fusion from first principles plasma theory. It outputs particle data that includes two, 2D arrays for electrons and ions respectively. Each row of the 2D array records eight attributes of a particle including coordinates, velocities, weight, and label. The last two attributes, process rank and particle local ID within the process, together form the particle label which globally identifies a particle. They are determined on each particle in the first simulation iteration and remain unchanged throughout the particle's lifetime. These two arrays are distributed among all cores and particles move across cores in a random manner as the simulation evolves resulting in an out of order particle array. In a production run at the scale of 16,384 cores, each core can output two million particles roughly every 120 second resulting in 260GB of particle data per output. GTC employs the ADIOS BP format [65], a log-structured, write-optimized file format for storing particle data.

As illustrated in Figure 14, three analysis and preparation tasks are performed on particle data. The first involves tracking across multiple iterations a million-particle subset out of the

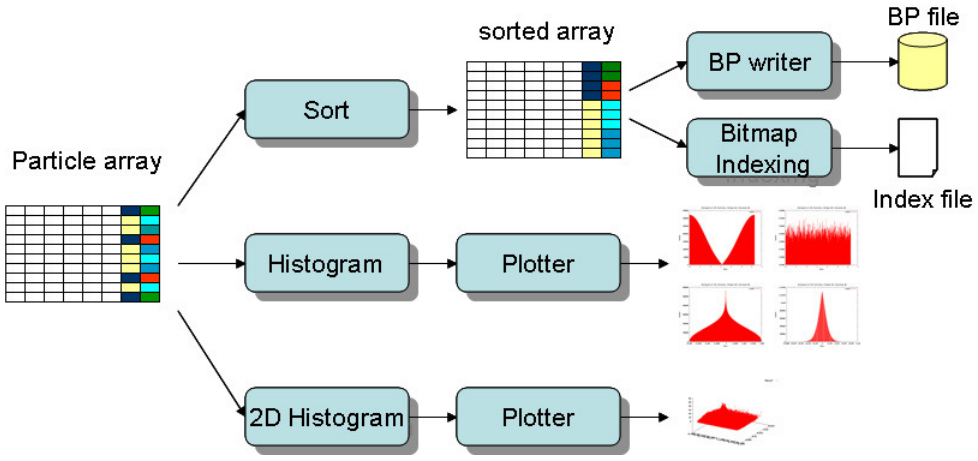


Figure 14: Illustration of PreDatA operations on GTC particle data

billions of particles, requiring searching among the hundreds of 260GB files by the particle index label. To expedite this operation, particles can be sorted by the label before searching. The second task performs a range query to discover the particles whose coordinates fall into certain ranges. A bitmap indexing technique [94] is used to avoid scanning the whole particle array and multiple array chunks are merged to speed up bulk loading. The third task is to generate 1D histograms and 2D histograms on attributes of particles [47] to enable online monitoring of the running GTC simulation. 2D histograms can also be used for visualizing parallel coordinates [47] in subsequent analysis. Our example focuses on integrating the 1D and 2D histograms with GTC.

This architecture was previously demonstrated in PreDatA [115] using node-to-node or even file on disk techniques to connect the workflow components. These previous experiments showed the importance of placing analytics carefully for the best overall system performance. For this work, we have repurposed this example to use the XEMEM connection using the new ADIOS transport method. This eliminates the need to move data off node while eliminating the need for source code changes to change from node-to-node to file on disk to the shared memory interface. To facilitate operating in the limited Kitten operating environment, the GTC-P proxy application for GTC is used. GTC-P is used for porting tests, performance

testing, and optimization investigations.

2.3.2.3 Neutronics Energy Spectrum Analysis

SNAP [2] is a proxy application, developed to simulate the performance workload of the discrete ordinates neutral particle transport application PARTISN [10]. PARTISN solves the linear Boltzmann equation for neutral particle transport within a material. The solution of the time-dependent transport equation is a function of seven independent variables describing the position, direction of travel, and energy of each particle, and time. PARTISN uses a domain decomposition strategy to parallelize the problem, distributing both the data and computations required to solve the Boltzmann equation. The inner loop of the application involves a parallel wavefront sweep through the spatial and directional dimensions, in the same fashion as the Sweep3D benchmark [3]. SNAP does not perform the actual physics calculations of PARTISN, rather it is designed to perform the same number of operations, use the same data layout, access the data in (approximately) the same pattern.

SNAP is coupled to an analytics code which evaluates the energy spectrum of the simulation at each time step. This application, coupled using TCASM, was originally described elsewhere [76]. At the end of each time step, the SNAP simulation publishes its data via TCASM's copy-on-write approach. The spectrum analysis code accesses each time step's data in turn and computes the spectrum, printing the results to standard output.

In the Hobbes environment, the application and analytics codes are unchanged. As preciously mentioned, TCASM itself has been modified to use XEMEM to share memory between enclaves, as opposed to using Linux VMA in the original implementation.

2.3.3 Evaluation

In order to demonstrate the effectiveness of our Hobbes OS/R architecture to support composed applications we have evaluated both the LAMMPS and GTC compositions on a single node. The experiments were conducted on one of the compute nodes of the "Curie" Cray XK7 system at Sandia National Labs. Curie consists of 52 compute nodes, each with a 16-core 2.1 GHz 64-bit AMD Opteron 6200 CPU (Interlagos) and 32 GB in 4 channels of DDR3

RAM. The compute nodes run Cray's customized Linux OS, referred to here as Compute Node Linux (CNL), which is based on a modified SUSE version 11 (SLES 11) kernel coupled with a BusyBox user space.

For each of our experiments we compared our Hobbes based multi-enclave environment against the standard CNL environment provided by Cray. Hobbes augments Cray's standard HPC-optimized Linux OS image with the Hobbes infrastructure, which consisted of two new kernel modules (XEMEM and Pisces) that needed to be loaded in the Cray OS and a number of Hobbes user-level tools for launching and managing new enclaves.

Our experiments consisted of multiple runs of each composed application in which the application components were mapped to different enclave topologies. Application binaries used for each configuration were identical, with the only change needed being an ADIOS configuration file update to select the underlying transport. For each environment we recorded the average runtime of the application along with the standard deviation in order to evaluate performance consistency. We present these results for both the LAMMPS and GTC applications.

2.3.3.1 LAMMPS We ran two components of our LAMMPS composition example, the LAMMPS executable and the separate Bonds executable, in several multi-enclave configurations and compared against the baseline of running both components in a single OS instance. The tested configurations along with performance results are shown in Table 5. The statistics reported were calculated from 10 runs. The baseline configuration was to run the LAMMPS and Bonds components as separate processes in the default Cray OS image. In the past these components have been coupled through the filesystem, using the ADIOS POSIX transport ('Cray-Linux (POSIX)' in the table). This is problematic because of the filesystem overhead and the difficulty of detecting when the LAMMPS output is ready for Bonds to start processing it. As can be seen, this configuration has approximately 7% higher overhead than the other configurations, which used shared-memory instead of the filesystem. Switching to the ADIOS XEMEM transport (developed by Hobbes) improved the baseline Cray OS performance significantly.

For the multi-enclave examples, the LAMMPS and Bonds components were split to run

LAMMPS Enclave	Bonds Enclave	Average	StdDev
Cray-Linux (POSIX)		165.02	0.20
Cray-Linux (XEMEM)		153.48	0.08
Cray-Linux	Kitten	153.50	0.15
Kitten	Cray-Linux	153.91	0.04
Cray-Linux	Linux-VM	153.35	0.17
Linux-VM	Cray-Linux	156.10	0.15
Kitten-Enclave1	Kitten-Enclave2	153.83	0.03

Table 5: LAMMPS multi-enclave runtimes (s)

in two separate OS images (enclaves). One component was run in the Cray OS while the other component was run in either a Kitten enclave or in a Palacios enclave hosting a Linux virtual machine (VM). Additionally we evaluated running the components in two separate Kitten enclaves. These enclaves were started by using the Hobbes tools to *offline* CPU and memory resources from the Cray OS and then 'booting' either a Kitten or Palacios enclave on the offlined resources (i.e., the enclaves space shared the node's resources). The components were then cross-enclave coupled using the Hobbes XEMEM memory sharing mechanism – LAMMPS exported a region of its memory with a well known name, which Bonds then attached to via the well known name.

As can be seen in Table 5, all of the multi-enclave configurations that we evaluated produced roughly the same performance as the single OS (single enclave) baseline with shared-memory coupling. This is what we had hoped to show – that we could run this composition across multiple OS images without significantly impacting performance. This enables the use of system software customized to the needs of the code. We plan to evaluate this aspect in future work looking at multi-node scalability.

2.3.3.2 Gyrokinetic Toroidal Code (GTC) The second set of experiments we ran focused on the GTC application. Similar to the LAMMPS experiments, we executed each application component on a collection of OS/R configurations to measure any difference in performance. Due to the OS/R feature requirements of GTC's analytics package we

GTC Enclave	Analysis Enclave	Average	StdDev
Cray-Linux (POSIX)		148.42	0.12
Cray-Linux (XEMEM)		147.40	0.09
Cray-Linux	Linux-VM	147.52	0.09

Table 6: GTC multi-enclave runtimes (s)

were unable to run the simulation directly on Kitten, and instead had to deploy it inside a Linux VM hosted on a Palacios/Kitten instance. In these tests, GTC runs for 100 timesteps performing output every five timesteps. It generates approximately 7.2 MB of data per process and the analysis routine generates histograms written to storage. For our examples, we run with a single process for both the simulation and the analysis routines as a proof of concept. The times presented when using POSIX represent solely the time for writing data to a RAM-disk file system while the XEMEM times include the time for generating all but the last set of histograms written to the RAM-disk file system. We chose not to include the histogram generation time as it is less than two seconds and the typical workflow coordination overheads and delays would unfairly penalize the POSIX files approach. For these tests, we perform five runs for each configuration and show the mean and standard deviation for each. The results of these experiments are shown in Table 6.

The experiments show that using the Hobbes XEMEM shared memory transport provides slightly improved and slightly more consistent performance over the native POSIX file based interface available on CNL. Furthermore, the XEMEM performance remains consistent even when the analytics workload is moved to a virtual machine based environment.

2.4 RELATED WORK

2.4.1 Overview of HPC OS/R Architectures

Two separate philosophies have emerged over recent years concerning the development of operating systems for supercomputers. On the one hand, a series of projects have investigated the ability to configure and adapt Linux for supercomputing environments by selecting removing unused features to create a more lightweight OS. Alternatively, other work has investigated the development of lightweight operating systems from scratch with a consistent focus on maintaining a high performance environment.

The most relevant efforts to our approach are FusedOS from IBM [78], mOS from Intel [110], McKernel from the University of Tokyo, and RIKEN from AICS [100]. FusedOS partitions a compute node into separate Linux and LWK-like partitions, where each partition runs on its own dedicated set of cores. The LWK partition depends on the Linux partition for various services, with all system calls, exceptions, and other OS requests being forwarded to Linux cores from the LWK partition. Similar to FusedOS, McKernel deploys a LWK-like operating environment on heterogeneous (co)processors, such as the Intel Xeon Phi, and delegates a variety of system calls to a Linux service environment running on separate cores. Unlike FusedOS, the LWK environments proposed by mOS and McKernel allow for the native execution of some system calls, such as those related to memory management and thread creation, while more complicated system calls are delegated to the Linux cores. These approaches emphasize compatibility and legacy support with existing Linux based environments, to provide environments that are portable from the standpoint of existing Linux applications. In contrast to these approaches, Hobbes places a greater focus on performance isolation by deploying co-kernels as fully isolated OS instances that provide standalone core OS services and resource management.

2.4.2 Overview of Composition Techniques

A wide range of virtualization systems and inter-VM communication techniques have been used to support multiple applications or enclaves with customized runtime environments.

For example, cloud systems based on virtualization support multiple independent virtual machines running on the same hardware. Communication between VMs in these systems is generally supported through virtual networking techniques [102, 27] or shared memory [107]. Furthermore, systems based on OS-level virtualization and resource containers (e.g., Docker [1]) can leverage OS inter process communication tools to support communication between containers.

The primary goal of each of these systems is to maximize throughput, while providing security and hardware performance isolation to the independent VMs or resource containers. However, Hobbes differs from these systems through its focus on providing performance isolation through each layer of the system stack, including the node's system software and resource management frameworks which are independent to each enclave. The result is that Hobbes can guarantee a higher degree of performance isolation, but can also selectively relax the isolation to support enclaves which *cooperate*, as needed by applications.

Much work in the HPC space has focused on facilitating application composition and coupling between simulations and analytics codes. Initial work has focused on providing streaming style data transfers between concurrently running simulation and analytics codes, both in-transit [34, 29, 104] and in situ [118, 19, 34]. More recent work has focused on providing management capabilities for these mechanisms to address interference when applications share physical resources [117, 6, 45], as well as resource allocation issues that span entire science workflows [30, 45]. The work focused on in this chapter can leverage and adapt these techniques to facilitate coordination of applications composed of multiple communicating enclaves in a virtualized environment via the XEMEM architecture.

2.5 SUMMARY

In this chapter we presented the design, implementation, and evaluation of an OS/R, Hobbes, that enables application composition across multiple independent application components. Hobbes facilitates the creation of *enclaves*, which are self-contained system software environments that manage an independent partition of node hardware. We provided a detailed

discussion of the XEMEM shared memory system which allows diverse and complex workflows to be composed across several different enclaves, each tailored to the specific needs of a workflow component. Critically, XEMEM supports orchestration across enclaves while maintaining performance isolation by removing all kernel involvement from the critical path of data movement during application execution. We evaluated the Hobbes OS/R, showcasing its ability to support real composed applications.

3.0 CHARACTERIZING VARIABILITY ON EXASCALE MACHINES

In the previous chapter, we discussed the details of the Hobbes OS/R, which was primarily designed to prevent software induced performance variability. However, there are many other sources of variability that arise from characteristics that are external to a node's system software. Our position is that these sources cannot be prevented by system software in the way that cross-workload OS interference and OS noise can be. There are several known sources of variability that arise from characteristics which the node-level software has little to no control over, including shared hardware resources, interconnects, and external system criteria such as job placement within an architecture. With this class of variability, the best the system software can do is to understand how and why variability occurs, and to provide infrastructure through which adaptive/reactive techniques can be applied to mitigate its effects.

Approaches in this vein have been studied in past systems, with mitigating techniques such as load balancing [49], work stealing [13, 48], and power redistribution [70]. For the most part, past efforts have been based on simplifying assumptions, not of why, but of *how* variability manifests - that is, how imbalance actually arises in real large scale workloads. Chief among these assumptions is that imbalance occurs in a spatially variant but temporally consistent manner; that is, although there may be differences in performance characteristics between nodes or processors, these differences persist over time, and thus one can project past observations of imbalance to future iterations of an application.

However, exascale systems are changing in substantial ways that will have implications for how variability occurs. Node architectures are much different and more complex than past architectures with hundreds of cores, multiple memory technologies such as high bandwidth memory cubes, co-processors/GPUs, and distributed on and off-chip interconnects to move

data between the diversity of components. External to a node, systems are incorporating different objectives incorporating power and energy constraints, driving innovations such as job-level power budgets that could themselves fluctuate over the lifetime of an application. Finally, as discussed extensively in Chapter 2, applications are increasingly composed and co-scheduled on the underlying nodes, creating competition not just for system software resources but potentially for hardware as well.

In light of these characteristics, we argue that a comprehensive approach to understanding and characterizing variability in an architecture is needed. For one, a framework that characterizes the extent to which spatial and temporal variability occur on a system would help to (in)validate assumptions that underlie existing and proposed approaches to mitigating variability. However, a characterization would also be useful for discovering new sources that arise in emerging architectures, or for understanding how different tunable system parameters impart variability. For example, node architectures now have capabilities to opportunistically boost CPU frequencies beyond their nominal peak values given the availability of thermal headroom on package (e.g., Intel TurboBoost [88]), as well as to optimize performance under a power budget (e.g., Intel RAPL [28]). It would be useful to be able to study the extent to which variability evolves as a function of these characteristics, and to understand whether key behaviors such as spatial and temporal variability change based on their usage. Similar arguments can be made for comparing many other tunable system parameters.

Beyond simply understanding how variability increases or decreases as a function of these characteristics, a more detailed statistical characterization can be a useful feedback mechanism to inform higher level management of resources. For example, researchers have shown that when an application is bottlenecked by different resources over time, these bottlenecks can be discovered by looking at how performance varies between processors [62] at different points in the application. These characterizations can be used to detect, for example, when an application is bound by memory bandwidth, and thus CPU frequencies can be reduced without reducing performance. We envision that many other such relationships can be discovered and used to inform higher level stochastic optimizations.

Based on these motivations, we introduce `varbench`, a new performance evaluation frame-

work designed to holistically characterize variability on a system. Varbench is designed to generate statistics that describe how performance varies on a given architecture in the context of a given workload. By using varbench, we are able to revisit the assumptions that have driven many existing approaches to mitigating variability, and provide evidence that these approaches are insufficient for exascale machines. Finally, the varbench framework is designed to be easily extensible, allowing users to, for example, incorporate different workloads into the framework, and to study how variability impacts these workloads on different architectures.

This chapter first explores some of the key assumptions underlying current approaches to mitigate variability and argues why these approaches are unlikely to be adequate. These issues are the topic of Section 3.1. Because of these issues, we further motivate the need for a new framework to measure and characterize variability on large scale machines. Section 3.2 introduces the varbench performance evaluation framework, discusses its key components, and illustrates how it can be used to make important classifications of variability on a machine. Based on these capabilities, Section 3.3 showcases several interesting observations that can be derived from the use of varbench in current machines, and suggests ramifications of these observations for exascale systems. We summarize this chapter in Section 3.4.

3.1 VARIABILITY IN EXASCALE MACHINES

We posit that performance variability will be an increasingly difficult challenge on exascale computers, not only due to the sheer scale of the machines, but also for qualitative reasons that will engender a change in how programs must react to variability in order to scale in its presence. This section will discuss three reasons why this is the case: (i) the complex and distributed nature of node hardware; (ii) multi-faceted system optimization criteria; and (iii) the emergence of compositional applications and workflows.

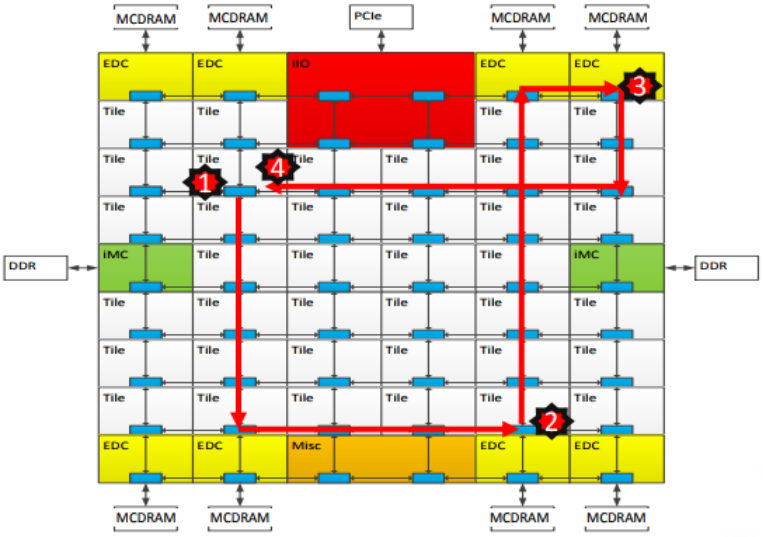


Figure 15: Memory routing in a Knight's Landing chip (image reproduced [95])

3.1.1 Complex, Heterogeneous and Distributed Node Architectures

One of the most influential innovations in supercomputing technology over the past five years or so has been the widespread adoption of heterogeneous computing resources. While GPUs have probably been the most influential, other technologies have started to gain adoption, including high bandwidth memory devices, FPGAs, and co-processors, such as those based on the Intel Knight's Landing (KNL) chip, now deployed on a wide range of top supercomputers [4]. These developments represent a significant departure from past systems which were mostly comprised of "skinny" nodes built with a small number of commercial off-the-shelf processors.

While these technologies are promising avenues to furthering the peak performance potential of large scale systems, they come with a drawback. Due to the increasingly distributed and interconnected nature of these architectures, accessing resources often requires access to and traversal of other *shared* node resources, such as system buses or on-chip interconnection networks. A key example, shown in Figure 15 can be seen in KNL, where memory accesses must traverse a 2-dimensional on-chip mesh, perhaps several times, in order to move data

to/from a core. Similar observations can be made when viewing slightly older generations of server architectures, with the addition of shared cross-processor interconnection links (e.g. Intel's QPI, AMD's HyperTransport), as well as more recently with the introduction of logical threads (or hyperthreads in Intel terminology) that share resources of an individual core (e.g., L1/L2 caches, FPUs).

The hypothesis that this chapter presents is that with increasingly shared and interconnected node architectures expected in future machines, performance variability that arises from intrinsic resource sharing will become a major source of variability. One of the major implications of this hypothesis is that *temporal variability*, where the performance of an individual processor changes over time, is set to become a much larger issue in these systems than it is currently understood to be.

3.1.2 System Objectives and Compositional Applications

While node architectures and their relation to performance variability are the primary focus of this chapter, there are additional criteria that suggest an increase in variance on future exascale machines. For one, the global system objectives in exascale machines will be multi-faceted, containing power and energy constraints, departing from nearly all previous HPC systems where the focus has been to optimize time to solution for a single application. This has driven a large portion of research both in global system-level power management [90, 70] as well as architectural advancements to intelligently limit the power consumption of different components [28].

The emergence of power as a scarce resource is also driving, in combination with growing I/O requirements, a change in the way applications are being programmed to utilize system resources. With HPC applications increasingly coupled both with high throughput sources of inputs (i.e., scientific instruments, real-time stock market streams, etc.) as well as post processing components that must analyze results, compositions that facilitate the integration of these workflows are gaining traction, as discussed extensively in Chapter 2.

The combination of these characteristics suggests that exascale machines may not be fully node partitioned, batch scheduled, and dedicated to a single application the way today's

supercomputers are. A clear implication of this is likely an uptick in performance variability due to nodes executing different combinations of workloads, and possibly with different power budgets to ensure energy and/or power efficiency.

3.1.3 Shortcomings of Existing Approaches to Variability

Of course, performance variability is not a new challenge. Previous approaches mitigate the impact of a particular source of variability - workload imbalance [25], network contention [16], operating system noise [42, 93], process variation [101, 18, 32], etc. - or more agnostically address variability by monitoring past application imbalance and using it to selectively inform future power redistributions. [90, 70]. Furthermore, asynchronous many task runtimes [13, 48, 49] that repartition workload among the nodes in response to imbalance have recently received much attention as a potential solution to variability.

We contend that while these efforts are a step in the right direction, their core theoretical frameworks are based on simplifying assumptions of how variability manifests in workload imbalance that, in light of the aforementioned exascale system characteristics, may not hold in the future. We enumerate some of the major assumptions here:

1. Variability creates "slow processes," which are commonly attributed to manufacturing defects [18].
2. Variability creates "slow nodes," which are attributed to, among many issues, manufacturing defects or placement within a network topology [106].
3. Variability persists over time. That is, if a process or node was "slow" or "fast" in a past iteration, it will be so again in the future. This assumption is made by many runtime systems and middlewares that redistribute resources based on past observed imbalance. [90, 70, 49]

With the criteria that motivated these assumptions now changing (homogeneous systems, single system objective, dedicated nodes/resources), we argue that a performance evaluation framework is needed that can characterize performance variability and allow us to determine whether or not these assumptions will continue to hold.

3.2 VARBENCH

In order to investigate these assumptions, we need a statistical framework that allows us to more holistically study the manifestation of variability. Specifically, this framework should allow us to answer the following questions:

- To what degree does performance vary in space (across identical processors or nodes)?
- To what degree does performance vary in time (across iterations of the same workload on the same processor)?
- To what degree does variability generate "slow spatial outliers" - i.e., processors that are consistently slower than the rest of the machine?
- To what degree does variability generate "slow temporal outliers? - i.e., iterations within an instance that are occasionally slower than the majority of iterations?
- To what degree do "slow nodes" influence imbalance on large scale machines?

We designed the `varbench` tool to characterize performance variability on a given machine. Varbench provides a set of workloads that have different requirements for resources from the underlying architecture. By examining the behavior of these workloads, a user can determine to what extent variability will impact different classes of workloads on their machine.

While we would like to examine these questions on real exascale machines, these systems are not yet available, so we are required to take a different approach. Fortunately, there are characteristics expected of exascale machines that can be seen in some of the more recent node architectures available today. One of these, as discussed in Section 3.1.1 is a high degree of **sharing** intrinsic to the system architecture. As a result, our primary focus in this section is to study variability in the context of a set of recent node architectures and to determine what, if any, useful observations or trends can be deduced by looking at how variability arises from sharing resources in different generations. We will use these trends to argue whether or not our current approaches to mitigating variability will be appropriate at exascale.

3.2.1 Core Methodology

In this section, we will discuss varbench's core features. 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. We leverage BSP-style workloads because they are most sensitive to variability and thus allow us to derive the most detailed measurements of imbalance.

Logically, varbench has three main components: *Kernels*, *instances*, and *iterations*. A *kernel* is the singular workload running during the course of the application. The same kernel runs on each node and processor in the machine. Section 3.2.3 will discuss the kernels used for this evaluation.

An *instance* can be thought of as a "rank" in traditional MPI terminology. Each individual processor in the machine runs a single instance. Thus, we 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). Varbench provides many options for determining precisely how instances are mapped to the underlying machine. In this chapter, unless otherwise noted, each instance is pinned to the finest granularity processor on the machine, and all memory accesses are pinned to the local socket. For example, on multi-socket NUMA architectures with hyperthreads, all instances are pinned to a single hyperthread, and all memory accesses are to addresses on the local NUMA socket.

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.

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

RV Statistic	Interpretation
`RV(S)` $= 0$	Low variance
$2 <$ `|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 7: Interpreting the RV statistic of a sample

3.2.2 Quantifying Resource Variability

The questions we seek to address require a statistical formulation of performance variability. Our approach is to understand performance as a sample drawn from an underlying probability distribution. Depending on the particular question being asked, the sample is selected in different ways. For example, when analyzing temporal variability, we select one sample per instance that consists of performance measurements for that instance across all iterations. Similarly, 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 coefficient of variance (CV), 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 3.1:

$$\texttt{RV(S)} = \left\{ \begin{array}{ll} 0, & \text{if } CV(S) < 0.01 \\ -1 * Kurtosis(S), & \text{if } Skewness(S) < 0 \\ Kurtosis(S), & \text{if } Skewness(S) \geq 0 \end{array} \right\} \tag{3.1}$$

To understand this metric, it is necessary to understand what CV, skewness, and kurtosis say about the shape of a sample. CV is defined as the ratio of a sample's variance to its mean, so samples with larger CVs have a larger degree of variance. Note that we leverage CV rather than simply variance because it is invariant to the mean, and thus is more readily comparable across applications and architectures.

Kurtosis is a measure of tail extremity, and thus indicates the degree to which sample

variance is driven by the presence of outliers [109]. 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 3 are *platykurtic*, while those with kurtosis greater than 3 are *leptokurtic*. Samples with kurtosis near 3 are said to be *mesokurtic*.

Finally, skewness is a measure of the (a)symmetry of a distribution, and can be used to determine whether the left hand side tail is longer, shorter, or similar in length to the right hand tail. Samples with skewness near zero are "balanced" with roughly equal density in the two tails, as in the normal distribution. Negative skew indicates that the sample left side tail is longer than the right hand side tail, while positive skew has the opposite meaning.

With this understanding, Table 7 demonstrates how to interpret the `RV` statistic. In cases where `RV` is 0, this indicates there is little variance in the sample. 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. The lepotkurtic cases have kurtosis significantly higher than the normal distribution, and thus these samples are more impacted by infrequent extreme outliers that occur "far" from the mean. The sign of the `RV` stat tells us whether these outliers are less than or greater than the mean, and thus, in the context of runtime, whether their occurrences are "faster" of "slower" than the mean. Finally, the platykurtic samples of course have high variance, meaning they are not "closely" distributed around a mean, but have low kurtosis, meaning the occurrence of extreme outliers is very low. Thus, these cases represent samples with either a single "broad" mode (e.g., a uniform distribution), or with multiple modes. We note that these thresholds are not inflection points that categorize samples in a rigid mathematical sense, but rather are simple guidelines for interpreting the statistic in most cases.

This formulation 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

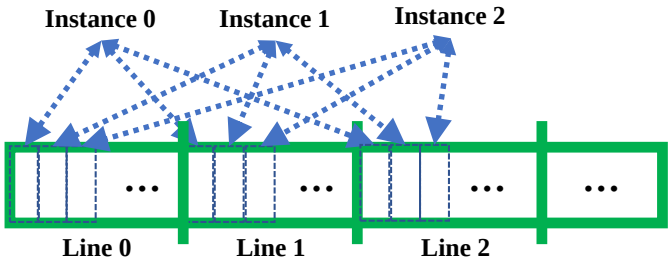


Figure 16: The Cache False Sharing (CFS) kernel

processors by calculating the `RV` stat over an iteration rather than an instance. In this way, the statistic is both simple and broadly useful for this analysis.

3.2.3 Kernels

A varbench *kernel* is the singular workload running on the machine. For the purposes of this analysis, we are interested in kernels that stress shared resources on the underlying architecture. 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 applications on modern architectures. To study LLC and interconnect sharing, we designed two kernels: *Cache False Sharing* and *Cache Capacity*. For memory subsystem resources, we used two kernels: *Random Access* and *Stream*. Finally, we also sought a more general kernel that does not target a specific resource, but rather represents a workload common to large scale HPC systems. For this, we leveraged the *Dgemm kernel*.

These kernels focus on the specific implications of sharing a particular resource. While large scale systems have many other shared resources, such as networks, I/O systems, and power budgets, we choose to focus primarily on memory and processor interconnects in this chapter, as these resources have been changing dramatically in recent years and provide interesting comparison points across architectures. We note that varbench can easily incorporate other workloads.

- *Cache False Sharing* (CFS) is designed to determine the impact of cache coherence traffic

required to share cache lines among instances. 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. At a high-level, this kernel is designed to measure the impact of frequent coherence traffic across inter-processor interconnects. This kernel is illustrated in Figure 16.

- *Cache Capacity* (CC) is designed to determine the impact of frequent LLC misses caused by capacity conflicts. 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. At a high level, this kernel is designed to measure the impact of concurrent memory requests bringing data from different cores into the LLC.
- *Random Access* is designed similarly to the Random Access benchmark from the HPCC suite, a benchmark that often maps directly to application performance [67]. In our system, each instance executes its own private version of the HPCC Random Access algorithm with no explicit sharing of data. This kernel has been shown to provide a good measurement of scalability for many HPC workloads.
- *Stream* is designed similarly to the STREAM benchmark from the HPCC suite, a benchmark that measures sustainable memory bandwidth in an architecture [67]. As in the case of Random Access, each instance executes its own private version of Stream without explicit 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 a workload common to HPC systems, that of measuring the performance of matrix-matrix multiplication. 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.

Processor Codename	Node Characteristics	Year Released
AMD "Opteron"	Dual socket; 6 cores @2.3 GHz 8 GB RAM per socket 64 KB per-core L1(i+d), 512 KB per-core L2, 6 MB L3	2003
Intel "Sandy Bridge"	Dual socket; 6 cores (12 HT) @ 2.2 GHz 12 GB RAM per socket 32 KB per-core L1(i+d), 256 KB per-core L2, 15 MB L3	2009
Intel "Ivy Bridge"	Dual socket; 6 cores (12 HT) @ 2.1 GHz 16 GB RAM per socket 32 KB per-core L1(i+d), 256 KB per-core L2, 15 MB L3	2013
Intel "Broadwell"	Dual socket; 18 cores (36 HT) @ 2.1 GHz 64 GB RAM per socket 32 KB per-core L1(i+d), 256 KB per-core L2, 45 MB L3	2015

Table 8: Characteristics of examined server architectures

3.3 PERFORMANCE ANALYSIS

In this section, we present results of the varbench benchmark on a variety of different node architectures. First, in Section 3.3.1 we analyze varbench on a set of dual-socket server server architectures that have commonly been used to run HPC workloads over the past decade. Then, in Section 3.3.2, we analyze varbench results from the Intel MIC architecture, illustrating how variability has evolved in the many-core era. Finally, in Section 3.3.3 we show how single node variability projects to program behavior at up to 512 nodes of a production HPC system.

3.3.1 Variability Across Generations of Server Architectures

The hypothesis of this chapter is that due to the increasing complexity and interconnectedness of node architectures, variability is evolving in a way that challenges existing assumptions. To investigate this claim, we executed the varbench benchmark on a set of 4 server architectures representative of what HPC workloads have been running on for the past decade or so. These architectures are based on the following 4 processor models: AMD Opteron, Intel Sandy Bridge, Intel Ivy Bridge, and Intel Broadwell. Further details of these architectures are given in Table 8.

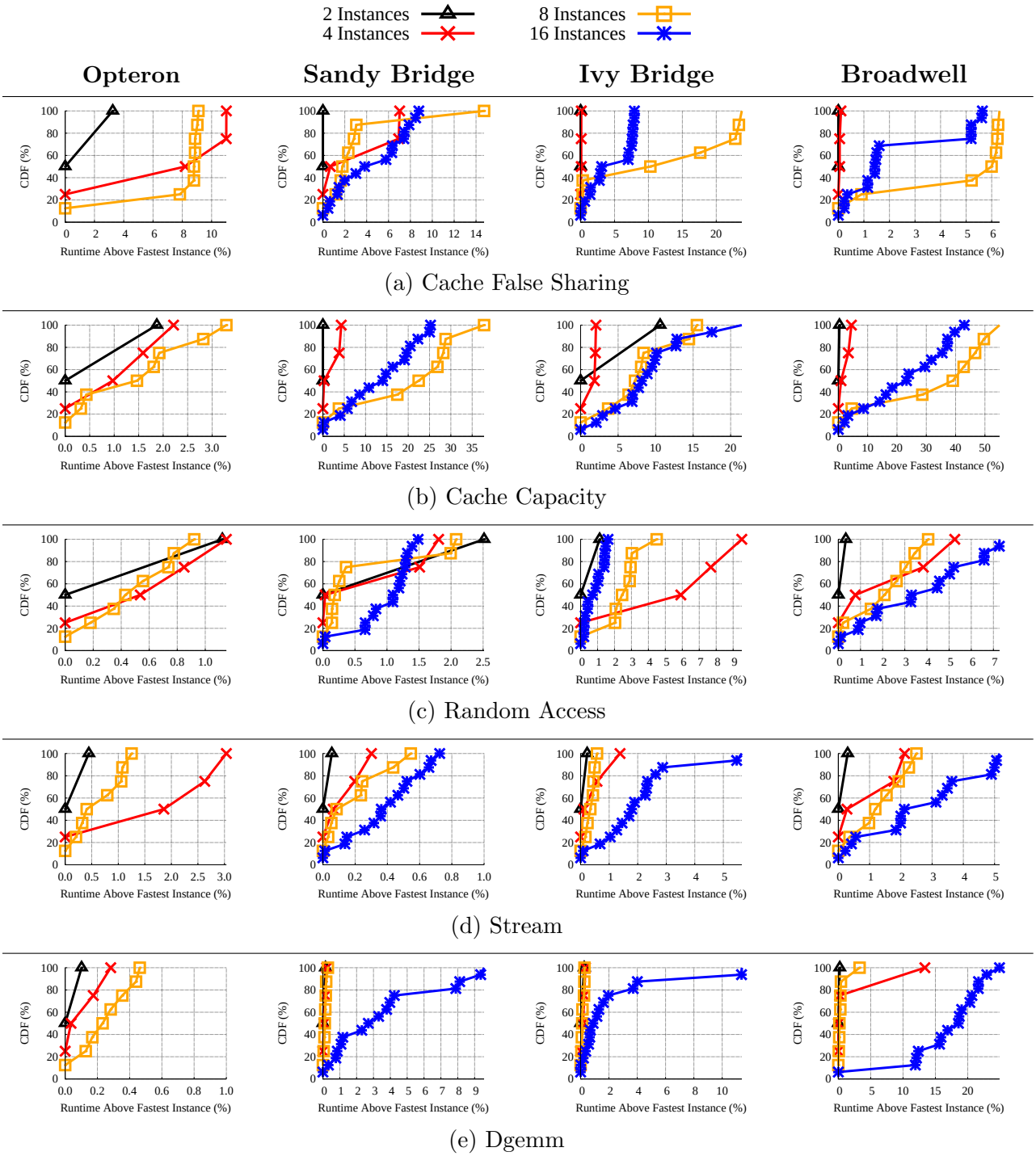


Figure 17: Spatial variability in various server architectures

3.3.1.1 Spatial Variability We first analyzed the characteristics of spatial variability on these architectures; that is, the difference in performance across multiple instances at a single iteration of each varbench kernel. Figure 17 shows, for each architecture and varbench kernel, a CDF of per-instance runtime, showing the runtime for each instance as overhead compared to the "fastest" instance. Furthermore, each figure shows the results for 2, 4, 8, and 16 instances. In each architecture, the 2 and 4 instance experiments only utilize one socket of the architecture, 8 instance experiments utilize both sockets, and the 16 instance experiments utilize both sockets as well as both hyperthreads for each core in the Intel architectures. In all cases, each instance is pinned to a specific logical core and only accesses memory from the local socket.[2]

There are two general observations to be made from Figure 17: (1) with few exceptions, cross-instance variability tends to increase with higher core counts; and (2) variability tends to increase in newer generations of server architectures, which can be observed by moving left-to-right across an individual kernel. We first study the latter phenomenon in the context of Figures 17(b) and 17(e). The Opteron architecture has significantly lower variability in these kernels compared to all other architectures. In Cache Capacity, this suggests that the Opteron's cache eviction policy is more fair than in the Intel systems. The Dgemm results likely reflect the impact of sharing the floating point unit between instances, as all cases with 8 or fewer instances show almost no spatial variability. Furthermore, the newest architecture, based on Intel Broadwell, shows significantly higher variability than the Sandy and Ivy Bridge architectures for these kernels, nearly doubling the maximum overhead at 16 instances.

In addition to understanding the maximum spatial variability across instances, it is also important to consider additional metrics that describe the shape of the distributions. Table 9 shows the `RV` statistic for each of the CDFs in Figure 17. In order to more easily make observations from this statistic, each cell is colored based on the value in that cell. The red and green cells illustrate that this experiment had "slow" or "fast" outliers, respectively, in a given iteration (the leptokurtic cases in Table 7). As the table shows, for the most

[2]In Cache False Sharing, half of the instances in the 8 and 16 instance cases access remote memory, as there is a single global array shared across all instances. Note, however, that in this benchmark, data is always resident in a cache line somewhere, and thus remote socket memory accesses are rare.

Architecture	Kernel	Number of Instances			
		2	4	8	16
Opteron	Cache False Sharing	1.000	-2.141	-5.959	-
	Cache Capacity	0.000	0.000	1.727	-
	Random Access	0.000	0.000	0.000	-
	Stream	0.000	-1.981	0.000	-
	Dgemm	0.000	0.000	0.000	-
Sandy Bridge	Cache False Sharing	0.000	-1.021	5.672	-1.416
	Cache Capacity	0.000	1.042	-2.060	-1.775
	Random Access	1.000	0.000	0.000	0.000
	Stream	0.000	0.000	0.000	0.000
	Dgemm	0.000	0.000	0.000	2.090
Ivy Bridge	Cache False Sharing	0.000	0.000	-1.255	-1.285
	Cache Capacity	1.000	0.000	2.217	3.183
	Random Access	0.000	-2.035	-3.461	0.000
	Stream	0.000	0.000	0.000	3.686
	Dgemm	0.000	0.000	0.000	5.007
Broadwell	Cache False Sharing	0.000	0.000	-2.389	1.674
	Cache Capacity	0.000	1.274	-2.002	-1.727
	Random Access	0.000	1.261	-1.791	-1.642
	Stream	0.000	0.000	0.000	-1.727
	Dgemm	0.000	2.333	5.971	-5.123

Table 9: Spatial resource variability (RV) statistic

part, these cases are not particularly prevalent, with only about 14.7% (5 out 34) of all cases with nonzero `RV` showing outliers of either kind. Rather, we see many more cases with normal (yellow cells) or platykurtic (orange cells) shapes, indicating the variability is more broadly distributed across instances in the system. This result strongly suggests that, when sharing resources of the architecture, variability does not generate "slow" outliers as commonly assumed, but rather has an impact on the performance of all instances.

3.3.1.2 Temporal Variability These experiments lend evidence in support of our hypothesis that more modern architectures are susceptible to more performance variability. However, in addition to spatial variability, we are also interested in determining the extent to which temporal variability arises, and to determine whether it also is more prevalent in recent systems. To study temporal variability, we focus not on performance across instances in a given iteration, but rather within a single instance across all iterations. We refer to the difference between the highest and lowest runtime of an individual instance over the entire kernel as the instance's range. For each instance in an experiment, we calculate its range

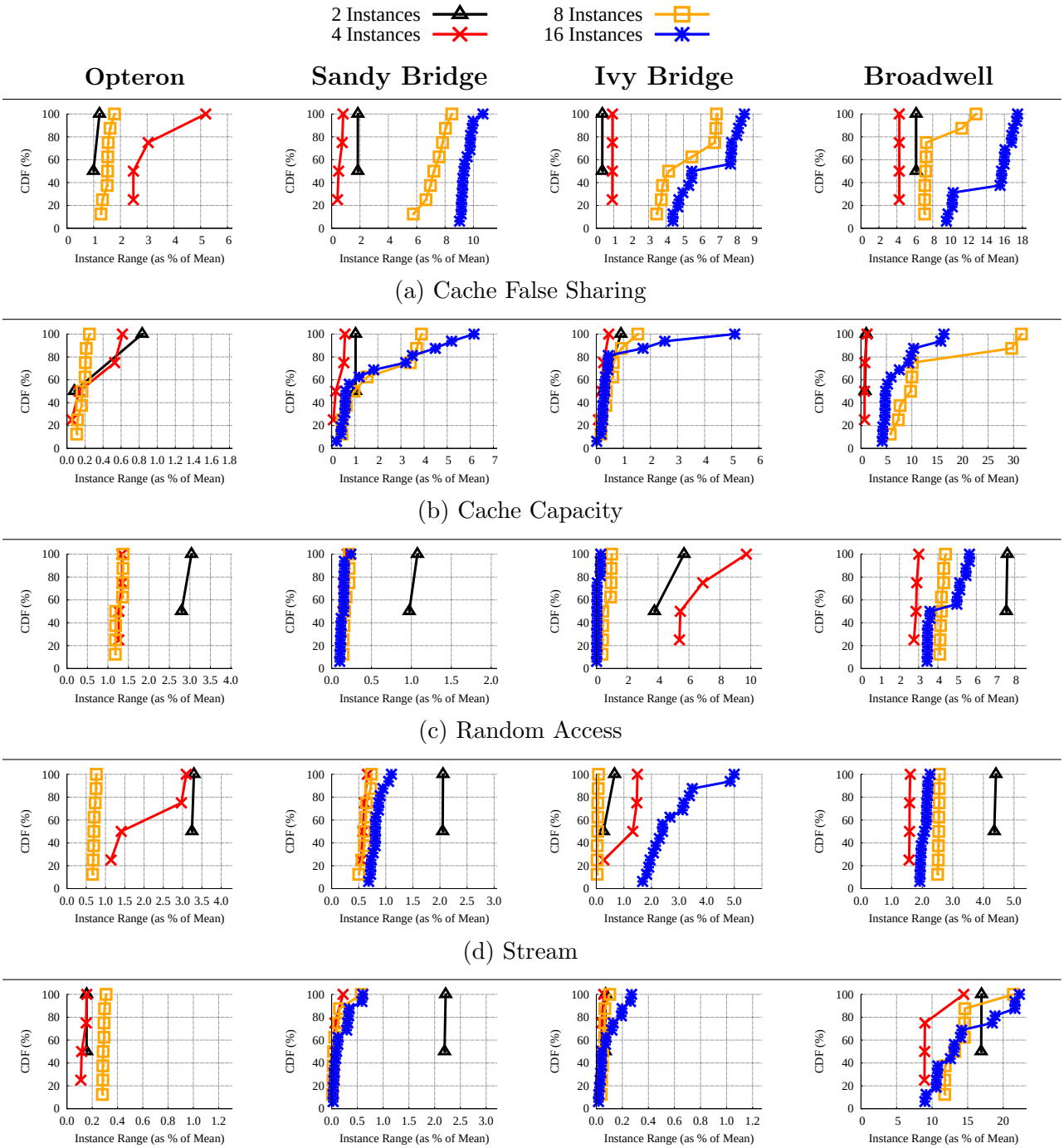


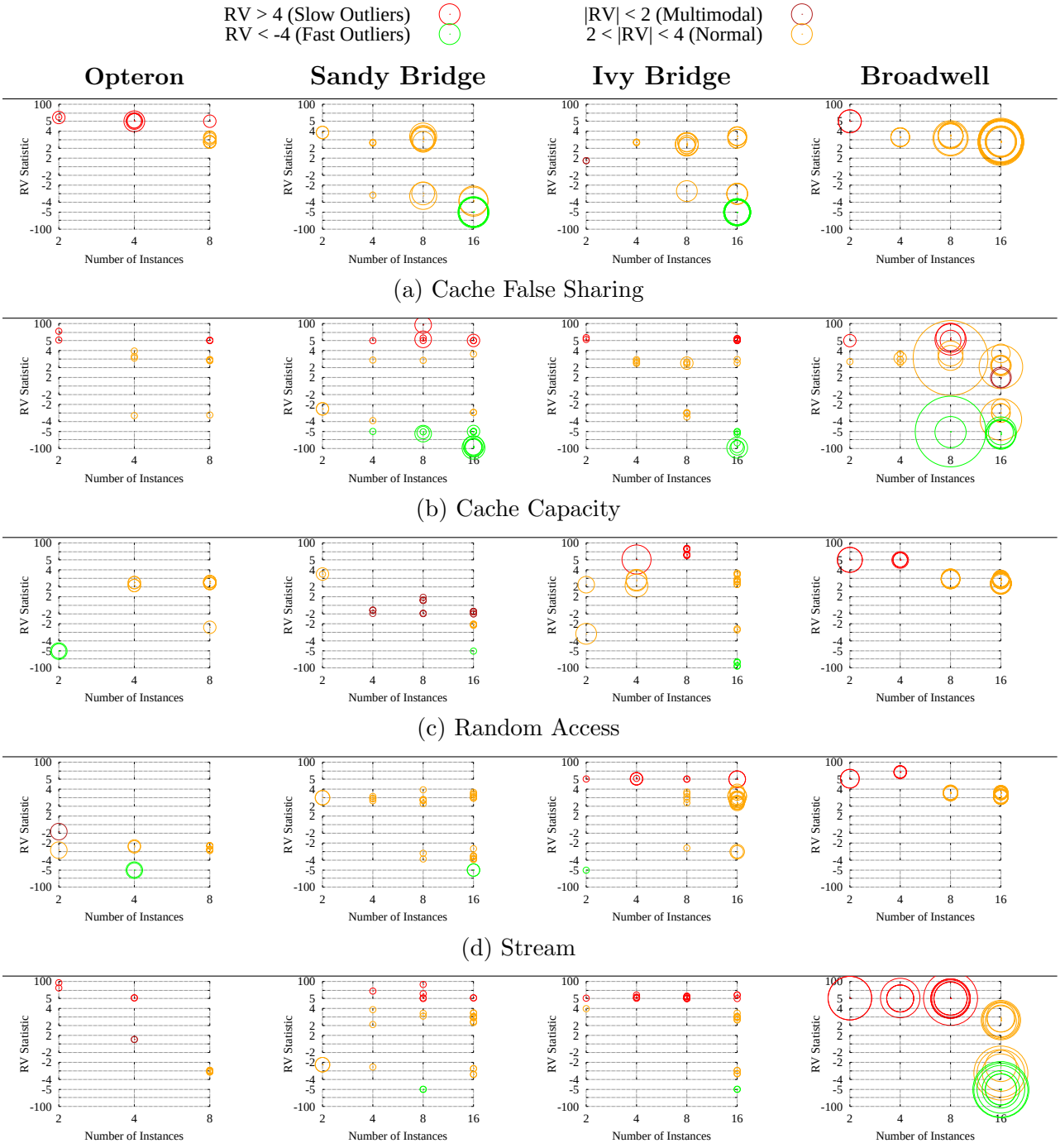
Figure 18: Temporal variability in various server architectures

as a percentage of its mean, and plot the CDF of each instance in Figure 18. As in the spatial discussion, we first focus on Figures 18(b) and 18(e), which show that, once again, variability for Cache Capacity and Dgemm becomes more pronounced in newer architecture generations. For the remaining kernels, cross-architecture differences are less pronounced, but once again, the Opteron architecture is almost always the most consistent, and Broadwell is the most susceptible to variability, particularly at higher instance counts.

These experiments strongly suggest that not only does variability lead to a wider distribution across processors in newer architectures, but also that each processor experiences a wider range of performance over time. This result has a major implication: it is, in general, not possible to statically characterize the variability of an architecture based on observations made at a single point in time. Importantly, this implication refutes assumptions made in several recent parallel runtimes that mitigate variability by focusing on recently observed imbalance [90, 70, 49] and projecting it as an indicator of future performance.

While this result is interesting in and of itself, it alone does not explain how variability with an instance is distributed. To provide this insight, we again utilize the `RV` statistic, this time calculated on a per-instance basis over all iterations for that instance. Figure 19 presents a scatter-plot for each kernel and architecture. Each circle on the plot represents two characteristics of an individual instance: its `RV` value, which indicates where it is placed vertically within a plot, and its diameter, which reflects how large the instance's range is in comparison to its mean; thus, circles that are large are from instances with large temporal variability. Additionally, circles are colored based on the category that the `RV` statistic falls into, following the same scheme from Table 9.

One of the important characteristics from this Figure is the lack of large red circles, representing "slow" outliers, in the `RV` range from 5 to 100. Broadwell is the only architecture that exhibits these to any significant degree, which occur in the 2, 4, and 8 instance cases of the Dgemm kernel. For these instances, there are a small number of iterations with significantly slower performance than normal, and are likely the result of software interference. Interestingly, at 16 instances with the addition of hyperthreads, these slow outliers tend to become "fast" outliers or become more normally distributed, suggesting that hardware sharing/interference is the more likely culprit.



(a) Cache False Sharing

(b) Cache Capacity

(c) Random Access

(d) Stream

(e) Dgemm

Figure 19: Temporal resource variability (RV) statistic

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 10: KNL configurations analyzed

This trend is present not just in Dgemm, but also in the remaining architectures and kernels. In no case does the largest instance count for any of the architectures exhibit "slow" temporal outliers. This is strong evidence to the contrary of another commonly held observation, which is that temporal variance, when it does exist, is attributed to transient software events such as OS interference. These results demonstrate that hardware competition creates interference that is distributed in an application and architecture specific manner that, once again, is difficult to statically characterize.

3.3.2 Variability in Many-core Architectures

Our analysis thus far has focused on typical dual socket server architectures that have composed most HPC systems over the past ten years. However, recent years have seen the birth of heterogeneous and distributed node architectures that both expose a much larger degree of parallelism to applications as well as more diverse memory technologies to provide the requisite bandwidth for the many cores on chip. One representative example is the Intel MIC architecture. In this section, we analyze a system based on the Intel Knight's Landing processor, focusing on how different configurations of the node's resources create variability for varbench workloads.

Our analysis will focus on specific KNL resources that can be configured in different manners and determine to what degree these configurations create variability. Specifically, we analyze a set of clustering modes and configurations of the on-package MCDRAM modules, detailed discussions of which are presented elsewhere [95]. The configurations we analyzed are listed in Table 10. For each of these configurations, we ran each varbench kernel with varying numbers of instances. Table 11 shows how we mapped instances to the processors

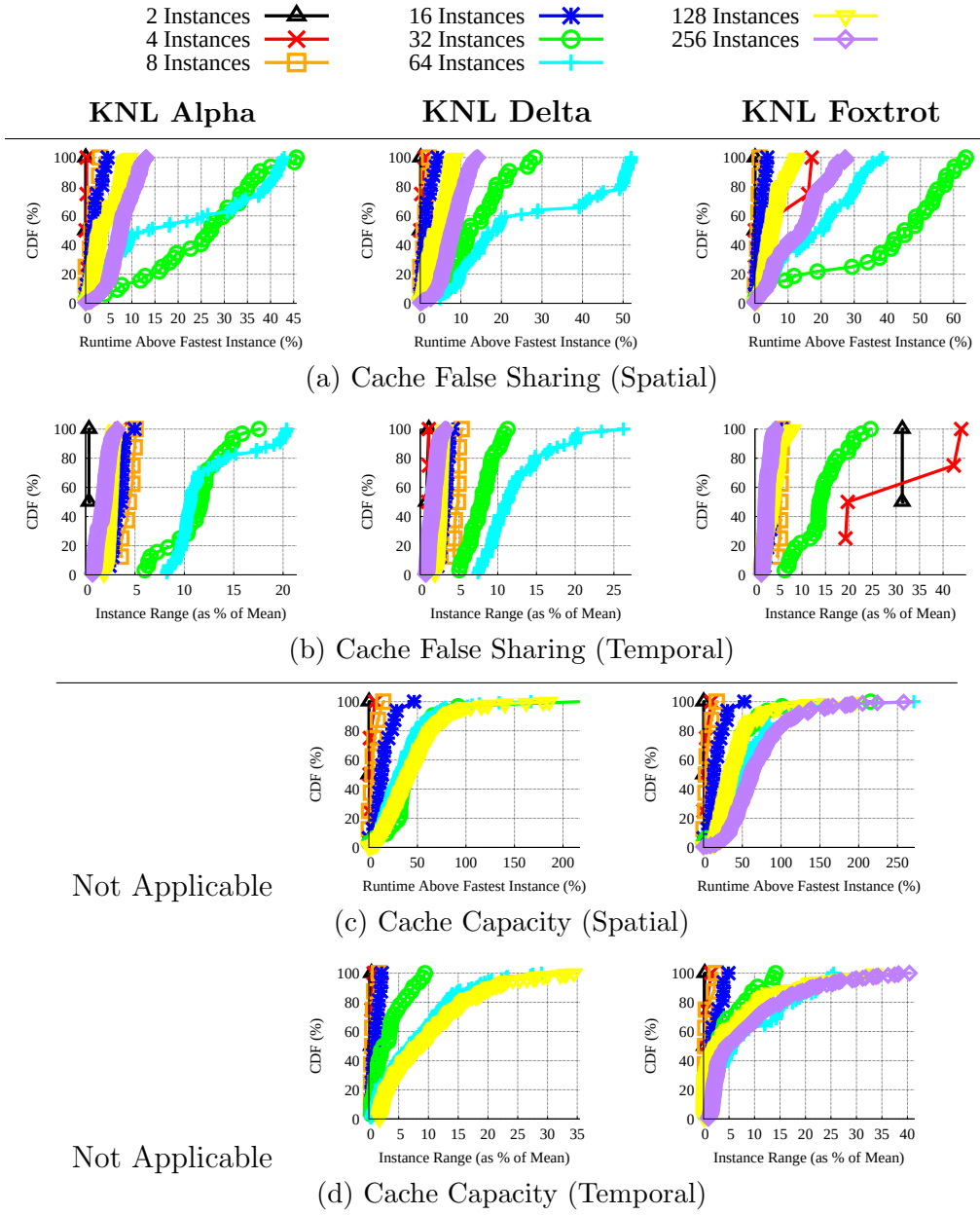
# Instances	# Tiles	# Cores / Tile	# Threads / Core
2	1	2	1
4	2	2	1
8	4	2	1
16	8	2	1
32	16	2	1
64	32	2	1
128	16	2	4
256	32	2	4

Table 11: Mapping of Varbench instances to the Knight's Landing architecture

on the chip. For configurations with fewer than 72 instances (the total number of cores on the chip), we choose to run two instances per tile, one on each of the tile's cores. For the 128 and 256 instance cases, we utilize multiple hardware threads of each core. It is important to keep in mind, then, that different instance counts reflect not just an increase in "scale" within the node, but also a different utilization of the tile and core-local resources, such as L1/L2 caches.

3.3.2.1 Cache Kernels Rather than separately characterize spatial and temporal variability for each workload and system configuration, we consider the two in concert in order to better understand the broader implications of the KNL configuration options for specific workloads. Figure 20 shows the manifestation of spatial and temporal variability in the two cache sensitive kernels, Cache False Sharing and Cache Capacity. Note that in contrast to the dual-socket architectures with shared last level caches, in these experiments, what constitutes the last level cache depends on the configuration of the architecture. For the Delta and Foxtrot configurations, MCDRAM is treated as an LLC in that it is not directly addressable by the OS, but rather transparently caches memory between the last-level on chip L2 caches and DDR4. In the Alpha configuration, MCDRAM is OS-addressable, and thus the last-level cache is the per-tile L2 cache.

Because of this, we configured our cache kernels to stress both the capabilities of the MCDRAM memory when treated as cache, as well as the on-chip L2 caches. For the CFS



(a) Cache False Sharing (Spatial)

(b) Cache False Sharing (Temporal)

(c) Cache Capacity (Spatial)

(d) Cache Capacity (Temporal)

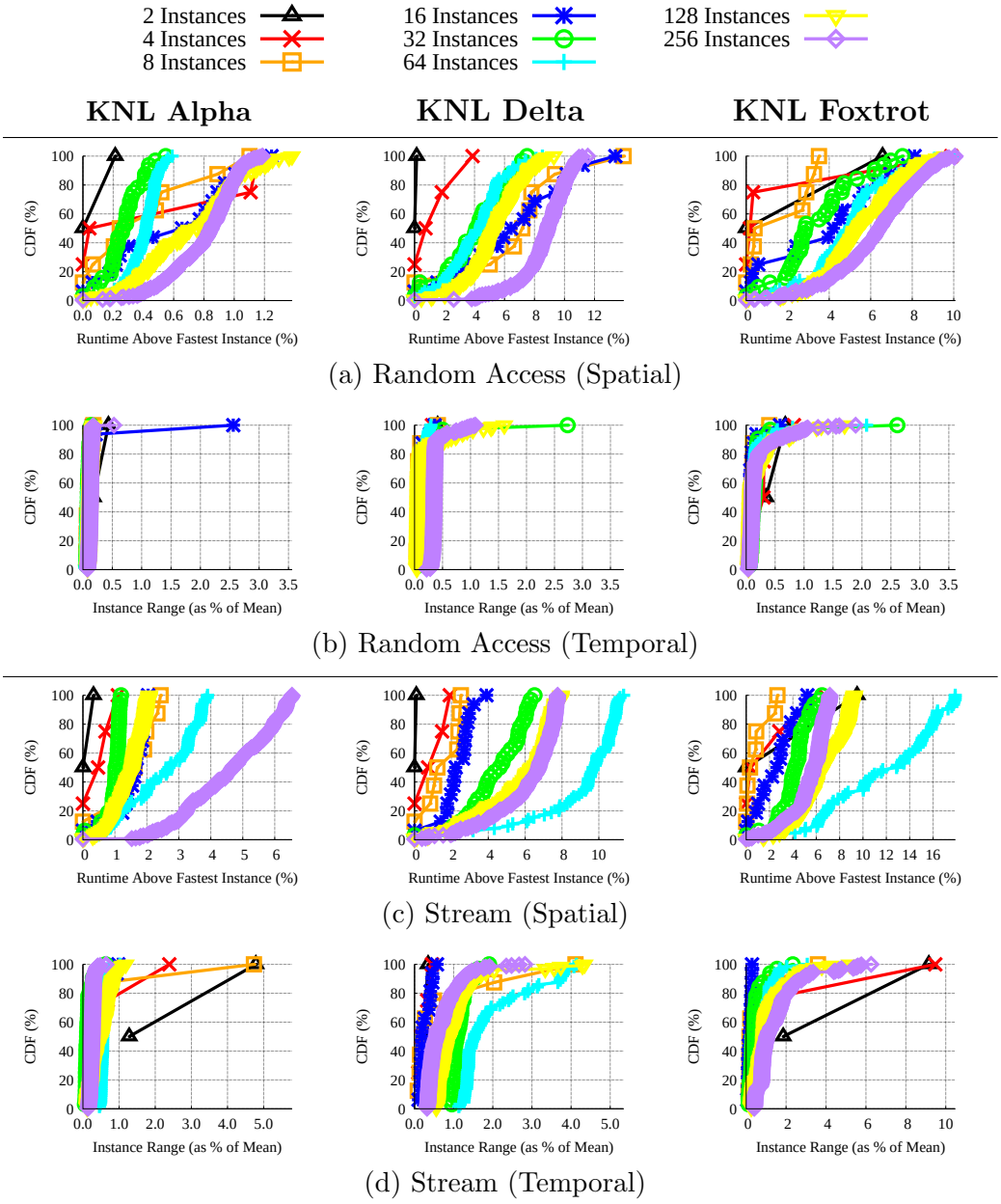
Figure 20: Variability in cache kernels on Knight's Landing

kernel, our goal is to study the impact of frequent coherence traffic being routed between tiles do the sharing of cache lines between private L2s. Thus, a single array is allocated that fits in a single tile's 1MB L2, and each instance accesses the same cache line from this array to generate cross-tile coherence traffic. On the other hand, in cache capacity, we wanted to measure the extent to which capacity misses in the MCDRAM cache would induce variability. Thus, for this kernel, the instances allocate 32GB of memory in aggregate, forcing capacity misses to DDR4 on roughly one half of their accesses.

We see first that the CFS workload creates a large degree of spatial variability (Figure 20(a)), particularly in the 32 and 64 instance experiments which generate the most cross-tile coherence traffic, with instances ranging from 45% to 60% slower than the fastest instance. Furthermore, we also see a significant degree of temporal variability (Figure 20(b)) for the 32/64 instance configurations. This suggests that performance for this type of workload is not necessarily a function of placement in the architecture, but rather a function of which L2 a particular cache line happens to exist in at some point in time. Interestingly, the 128 and 256 instance experiments yield less variability than the 32 and 64 cases. The former introduce a greater degree of tile-local resource sharing as these configurations pack 8 instances onto each tile. The result is that the resource bottlenecks shift to tile-local resources that are only contended by a smaller number of instances running on the tile.

Figure 20(c) demonstrates that Cache Capacity creates even more spatial variability, with "slower" instances taking more than twice as long to complete an iteration than faster instances. We also see a large degree of temporal variability for some instances, with nearly half of the instances achieving mostly consistent temporal performance ($< 5\%$ of instance range), while the other half have ranges between 5% and 40% of their means (Figure 20(d)). This result demonstrates that the on-chip interconnects and routing protocols used to communicate memory requests, as well as caching/eviction policies in the MCDRAM module, create significant spatial and temporal variability.

3.3.2.2 Memory Kernels We next focus on the two memory subsystem kernels, Random Access and Stream, that also stress the capabilities of the memory modules, but are not explicitly designed to measure the impact of coherence and eviction policies. Figure 21



(a) Random Access (Spatial)

(b) Random Access (Temporal)

(c) Stream (Spatial)

(d) Stream (Temporal)

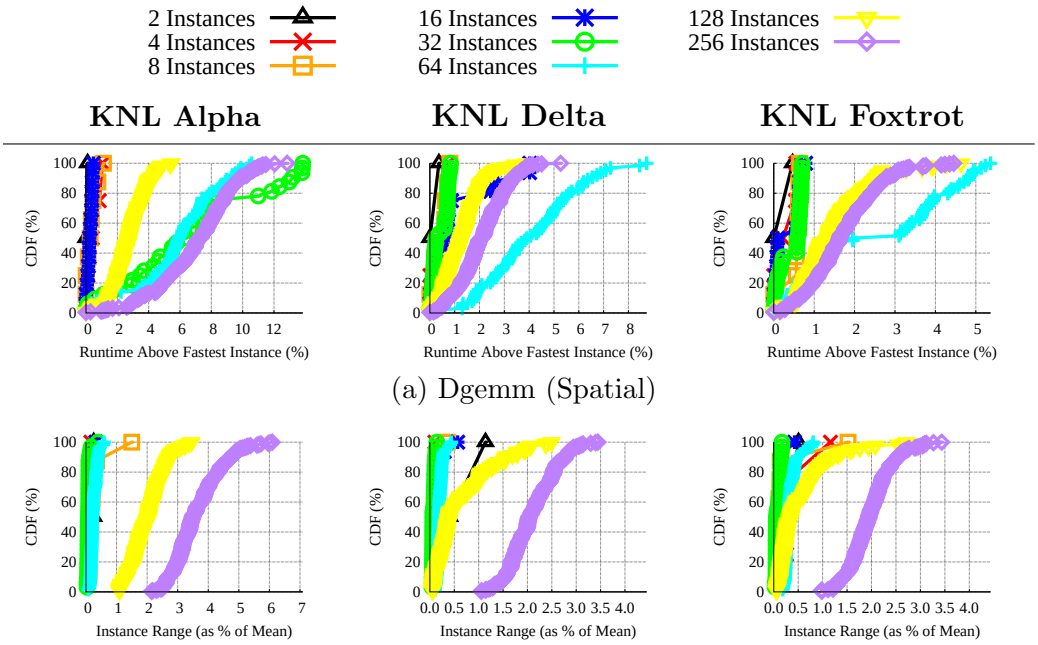
Figure 21: Variability in memory subsystem kernels on Knight's Landing

illustrates their results. Focusing first on Random Access, we see that, in contrast to the two cache kernels, the choice of KNL memory mode has a large impact on spatial variability (Figure 21(a)). In the KNL Alpha configuration, all memory accesses that miss in the tile's L2 cache bypass the MCDRAM modules and go directly to one of the two DDR4 memory controllers. In this scenario, performance is highly uniform across instances. However, in the Delta and Foxtrot configurations, these memory accesses go to MCDRAM, and we see a much larger degree of variability between instances. Furthermore, we also see that, in contrast to the cache kernels, temporal variability is mostly non-existent in this kernel. This indicates that placement within the architecture or differences in cross-instance memory access patterns drive variability, as opposed to interference in routing and communication on the chip.

Stream's performance (Figures 21(c) and 21(d)) mostly reinforces the observations made in the Cache Capacity experiments. Though at some level these workloads are similar, Stream's KNL Delta and Foxtrot results show significantly less variability than what we observed in the Cache Capacity workloads. This reinforces our conclusion that variability arises from frequent cross-chip messages to route memory requests from MCDRAM to DDR4 (as required for Cache Capacity), not contention for the MCDRAM/DDR4 memory modules themselves.

3.3.2.3 Dgemm Lastly, Figure 22 shows the results of the Dgemm experiments. Because Dgemm is not designed to explicitly target any particular shared resource, it's performance is likely to reflect some combination of characteristics from the other kernels based on which resource happens to be its bottleneck.

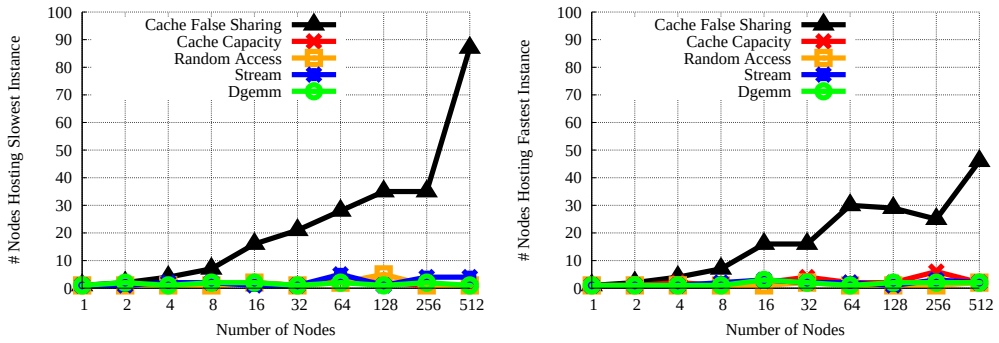
It makes sense to interpret Dgemm's results in two sets: the 128/256 instance cases, and everything else. For the former, recall that these are the only configurations in which more than 1 SMT thread per core is utilized. Thus, based on the understanding that Dgemm is floating point intensive and there is only one FPU per core shared by each SMT core, it makes sense to see that the 128/256 cases have distinct temporal variability profiles compared to the remaining experiments. It is clear by observing the spatial variability results that almost all variability in this kernel is temporally variant.



(a) Dgemm (Spatial)

(b) Dgemm (Temporal)

Figure 22: Variability in Dgemm on Knight's Landing



(a) Distribution of Slowest Instance (b) Distribution of Fastest Instance

Figure 23: Distribution of tail instances in a 512 node machine

For the remaining experiments ($\leq$ 64 instances), temporal variability is mostly non-existent, which suggests variability is primarily a function of static characteristics, such as placement within the architecture. In fact, these results are most similar to the Stream results, which suggests that Dgemm's bottleneck in these cases is simply memory bandwidth from either the DDR4 or MCDRAM controllers. Dgemm thus is not particularly prone to variability arising from on-chip message routing in the KNL architecture.

3.3.3 Variability at Scale

The last component of our analysis is to understand how variability impacts the scalability of a workload. In this section, we analyze the performance of varbench on up to 512 nodes of a production HPC system, focusing on the implications of single node variability for performance at scale.

3.3.3.1 Infrastructure The system we used for our analysis is the 1.2 petaflop "Serrano" capability machine located at Sandia National Laboratories. This machine is composed of 1,122 nodes with dual-socket, 36 core Broadwell processors with 64 GB of memory per socket, interconnected via the Intel Omni-Path network.

3.3.3.2 Variability Across Nodes

One of the commonly made assumptions in the HPC community is that "slow nodes" inhibit the progress of applications at scale. In order to investigate this hypothesis, we analyzed the tail behavior across instances for each of the varbench kernels. For each kernel, we determine the number of nodes that at any point in time was the node on which the slowest instance in the system executed (that is, the last instance to complete an iteration). We also analyzed the extent to which fast nodes exist, defining a fast node as a node that consistently hosts the first instance to complete an iteration. Finally, to provide a broader view of the tails of the distribution, we also analyze the number of nodes which at some point host the 64 slowest or 64 fastest instances to complete an iteration.

Figure 23 illustrates these numbers. First focusing on Figure 23(a), it is clear that all kernels but Cache False Sharing are impacted by the presence of slow nodes, as only a small percentage of nodes ever host the slowest instance in the system. Cache False Sharing is unique in that it is neither CPU nor memory intensive, but rather measures the implications of cache coherence traffic. This suggests that LLC characteristics are mostly consistent across nodes even in the presence of "slow" or faulty nodes, while memory and CPU resources are not. We see similar behavior when looking at the fastest instance in Figure 23(b). These results suggest that outlier nodes do exist in this architecture, and thus future runtime systems should be cognizant both of significant cross-node variability as well as intra-node variability.

3.3.3.3 Impact of Variability on Runtime

Finally, to understand the impact of single node variability on runtime, we calculated the runtime of each kernel as the sum of runtime across all 100 iterations, where the runtime of an iteration is the maximum runtime of all instances in that iteration.

Figure 24 shows the runtime results for each kernel. This figure demonstrates, importantly, that the extent to which a single node generates variability is a good indicator of how well a kernel will scale. Recalling the results from the Broadwell architecture in Section 3.3.1, which is identical to the architecture in our multi-node system, we saw that the Random Access and Stream kernels performed most consistently both in space and time, and

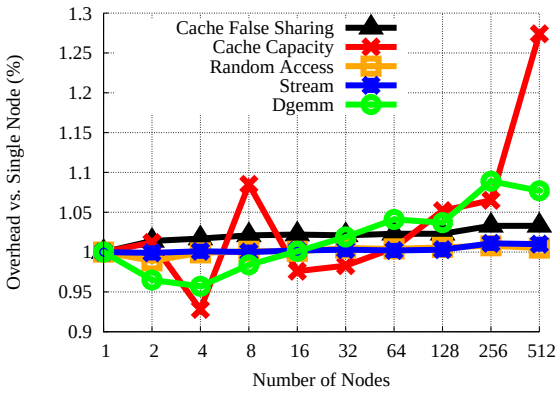


Figure 24: Scalability of Varbench kernels

both of these kernels scale well to 512 nodes. Cache Capacity and Dgemm showed the most variability, both in space and time, and they both demonstrate erratic behavior, ultimately suffering 27% and 8% overhead at 512 nodes. Finally, Cache False Sharing demonstrated variability between these two classes, and it exhibits modest slowdowns on this machine.

3.4 SUMMARY

In this chapter, we introduced the varbench performance variability evaluation framework. Varbench allows a user to characterize the performance variability of an architecture along two key dimensions: space and time. By using varbench, we studied the performance variability that arises from intrinsic resource sharing in modern server architectures as well as a recent many-core system. Our results indicate that performance variability is becoming a larger issue as architectures become more distributed and complex. Furthermore, our results bring into question some assumptions commonly made in the HPC runtime systems community, including that variability creates "slow" outliers and that variability is persistent over time. Finally, we showed evidence that small scale variability correlates with performance degradation when scaling to many nodes of a distributed system.

4.0 MODELING VARIABILITY WITH CRITICALITY MODELS

In Chapter 3, we examined several common assumptions about how variability manifests on today's machines, and called into question the validity of these assumptions for exascale systems. Our research indicates that the way in which variability impacts program behavior is both highly dependent on workload characteristics as well as the architecture on which the workload executes. In this chapter, we leverage this knowledge to derive models of performance variability called *criticality models* [58]. The core idea behind criticality models is to provide a scalable mechanism by which a large scale machine can derive predictions of where critical processes are running - that is, processes most impacted by variability.

Ours is not the first effort to model criticality. Existing work has shown that a modeling based approach can be used to drive power management decisions in order allocate more power to tasks that are more impacted by variability, thereby improving energy utilization, runtime, and scalability [90, 103, 11, 70]. Other efforts have shown that accurate models of variability can guide load-balancing efforts, whereby less impacted partitions steal work to speed up computation [33, 7]. In general, these efforts indicate that applications can scale, not by eliminating variability, but by detecting and reacting to it, either in the underlying system software or directly in the application source. However, criticality models have a distinguishing characteristic that is uniquely valuable in the context of future exascale machines: they are built in such a way as to reflect the prevalence of *temporal variability*. In contrast, existing efforts assume simpler models that reflected the characteristics of variability on past generations of systems. But in light of the concerns raised by our varbench analysis in the previous chapter, we claim these assumptions are not a good fit for extreme scale.

To reflect the expected rise in temporal inconsistencies, criticality models use low-level

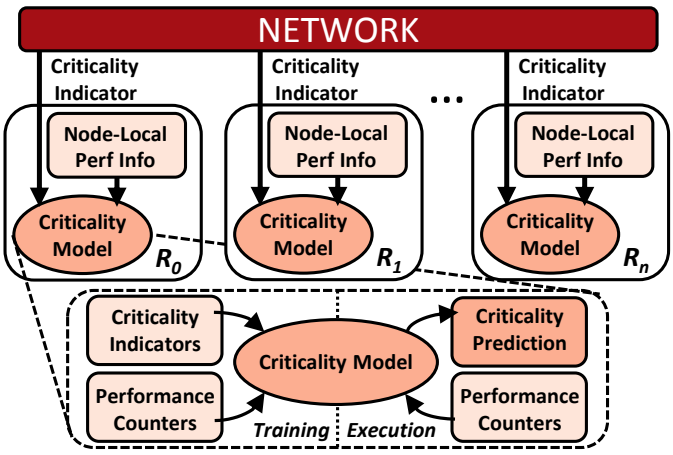


Figure 25: A high-level view of criticality models.

performance measurements and system-wide criticality labels based on application timing variability in order to learn how variability manifests itself, allowing it to be quickly and easily measured by a process. Once generated, criticality models can be used at anytime to estimate how critical a node's performance is to the progress of the application as a whole. A high-level view of our criticality modeling approach is shown in Figure 25. Each rank contains an internal criticality model used to predict its likelihood of being critical to the application's progress (i.e. being one of the stragglers). By grouping ranks based on their criticality predictions, the models collectively produce a *critical set* of ranks whose members are most likely to delay progress. Most importantly, once generated, criticality models require no communication between ranks – each rank executes its model independently and makes its own predictions of critical set membership. As a result, criticality models are more scalable than approaches that require frequent communication to determine criticality. Furthermore, because criticality models are agnostic to sources of variability, they are well-suited to address the myriad sources that will impact extreme scale systems, including complex causes of temporally inconsistent variability.

This chapter makes the following main contributions: 1) It proposes criticality models as a mechanism for capturing execution imbalance caused by performance variability and

discusses areas where criticality models can be used to support higher-level exascale services; 2) It evaluates the classification accuracy of criticality models on a representative set of HPC mini-applications and benchmarks; 3) It shows that criticality models can predict variability more accurately than simpler models which consider variability to be temporally consistent.

4.1 CRITICALITY MODELS

The previous section demonstrated that performance variability is a major impediment to the performance of current large scale HPC systems. Unfortunately, there are indications that performance variability will be even more of an issue at exascale. Given the emergence of power as a scarce computational resource, exascale systems may be power over-provisioned in a non-uniform manner so as to limit system power consumption [89, 91, 79, 80]. Furthermore, nodes will be more heterogeneous, both within the resources of a given node (e.g., big/small cores, co-processors, etc.) and among different nodes. Finally, multiple workloads will likely be consolidated on the same nodes to fully utilize the diverse infrastructure [116], as well as to conserve power normally required to move data between nodes [97]. Collectively, these issues will lead to increased variability across the system.

4.1.1 A Case for Criticality Models

Based on the sheer number and complexity of sources that will induce variability, a top-down approach to eliminating all sources will be challenging to achieve. Thus, we propose *criticality models*, a learning based mechanism that allows the system to generate holistic models of performance variability as it occurs during application runtime. Criticality models use observations of how performance variability manifests across a large scale system to automatically learn the local node-level performance characteristics that indicate criticality. Criticality models are designed to provide a mechanism by which applications and runtime systems can detect and react to variability in an intelligent manner rather than preventing performance variability from occurring.

An accurate model of criticality will be useful across a variety of domains. Researchers have demonstrated that models of criticality lend themselves to power management approaches, whereby tasks deemed less critical (i.e., less impacted by variability) can execute at lower CPU frequencies, thereby saving power and energy [90, 103]. Furthermore, this power and can be "shifted" to tasks that are predicted to be more critical, thereby improving performance and energy efficiency without increasing total system power consumption [70, 11].

Accurate models of criticality can also be used to guide work stealing approaches, where processors executing non-critical tasks perform additional work to reduce runtime [33, 7]. Furthermore, while these mechanisms are usually not incorporated into application source code, there are some applications that directly attempt to rebalance workloads as a result of performance variability [83, 25]. While current load balancing techniques directly measure progress at synchronization points, an accurate model of criticality could provide more fine-grained opportunities to rebalance workloads in between synchronization points, leading to performance gains.

Finally, simply providing information about how and where variability is occurring could be useful to applications and their users. Many sources of variability are complex and require significant time to diagnose [84, 16, 46]. An accurate model of criticality can help in debugging efforts, where users or administrators could be informed precisely where bottlenecks are located, how they are manifesting in low level performance measurements, and how they are distributed among the system (e.g., spatial and/or temporal inconsistencies).

Our criticality models are designed with three key features: 1) the use of statistical modeling to select node-level characteristics causing criticality; 2) a distributed model generation approach using global observations of performance variability; 3) a parallel and autonomous model execution framework.

4.1.2 Statistical Model Training

Due to the complex nature of variability at exascale, generating heuristic based models that can capture all sources of variability is challenging. Instead, we propose the use statistical models to generate models. The key advantage of this approach is that features indicative

of variability can be automatically learned and selected during the model generation phase.

Statistical modeling techniques, such as those commonly used in machine learning, can generate accurate models for a wide range of applications and system architectures. Given that different applications are sensitive to different types of performance variability, a "one size fits all" model cannot perform well for all combinations of applications and systems. Thus, learned models can be superior in terms of model performance as well as human effort in generating tailored models for new applications and systems whose performance characteristics may not be easily understood.

4.1.3 Distributed Model Generation

Criticality models are generated in a distributed fashion. Each node in the system is responsible for periodically collecting low-level performance measurements as an application is running. These measurements may take the form of hardware performance counters, software events (e.g., percentage of time context switching, breakdown of user mode versus kernel mode time), and/or counters related to the network (e.g., bytes per second received at a network endpoint over an interval of time). Measurements can also contain application-level information, such as the most recent MPI call made by a rank.

Models can be generated from these measurements either online or offline. In either case, the key is that each rank-specific performance measurement must be labeled with an indicator of criticality to feed to the model generation phase. For online generation, each rank labels its performance measurements when it reaches the next global communication point. Labels are based on the rank's communication time (e.g., MPI slack) in comparison to the communication time required by all other ranks. Alternatively, models can be generated offline from a set of runs of the application. In this case, the criticality labels are still derived from discrepancies in communication time, but offline generation provides two additional benefits: (1) applications do not need to determine global time discrepancies to train the model at runtime, but rather can store local time information to be post processed later; and (2) larger sets of data points from multiple runs, and thus more observations of performance variability, can be generated and can lead to more accurate models.

4.1.4 Parallel and Autonomous Model Execution

Once criticality models have been generated, each node executes its model by collecting node-local performance measurements and generating a local prediction of criticality based on these measurements. The main benefit of this approach is a very high degree of scalability – nodes do not need to communicate with each other to derive predictions of their own criticality. This design is based on the observation that, as systems scale to potentially millions of nodes, requiring inter-node communication to execute a criticality model will be inherently unscalable, particularly if the results of the prediction need to be generated in a timely manner (e.g., redistribute power before the next collective). Our criticality models eschew global communication, instead allowing each node to predict its criticality based only on measurements that it can quickly and easily collect as the application is running.

The downside of this approach is that, in cases where an absolute consensus regarding criticality is required, there is a need for an additional layer of communication. For example, if criticality models are used to guide a "hard" power capping mechanism, node-level predictions may need to be aggregated by a higher-level framework so that power is not over-allocated if too many nodes claim to be critical. In general, if global consensus is required, the models are still valuable, but do not provide a complete solution.

However, in many cases such a consensus is not necessary. If resource constraints are more relaxed (e.g. "soft" power capping) the system could use a hierarchical or gossip-based approach [96] to communicate criticality predictions across a small partition of the system, such that decisions can still be made quickly. Furthermore, a more general power management mechanism could use criticality models to achieve power or energy efficient computing, even if a cap cannot be directly guaranteed without communication. In addition to power management, as discussed previously, work stealing, application-level workload redistribution, and performance debugging are all areas where global consensus is not required, and thus the benefits of criticality models will be directly attainable.

4.2 PROOF OF CONCEPT: CRITICALITY MODELING ON A SMALL CLUSTER

In this section, we show how criticality models can be built and executed on trace runs from MPI applications on a small scale cluster. While criticality models target variability on large scale systems, our initial experimental analysis focuses on demonstrating the effectiveness of the models in identifying the low level indicators of criticality.

4.2.1 Performance Measurements and Profiling

Criticality models are generated from low-level performance measurements collected independently by each rank. For this analysis, we have limited performance measurements to hardware performance counters, but as noted in Section 4.1.3 other types of measurements can be incorporated. Models are first trained with performance measurements from ranks deemed to be "critical" (i.e., in the critical set) or "not critical" (i.e., not in the critical set). For this analysis, these ground truth annotations are provided in an offline manner by post-processing MPI traces generated from application runs, using MPI timing information (e.g., slack) via the DUMPI [59] library. Whenever a rank reaches an MPI collective, it gathers performance measurements for a configurable amount of time. Once it reaches the next MPI collective, it records the amount of time until that collective completes and uses this as a proxy for the MPI slack for that collective.

4.2.1.1 Instruction Based Sampling We choose to collect performance counters using the Instruction Based Sampling (IBS) mechanism found in modern AMD processors [8]. When IBS is enabled, the processor generates an interrupt after a configurable number of cycles (or instructions) has elapsed and provides information about how the instruction was processed in the core's pipeline (e.g., TLB and cache hit rates, number of cycles elapsed since the instruction was issued, etc.). The main benefit of using IBS for this analysis is that IBS interrupts provide a wide range of instruction-level information that gives a clear picture of the current state of the microarchitecture. Specifically, we collected over 20 different

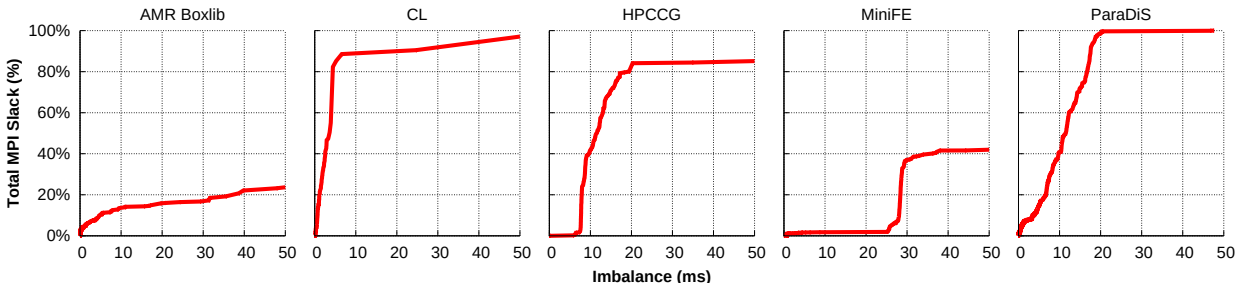


Figure 26: Cumulative sum of MPI slack by imbalance observed in the collective

performance measurements from each IBS instruction.[1] While many of these measurements can be made via regular performance counters, the primary benefit of using IBS is that a larger number of counters can be collected for an instruction than can be collected via the registers used for regular counters. The result is that IBS is a particularly good match for our learning based framework. As more parameters are fed to the model during the generation phase, it becomes more likely that influential hardware characteristics (those that are good predictors of criticality) can be identified automatically.

4.2.2 Constructing the Models

In this analysis, criticality models are constructed in an application specific manner. We select a representative set of mini-applications from the Mantevo suite [41], as well as two real applications. All models are trained offline from application traces collected on a small 4-node experimental cluster. Each node has a 4-core AMD A10-7850K processor with 16 GB RAM, with each node interconnected with MT27600 Mellanox InfiniBand cards. Each application is compiled against the DUMPI library to generate detailed MPI traces. Each rank is also configured to collect IBS performance counters during its execution. The applications are as follows: (1) AMR Boxlib – performs a single time step of an AMR (adaptive mesh refinement) run with compressible hydrodynamics and self-gravity. Snapshot of production AMR application from LBNL; (2) Cloverleaf (CL) – mini-app that solves the compressible

[1]The full list of IBS performance measurements can be found on page 597 of the AMD BKDG [8]

Euler equations on a Cartesian grid, using an explicit, second-order accurate method; (3) HPCCG – proxy application for unstructured implicit finite element code. Performs iterative conjugate gradient method for a configurable number of time steps; (4) MiniFE – proxy application for unstructured implicit finite element code. Similar to HPCCG, but provides a more complete vertical covering of steps in this class of application; (5) ParaDiS – application that simulates dislocation dynamics to calculate plastic strength of materials [83, 22].

The key parameter guiding the construction of the models is the *imbalance* observed over the course of an application's execution. We define the per-collective *imbalance* as the maximum MPI slack for a given collective. Figure 26 shows the cumulative sum of MPI slack in these applications, with imbalance in the associated collectives plotted on the x axes. The figure shows how much individual collectives of particular levels of imbalance contribute to the total slack in the application. Intuitively, growing quickly in the y-axis indicates that periods of small imbalance collectively add up to contribute significantly to the total slack experienced by the application, while growing slowly in the y-axis indicates that larger amounts of imbalance contribute more. As the figure demonstrates, CL, HPCCG, and ParaDiS all exhibit some of the former behavior, where small imbalance periods collectively contribute much of the slack. On the other hand, AMR Boxlib and MiniFE are impacted more by collectives with larger imbalance.

To effectively model applications that exhibit either type of imbalance behavior, we choose 10 ms as a threshold for determining which collectives exhibit a large degree of imbalance and should be used as input to the model generation phase. The intuition is that collectives with high imbalance should be used to train the model, so that future periods of large imbalance can be detected with high probability. Formally, if R is the set of all ranks, K the set of all collectives, $S_{r,k}$ the MPI slack experienced by rank r at collective k, I_k the imbalance at collective k, then the annotation for rank r's performance counters at collective

k, $PC_{r,k}$ is given by Equation 4.1:

$$\forall r \in R, k \in K \text{ s.t. } I_k >= 10 \text{ ms}, PC_{r,k} = \begin{cases} \text{“Critical”} & \text{, if } S_{r,k} < (I_k * 0.25) \\ \text{“Not critical”} & \text{, otherwise} \end{cases} \tag{4.1}$$

That is, ranks whose slack is less than a quarter of the collective's imbalance (slack of the fastest rank) have their performance measurements classified as belonging to a critical rank, while all others have their counters classified as belonging to a not critical rank. Currently, the criticality models are generated with a logistic regression, which takes a single IBS measurement as input and predicts whether the instruction was generated from a critical or non-critical rank. We plan to investigate other, more complex classification tools (support vector machines, neural networks, etc.) in the future.

4.2.3 Evaluating the Models

Having constructed the models, we evaluate them using additional trace data. After collecting performance counters for 5 ms after a collective, each rank generates a criticality prediction that remains in effect until the next collective is reached. Practically speaking, this means that predictions are not made for computational periods (between MPI collectives) shorter than 5 ms; thus, we do not evaluate the models on these periods. As the data in Figure 26 shows, this value still allows criticality predictions to be generated quickly enough to be effective for every application but CL. In the case of CL, most computational periods are too short or too balanced for any criticality decision to be useful to the application.

4.2.3.1 Classification Accuracy To evaluate the models, we perform a series of cross-validations. Our goal in these evaluations is not only to determine how well variability can be modeled by performance counters, but also whether the absolute values of imbalance have any effect on accuracy. Each application is executed three times to generate data sets for training and testing. The input data for each application (performance counters and criticality annotations) is aggregated from these runs and then randomly sampled without

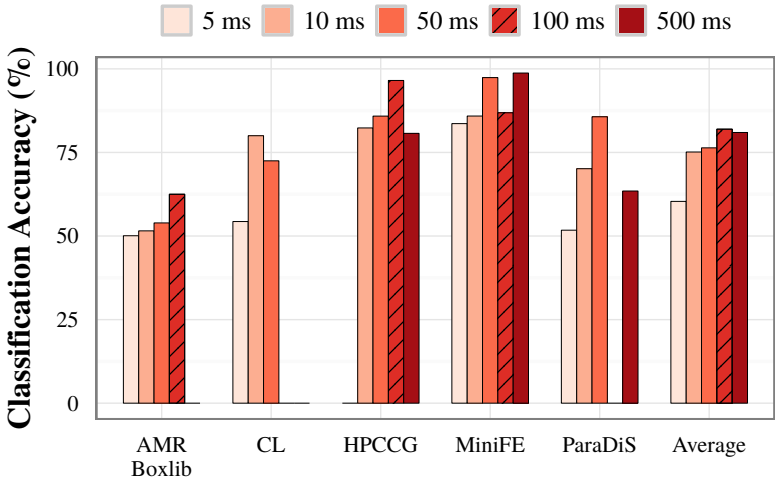


Figure 27: Classification accuracies of criticality prediction models built from node-level performance counters. Missing bars indicate that no periods with the specified imbalance occurred in the application.

replacement such that 75% of the data is used for training and the remaining 25% for testing. We then perform k-fold cross-validation with k=10 to generate classification accuracies.

Figure 27 shows the results of the cross-validations. An individual classification is deemed accurate if it matches the annotation for the rank which generated the performance counter ("critical" or "not critical"). The results are partitioned based on the imbalance seen in the collective that the sample was generated in. For example, the leftmost bar shows the classification accuracy for all samples in AMR Boxlib that were generated from a collective with at least 5 ms but less than 10 ms of imbalance. Missing bars indicate that no periods with the specified imbalance occurred in the application.

The most obvious result is that, in general, larger levels of imbalance result in better classification accuracy. This suggests that the models are able to better identify predictive characteristics from performance counters that are generated during periods of relatively large imbalance and that imbalance does indeed manifest in the node-level performance discrepancies for these applications. AMR Boxlib is the only application whose criticality model classified with less than 60% accuracy, unless the sample came from a collective with

	Prediction Frequency (%)					
T	AMR Boxlib	CL	HPCCG	MiniFE	ParaDiS	Average
0.5	100	100	100	100	100	100
0.75	55.56	95	82.56	70.12	67.09	74.07
0.9	55.56	90	73.31	54.21	65.49	67.71

Table 12: Prediction frequency based on criticality threshold

at least 100 ms of imbalance. Additional performance metrics such as network performance counters or MPI call stacks may be able to better explain imbalance in this application and we plan to investigate such metrics in the future. However, for the remaining applications, as long as there is at least 10 ms of imbalance, the classification accuracy is at least 70% and in some cases (HPCCG, MiniFE) much higher.

4.2.3.2 Generating Rank-Level Predictions As a side effect of using IBS measurements, the models operate on the granularity of IBS interrupts, which only capture the instantaneous state of the node during an instruction's execution. As a result, we need to aggregate multiple predictions to generate a prediction of the rank's criticality over a window of time. We note that general performance counters automatically aggregate a small set of measurements over a given window of time, and therefore would be straightforward to incorporate.

To convert IBS predictions to an overall prediction of criticality for the rank, we simply aggregate the predictions into two categories: the number of "critical" predictions (C_r) and the number of "not critical" predictions (NC_r). Then, an aggregate criticality prediction for the rank (P_r) is made based on the relative number of predictions for each category and a

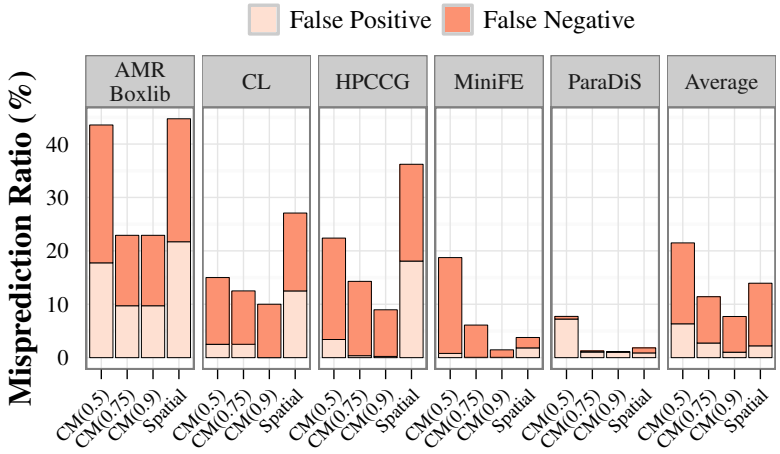


Figure 28: Accuracy generating rank-level predictions with criticality models (CM), compared to a simplified model that considers only spatial variability (Spat.).

confidence threshold (T) as shown by Equation 4.2:

$$\forall r \in R, P_r = \begin{cases} \text{“Critical”} & \text{, if } \dfrac{C_r}{C_r + NC_r} > T \\ \text{“Not critical”} & \text{, if } \dfrac{NC_r}{C_r + NC_r} > T \\ \text{“None”} & \text{, otherwise} \end{cases} \tag{4.2}$$

Thus, for a given collective, each rank can either be predicted as “critical” or “not critical” or there could be no prediction generated by the model if the confidence threshold is not met. Given that this aggregation yields three possible predictions, we define false positive and false negative predictions from the standpoint of the model’s definition of criticality as follows: “critical” predictions for ranks that are “not critical” are false positives, while “not critical” predictions for ranks that are “critical” are false negatives.

The choice of criticality threshold T represents a trade-off between frequency and accuracy of the modeling framework. Intuitively, lower thresholds will cause the model to generate (non “None”) predictions more frequently at the cost of potentially worse accuracy, while higher thresholds increase accuracy at the cost of not being able to always generate a prediction. Table 12 shows the frequency at thresholds of 0.5, 0.75, and 0.9, where frequency

is defined as total number of predictions made divided by the number of prediction opportunities (number of computational periods between collectives longer than 5 ms as discussed in the first paragraph of Section 4.2.3).

Figure 28 shows the false positive and false negative rates generated by our criticality models (CM). The first three bars in each cluster report the rates based on the given confidence threshold – periods where ranks do not generate predictions do not impact the reported rates. T=0.5 provides a baseline as this effectively disables the confidence threshold, and as expected results in the lowest prediction accuracy for each application. Increasing T from 0.75 to 0.9 leads to the highest classification accuracy, on average reducing false positives from 2.74% to 1.02%, and false negatives 8.67% to 6.69%, at the cost of reducing the average prediction frequency from 74.07% to 67.71% as shown in Table 12. We feel that such a reduction in frequency is warranted given the improved accuracy, and thus we focus on T=0.9 for the remaining analysis.

With T=0.9, all applications but AMR Boxlib experience a false positive rate lower than 1.02%. This indicates that the models effectively identify performance counters that positively indicate critical ranks, as predictions of criticality are very likely to be accurate. On the other hand, false negative rates for these applications are slightly higher, but are still limited due to the confidence threshold. All applications except AMR Boxlib have false negative rates below 10%, with MiniFE (1.4%) and ParaDiS (0.89%) substantially lower. Even for AMR Boxlib, which achieved only modest instruction-level classification accuracy (Figure 27), the threshold significantly reduces the number of mispredictions at the rank level.

4.2.3.3 Comparison with Simpler Spatial Modeling

We also seek to compare our criticality models with a simpler prediction engine inspired by previous work [90]. This model, which we refer to as a spatial prediction model, simply remembers the criticality set (those ranks within 25% of the collective's imbalance) from the previous global collective and uses it to predict the criticality set of the next collective. While this model is not the exact model used in the previous work, it is similar in the targeting of spatial variability which has occurred previously in the application. Figure 28 shows the accuracy of this model (Spatial),

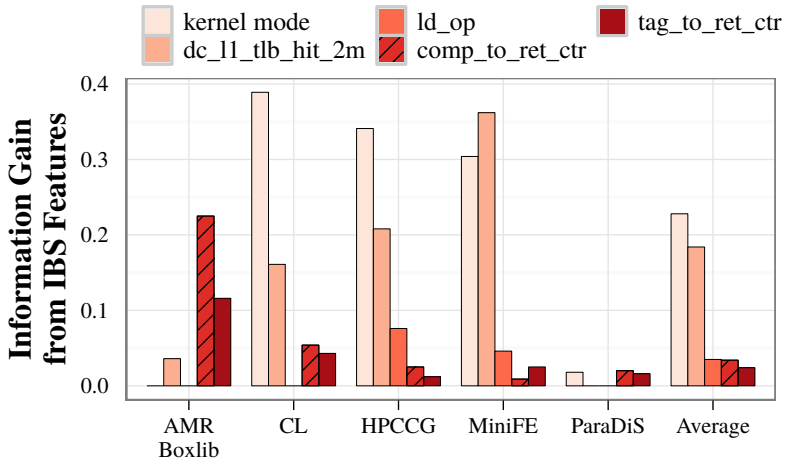


Figure 29: Information gain from IBS performance events. Larger values better model performance variability.

with false positive and false negative rates reported in a similar fashion. We note that this model generates predictions if and only if our model generates a prediction when T=0.5, which means all periods longer than 5 ms are predicted. We see this is a fair comparison because criticality thresholds allow our framework to eschew predictions in cases where there may not be significant imbalance in the application, a characteristic that cannot be easily mimicked by (nor would be particularly useful for) approaches that assume variability to be temporally consistent. It can be seen that, for AMR Boxlib, CL, and HPCCG, our criticality models better limit false positive and false negative rates because (1) they capture temporal performance variability, and (2) the confidence threshold limits mispredictions. Conversely, MiniFE and ParaDiS are modeled well by both approaches. This suggests that spatial performance variability is the dominant indicator of criticality in these applications. On average, with T=0.9 our criticality models reduce false positive rates 2.21% to 1.02%, and false negative rates from 11.72% to 6.69%. For an application more impacted by temporal variability, such as HPCCG, criticality models reduce false positives from 18.09% to 0.23%, and false negatives from 18.13% to 8.74%.

4.2.3.4 Predictive Performance Counters

Finally, we also seek to determine which types of performance counters gleaned from IBS sampling are most influential in generating the criticality models. Figure 29 shows the information gain from the five most influential performance events in the IBS measurements. The figure shows that the two most indicative signs, on average, are whether a sample is generated from kernel space (`kernel mode`) and whether the physical address for a tagged load/store is in a 2MB page table entry in data cache L1 TLB (`dc_l1_tlb_hit_2m`). However, for some applications (HPCCG and MiniFE) an instruction being a load operation (`ld_op`) provides insight as to whether it is generated by a "critical" or "non critical" rank. Finally, the number of cycles from instruction completion to retirement (`comp_to_ret_ctr`) and from instruction tagging to retirement (`tag_to_ret_ctr`) also have predictive capability.

4.3 RELATED WORK

Previous work in criticality modeling either models thread criticality using low-level performance measurements within multiple cores of a single SMT system [17, 35, 72], or utilizes a simpler model of criticality which mainly targets spatially inconsistent variability in distributed multi-node systems [90, 11, 70]. Our approach is novel in the sense that it considers many sources of variability, including temporal variability, but also is built to be utilized by an application parallelized across many independent nodes. While the concept of modeling performance variability or criticality within an application is not new, to our knowledge there have been no attempts to holistically model both temporally and spatially inconsistent variability as it occurs across multiple nodes of a large scale system.

Motivated by the fact that many applications present imbalance during execution, Adagio [90] is an approach that autonomously decides the voltage and frequency states of a given task on each node to reduce energy consumption without hurting performance. Adagio is driven by a predictor that assumes temporal consistency with respect to variability (i.e., if a *task* was slow in the past it will be slow in the future). Conductor [70, 11] is a runtime that uses Adagio to guide non-uniform power distributions under an application-level power cap.

Additionally, Energy-Aware MPI [103] (EAM) is an MPI implementation utilizing models of communication primitives to determine when to set low power modes to CPUs.

There are two primary difference between criticality models and these works. In comparison to Adagio/Conductor, criticality models target temporally inconsistent performance variability, as opposed to using previous executions of a particular task as indications of how that task will perform in the future. On the other hand, EAM does target temporal variability, but does so by focusing on human-derived models of performance which are primarily driven by application communication characteristics (e.g. MPI primitives). While these models are highly accurate for many applications, they ignore characteristics that are external to applications (e.g., resource contention) that will impact exascale systems. Furthermore, in contrast to each of these works, criticality models support a wider range of research efforts such as work stealing [33, 7] and workload redistribution [83, 25].

There has been considerable research in power management techniques for HPC systems, including scheduling multiple jobs in power-constrained systems [70, 11], using dynamic voltage and frequency scaling (DVFS) to dynamically reduce power [43, 62], and applying DVFS and job scheduling to meet a cluster-level power budget [80, 89]. However, these works do not investigate the impact of performance variability on HPC applications and thus leave opportunities for performance and energy improvements at extreme scale.

Finally, in addition to the above mentioned works, there is significant research in power shifting and DVFS at the node level [72, 64, 63, 24, 81]. These node-centric power optimization techniques are orthogonal to and could be coupled with criticality models.

4.4 SUMMARY

Performance variability will be a significant impediment to the runtime and energy efficiency of future HPC systems. We introduced criticality models to address the increasing complexity caused by temporally and spatially inconsistent variability. Criticality models are designed to learn how causes of variability manifest across a system and provide a scalable, low latency mechanism to inform higher level services how and where variability occurs. We evaluated

criticality models on a small cluster and showed that they model performance variability more effectively than a simpler model that assumes variability to be temporally consistent, improving prediction accuracy by up to 17% for a set of HPC benchmarks.

5.0 CONCLUSION

The only way to utilize the world's largest and most powerful computers is through parallel programming, and the predominant parallel programming model for many application domains is Bulk Synchronous Parallelism (BSP). With exascale systems expected to arrive by the early 2020s, parallel computing workloads will have unprecedented access to computational resources. However, it will be a major challenge to efficiently utilize these resources due to the issue of performance variability.

In this dissertation, we explained that to mitigate variability requires an understanding of how its myriad sources differ along a couple of key characteristics: whether a particular source of variability is generated by software or hardware, and whether the imbalance a source generates varies over space (between processors at the same point in time), or over time (in the same processor at different points in time). We presented three major contributions that address different sources of variability based on this high level taxonomy.

Our first major contribution was to study performance variability that arises from software. We showed that this class of variability can be largely eliminated by carefully designing the operating system around the key principle of performance isolation. We discussed the design and implementation of a new operating system called Hobbes, which partitions node resources into multiple isolated *enclaves*. Each workload that executes on a Hobbes-capable machine can leverage its own private system software environment, including custom operating system kernel, so as to remain fully isolated from performance interference from other workloads. Furthermore, we demonstrated how Hobbes can selectively relax this isolation, when explicitly requested by applications, to facilitate communication across enclave boundaries, a key capability that will be required by exascale workloads. The XEMEM system provides this capability by mapping shared memory regions into multiple enclave address

spaces. Importantly, communication through these shared memory regions does not require cross-kernel communication, but rather is driven entirely by user-level reads and writes to memory. Thus, the XEMEM system precludes kernel level interference from generating variability.

While this level of performance isolation is key to reducing software induced performance variability, our taxonomy showed that there are many other sources of variability expected in exascale systems that probably cannot be prevented, no matter how careful the design of the operating system. Examples include intrinsically variable node hardware resources, as well as resources that are shared across the entire system, including interconnects and power. Based on these considerations, we expect that exascale system software must be able to *detect and react* to variability rather than attempt to prevent all possible sources from occurring. While at a high level the notion of detection and reaction is not novel, we find that current approaches in this vein make assumptions of how variability occurs that will probably not hold on exascale machines, including that imbalance is predominantly spatially variant, the result of imbalanced application workload distribution, or of "slow" processors and nodes that consistently delay progress over the lifetime of an applications. With exascale systems incorporating more complex, heterogeneous, and distributed node architectures, as well as diverse system objectives leading to co-scheduled and/or power constrained applications, these assumptions likely no longer reflect the nature of variability.

Accordingly, we presented an approach to holistically characterize variability in an exascale system in order to revisit these assumptions and to push towards a more accurate understanding of how imbalance manifests on a machine. We designed and implemented a new performance evaluation framework called `varbench`, which frames variability in a machine across both space and time. Using varbench, we demonstrated that node architectures have evolved in such a way that's driving an increase in both spatial and temporal variability. We also demonstrated that variability occurs in different ways based on application workload characteristics, suggesting that exascale systems must incorporate understanding of per application behavior to mitigate variability.

Finally, based on these observations, the final component of this dissertation presented a technique called *criticality modeling*. Criticality models are not based on simplifying as-

sumptions about how performance variability impacts a system, but rather are built to reflect the propensity of different classes of variability to arise on a specific architecture in the context of a specific application. Criticality models observe the manifestation of imbalance within a given architecture, and use statistical modeling techniques to determine which low-level hardware characteristics correlate with observed imbalance. With criticality models, a higher-level service can make predictions about how imbalance will manifest without making assumptions, but rather based on relationships learned by monitoring its behavior over time.

Together, these efforts collectively provide mechanisms to mitigate variability and promote scalability across large scale parallel machines.

BIBLIOGRAPHY

[1] Docker. Online, accessed: 2015-05-15. http://www.docker.com.

[2] SN Application Proxy. Online, Accessed: 2015-03-20. https://github.com/losalamos/SNAP.

[3] Sweep3D Benchmark. Online, Accessed: 2015-03-20. http://wwwc3.lanl.gov/pal/software/sweep3d/.

[4] Top500: The List. Online, Accessed: 2017-09-07. https://www.top500.org.

[5] Martin Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation*, (OSDI '16), 2016.

[6] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. DataStager: Scalable Data Staging Services for Petascale Applications. *Cluster Computing*, 13:277–290, 2010.

[7] Bilge Acun and Laxmikant Kale. Mitigating Processor Variation through Dynamic Load Balancing. In *Proceedings of the 1st IEEE International Workshop on Variability in Parallel and Distributed Systems*, (VarSys '16), 2016.

[8] Advanced Micro Devices, Inc. BIOS and Kernel Developers Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors. online, 2015. http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf.

[9] Hakan Akkan, Latchesar Ionkov, and Michael Lang. Transparently Consistent Asynchronous Shared Memory. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, (ROSS '13), 2013.

[10] Ray Alcouffe, Randal Baker, Jon Dahl, Scott Turner, and Robert Ward. PARTISN: A Time-dependent, Parallel Neutral Particle Transport Code System. Technical report, Los Alamos National Laboratory, 2005.

[11] Peter Bailey, Aniruddha Marathe, David Lowenthal, Barry Rountree, and Martin Schulz. Finding the Limits of Power-Constrained Application Performance. In *Proceed-*

ings of the 27th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis, (SC '15), 2015.

[12] Nathan Barton, Joel Bernier, Jaroslaw Knap, Anne Sunwoo, Ellen Cerreta, and Todd Turner. A Call to Arms for Task Parallelism in Multi-Scale Materials Modeling. *International Journal for Numerical Methods in Engineering*, 86(6):744–764, 2011.

[13] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the 25th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '12), 2012.

[14] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, and Susan Coghlan. Operating System Issues for Petascale Systems. *ACM SIGOPS Opererating Systems Review*, 40(2), April 2006.

[15] Abhinav Bhatele, Sebastien Fourestier, Harshitha Menon, Laxmikant Kale, and Francois Pellegrini. Applying Graph Partitioning Methods to Measurement-based Dynamic Load Balancing. Technical report, University of Illinois at Urbana-Champaign, February 2012.

[16] Abhinav Bhatele, Kathryn Mohror, Steven Langer, and Katherine Isaacs. There Goes the Neighborhood: Performance Degradation due to Nearby Jobs. In *Proceedings of the 25th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '13), 2013.

[17] Abhishek Bhatthacharjee and Margaret Martonosi. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. In *Proceedings of the 36th International Symposium on Computer Architecture*, (ISCA '09), 2009.

[18] Shekhar Borkar. Designing Reliable Systems from Unreliable Components: the Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, 2005.

[19] David Boyuka, Sriram Lakshminarasimhan, Xiaocheng Zou, Zhenhuan Gong, John Jenkins, Eric Schendel, Norbert Podhorszki, Qing Liu, Scott Klasky, and Nagiza Samatova. Transparent in Situ Data Transformations in ADIOS. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, (CCGrid '14), 2014.

[20] Ron Brightwell, Ron Oldfield, Arthur Maccabe, and David Bernholdt. Hobbes: Composition and Virtualization as the Foundations of an Extreme-scale OS/R. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, (ROSS '13), 2013.

[21] Ron Brightwell, Kevin Pedretti, and Trammell Hudson. SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-Core Processor.

In *Proceedings of the 21st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '08).

[22] V. Bulatov et al. Scalable Line Dynamics in ParaDiS. In *Proceedings of the 16th Annual IEEE/ACM International Conference for High Performance Computing, Storage, Networking and Analysis*, (SC), 2004.

[23] Dong Chen, Noel Eisley, Philip Heidelberger, Robert Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David Satterfield, Burkhard Steinmacher-Burow, and Jeffrey Parker. The IBM Blue Gene/Q Interconnection Network and Message Unit. In *Proceedings of the 24th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '11), 2011.

[24] Ryan Cochran, Can Hankendi, Ayse Coskun, and Sherief Reda. Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, (MICRO), 2011.

[25] P. Colella, D. Graves, J. Johnson, H. Johansen, N. Keen, T. Ligocki, D. Martin, P. Mccorquodale, D. Modiano, P. Schwartz, T. Sternberg, and B. Straalen. *Chombo Software Package for AMR Applications - Design Document*, 2013.

[26] Henggang Cui, Hao Zhang, Gregory Ganger, Phillip Gibbons, and Eric Xing. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server. In *Proceedings of the Eleventh European Conference on Computer Systems*, (EuroSys '16), 2016.

[27] Zheng Cui, Lei Xia, Patrick Bridges, Peter Dinda, and John Lange. Optimizing Overlay-based Virtual Networking Through Optimistic Interrupts and Cut-through Forwarding. In *Proceedings of the 25th ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, (SC '12), 2012.

[28] Howard David, Eugene Gorbatov, Ulf Hanebutte, Rahul Knanna, and Christian Le. RAPL: Memory Power Estimation and Capping. In *Proceedings of the 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design*, (ISLPED '10), 2010.

[29] Jai Dayal, Drew Bratcher, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Xuechen Zhang, Hasan Abbasi, Scott Klasky, and Norbert Podhorszki. Flexpath: Type-Based Publish/Subscribe System for Large-Scale Science Analytics. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, (CCGrid '14), 2014.

[30] Jai Dayal, Jianting Cao, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Fang Zheng, Hasan Abbasi, Scott Klasky, Norbert Podhorszki, and Jay F. Lofstead. I/O containers: Managing the data analytics and visualization pipelines of high end codes. In *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*.

[31] Jeffrey Dean and Luiz Barroso. The Tail at Scale. *Commuincations of the ACM*, 56(2):74–80, 2013.

[32] 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. 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):184–193, 2011.

[33] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P. Sadayappan, and Chau-Wen Tseng. Dynamic Load Balancing of Unbalanced Computations Using Message Passing. In *Proceedings of the 6th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, (PMEO-PDS '07), 2007.

[34] Ciprian Docan, Manish Parashar, and Scott Klasky. DataSpaces: an Interaction and Coordination Framework for Coupled Simulation Workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Parallel and Distributed Computing*, (HPDC '10), 2010.

[35] Kristof Du Bois, Stijn Eyerman, Jennifer Sartor, and Lieven Eeckhout. Criticality Stacks: Identifying Critical Threads in Parallel Programs using Synchronization Behavior. In *Proceedings of the 40th International Symposium on Computer Architecture*, (ISCA '13), 2013.

[36] Noah Evans, Brian Kocoloski, John Lange, Kevin Pedretti, Shyamali Mukherjee, Ron Brightwell, Michael Lang, and Patrick Bridges. Hobbes Node Virtualization Layer: System Software Infrastructure for Application Composition and Performance Isolation. In *Proceedings of the 28th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '16), 2016.

[37] Noah Evans, Kevin Pedretti, Brian Kocoloski, John Lange, Michael Lang, and Patrick Bridges. A Cross-Enclave Composition Mechanism for Exascale System Software. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, (ROSS '16), 2016.

[38] Noah Evans, Kevin Pedretti, Shyamali Mukherjee, Ron Brightwell, Brian Kocoloski, John Lange, and Patrick Bridges. Remora: A MPI Runtime for Composed Applications at Extreme Scale. In *Proceedings of the Workshop on Exascale MPI*, (ExaMPI '16), 2016.

[39] Kurt Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing Application Sensitivity to OS Interference using Kernel-Level Noise Injection. In *Proceedings of the 21st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '08), 2008.

[40] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert Wisniewski. Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK. In *Proceedings of the 23rd ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '10), 2010.

[41] Michael Heroux et al. Welcome to the Mantevo Project Home Page, https://software.sandia.gov/mantevo.

[42] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 22nd Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '10), 2010.

[43] Song Huang and Wu Feng. Energy-Efficient Cluster Computing via Accurate Workload Characterization. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, (CCGrid), 2009.

[44] Khaled Ibrahim. Characterization of the DOE Mini-apps. Online, 2014. http://portal.nersc.gov/project/CAL/designforward.htm.

[45] Tong Jin, Fan Zhang, Qian Sun, Hoang Bui, Manish Parashar, Hongfeng Yu, Scott Klasky, Norbert Podhorszki, and Hasan Abbasi. Using Cross-layer Adaptations for Dynamic Data Management in Large Scale Coupled Scientific Workflows. In *Proceedings of the 26th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '13), 2013.

[46] Ana Jokanovic, Jose Sancho, German Rodriguez, Alejandro Lucero, Cyriel Minkenberg, and Jesus Labarta. Quiet Neighborhoods: Key to Protect Job Performance Predictability. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium*, (IPDPS '15), 2015.

[47] Chad Jones, Kwan-Liu Ma, Allen Sanderson, and Lee Myers Jr. Visual Interrogation of Gyrokinetic Particle Simulations. *Journal of Physica: Conference Series*, 78(012033), 2007.

[48] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *Proceedings of the International Conference on Parallel Processing Workshops*, (ICPPW '09), 2009.

[49] Laxmikant Kale and Gengbin Zheng. *Advanced Computational Infrastructures for Parallel and Distributed Applications*. Wiley, Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects, 2009.

[50] Suzanne Kelly, John Van Dyke, and Courtenay Vaughan. Catamount N-Way (CNW): An Implementation of the Catamount Light Weight Kernel Supporting N-cores Version 2.0. Technical report, Sandia National Laboratories, 2008. http://www.sandia.gov/~smkelly/SAND2008-4039P-CNW-ProjectReportV2-0.pdf.

[51] Brian Kocoloski and John Lange. Better Than Native: Using Virtualization to Improve Compute Node Performance. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, (ROSS '12), 2012.

[52] Brian Kocoloski and John Lange. Improving Compute Node Performance Using Virtualization. *International Journal of High Performance Computing Applications*, 27(2):124–135, 2013.

[53] Brian Kocoloski and John Lange. HPMMAP: Lightweight Memory Management for Commodity Operating Systems. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, (IPDPS '14), 2014.

[54] Brian Kocoloski and John Lange. XEMEM: Efficient Shared Memory for Composed Applications on Multi-OS/R Exascale Systems. In *Proceedings of the 24th International ACM Symposium on High Performance Distributed Computing*, (HPDC '15), 2015.

[55] Brian Kocoloski and John Lange. Lightweight Memory Management for High Performance Applications in Consolidated Environments. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):468–480, 2016.

[56] Brian Kocoloski, John Lange, Hasan Abbasi, David Bernholdt, Terry Jones, Jai Dayal, Noah Evans, Michael Lang, Jay Lofstead, Kevin Pedretti, and Patrick Bridges. System-Level Support for Composition of Applications. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, (ROSS '15), 2015.

[57] Brian Kocoloski, Jiannan Ouyang, and John Lange. A Case for Dual Stack Virtualization: Consolidating HPC and Commodity Applications in the Cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, (SOCC '12), 2012.

[58] Brian Kocoloski, Leonardo Piga, Wei Huang, Indrani Paul, and John Lange. A Case for Criticality Models in Exascale Systems. In *Proceedings of the 18th IEEE International Conference on Cluster Computing*, (CLUSTER '16), 2016.

[59] Sandia National Laboratories. Using DUMPI. online, 2015. http://sst.sandia.gov/using_dumpi.html, Accessed on 16 April 2015.

[60] John Lange, Kevin Pedretti, Peter Dinda, Patrick Bridges, Chang Bae, Philip Soltero, and Alexander Merritt. Minimal Overhead Virtualization of a Large Scale Supercomputer. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, (VEE '11), 2011.

[61] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Andy Gocke, Steven Jaconette, Mike Levenhagen, and Ron Brightwell. Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, (IPDPS '10), 2010.

[62] Wim Lavrijsen, Costin Iancu, Wibe de Jong, Xin Chen, and Karsten Schwan. Exploiting Variability for Energy Optimization in Parallel Programs. In *Proceedings of the Eleventh European Conference on Computer Systems*, (EuroSys '16), 2016.

[63] Charles Lefurgy, Xaiorui Wang, and Malcolm Ware. Power Capping: a Prelude to Power Shifting. *Cluster Computing*, 11(2):183–195, 2008.

[64] Jian Li, Jose Martinez, and Michael Huang. The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, (HPCA), 2004.

[65] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Input/Output APIs and Data Organization for High Performance Scientific Computing. In *Proceedings of the 3rd Parallel Data Storage Workshop*, (PDSW '08), 2008.

[66] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Adaptable, Metadata Rich IO Methods for Portable High Performance IO. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, (IPDPS'09), 2009.

[67] Piotr Luszczek, Jack Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. Introduction to the HPCChallenge Benchmark Suite. Technical report, University of Tennessee, March 2005.

[68] Kwan-Liu Ma, Chaoli Wang, Hongfeng Yu, and Anna Tikhonova. In-Situ Processing and Visualization for Ultrascale Simulations. In *Journal of Physics: Proceedings of DOE SciDAC 2007 Conference*, June 2007.

[69] Grzegorz Malewicz, Matthew Austern, Aart Bik, James Denhert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, (SIGMOD '10), 2010.

[70] Aniruddha Marathe, Peter Bailey, David Lowenthal, Barry Rountree, Martin Schulz, and Bronis de Supinski. A Run-Time System for Power-Constrained HPC Applications. In *Proceedings of the 30th International Conference, ISC High Performance*, 2015.

[71] Maxime Martinasso and Jean-Francois Mehaut. A Contention-Aware Performance Model for HPC-Based Networks: A Case Study of the Infiniband Network. *Lecture Notes in Computer Science*, 6852, 2-11.

[72] Timothy Miller, Xiang Pan, Renji Thomas, Naser Sedaghati, and Radu Teodorescu. Booster: Reactive Core Acceleration for Mitigating the Effects of Process Variation and Application Imbalance in Low-voltage Chips. In *Proceedings of the 18th IEEE International Symposium on High-Performance Computer Architecture*, (HPCA '12), 2012.

[73] Alessandro Morari, Roberto Gioiosa, Robert Wisniewski, Bryan Rosenburg, Todd Inglett, and Mateo Valero. Evaluating the Impact of TLB Misses on Future HPC Systems. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, (IPDPS '12), 2012.

[74] National Center for Atmospheric Research. CESM: Community Earth System Model. Online, 2015. http://www2.cesm.ucar.edu/.

[75] Leonid Oliker, Jonathan Carter, Michael Wehner, Andrew Canning, Stephane Ethier, Art Mirin, Govindasamy Bala, David Parks, Patrick Worley Shigemune Kitawaki, and Yoshinori Tsuda. Leading Computational Methods on Scalar and Vector HEC Platforms. In *Proceedings of the 18th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '05), 2005.

[76] Douglas Otstott, Noah Evans, Latchesar Ionkov, Ming Zhao, and Michael Lang. Enabling Composite Applications Through an Asynchronous Shared Memory Interface. In *Proceedings of the 2014 IEEE International Conference on Big Data*, (BigData '14), 2014.

[77] Jiannan Ouyang, Brian Kocoloski, John Lange, and Kevin Pedretti. Achieving Performance Isolation with Lightweight Co-Kernels. In *Proceedings of 24th ACM International Symposium on High Performance Parallel and Distributed Computing*, (HPDC '15), 2015.

[78] Yoonho Park, Eric Van Hensbergen, Marius Hillenbrand, Todd Inglett, Bryan Rosenburg, Kyung Dong Ryu, and Robert Wisniewski. FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment. In *Proceedings of the 24th IEEE International Symposium on Computer Architecture and High Performance Computing*, (SBAC-PAD '12), 2012.

[79] Tapasya Patki, David Lowenthal, Barry Rountree, Martin Schulz, and Bronis de Supinski. Exploring Hardware Overprovisioning in Power-Constrained, High Performance Computing. In *Proceedings of the 27th ACM International Conference on Supercomputing*, (ICS '13), 2013.

[80] Tapasya Patki, David Lowenthal, Anjana Sasidharan, Matthias Maiterth, Barry L. Rountree, Martin Schulz, and Bronis de Supinski. Practical Resource Management in Power-Constrained, High Performance Computing. In *Proceedings of the 24th International ACM Symposium on High-Performance Parallel and Distributed Computing*, (HPDC '15), 2015.

[81] Indrani Paul, Wei Huang, Manish Arora, and Sudhakar Yalamanchili. Harmonia: Balancing Compute and Memory Power in High-performance GPUs. In *Proceedings of the 42nd International Symposium on Computer Architecture*, (ISCA), 2015.

[82] Amir Payberah. Large Scale Graph Processing Pregel, GraphLab and GraphX. Online, 2016. https://www.sics.se/~amir/id2221/slides/graph_pregel_graphlab_graphx.pdf.

[83] Olga Pearce, Todd Gamblin, Bronis de Supinski, Martin Schulz, and Nancy Amato. Quantifying the Effectiveness of Load Balance Algorithms. In *Proceedings of the 26th ACM International Conference on Supercomputing*, (ICS '12), 2012.

[84] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the 15th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Anaylsis*, (SC '03), 2003.

[85] Steve Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[86] Bogdan Prisacari, German Rodriguez, Philip Hiedelberger, Dong Chen, Cyriel Minkenberg, and Torsten Hoefler. Efficient Task Placement and Routing of Nearest Neighbor Exchanges in Dragonfly Networks. In *Proceedings of 23rd ACM International Symposium on High Performance Parallel and Distributed Computing*, (HPDC '14), 2014.

[87] Gonzalo Rodrigo, Erik Elmroth, Per-Olov Ostberg, and Lavanya Ramakrishnan. Enabling Workflow-Aware Scheduling on HPC Systems. In *Proceedings of the 26th ACM International Sumposium on High-Performance Parallel and Distributed Computing*, (HPDC '17), 2017.

[88] Efraim Rotem, Alon Naveh, and Avinash Anathakrishnan. Power-Management Architecture of the Intel Microacrhitecture Code-Named Sandy Bridge. *IEEE Micro*, 32(2):20–27, 2012.

[89] Barry Rountree, Dong Ahn, Bronis de Supinski, David Lowenthal, and Martin Schulz. Beyond DVFS: A First Look at Performance under a Hardware-Enforced Power Bound. In *Proceedings of the 26th International IEEE Parallel and Distributed Processing Symposium Workshops PhD Forum*, (IPDPSW '12), 2012.

[90] Barry Rountree, David Lowenthal, Bronis de Supinski, Martin Schulz, Vincent Freeh, and T. Bletsch. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd ACM International Conference on Supercomputing*, (ICS '09), 2009.

[91] Osman Sarood, Akhil Langer, Laxmikant Kale, Barry Rountree, and Bronis de Supinski. Optimizing Power Allocation to CPU and Memory Subsystems in Overprovisioned HPC Systems. In *Proceedings of the 15th IEEE International Conference on Cluster Computing*, (CLUSTER '13), 2013.

[92] Rodney Schmidt, Kenneth Belcourt, Russell Hooper, Roger Pawlowski, Kevin Clarno, Srdjan Simunovic, Stuart Slattery, John Turner, and Scott Palmtag. An Approach

for Coupled-Code Multiphysics Core Simulations from a Common Input. *Annals of Nuclear Energy*, 84:140–152, 2014.

[93] Galen Shipman, Patrick McCormick, Kevin Pedretti, Stephen Olivier, Kurt Ferreira, Ramanan Sankaran, Sean Treichler, Alex Aiken, and Michael Bauer. Analysis of Application Sensitivity to System Performance Variability in a Dynamic Task Based Runtime. In *Proceedings of the Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures*, 2015.

[94] Rishi Sinha and Marianne Winslett. Multi-Resolution Mitmap Indexes for Scientific Data. *ACM Transactions on Database Systems*, 32(3), 2007.

[95] Avinash Sodani. Knight's Landing (KNL): 2nd Generation Intel Xeon Phi Processor. In *Proceedings of the IEEE Symposium on High Performance Chips*, (HC27), 2015.

[96] Philip Soltero, Patrick Bridges, Dorian Arnold, and Michael Lang. A Gossip-based Approach to Exascale System Services. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, (ROSS '13), 2013.

[97] Rick Stevens, Andrew White, et al. Architectures and Technology for Extreme Scale Computing. Technical report, U.S. Department of Energy, 2009.

[98] Alejandro Strachan, Sankaran Mahadevan, Vadiraj Hombal, and Lin Sun. Functional Derivatives for Uncertainty Quantification and Error Estimation and Reduction via Optimal High-Fidelity Simulations. *Modelling and Simulation in Materials Science and Engineering*, 21(6), 2013.

[99] Devesh Tiwari, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter Desnoyers. Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, (FAST '13), 2013.

[100] Hirofumi Tomita, Mitsuhisa Sato, and Yutaka Ishikawa. Japan Overview Talk. In *Proceedings of the 2nd International Workshop on Big Data and Extreme-scale Computing*, (BDEC '14), 2014.

[101] James Tschanz, James Kao, Siva Narendra, Raj Nair, Dimitri Antoniadis, Anantha Chandrakasan, and Vivek De. Adaptive Body Bias for Reducing Impacts of Die-to-Die and Within-Die Parameter Variations on Microprocessor Frequency and Leakage. *IEEE Journal of Solid-State Circuits*, 37(11):1396–1402, 2002.

[102] Mauricio Tsugawa and Jose Fortes. A Virtual Network (ViNe) Architecture for Grid Computing. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, (IPDPS '06), 2006.

[103] Akshay Venkatesh, Abhinav Vishnu, Khaled Hamidouche, Nathan Tallent, Dhabaleswar Panda, Darren Kerbyson, and Adolfy Hoisie. A Case for Application-oblivious

Energy-efficient MPI Runtime. In *Proceedings of the 27th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '15), 2015.

[104] Venkatram Vishwanath, Mark Hereld, and Michael Papka. Toward Simulation-Time Data Analysis and I/O Acceleration on Leadership-Class Systems. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, (LDAV '11), 2011.

[105] H. Viswanathan, E. K. Lee, I. Rodero, D. Pompili, M. Parashar, and M. Gamell. Energy-Aware Application-Centric VM Allocation for HPC Workloads. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing*, (IPDPS '11), 2011.

[106] Dylan Wang, Abhinav Bhatele, and Dipak Ghosal. Performance Variability Due to Job Placement on Edison. In *Proceedings of the 27th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '14), 2014.

[107] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. XenLoop: A Transparent High Performance Inter-VM Network Loopback. In *Proceedings of the 17th ACM International Symposium on High Performance Parallel and Distributed Computing*, (HPDC '08), 2008.

[108] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yudou Wu, Andy Riffel, and John Owens. Gunrock: a High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (PPoPP '16), 2016.

[109] Peter Westfall. Kurtosis as Peakedness, 1905 - 2014. R.I.P. *The American Statistician*, 68(3):191–195, 2014.

[110] Robert Wisniewski, Todd Inglett, Pardo Keppel, Ravi Murty, and Rolf Riesen. mOS: An Architecture for Extreme-Scale Operating Systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, (ROSS '14), 2014.

[111] Matthew Wolf, Zhongtang Cai, Weiyun Huang, and Karsten Schwan. SmartPointers: Personalized Scientific Data Portals in Your Hand. In *Proceedings of 15th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '02), 2002.

[112] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. The SGI Altix 3000 Global Shared-Memory Architecture. Technical report, Silicon Graphics International Corporation, 2003.

[113] Reynold Xin, Joseph Gonzalez, Michael Franklin, and Ion Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *Proceedings of the 1st International Workshop on Graph Data Management Experiences and Systems*, (GRADES '13), 2013.

[114] Matei Zahari, Mosharaf Chowdhury, Michael Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, (HotCloud '10), 2010.

[115] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Qing Liu, Scott Klasky, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. PreDatA - Preparatory Data Analytics on Peta-Scale Machines. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, (IPDPS '10), 2010.

[116] Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, Greg Eisenhauer, Karsten Schwan, Hasan Abbasi, and Scott Klasky. GoldRush: Resource Efficient In Situ Scientific Data Analytics Using Fine-Grained Interference Aware Execution. In *Proceedings of the 26th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (SC '13), 2013.

[117] Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, Greg Eisenhauer, Karsten Schwan, Hasan Abbasi, and Scott Klasky. GoldRush: Resource Efficient in Situ Scientific Data Analytics Using Fine-grained Interference Aware Execution. In *Proceedings of the 26th International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[118] Fang Zheng, Hongbo Zou, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Jai Dayal, Nguyen Tuan-Anh, Jianting Cao, Hasan Abbasi, Scott Klasky, Norbert Podhorszki, and Hongfeng Yu. FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium*, (IPDPS '13), 2013.