Reducing Kernel Surface Areas for Isolation and Scalability

Daniel Zahka, Brian Kocoloski
Department of Computer Science & Engineering
Washington University in St. Louis
{daniel.zahka,brian.kocoloski}@wustl.edu

Kate Keahey
Mathematics & Computer Science Division
Argonne National Laboratory
keahey@anl.gov

ABSTRACT
Isolation is a desirable property for applications executing in multi-tenant computing systems. On the performance side, hardware resource isolation via partitioning mechanisms is commonly applied to achieve QoS, a necessary property for many noise-sensitive parallel workloads. Conversely, on the software side, partitioning is used, usually in the form of virtual machines, to provide secure environments with smaller attack surfaces than those present in shared software stacks.

In this paper, we identify a further benefit from isolation, one that is currently less appreciated in most parallel computing settings: isolation of system software stacks, including OS kernels, can lead to significant performance benefits through a reduction in variability. To highlight the existing problem in shared software stacks, we first developed a new systematic approach to measure and characterize latent sources of variability in the Linux kernel. Using this approach, we find that hardware VMs are effective substrates for limiting kernel-level interference that otherwise occurs in monolithic kernel systems. Furthermore, by enabling reductions in variability, we find that virtualized environments often have superior worst-case performance characteristics than native or containerized environments. Finally, we demonstrate that due to their isolated software contexts, most virtualized applications consistently outperform their bare-metal counterparts when executing on 64-nodes of a multi-tenant, kernel-intensive cloud system.

CCS CONCEPTS
• General and reference → Measurement; Experimentation;
• Computer systems organization → Multicore architectures;

ACM Reference Format:

1 INTRODUCTION
Though clouds have proven elusive for many high performance computing (HPC) applications, recent trends show promise for future convergence of cloud and HPC systems. One can view this perspective from either of two directions. First, existing cloud applications increasingly require high performance hardware, in large part due to the continued growth of computation intensive machine learning and graph processing algorithms. At the same time, ongoing trends in HPC towards data-intensive and interactive workloads suggest that tighter integration with cloud systems, which provide unrivaled on-demand access to resources and data processing abilities, may benefit existing HPC communities, an idea that some HPC centers have recently pursued [9, 30].

In either case, we expect many large scale parallel applications will be executing in both commercial and academic cloud-based systems. One of the primary challenges that comes with such environments is to effectively manage multi-tenancy. Generally speaking, multi-tenancy creates problems related to performance and security. To the former, much research has studied the implications of hardware resource interference on performance, both within node-level resources [16, 28] as well as on inter-node resources such as interconnects [25] and parallel filesystems [20]. Such issues are particularly problematic for applications sensitive to performance variability; examples can be seen in latency-sensitive applications [5], common in cloud datacenters, as well as parallel applications that leverage bulk synchronous parallel (BSP) algorithms, known to suffer from “straggler” effects. To remedy these issues, research focuses mostly on performance isolation – that is, to administer resources in such a fashion that their performance characteristics are consistent. Of note is that hardware increasingly provides knobs to enforce isolation [1], and such areas are likely to retain prominence in future settings.

On the other hand, security is also a major challenge in multi-tenant environments, but security is fundamentally a cross-stack issue that hardware alone cannot fully address. As a result, much research in this space also centers on isolation, but the isolation is in system software, which is used to limit attack vectors between software stacks of different tenants [26]. For this reason, the debate between virtual machines and containers is still quite active, with several nascent projects attempting to provide the ease-of-use, flexibility, and bare-metal performance of containers while retaining the security benefits of virtual machines [2–4].

In this paper, we argue that system software isolation has significant but largely underappreciated benefits outside of software security. We will demonstrate that system software isolation can actually have a significant performance impact on parallel workloads at scale. From our perspective, the idea that system software isolation, whereby hardware virtualization is used to administer isolated software stacks, can lead to benefits over non-virtualized environments is not a broadly understood or accepted idea – particularly not within HPC contexts, where workloads are hardware sensitive, and thus proximity to hardware considered paramount.
This is evidenced in part by the significant success of "HPC container" projects such as Singularity [17] and Shifter [11] that are considered lighter weight than approaches leveraging hardware virtualization. To that end, our core research finding is that providing close proximity to hardware – as Linux containers are reputed to do – is not a panacea for HPC system software, because many parallel applications require consistency to achieve good parallel performance at scale, and these two goals are not necessarily easy to achieve in concert.

To illustrate the problem that isolation in system software helps to solve, this paper first focuses on measuring cross-tenant performance variability that arises in shared Linux kernels. To do so, we present a new performance analysis methodology that leverages synthetic, coverage-guided program generation to create corpuses of system calls, collectively designed to stress a large portion of the kernel codebase. By carefully deploying these programs across multiple cores of a 64-core machine in a fashion that synchronizes their concurrent requests for kernel services, we find that several kernel subsystems are prone to generate significant variability in multi-tenant system environments.

Beyond leveraging our methodology to measure kernel-level variability, we also identify a key parameter which predictably influences the occurrence and scope of variability within the kernel: the kernel surface area. We define the kernel surface area as a multidimensional parameter denoting, for each resource in the system, the amount of that resource managed by the kernel. We find that this notion closely correlates with variability measured in several kernel subsystems; that is, by configuring VM boundaries to limit kernel surface area, we can reliably reduce variability in several kernel subsystems in ways that benefit higher-level applications.

Finally, we demonstrate that VMs, via their ability to reduce software-level variability, lead to better 99th percentile performance in multi-tenant environments for latency-sensitive workloads. Furthermore, we also demonstrate that this reduction in low-level variability translates to significant performance gains when executing large scale workloads sensitive to worst-case performance outliers, which are common in HPC settings. In a 64-node kernel-intensive multi-tenant environment, nearly every application we measured achieved superior performance in an isolated, virtualized environment compared to a shared, containerized environment due to reduction of low-level kernel variability.

To summarize, we make the following novel contributions:

- We present a new performance evaluation methodology based on synthetic, coverage guided program generation and synchronous call execution to extract latent sources of variability in shared system software.
- We use our methodology to characterize variability in the Linux kernel via performance evaluation of a diverse set of kernel subsystems.
- We identify the kernel surface area as an influential parameter in determining kernel-level variability. Reductions in kernel surface area reliably lead to reductions in variability, and vice versa, despite no changes to the application workload.
- We demonstrate that VMs, via their smaller kernel surface areas and reduced susceptibility to cross-tenant interference, often enable significant performance improvements for large scale kernel-intensive applications.

2 RELATED WORK

Variability in clouds. Variability is a challenging problem in public cloud systems, particularly for web-based applications providing interactive client-facing services. Research has documented sources of variability in clouds, including networks [28] and local-node hardware contention caused by multi-tenancy [16]. More generally, variability across different instances in IaaS systems has been documented in several efforts [10, 19]. In contrast to these efforts, we focus on variability that specifically arises in shared software systems due to contention from multi-tenancy. Our methodology is designed explicitly to measure software overheads that standard approaches relying on benchmarks cannot easily measure, due their reliance on a combination of factors, including sensitivity to hardware interference that makes attribution of overheads to software vs hardware a complicated task.

Variability in HPC. Variability is also well-studied within the HPC community, where systems are susceptible to outlier processors due to the reliance on bulk synchronization and communication that many applications have. Many sources of variability have been studied, ranging from low-level intrinsic hardware issues, such as manufacturing variation [6] and resource contention to software issues such as OS scheduling “noise” [14, 24]. From the kernel’s perspective, HPC systems have historically not been OS-intensive; in fact optimizations within HPC apps such as OS bypass networking [27] remove the OS from the critical path of cross-node communication in the application. However, as HPC applications continue to adopt cloud-based resources and become more I/O driven [9, 30], and thus more reliant on kernel services, issues related to kernel variability will begin to impact them more directly.

Virtual machines vs containers. Questions pertaining to the performance and security benefits of VMs and containers are well-studied (e.g., [22]). In this context, our unique contribution is to focus on performance issues that result from contention in the kernel itself, and the impact that isolation has for removing this contention. To our knowledge, our approach is the first to consider performance differences that arise at this level of the stack.

Containers within the HPC community have gained significant adoption; e.g., Singularity [17] and Shifter [11]. Such approaches primarily target software deployment benefits and environment configurability that containers provide, but are not explicitly targeted for multi-tenant environments with concurrently executing workloads. Recent research in this context, but outside of HPC, generally consists of “lightweight VMs” that provide both the benefits of environment flexibility and rich ecosystem that containers provide while retaining the security benefits of VMs. Projects in this space have not received significant academic attention, but have much industry buy-in due to the security focus; examples include Kata containers [3], IBM’s Nabla [4], and Amazon AWS’s Firecracker [2]. While we focus on Docker [23] and KVM [13], such technologies would be interesting to evaluate in a similar fashion.

3 MEASURING KERNEL VARIABILITY

This section describes our methodology for measuring latent sources of variability in system software, particularly those which manifest
from contention in shared, multi-tenant workload configurations. Given its ubiquity in cloud and HPC environments, we focus on the Linux kernel, though our methodology would apply to other kernels or hypervisors tasked with managing multi-tenancy.

Systematic evaluation of kernel variability is challenging for multiple reasons. First, the only manner in which workloads can directly invoke kernel services is through the system call API. This prevents direct invocation of the majority of low level subroutines within the kernel, which combined with the sheer size of the kernel makes it challenging to systematically evaluate specific subsystems. Second, because we are interested in software overheads that arise from multi-tenancy, we need to be able to determine whether observed overhead/variability is attributable to software performance, including, e.g., scheduling, synchronization, and software caches (e.g., the page cache, slab cache, etc.), as opposed to hardware. Traditional performance evaluation approaches using benchmarks and applications are typically hardware intensive, and even if they are sensitive to kernel performance, it is difficult to differentiate overheads between software and hardware. And third, because we suspect that variability in kernel performance may arise due to contention from parallel invocations of services, it is important that our approach be able to control parallelism and timing at a fine granularity in order to deterministically generate parallel requests. Based on these observations, we designed a new framework to measure kernel variability. Our framework has the following key characteristics:

1. We dynamically generate workloads consisting solely of system calls with the goal of making them minimally hardware-intensive.
2. We leverage existing work on coverage-guided system call fuzzing [7] to generate workloads that invoke many different kernel subsystems.
3. We provide runtime control over the level of parallelism with which requests for kernel services are issued.
4. We leverage barrier synchronization to control, at a fine granularity, the times at which kernel services are invoked across concurrent processes.

A high-level view of our methodology is shown in Figure 1. This figure illustrates the interactions between the three main components of our evaluation approach: (1) a generator that creates programs synthesized from a system call fuzzer [7]; (2) a harness which controls concurrency and timing of system call invocations as well as collects data from experiments; and (3) an environment which executes system call corpuses via the harness. We discuss each of these components in detail below.

### 3.1 Coverage-Guided System Call Generation

Our approach to measuring variability of low level software is to generate programs that only issue sequences of system calls. We made this decision to address two of the aforementioned challenges associated with measuring kernel performance: (1) system calls are the only explicit vehicle through which programs can invoke the kernel, and the only through which the user has control of the times at which kernel services are invoked; (2) higher level benchmarks and applications that do more than issue system calls, even if they are kernel intensive, make performance analysis a more complicated task due to the impact of hardware contention on performance. Furthermore, given the large number of system calls provided in the kernel, manual generation of programs that stress all areas of the kernel would be challenging and time consuming, and so we generate them automatically.

To generate programs that cover a wide set of kernel services, we leverage an existing system call fuzzer, Syzkaller [7]. In its most basic form, fuzzing, which is used primarily for software testing, entails feeding random inputs to a program to find bugs. More complicated fuzzers use more “intelligent” approaches to fuzzing to expedite the process of bug detection, including by providing templates to reduce the domain size of the arguments passed to the program. Syzkaller takes this approach one step further by relying on coverage signals to indicate which specific functions – and indeed even finer granularity information, including which basic blocks [32] – are traversed by the system call handler in response to a set of inputs. Using this information, Syzkaller iteratively builds complex corpuses that cover large portions of the kernel code base.

Composing workloads. While Syzkaller is designed to find bugs within kernel code, we leverage the sequences of system calls it generates to broadly measure kernel performance. Because Syzkaller corpuses are coverage guided, with the aim of covering a large portion of kernel code, they consist of a significant breadth of different system calls as well arguments to those calls to trigger different functions within the kernel. The diversity of calls and breadth of coverage make these programs suitable for stressing a variety of kernel subsystems in a fashion that would be difficult to replicate via hand-generated programs.

To generate workloads, we first deploy Syzkaller on a platform to generate a diverse corpus. Once generated, the corpus consists of separate sequences of system calls called programs, each of which is produced by the Syzkaller generation phase and invokes at least some basic block of the kernel that is not covered by any other program. We extract each program into a common shared library, libsysyzcorsso, which we integrate into our evaluation apparatus as shown in Figure 1.
3.2 Varbench

Our apparatus leverages a harness to issue system calls and performs synchronization operations to control concurrency across multiple cores issuing calls. We chose to base our harness on an existing framework called varbench [15], a performance analysis toolkit designed to measure and characterize variability in parallel systems. Though originally designed to measure variability in shared low-level hardware components, such as last-level CPU caches and memory channels, varbench was designed in a modular fashion, making it straightforward to incorporate a new workload consisting of system call programs from our generator.

Fine-grained concurrency control. Varbench provides support for parallel deployment of a workload across multiple cores, and allows for frequent fine-grained synchronization between cores. The latter is an especially important capability for our harness because potential sources of variability within the kernel, such as software caches and locks, are more likely to exhibit variability in performance if faced with concurrent access. Varbench thus allows us to deploy the same sequences of system calls across multiple cores, and to insert synchronization points between programs from our generator to ensure that every program (i.e., sequence of system calls) starts at the same time on every core.

No dependence on evaluation environment. Beyond evaluating Linux kernel variability, our framework should be capable of evaluating the merits of widely adopted approaches to system software deployment, including virtualization and containerization. Varbench is useful for evaluating such systems because its synchronization mechanisms are performed in user-space via MPI, rather than via a node-local parallel runtime (e.g., OpenMP, pthreads) that relies on OS processes or threads to perform synchronization. For example, when we evaluate virtualized and containerized environments, each VM or container runs a separate operating system or a separate container but still executes the same varbench application. Because varbench relies on MPI, each of these VMs_CONTAINERS can be assigned a separate IP address and still participate in the global synchronizations needed for fine-grained concurrency control.

3.3 Attributing Variability to Software

When we use our framework to measure variability, we would like to be able to make claims that the system software being evaluated, via its design, is indeed responsible for variability, as opposed to other affects such as intrinsic hardware noise or variability from hardware contention that is perhaps impossible to eliminate given a particular set of concurrently executing workloads. Many design decisions are made in the development of a shared software system that may contribute to performance variability in subtle but impactful ways; examples include choice of synchronization constructs (e.g., spinlocks, sleeping locks, and lock-free data structures), methods for communication between cores (e.g., message passing via IPIs, cache-coherent shared memory), mappings of processes to cores, frequency of timer interrupts and kernel preemption, mappings of interrupt request lines to CPUs, etc. The full list of potential factors is too exhaustive to enumerate in a system as complex as a general purpose OS kernel.

Table 1: VM configurations deployed to study influence of different kernel surface areas

<table>
<thead>
<tr>
<th>Configurations</th>
<th># VMs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td># Cores/VM</td>
<td></td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>GB RAM/VM</td>
<td></td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Kernel surface area. Given this complexity, there are two ways in which our methodology is explicitly designed to support attribution of variability to system software design. The first, which has already been discussed, is to rely on synthetic workloads which are minimally hardware-intensive. The second is to support comparative analysis based on a simple parameter called the kernel surface area. The kernel surface area is a multi-dimensional value indicating, for each hardware resource, the number of those resources managed by the OS kernel. A simplification of this idea is to just consider the number of cores and pages of memory managed by the kernel. Our approach is to systematically vary the kernel surface area without varying the application workload, including its resource requirements and the degree of parallelism, and to analyze how surface area relates to system call variability. A simple example is shown in the Environment portion of Figure 1. Assuming a four-core system, the same application workload could be deployed across two two-core virtual machines, or one four-core virtual machine, with differences in performance attributed to a 50% reduction in kernel surface area in the two two-core virtual machine case.

This notion of kernel surface area illustrates our guiding hypothesis, which is that applications will contend for kernel resources despite the lack of any true sharing requirements in the application; i.e., all cores execute programs which, though they are designed to concurrently target specific software subsystems, are perfectly capable of executing on isolated OS instances within separate virtual machines, or even separate physical machines. By leveraging hardware virtualization, we are able to run copies of the same guest OS image in multiple VMs in order to reduce the guest kernel surface area without modifying the application’s resource requirements.

4 HIGH-LEVEL COMPARATIVE ANALYSIS

In this section, we describe our experimental environment and present a high-level analysis of kernel variability in native, virtualized, and containerized environments.

4.1 Evaluation Platform

Hardware environment. All experiments were performed on a Dell PowerEdge R6415 server, which provides an AMD EPYC 7000 series processor with 32 cores / 64 hardware threads, and 64 GB of DDR4 DRAM split over four memory channels.

Virtual machine configurations. In order to evaluate the impact of kernel surface area on system call variability, we deployed our application on a set of different VM configurations with varying numbers of cores and GB of memory. Table 1 shows the set of configurations we evaluate. In all cases, a total of 64 cores and 32 GB of memory are virtualized and used to run the benchmark. In the configuration with only 1 VM, all 64 cores and 32 GB of memory are assigned to it, while in the configuration with 64 VMs,
each VM consists of just 1 core and 512 MB of memory. All other configurations represent a middle ground between the two extremes and allow us to track the progression of variability at increasingly smaller kernel surface areas.

In all VM configurations, we leverage techniques in an attempt to reduce system variability and improve application performance. For example, all VM cores are pinned to physical CPU cores, all memory is mapped with 2MB large pages via hugetlbfs, and we leverage a virtio block device for the guest hard drives. We also utilize a TUN/TAP network device to provide a virtual Ethernet network for MPI communication across VMs, though we note that this communication is only needed to enable synchronization and does not influence system call performance.

**Container configurations.** In addition to the VM configurations, we also evaluate environments based on Docker containers, which are designed to support ease of deployment of customized user-level environments and software packages, as opposed to providing performance isolation between containers. While containers are fundamentally different than VMs from the perspective of kernel isolation (i.e., all containers shared the same OS kernel), they do leverage features such as kernel namespaces and control groups to provide some level of environment isolation. Thus, we also evaluate the variability associated with containerized systems by performing a similar analysis to the VM configuration study, with varying numbers of containers, ranging from 1 to 64, concurrently deployed to run the application benchmark. We again pin each container’s application processes to specific CPUs so as to limit variability associated with, e.g., scheduling and migration.

### 4.2 System Call Performance

We first deployed our application in three different configurations: native Linux, 64 1-core KVM VMs, and 64 1-core Docker containers, to gain a high-level understanding of the influence of VM isolation on kernel variability. In total, the Syzkaller programs we generated consisted of 27,408 system calls. Each system call is made from the same position in its program with the same arguments each time it is invoked. Each Syzkaller program was executed on all 64-cores over the course of 100 different iterations, meaning that there are at least 1 640 invocations of each generated system call.

Table 2 shows the performance of these programs. For every unique system call in the program, we tabulate each of its invocations across all 64 cores and all 100 iterations, and calculate its median, 99th percentile, and worst-case performance. We then discretize the results in order to understand how system calls of different time scales are influenced by contention within the kernel. For example, the 1µs column illustrates that 11.12% of all system calls in the application have median runtimes less than 1µs, while only 8.34% and 7.35% of system calls have medians less than 1µs in the 64 VM / 64 container environments.

The Linux and Docker configurations are superior to KVM in limiting median performance of short time scale system calls. We observe the same behavior when considering 99th percentile performance, as nearly 20% fewer system calls have 99th percentiles under 100 µs in KVM than in either Linux or Docker. However, we also observe that KVM limits 99th percentile performance at larger time scales; specifically, 99.16% of system calls have 99th percentiles under 1 ms in KVM, but only about 97.8% are limited to 1 ms in Linux and Docker. When looking at worst-case performance, we again see that nearly 2% more system calls have maximum runtimes above 10 ms in Linux and Docker than in KVM. This evidence suggests that a minimized kernel surface area, provided by VM “boundaries” between each application process, is effective at preventing tail degradation for a subset of system calls.

### 4.3 System Model

Collectively, these experiments illustrate two key points about the role of virtualization in system call performance. The first is that the presence of hardware virtualization adds a limited amount of overhead to many system calls. This is reflected in the lower percentage of system calls with medians below 10 µs and 100 µs, as well as fewer system calls with 99th percentiles below 100 µs as shown in Table 2. This overhead can be attributed to performance penalties associated with virtualization that comes with handling VM entries/exits, virtualizing CPU/memory resources, etc.

However, the second key point is that this hardware-induced overhead does not cascade to significant performance penalties at higher time scales (on the order of 10s of ms and higher), but rather is a bounded cost determined by the efficiency of the hardware. This is in contrast to the performance observed in the native Linux and Docker configurations, in which most system calls have better median performance, but some have drastically worse 99th percentile and worst-case behavior.

Collectively, these results suggest the following conceptual model of kernel performance: hardware virtualization contributes bounded overhead to most system calls, while software interference contributes less frequent but potentially unbounded overhead.

### 5 INFLUENCE OF KERNEL SURFACE AREA

We now focus on the following questions: can system call variability actually be attributed to kernel surface area? If so, which kernel subsystems most benefit from reductions in surface area?
To answer these questions, our approach is to measure system call variability using the same system call corpus presented in the previous section, but deployed across a variety of different environments that systematically vary kernel surface area and the number of isolated VMs. We analyzed the spectrum of configurations shown in Table 1, ranging from 1 64-core VM to 64 1-core VMs. By studying variability across this full set, we can attribute consistently observed performance trends to changes in the surface area.

Our goal is to understand which kernel subsystems are most heavily influenced by kernel surface area. To that end, we first characterized each Linux system call into a category (or set of categories) that broadly reflect the system call’s purpose: (a) process management/scheduling, (b) memory management, (c) file I/O, (d) filesystem management, (e) inter-process communication (IPC), and (f) permission/capabilities management. Some system calls are categorized in multiple groups; e.g., we consider chmod both filesystem and permission/capability related. Due to the sheer size of the system call API (there are over 300 system calls in the 4.16 kernel we analyzed\(^2\)), we do not present the full mapping of calls to categories here; most groups can be easily determined by reading a system call’s man page.\(^3\)

Finally, we focus our analysis on those system calls with medians of at least 10 $\mu$s in the native Linux configuration. We found that most calls with shorter medians are mmap calls that allocate small buffers, which themselves are passed as inputs to other system calls in the corpus. We analyzed these calls and found no significant variation across different VM configurations, and thus do not present them in the following section to as not not obfuscate more interesting observations.

### 5.1 Analysis of Kernel Subsystems

Figure 2 illustrates the 99th percentile runtimes for each system call, with subfigures organized by system call category. Each individual violin plot corresponds to a particular VM configuration (number of VMs) and shows both a box plot and kernel density plot of each system call in that particular VM configuration. The thick black box within a violin represents the interquartile range, the thin black line that extends from the black box shows the 95% confidence interval, and the white dot shows the median. We note that kernel surface area decreases moving left to right across a figure.

Each subsystem is impacted differently by kernel surface area; we break down our analysis based on common themes.

#### Reduction in extreme outliers.
We first focus on process management/scheduling and filesystem operations (Figures 2(a) and 2(d), respectively), which are the two subsystems that experience the greatest reduction in extreme variability as smaller kernel surface areas are deployed. Both of these cases provide evidence for our system model, which is that reductions in surface area lead to a reduction in 99th percentile outlier performance for some system

\(^2\)https://www.kernel.org

\(^3\)http://man7.org/linux/man-pages/man2/syscalls.2.html
calls. This can be seen in the fact that the top end of each plot
is gradually and consistently reduced as VM count grows from
1 up to 8, before stabilizing at 16/32 VMs and dropping again at
64. Additional evidence to this point can be seen in the increasing
thickness of the bottom-half of each violin at higher VM counts.
Higher thickness reflects greater density (i.e., greater percentage
of system calls with 99th percentiles in that range), and thus due
to the log scale on the y-axis, demonstrates that the low-VM plots
have not only higher extrema but also larger densities near the
upper portions of the plots.

The median (white dot) of all 99th percentiles is not drastically
impacted by kernel surface area in these subsystems, though the
64-VM configuration does show the lowest median in both cases. In
some cases, the median actually rises by very small amounts. This
suggests that a majority of calls are likely unaffected by smaller
surface areas, but those that do obtain significant benefits from it.

Modest but consistent benefit. Two kernel subsystems are better
categorized as receiving more modest but also more consistent
benefits from smaller surface area: memory management and per-
missions/capabilities management (Figures 2(b) and 2(f), respec-
tively). This is evidenced by a consistent reduction in the median
of all 99th percentile performance consistently reducing moving left
to right. The memory management results also show a drastic re-
duction in system call latencies in the 64-VM case, with the median
dropping from the 10s of ms range to the 100 µs range. This likely
reflects the fact that some memory management system calls re-
quire costly synchronization operations in multi-core systems: e.g.,
nummap requires cross-core TLB shootdowns to invalidate cached
page table mappings, which are obviated in a uniprocess system.
While not generally attributable to kernel surface area, this does
illustrate a benefit of uniprocessor environments.

The permission and capabilities class experiences a similar grad-
ual but consistent reduction in latency, with the whole mass moving
from nearly 10 ms in the 1 64-core VM configuration to just over 1
ms in the 64 1-core VM configuration.

Modest but inconsistent benefit. Inter-process communication
system calls, shown in Figure 2(e), exhibit modest but less consistent
reductions in latency with small kernel surface areas. These calls
are mostly similar to the process management and filesystem cases,
albeit without the major reductions in extreme outliers. The relative
increase in thickness reflects a larger density of mass around the
median in these cases, and the interquartile range consistently
drops with VM counts of 8 and higher. Again, these calls also show
benefits from single-core VM configurations.

Finally, file I/O latency, shown in Figure 2(c), does not exhibit
any clear trend as a function of surface area. There is a marginal
reduction in median, and the 64 1-core VM configuration again
shows some uniprocessor benefits, but the impact is small and
there is little to no latency reduction in high latency calls.

Takeaways. Based on these results, we observe that kernel sur-
face area has a significant impact on system call performance for
many system calls, particularly those related to process manage-
ment/scheduling, filesystem, and memory management. Further-
more, these results confirm that hardware virtualization, which
adds limited overhead to many system calls, is a useful tool for

<table>
<thead>
<tr>
<th># ctnrs</th>
<th>µs</th>
<th>10µs</th>
<th>100µs</th>
<th>1ms</th>
<th>10ms</th>
<th>&gt;10ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.02</td>
<td>11.89</td>
<td>66.57</td>
<td>98.24</td>
<td>1.76</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10.38</td>
<td>64.36</td>
<td>98.18</td>
<td>1.82</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0.01</td>
<td>13.07</td>
<td>72.74</td>
<td>98.16</td>
<td>1.84</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0.02</td>
<td>10.55</td>
<td>70.38</td>
<td>98.09</td>
<td>1.91</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0.02</td>
<td>10.72</td>
<td>73.01</td>
<td>98.04</td>
<td>1.96</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>0.02</td>
<td>6.53</td>
<td>54.59</td>
<td>97.66</td>
<td>2.34</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>0.01</td>
<td>5.63</td>
<td>47.22</td>
<td>97.64</td>
<td>2.36</td>
</tr>
</tbody>
</table>

Table 3: Breakdown of worst-case (max) system call performance in Docker with varying container configurations

providing performance isolation and reducing outlier performance
for workloads making use of these subsystems.

5.2 Analysis of Container Isolation

Lastly, due to their emergence as popular deployment methods in
HPC (e.g., Singularity [17], Shifter [11]), we find it important to
evaluate the performance of OS containers in a similar fashion. We
deployed container configurations varying from 1 64-core container
to 64 1-core Docker containers and executed the same system call
corpus from the native Linux and KVM experiments. As expected,
we saw no significant difference in median or 99th percentiles
performance across any of the configurations (results omitted for
space). However, as illustrated in Table 3, we observed that worst-
case system performance actually suffers in the configurations
with a large number of small containers. Nearly 20% more system
calls have worst-case runtimes above 1 ms in the 64 container
configuration than the 1 container configuration, and about 0.5%
more system calls have worst-case runtimes above 10 ms. Thus,
though containers do leverage techniques control groups to enforce
hardware isolation, they are still managed by a single monolithic
kernel, and these results show that the resulting environment is
still subject to kernel variability.

6 IMPLICATIONS FOR NOISE-SENSITIVE
APPLICATIONS

Finally, we turn our focus to the following question: for which
applications does this type of kernel variability matter?

We posit that the class of affected applications is actually quite
broad, especially considering that application performance is not
the only relevant goal; e.g., intrinsic variability in trusted software
makes system security techniques such as anomaly detection [29]
much more challenging. With that said, our evaluation focuses on per-
formance, and we analyze two classes of applications: (i) latency-
sensitive workloads where tail request latency is an important
measurement of scalability, and (ii) large scale bulk synchronous
parallel workloads. The former traditionally take the form of clien-
t/server workloads and are common in production datacenters and
clouds. In these settings, a server runs in the provider’s datacenter,
and clients (either internal or external) issue requests and wait for
responses from the server. The primary metric in these systems
is 95th or 99th percentile request latency, which provide a strong
measures of the system’s reliability and scalability.

The second class of applications suffer from variability because
they rely on global synchronization operations, and thus even seem-
ingly minor perturbations in a node’s runtime can render large

Reducing Kernel Surface Areas for Isolation and Scalability  
ICPP 2019, August 5–8, 2019, Kyoto, Japan
portions of the system idle due to straggler effects [18]. Given the increasingly data-driven nature of modern HPC workloads (both in computational sciences as well as large scale machine learning) the kernel will become a more critical performance contributor due to its filesystem, I/O and memory subsystem services that data-centric applications rely on.

For these reasons, we selected the tailbench [12] benchmark suite. As the name suggests, tailbench is designed to measure tail latency performance. However, several of the tailbench applications are interesting from a parallel computing perspective, including neural network training, speech recognition, and statistical machine translation, and thus we configured them in different manners to evaluate both a tail sensitive cloud environment (Section 6.2) as well as an HPC-centric large scale environment (Section 6.3). We first discuss the benchmarks used before discussing the results.

### 6.1 Tailbench Workloads

Tailbench workloads can be deployed in a variety of fashions [12]; we deploy the client/server mode where both client and server execute within the same container and communicate via socket I/O over a loopback network device. While the details of the service requests vary based on the individual benchmarks, we configured client requests rates so that the server would execute at roughly 75% utilization. Table 4 briefly describes these benchmarks; detailed discussions can be found in the original paper [12].

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xapian</td>
<td>search engine</td>
</tr>
<tr>
<td>masstree</td>
<td>in-memory key-value store</td>
</tr>
<tr>
<td>moses</td>
<td>statistical machine translation system</td>
</tr>
<tr>
<td>sphinx</td>
<td>speech recognition system</td>
</tr>
<tr>
<td>img-dnn</td>
<td>handwriting image recognition program</td>
</tr>
<tr>
<td>specjbb</td>
<td>Java middleware benchmark</td>
</tr>
<tr>
<td>silo</td>
<td>in-memory transactional database</td>
</tr>
<tr>
<td>shore</td>
<td>disk-based transactional database</td>
</tr>
</tbody>
</table>

Table 4: Tailbench applications

### 6.2 Single Node Tail Latency Evaluation

**Evaluation Environment.** Our evaluation is focused on the potential for interference effects within the OS kernel. To that end, we deployed our varbench system call programs as "noise" generators that stress the kernel while tailbench workloads are concurrently executing, and measured tailbench performance both with and without concurrently executing varbench programs. For these experiments, we evaluated tailbench using the same hardware platform from the system call evaluation section: a 32-core/64-thread AMD EPYC machine with 64 GB RAM.

We evaluated two different environments: one relying on KVM VMs to provide inter-workload isolation, and one relying on Docker containers to provide isolation. In the KVM configurations, we deployed 4 VMs in total, each with 16 cores and 8 GB of memory. The VMs are configured similarly to the previous evaluation, with vCPUs pinned to pCPUs and memory mapped via hugetlbfs; furthermore, all memory for a VM is allocated from a single memory channel to reduce hardware interference between VMs. 3 of the VMs work together to execute a 48-core varbench workload running synthetic system call programs, while the remaining VM runs a single tailbench application. We note that this final VM actually runs the tailbench workload within a single Docker container that executes within the VM, as this is the easiest way to provide the software environments needed for the tailbench applications.

For the Docker configuration, we used a similar setup, with 4 containers deployed. Again, 3 of the containers run a 48-core varbench synthetic program set, while the remaining container runs each tailbench application. For all containers, we used CPU pinning and memory control groups to limit hardware resource interference and to provide a similar hardware configuration to the VM experiments. In all cases, one tailbench application executes at a time, and we configure all tailbench clients to run for a period of about 3 minutes. To reduce the impact of cold-start, we ran each client twice and ignored the results of the first run. In each application, the client issues requests for a period of nearly three minutes, and we measure 99th percentile request latency. Furthermore, each client also includes a dedicated warm-up phase that does not factor into the performance results.

**Results.** Figure 3 shows the results of our experiments. Figure 3(a) shows 99th percentile latencies for KVM and Docker in an isolated configuration where only one VM or container is running tailbench workloads. Conversely, Figure 3(b) shows 99th percentile latencies when running the benchmarks concurrently with the 48-core varbench program. Lastly, Figure 3(c) shows the percentage increase in 99th percentile latencies between the isolated environment and the environment with competition.

The first observation we make is that, in the isolated configuration, Docker outperforms KVM on every workload. This is an expected result due to the hardware overhead associated with virtualization, and the single-tenant environment precluding any contention. The shore benchmark shows the most degradation, and this relies heavily on SSD performance; the SSD is virtualized in software via virtio in our KVM configuration, which explains the large performance overhead.

However, in the contention configuration, we see the benefits of VM-level isolation. Three of the applications – xapian, sphinx, and moses – now show superior 99th percentile performance in KVM, with the latter two exhibiting 36% and 39% better 99th percentile latency, respectively. Each of the remaining applications – masstree, img-dnn, silo, spec-jbb, and shore – still perform better on Docker, but each by a smaller margin than in the isolated environment. The relative performance differences between environments are captured in Figure 3(c), which clearly illustrates the performance isolation benefit provided by KVM. Every application experiences a larger margin of performance degradation under high contention in the Docker environment compared to KVM. Some results are drastic: sphinx and moses experienced performance degradations of 98% and 146% respectively. On average, 99th percentile increases by over 50% in Docker when kernel-intensive activity is present, while KVM 99th percentiles increase by no more than 15% in any application.

**Takeaways.** We draw two conclusions from these results. First, we see compelling evidence that kernel-level contention is problematic in the containerized configurations. Regardless of the KVM results, Docker’s drastic increases in 99th percentile latency despite limited to no hardware contention between workloads, shows that
the OS kernel is a source of interference. Second, hardware virtualization provides a promising mechanism to improve performance isolation, but the relative isolation benefit is not always significant enough to outweigh the latent virtualization cost at this small scale. While a few applications (i.e., sphinx and moses) certainly prefer KVM, others show no significant difference, and some still benefit from the container’s proximity to hardware (i.e., silo, shore). While at this scale results are mixed, we believe this is a significant result, particularly given that virtualization performance continues to improve in newer hardware and that hardware is increasingly self-virtualizable [8].

### 6.3 Benefits of Isolation at Large Scale

Small scale performance isolation improvements, even if modest, can have substantial impacts on the performance of applications at large scale [24]. The last component of our evaluation will demonstrate that all but one of the tailbench workloads we evaluate, when executing in parallel on a 64-node system, prefer a virtualized and isolated configuration over a containerized counterpart.

**Evaluation Environment.** Our final set of experiments were performed on a 64-node partition of the NSF Chameleon Cloud computing testbed [21]. Each node was identically configured with an Intel Haswell dual-socket processor with 24 cores / 48 hyperthreads and 128 GB of RAM, split evenly per each socket. We again deployed each tailbench application both in isolation as well as with competition from a co-running system call corpus, with each application executing in either a separate Docker container of KVM VM and pinned to memory and cores of a single NUMA socket to prevent hardware interference.

In order to configure the tailbench applications to execute at scale, we wrote a simple harness using MPI to deploy each client/server application in parallel across each node. Every individual node’s client/server application only issued local requests – that is, no inter-node traffic was part of the critical path of the application. Our harness executed each application for 50 iterations and applied barrier synchronization between each iteration. In this way, the applications had similar timing and synchronization requirements to BSP applications that are common in large scale HPC systems. Finally, instead of measuring 99th percentile request latency, we configured each client on each node to issue a fixed number of requests to the local server, and proceed to the barrier synchronization step once all of its requests were completed. The servers did not have SSDs, so we did not run the shore benchmark. We also encountered Java runtime failures when running specjbb on these machines, thus we omit these results as well.
Results. Figure 4 illustrates the results of our experiments. Figures 4(a) and 4(b) show application runtimes for the isolated environment and the multi-tenant environment with system call competition. As in the single node environment, we see that most applications have a small affinity for Docker when no competition is present in the system. However, in contrast to the single node cases, we see that all but the silo application prefer the KVM environment when contention is added to the system. Four of the six applications—xapian, img-dnn, sphinx, and moses—show significant performance improvements in KVM at this scale, with runtimes 20%, 5%, 30%, and 10% lower, respectively than in Docker. Figure 4(c) shows the relative performance loss from the isolated to multi-tenant configuration, again showing that all but silo experienced significantly smaller performance losses in KVM.

One benchmark, silo, did experience significantly worse performance in KVM than Docker. The most likely reason is that silo, as an OLTP database workload, is difficult to efficiently virtualize, because it: (i) has kernel code paths which are hard to efficiently virtualize without VM exits/host intervention; and (ii) is very sensitive to processor cache and TLB performance, which in turn suffer from frequent VM exits [31]. Clearly, in this case the hardware virtualization overhead is still a dominant performance factor. With that said, we emphasize two trends that will continue to push workloads towards the isolated virtualized side of this tradeoff. First, as discussed previously, hardware continues to implement more support for virtualization, meaning the number of situations such as this that require significant hypervisor intervention will shrink [8]. Secondly, core counts, and thus the potential for and impact of interference will likely increase further in future systems.

Nevertheless, these results show that almost all applications prefer to withstand base hardware virtualization overhead to gain the benefits of isolation. VMs are not a perfect solution, but they show that better system software isolation can lead to significant performance benefits at scale.

7 CONCLUSION

This paper presented a new performance evaluation methodology to analyze the presence of variability within shared multi-tenant software systems. We also formulated the notion of kernel surface area, and used our methodology to show that it correlates with the levels of variability for many kernel subsystems. Finally, we demonstrated that the isolation provided by VM-boundaries leads to more consistent tail behavior for a subset of tail-latency sensitive workloads, leading to some applications preferring a virtualized solution.

4 REFERENCES


Daniel Zahka, Brian Kocoloski and Kate Keahey