Modeling Speedup in Multi-OS Environments

Brian R. Tauro, Conghao Liu, and Kyle C. Hale
{btauro@hawk, cliu115@hawk, khale@cs}.iit.edu
Department of Computer Science
Illinois Institute of Technology

***Abstract*—For workloads that place strenuous demands on system software, novel operating system designs like unikernels, library OSes, and hybrid runtimes offer a promising path forward. However, while these systems can outperform general-purpose OSes, they have limited ability to support legacy applications. Multi-OS environments, where the application's execution is split between a compute plane and a data plane operating system, can address this challenge, but reasoning about the performance of applications that run in such a split execution environment is currently guided only by expert intuition and empirical analysis. As the level of specialization in system software and hardware continues to increase, there is both a pressing need and ripe opportunity for investigating analytical models that can predict application performance and guide programmers' intuition when considering multi-OS environments. In this paper we present such a model to place bounds on application speedup, beginning with a simple, intuitive formulation, and progressing to a more refined, predictive model. We present an analysis of the model, apply it to a diverse set of benchmarks, and evaluate it using a prototype measurement tool for analyzing workload characteristics relevant for multi-OS environments.**

I. Introduction

A growing number of applications and runtimes place intense demands on systems which push the traditional hardware and software stack to its limits. The needs of these applications often cannot be met by general-purpose operating systems (GPOSes), either owing to overheads caused by mismatched abstractions [16], [17], system interference and jitter from "OS noise" [8], [30], or unnecessary complexity introduced by a general-purpose OS design [15].

For decades the HPC community has considered—and in several cases has deployed in production [24], [11], [9], [37], [27]—*lightweight kernels*, which employ a simpler and more performant kernel design. A similar trend permeates commodity computing today, where an increasing number of systems dedicated to one application or small set of applications obviates the need for a GPOS [25], [33]. Unikernels (and their intellectual predecessor, Exokernels [6], [7]), take advantage of this fact to provide an OS tailored to a specific application. In some cases, application developers can even compile their programs directly from a high-level language (such as OCaml) into a bootable, application-specific OS image [29]. Amazon, for example, has recently developed a custom version of Linux dedicated to running containers via a lightweight hypervisor[1].

While these specialized OSes (SOSes) have been shown to increase performance, in some cases significantly [29], [28], [33], [16] one of the biggest challenges facing their widespread adoption is their non-conformance to POSIX or the Linux ABI. This means that for users to take advantage of a new OS, developers must first port applications to work with the kernel's system interface. The previously described scenario in which high-level language (HLL) applications are compiled into a kernel is actually ideal when exploring novel OS environments, as the HLL compiler[2] controls the degree to which an application written in the language leverages OS interfaces (namely, system calls). The burden of supporting a new OS environment thus lies solely on the HLL compiler/runtime designer. However, for low-level systems and scientific computing languages, such as C and Fortran, or for HLLs which make extensive use of native interfaces (e.g. Java JNI), the situation is more complicated. Developers writing their programs in low-level languages are free to use any subset of the application binary interface (ABI), and indeed, can even forgo standard libraries altogether and issue system calls directly using inline assembly. These applications therefore require more effort when porting to a new OS, particularly one which lacks POSIX support entirely, or one which employs a non-traditional execution environment (e.g. a single address space or no user/kernel isolation).

One approach to ameliorate this situation involves *delegation* of a portion of the system call interface to another OS. This approach, sometimes referred to as a multi-kernel [3] or co-kernel [31] setup, partitions the machine (either virtually [17], [18], [2] or physically [31], [26], [9], [39], [10]) such that different OSes control different resources. Usually this means that a GPOS (such as Linux) acts as the *control plane*—setting up the execution environment, launching applications, and handling system services—while an SOS acts as the *data plane* or *compute plane*. The rationale is that the majority of the application's execution will be in the compute plane, and any system services unsupported by the SOS will be delegated to the GPOS via some forwarding mechanism. We discuss this approach in more detail in Section II.

As hardware designers employ increasing levels of specialization [20], OS developers will likely follow suit. Rather than deploying monolithic kernels with drivers for an array of accelerators, we can expect to see OS deployments consisting of myriad kernels, each with its own performance properties and target applications. Thus, the ability to understand how applications perform in multi-OS environments will become

[1] https://firecracker-microvm.github.io/

[2] Or the implementation of the language runtime for interpreted/JIT-compiled languages

increasingly important.

While others have presented empirical analysis of delegation [12], and several multi-OS designs exist [31], [39], [10], [32], there has not yet been an attempt to model these environments analytically. This presents an opportunity, as "bounds and bottleneck" analysis can provide valuable insight and intuition for novel computing paradigms. Amdahl's law [1] and its successors [14], [36] provided keen insight on the limitations of parallel program performance at a time when parallel systems and algorithms were emerging. Roofline models [38] provide a useful visual tool for understanding how application performance relates to architectural limitations. Intuitive tools like the "3 C's" of cache misses, while not rigorously formulated, can provide invaluable insight into program performance and guide developers' intuition for tuning performance.

In this paper we provide a mathematical tool along these lines which we hope will give insight into the limitations of program performance in multi-OS environments. In Section III, we present and analyze a naïve yet intuitive model to represent application speedup in a multi-OS setup and subsequently refine it to present a more accurate picture of reality. We use our model to analyze the behavior of real-world benchmarks in Section IV. In Section V, we discuss the model, including how it might be used by developers, its limitations, and potential further refinements. We outline prior art in Section VI and draw conclusions in Section VII.

II. Background

Multi-OS environments are arranged such that compute intensive portions of an application run atop a specialized OS (compute plane), and system services not supported by that SOS are *delegated* to a general-purpose OS (control plane), which handles the calls and returns the results. There are two primary concerns when considering this setup: (1) which calls to forward and (2) *how* to forward them. The first concern might depend on the nature of the calls. For example, if there is no filesystem support in the SOS, system calls like `read()`, `write()`, and `open()` must be delegated. In HPC environments such I/O offload is often employed to reduce load on the parallel file system (PFS) induced by concurrent client requests from compute nodes. Instead, filesystem requests are delegated to an I/O node. In other cases, the choice of which system calls to delegate might hinge on time and resources available to the OS developer. In the interest of time, he or she might implement commonly invoked system calls in the SOS to optimize for the critical path, but choose to delegate those invoked infrequently. Gioiosa et al. showed how this choice can be guided by empirical analysis [12].

The second concern (regarding the delegation *mechanism*) depends on the capabilities of the hardware and software stack, and on the use case. For example, modern Linux kernels allow for offlining a subset of CPU cores which can then be controlled by an SOS. IHK/McKernel [10], Pisces [31], FusedOS [32], and mOS [39] leverage this feature. In this

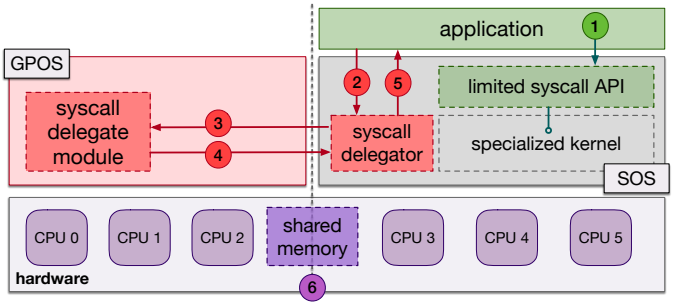


Fig. 1: High-level architecture of a multi-OS system leveraging the delegate model, where unsupported system calls are delegated to a general-purpose OS.

case, because the two OS kernels run on the same node, delegation can occur between an SOS and the GPOS using shared pages. If the SOS and GPOS are running on separate nodes, however, delegation must occur over the network, and involves marshalling arguments and initiating RPC between nodes unless the system supports distributed shared memory. Remote delegation is necessary for the I/O offloading example mentioned above. In cases where the machine is partitioned using virtualization, as in Libra [2] and Hybrid Virtual Machines [17], delegation may occur either via VMM-managed shared pages or via explicit hypercalls from guests.

Figure 1 shows an environment that supports *local* system call delegation in a hexa-core machine. The machine is physically partitioned between a GPOS and SOS such that they own a subset of the physical resources (memory and processors). In this case, the GPOS runs on cores 0–2 and the SOS runs on cores 3–5 (the compute cores). Memory is assumed to be partitioned such that the physical address space is split between them. When an application invokes a system call supported by the SOS (1), the SOS kernel handles it directly. However, when the application invokes an unsupported system call, it vectors to a handler in the specialized kernel (2), which communicates with a component (3) in the GPOS (such as a kernel module), which fields the original request. In this case, the communication between the SOS and GPOS is facilitated by a shared page of memory. This delegation process involves some subtlety, as the context of the calling (delegator) process must be mirrored by the handling (delegate) component, and the system call handler service might exist in the GPOS kernel or in a user-space process running atop the GPOS, as in User-level Servers in mOS. After the delegate handles the system call, it sends back the results to the SOS (4), which then returns the result to the application (5).

While there are technical differences between existing multi-OS systems, they all share two primary characteristics that we will use in modeling them. The first is that they assume some difference in performance when a program runs in the GPOS and in the SOS. The second is that some delegation or forwarding mechanism exists to allow the two kernels to communicate.

III. The Delegate Speedup Model

We now present two versions of a model which represents the speedup of an application in a multi-OS environment. For the following, we assume a single-threaded application (more on this in Section V) whose computation portion runs in a specialized (compute plane) OS, and whose system portion (namely, system calls) run on a general-purpose (data plane) OS. Thus, *all* system calls are initially assumed to be delegated to the GPOS.

A. Naïve Model

We begin by outlining the simplest and familiar scenario, namely where there is no system call forwarding. In this scenario, the program is executed entirely on a GPOS. Let T_{orig} be the execution time of the program in this environment. It will be useful for us to represent T_{orig} as follows:

$$T_{orig} = T_{orig} \cdot p + (1 - p) \cdot T_{orig} \tag{1}$$

Here p is the percentage of the workload related to system calls. This, for example, could be calculated by dividing the number of instructions executed in the kernel[3] by the total number of instructions retired during the run of the program. Such measures are commonly available from hardware performance counters, but the source of these terms is beyond our scope.

Now we consider a scenario wherein we execute the application in an SOS, but forward all system calls to a GPOS. Let T_{new} be the new execution time, and let γ be a constant factor that represents the speedup from running the *computational* portion of the workload in the SOS relative to running the same portion in the GPOS. We hereafter refer to γ as the *gain* factor of the SOS.

$$T_{new} = T_{orig} \cdot p + \frac{T_{orig}}{\gamma} \cdot (1 - p) \tag{2}$$

Using the familiar speedup ratio (T_{orig}/T_{new}), we arrive at the overall speedup (represented by S_n, where the n corresponds to "naïve"):

$$\boxed{S_n = \left(p + \frac{1 - p}{\gamma}\right)^{-1}} \tag{3}$$

This intuitive model has power in its simplicity. Consider the case where $p \ll 1$. This corresponds to a workload that spends very little of its time performing system calls (control plane), and thus spends most of its time computing. We might say that this application has very high "operational intensity," spending more time in userspace than in kernel space. In this case, the application receives a significant benefit[4] from the SOS, and the overall speedup reduces to γ. However, to understand how, e.g., an I/O-intensive application behaves, we observe that as p approaches 1, so does the overall speedup. The intuition here is that a system-only workload will receive *no* benefit from the SOS, and will thus spend most of its time in the control plane. It is important to note just how important p is for this model. Consider that as γ tends towards infinity, this speedup relation tends to $\frac{1}{p}$.

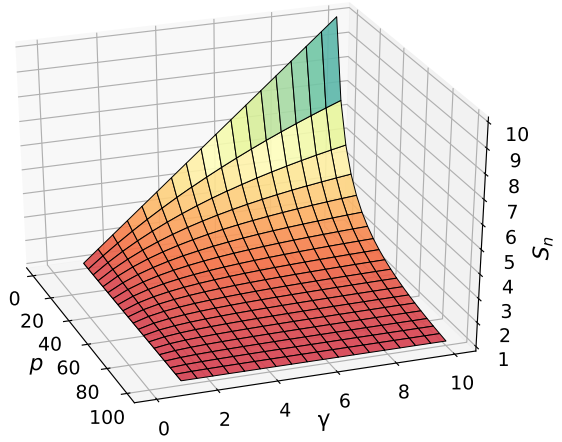


Fig. 2: Speedup in a multi-OS environment (S_n) given the proportion of time an application spends on system calls (p) and the gain factor (γ) from running in the specialized OS.

This succinctly captures the bounded speedup of the multi-OS environment, and echoes the insight provided by Amdahl's Law. Essentially what this says is that *even with infinite improvement* of the computational portion of a workload by the specialized OS, the speedup of the application is bounded by how much it relies on the legacy system interface. Figure 2 depicts how this model behaves as p and γ vary. Notice how as p grows smaller, we reach perfect linear speedup. The gain factor (γ) is the interesting part of this model, and it very closely resembles the parameter representing parallelism in Amdahl's Law. Semantically, however, they are quite different. In practice, we do not expect the gain to be very large (likely < 10), but the interplay between γ and p are still significant for application performance.

One can also use this model from the perspective of a kernel developer, in which case it can be used to determine where to focus development efforts. For example, even if monumental effort is spent improving the computational aspect of a workload (e.g. by focusing on developing efficient threading libraries), it might make very little impact if the kernel will run I/O-intensive applications. This echoes the well-known principle, "optimize for the common case."

To understand how this model might translate into real application performance, we first determined p for a selection of benchmarks in the SPEC CPU 2017 suite, and projected real application speedup given a fixed gain (γ) factor.

[3]This would include transparent system events such as page fault exceptions and interrupt handling, but we ignore this detail.

[4]This, for example, might be due to guaranteeing cooperative scheduling or might be due to an address space setup amenable to (or tailored to) the application.

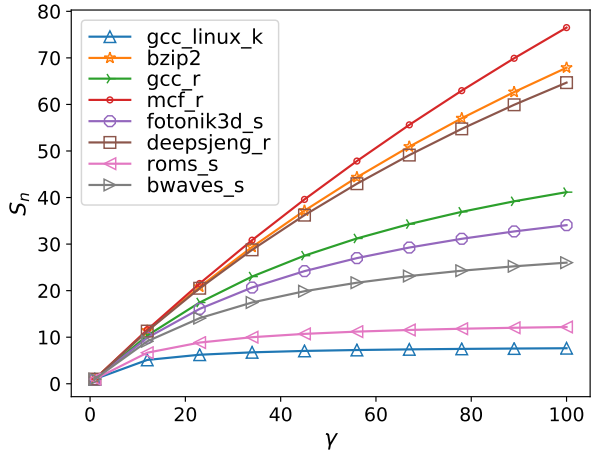


Fig. 3: Speedup projected by our simple model for SPEC CPU 2017 benchmarks when the gain factor (γ) is varied.

TABLE I: SPEC CPU 2017 benchmarks and empirically determined system call portion (p).

Benchmark	Description	p
`gcc_linux_k`	Linux kernel compilation	12.23%
`rom_s`	Regional Ocean Modeling System	7.27%
`bwaves_s`	Blast wave simulation	2.87%
`fotonik3d_s`	Computational Electromagnetics (CEM)	1.95%
`gcc_r`	code generation for IA-32	1.4%
`deepsjeng_r`	Deep Sjeng chess engine (tree search)	1.12%
`bzip2`	bzip compression	0.48%
`mcf_r`	combinatorial optimization	0.31%

To determine p empirically, we used the `time` command for a reference run of the individual benchmark, and determined p by taking the ratio of system time to total (user and system) time. We note that this will actually be an *overestimate* of p, as some portions of time spent in the kernel will *not* be due to system calls, but rather due to other paths through the kernel such as interrupts and exceptions (e.g. page faults). Furthermore, p may vary for a particular benchmark when its inputs are changed. Table I shows the empirically determined values of p and descriptions for each benchmark we used. Figure 3 shows the results of our experiment. Linux kernel compilation (`gcc` in the graph) stresses the system interface the most (due to heavy file I/O), and thus achieves very little speedup, even with a significant initial speedup from the SOS, represented by γ. The other benchmarks are heavily skewed towards computation (which is not surprising given the nature of these benchmarks), and thus achieve sub-linear speedup as γ increases.

B. Extended Model

While our naïve model can be used as an intuitive tool, there are several simplifications that make it unrealistic in terms of predicting performance:

1) The cost of forwarding system calls is ignored. In the existing model, this means that we assume *all* of them are forwarded.
2) Different system calls have different costs (in terms of execution time).
3) A given system call will have different costs for different invocations (in most cases determined by its arguments). Consider, for example, the `read()` system call.
4) System calls which are *ported* to the SOS might have different cost than the original GPOS version.
5) It is inaccurate to say that the speedup factor (γ) applies uniformly to all non-system instructions in the program. For example, the SOS environment might have a simplified paging setup (e.g. identity-mapped, 1GB pages) which significantly reduces TLB misses for instruction fetches and loads and stores, but integer/floating point instructions will be unaffected.
6) Setups where there are more than one GPOS and more than one SOS are not considered.

In this section, we refine our model by addressing (1), (2), and (3) above. We intend to refine the model further in future work to account for the remaining simplifications.

We first must capture the fact that different system calls can have different costs ((2) and (3)). Let $\mathbb{S} = \{s_1, s_2, \ldots, s_n\}$ be the set of all system calls invoked during a particular run (with fixed inputs) of program P. We introduce a function $g : \mathbb{S} \to \mathbb{R}$, such that $g(s)$ gives the absolute time taken for *all invocations* of system call s in the GPOS for the run of program P. For example, if one program run contained several invocations of `mmap()` (which is common), $g(mmap)$ will represent the time taken for *all* such invocations[5] when run on a general-purpose OS.

Let C represent the absolute time taken by the computational portion of the program (that is, all instructions not executed in the context of a system call).

We can then calculate the total execution time in the default case, where the program runs entirely in the GPOS and no system call delegation occurs (represented by t_{nd}) as follows:

$$t_{nd} = C + \sum_{s \in \mathbb{S}} g(s) \tag{4}$$

We then must capture the notion of system call delegation. We introduce two functions $f : \mathbb{S} \to \mathbb{R}$ and $b : \mathbb{S} \to \{0, 1\}$. The function $f(s)$ represents the time required to forward system calls, defined as:

$$f(s) = 2 f_c n(s) \tag{5}$$

Here f_c is a constant that represents the *base* forwarding cost for all system calls using a particular forwarding mechanism, and the function $n : \mathbb{S} \to \mathbb{N}$ represents the number of times system call s is invoked. f_c is scaled by a factor of two to account for the round-trip from the SOS to the GPOS. That is, a system call is forwarded from the SOS to GPOS, executed on the GPOS, and the results are sent back to the SOS, so the forwarding overhead is incurred twice. f_c will

[5] sum of the execution times of all mmap() invocations

vary widely depending on the software mechanisms which implement forwarding and the underlying interconnect over which system calls are forwarded[6]. For example, for delegation over shared memory (Section II), f_c would likely be no more than a few nanoseconds. For delegation over a network, this number might be closer to several μs or several ms, depending on the network characteristics.

The second function, $b(s)$ tells us whether or not a particular system call is delegated:

$$b(s) = \begin{cases} 0, & \text{if } s \text{ is not delegated} \\ 1, & \text{otherwise} \end{cases} \tag{6}$$

Recall that the program primarily runs in the context of the SOS, and so receives some performance benefit (represented before by the factor γ) as a result. Thus, as before, we only apply γ to the computational portion of the workload (C). For the system call portion of the workload, we must differentiate between delegated system calls and local system calls (those which have corresponding implementations in the SOS). We can represent the absolute time taken by all *local* system calls (t_{local}) by introducing another function g' which captures this notion. We use $g'(s)$ to represent the time taken for all invocations of system call s given the custom version of s implemented in the SOS.

$$t_{local} = \sum_{s \in \mathbb{S}} (1 - b(s)) g'(s) \tag{7}$$

We then represent the absolute time taken by all *delegated* system calls (t_{remote}) as follows:

$$t_{remote} = \sum_{s \in \mathbb{S}} b(s)(g(s) + f(s)) \tag{8}$$

We can now calculate the total time taken (t_d) for a setup where *some* system calls are delegated:

$$t_d = \frac{C}{\gamma} + t_{remote} + t_{local} \tag{9}$$

Thus, we can represent our new speedup (S_r) as

$$S_r = \frac{t_{nd}}{t_d} \tag{10}$$

Expanding this out, we get

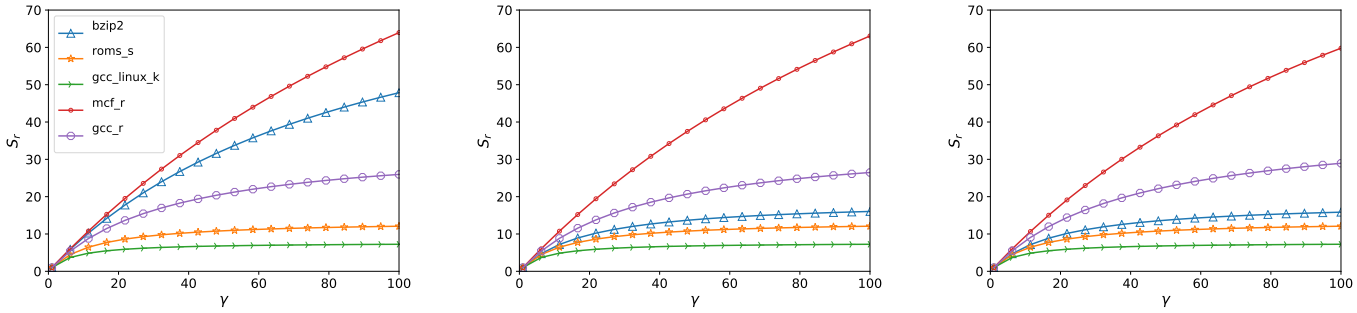
$$\boxed{S_r = \frac{C + \sum_{s \in \mathbb{S}} g(s)}{\frac{C}{\gamma} + \sum_{s \in \mathbb{S}} b(s)(g(s) + f(s)) + (1 - b(s))g'(s)}} \tag{11}$$

Intuitively, the more system calls that are forwarded (those which have $b(s) = 1$), the more overhead is incurred, and speedup is curtailed. Notice that in the denominator, the time taken by the computational portion is scaled by a factor of $\frac{1}{\gamma}$. An ideal scenario would have $g'(s)$ take *less* time than $g(s)$, meaning that an implementation of a system call in the SOS would be more efficient than its counterpart in the GPOS. However, going forward, we make the simplifying assumption that $g(s) = g'(s)$, meaning that both implementations take the same amount of time.

Figure 4 illustrates speedup projections (represented by S_r) using our refined model for a subset of the benchmarks shown in Table I. We initially fix the forwarding cost, f_c, at 10 μs. This is representative of a scenario where forwarding occurs between separate physical nodes over a network using a low-latency interconnect such as Infiniband. We vary γ to illustrate the effects of running the application in the SOS. While a gain factor of 100 is unlikely, these graphs help illustrate the limits of application speedup. Each benchmark in the suite invokes a different set of system calls, and here we are interested in seeing the effects of choosing different sets of system calls to forward. In this case, we show three scenarios. A fixed proportion of 20% of system calls are forwarded. In each graph we vary *which* 20% are forwarded. In the first two scenarios, system calls are chosen according to how many times they are invoked. Figure 4a shows the projected speedup when system calls invoked *infrequently* by the application are chosen to be forwarded. This is the most ideal scenario, as the forwarding overheads will not be incurred often. For comparison, Figure 4b shows the speedup when we choose system calls which are invoked *most frequently* (the worst case). Finally, Figure 4c depicts the results when we make a random choice. Note how different benchmarks are affected differently by the forwarding schemes. `mcf_r` achieves a high speedup no matter the scheme. This is because it uses the least system calls of all the benchmarks (as shown it Table I, it only spends 0.31% of its time in system calls). `bzip2`, however, shows a more remarkable difference when we compare the min. frequency case with the max. frequency case. To see why, it is illustrative to study Figure 6, which shows a breakdown of system call usage for several of the benchmarks. Looking at the CDF for `bzip2`, it is clear that its speedup is curtailed because it uses a small set of system calls very often (this particular application invokes `read()` and `write()` almost exclusively). It is thus critical that those system calls are *not* forwarded. When they are, as in Figure 4b, performance is severely affected. It is also clear from the figures that applications which have a more varied system call distribution will be less affected by selective forwarding schemes. More generally, workloads showing system call invocations with more statistical structure will be more amenable to selective forwarding schemes. This aligns with intuition and prior empirical results from Gioiosa et al. [12]. It is also interesting to note that random selection shows a speedup only slightly better than the worst case.

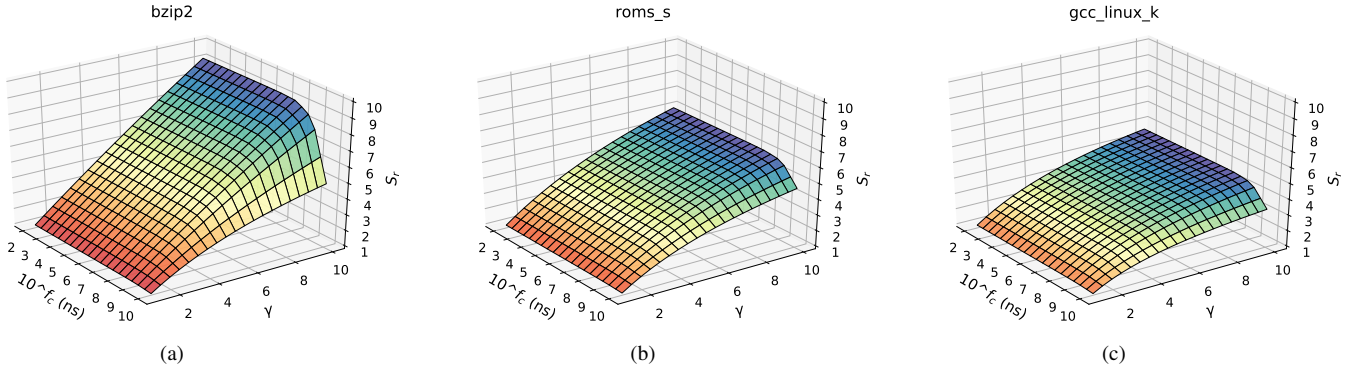
In Figure 5, we choose three benchmarks from the SPEC suite, and show in the surface plots how their projected speedup changes as we vary both the forwarding cost (f_c) and

[6]Here we make the simplifying assumption that this cost is *independent* of the system call itself, but this is not strictly true. A forwarding mechanism that uses marshalling (e.g. over a network) will incur more forwarding costs for a system call with more arguments.



(a) min. frequency (b) max. frequency (c) random

Fig. 4: Projected speedup as the gain (γ) varies for SPEC CPU 2017 benchmarks. This assumes a fixed forwarding cost (f_c) of 10 μs and a fixed proportion (20%) of forwarded system calls. Three schemes for choosing which system calls to forward are shown. From left to right, we select system calls by least frequently invoked (a), most frequently invoked (b), and randomly (c).



(a) (b) (c)

Fig. 5: Projected speedup as the gain (γ) and forwarding cost (f_c) are varied for SPEC benchmarks. Here the proportion of forwarded system calls is fixed at 20%, and which calls to forward is determined according to those least frequently invoked (min. frequency). Note the log scale on the f_c axis.

the gain factor from execution in the SOS. Here we forward the lowest 20% of system calls (those invoked most *infrequently*). It is important to note the log scale on the f_c axis, so the lower end of the scale indicates forwarding costs in the nanosecond range, the middle approaches milliseconds, and the higher end approaches roughly ten seconds. Along the γ axis, all benchmarks achieve a speedup, but note that from left to right these benchmarks have characteristics less amenable to system call delegation, respectively. `bzip2` achieves the highest speedup, both because it has a low system call portion, and because that portion involves very few system calls that are invoked frequently. Linux kernel compilation with `gcc` has the highest system call portion, and a more uniform distribution of system calls, which leads to a curtailed speedup. Forwarding cost only becomes a significant factor in all cases when it becomes greater than tens of milliseconds.

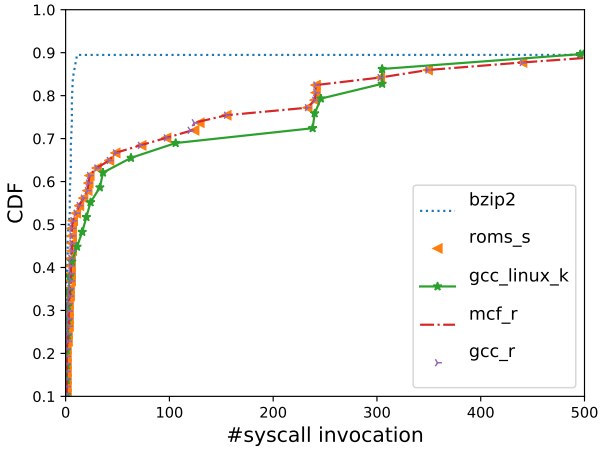


Fig. 6: Application system call profile for selected benchmarks.

Figure 7 shows another perspective on forwarding overheads. Here, we show the speedup projected by our model as we vary both the forwarding cost (f_c) and the percentage of

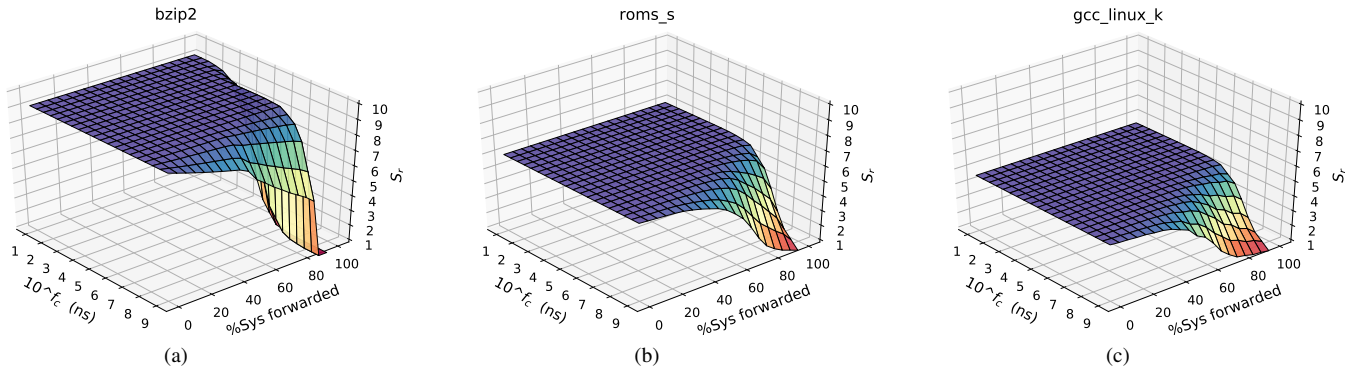


Fig. 7: Projected speedup as the forwarding cost (f_c) and the proportion of forwarded calls are varied, with the gain (γ) fixed at 10. Note the log scale on the f_c axis.

forwarded system calls (assuming that percentage consists of infrequently invoked system calls). We make two observations, both of which again align with intuition. The first is that workloads with more skewed system call distributions are more amenable to delegation, even with a relatively large portion of the system call interface is forwarded. The second is that when these workloads are *not* properly accounted for (i.e. when the *wrong* calls are forwarded, the performance degradation is dramatic, as is shown in the curve for `bzip2`. An interesting note about this visualization is that the "topography" of the speedup curves directly reflect the structure in the application's system call invocation trace. `bzip2`'s surface has steeper drop-offs when an increasing number of system calls are forwarded, indicating a heavy skew in the system call distribution. The smoother "rolling hills" of the other two benchmarks indicate a more even distribution of calls.

In Figure 8, we vary the gain and the proportion of forwarded calls. The interesting point here is that that with the fixed forwarding cost of 10 μs, we only see an effect for `bzip2` when almost *all* calls are forwarded (thus capturing the frequently invoked `read()` and `write()` calls). Kernel compilation and `roms_s` are largely unaffected by such small forwarding overheads.

In Figure 9, we again show the effects of different forwarding policies, but as a function of varying forwarding overheads. `mcf_r` and `bzip2` both have steep dropoffs in speedup when the wrong set of system calls is forwarded. When *infrequently* invoked system calls are delegated to the GPOS, forwarding overheads must reach between 100 ms and 1 sec before making a significant impact.

IV. Experimental Analysis

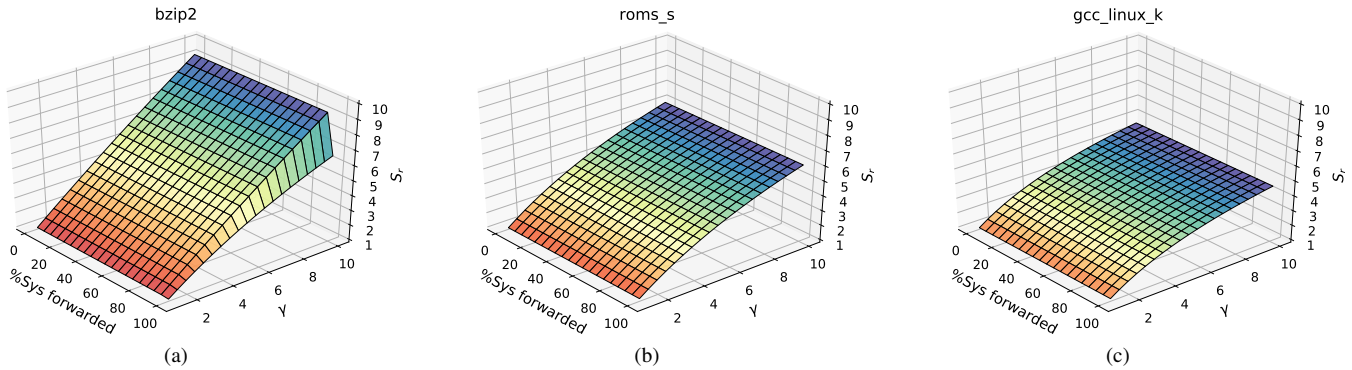
The primary obstacle in measuring application performance directly in a multi-OS environment is the OS development burden required to implement functionality in the SOS. The simplest case is when the *entire* system interface is delegated to a GPOS. In this case, the application benefits solely from the properties of the execution environment provided by the SOS (e.g. simplified, deterministic paging), and no development effort is spent porting system calls to the SOS. However, this scenario is not ideal, as the results from the previous section indicate. However, it would be useful to project performance *before* undertaking the development effort to do so. In this section, we outline a tool that enables this projection, allowing users interested in multi-OS setups to perform "what-if" analysis. Users can run their unmodified programs using our tool to project performance in a multi-OS setup.

Note that this tool does not actually leverage two separate operating systems. Instead, we leverage a Linux kernel module that employs a kernel thread on the *same OS* to serve as a delegate (standing in for the control-plane OS); this delegate fields system call requests from a running process. At a high-level, using our tool, a configurable subset of system calls are intercepted by the kernel, which forwards them to this delegate thread with a tunable forwarding cost. With this architecture we can experiment with different delegation schemes to determine performance *without* spending effort porting an application to a new OS.

Our tool, called `mktrace`, is freely available online[7], runs on Linux, and only requires that users load a kernel module before using it.

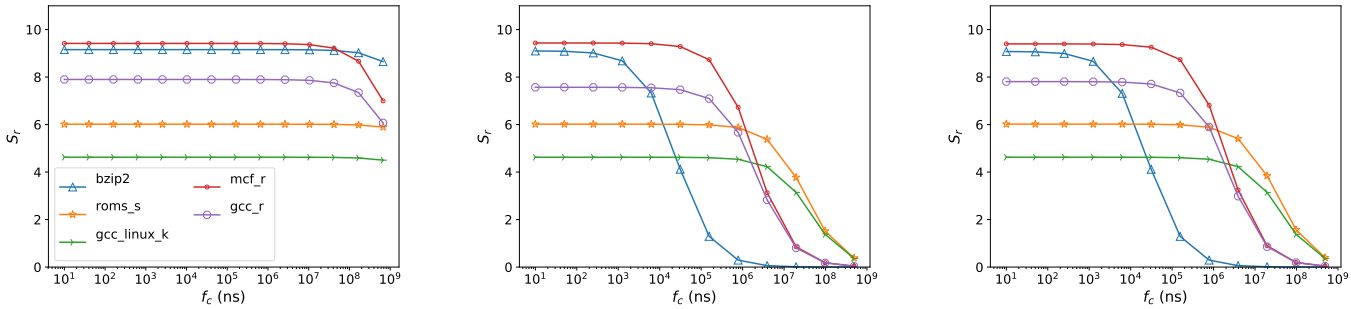
The primary technique used by our tool is *system call interposition*. This technique has been used primarily for secure monitoring of kernel activity, both using in-kernel or user-level approaches [13], [21], [23], [34] and out-of-kernel by leveraging an underlying virtual machine monitor (VMM) [19], [5], [35]. Typical interposition tools provide hooking points for system call entry and exit, but we only need to capture entries in order to forward them. Figure 10 illustrates how our tool works. After a user loads our system call interceptor (a kernel module), system calls can be selectively forwarded, with tunable forwarding cost. This is achieved by patching the kernel's system call table. In the figure, a regular system call `bar()` (a) is invoked, which vectors via a system call table entry (b) to the kernel's handler for that system call (c). When a forwarded system call `foo()` (1) is invoked, our patched

[7]https://github.com/hexsa-lab/mktrace



(a) (b) (c)

Fig. 8: Projected speedup as the gain (γ) and the proportion of forwarded calls are varied. Here the forwarding overhead (f_c) is fixed to 10 μs.



(a) min. frequency (b) max. frequency (c) random

Fig. 9: Projected speedup as the forwarding cost (f_c) varies. Here we fix the gain factor (γ) to 10 and assume that 20% of system calls are forwarded. Again, forwarding is based on minimum frequency (a), maximum frequency (b), and random selection (c).

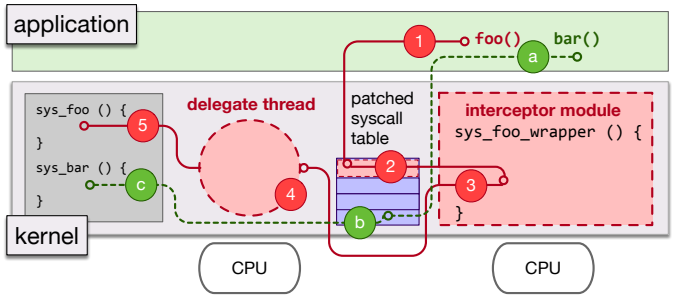


Fig. 10: High-level overview of mktrace.

system call table entry (2) vectors instead to our module (3), which mirrors register state (arguments) and the execution environment in the calling process (e.g. address space). Our module then forwards the system call to a *delegate thread*, backed by a separate kernel thread on a separate CPU (4), which spins for a configurable amount of time (representing the forwarding delay), and then invokes the kernel's original system call handler (5). The results are sent back to the calling process and execution continues normally.

A. *Experimental Setup*

We conducted our experiments on a system called *fusion*, which has a 1.9GHz AMD Opteron 6168 CPU with 48 cores and 256 GB of memory spread out across four sockets and eight NUMA nodes. It runs Fedora 26 with stock Linux kernel version 4.16.11. For these experiments, we selected several benchmarks from the initial set: `bzip2`, compiling Linux kernel version 5.1.4 with `gcc`, `roms_s`, and `mcf_r`.

Since we are not actually running these applications in a real multi-OS system, we cannot empirically observe the speedup factor γ given by running the application in the SOS. To approximate this factor, we designed a synthetic benchmark which performs phases of variable amounts of computation followed by phases of fixed system call invocations. The computation is a simple Monte Carlo calculation of π, which is dominated by floating point operations. To artificially induce a speedup, we simply vary the amount of computation (by reducing the number of trials in the approximation) according to a γ value. Thus, a higher value of γ is approximated by a concomitant reduction in the amount of work done in the computation phase.

B. Experiments

We first run the benchmarks described above both in the normal fashion on Linux, and then using our `mktrace` tool, which approximates delegation by forwarding system calls to a kernel thread on another core. We measured the maximum overhead of forwarding using this mechanism (represented by f_c in previous sections) to be 40 μs. Figure 11 shows the results.

In all cases, because of the small forwarding overhead, there is very little impact on performance. We do not see the upward trend in performance shown in previous graphs because here γ is not being accounted for. Thus, a good result here is when the bars are the same, indicating that forwarding did not *reduce* performance significantly. As projected by our model, when the most frequently invoked system calls are forwarded, `bzip2` is the most affected. However, because of the small forwarding overheads, there is only a 6% performance drop in this case.

As described above, we now attempt to incorporate the factor γ by approximating gain using a variable amount of computation. First, we show the system call profile for our synthetic benchmark in Figure 12. We designed the benchmark to be similar in profile to `bzip2`, so that there are a few very frequently invoked system calls and several infrequently invoked calls as well. The distribution is heavily skewed towards `write()` (1.5 million invocations), with other calls being invoked less than a few hundred times.

Figure 13 shows the projected speedup from both the simple model and the refined model compared to the speedup empirically determined by running our synthetic benchmark with varying gain using our `mktrace` tool. Here again we are forwarding only the 20% least frequently invoked system calls. Noting the system call profile, this means that we can expect forwarding to affect only a few relatively unimportant system calls, which is reflected by the proximity of the lines for our two models (the refined speedup has a slightly smaller speedup due to forwarding, which is not taken into account with the naïve model).

The curve labeled "empirical speedup" shows the results for running this synthetic benchmark with and without `mktrace`. When smaller gain factors are used, the experiments track the speedup projected by our models. For example, for $\gamma = 2$ (a fairly realistic scenario), the projected speedup is within 9% of the experimental results. As γ grows to larger values, our models become less accurate. We suspect that this inaccuracy stems from indirect performance effects caused by our existing `mktrace` implementation. For example, TLBs are flushed on every forwarding event to make sure system calls related to paging are serviced correctly. The performance hit caused by the subsequent TLB misses are not captured by our models.

V. Discussion and Limitations

What is clear from the previous section is that using the models we presented, one can develop intuition for how an application will behave in a multi-OS setup. While the refined model does not predict performance precisely for our proxy setup, it demonstrates the overall trends, and can guide development efforts when designing a custom OS. However, there are still several limitations with these models. The primary limitation is that our current models assume a single delegator thread and a single delegate thread (one on the GPOS and one on the SOS). While this is a reasonable setup for serial workloads, it is unrealistic for parallel applications, where system call requests from the delegator might come from several cores or several machines. In this case, a single delegate (GPOS) thread servicing these requests would be inadequate. In some cases, *several* applications might be running at once, making delegation requests. This can easily introduce a choke point in the delegate, and requires careful consideration. We do not currently capture these concerns in our model, but we plan to incorporate them into future iterations.

Part of the inaccuracy of the model comes from the remaining simplifying assumptions described in Section III-B. In particular, it we assume that system call implementations in the specialized OS have the same cost as when implemented in the GPOS. In future work we intend to address this and explore other ways of modeling the gain offered by execution in the specialized OS other than a simple multiplicative factor.

VI. Related Work

As far as we are aware, we are the first to model speedup analytically for multi-OS environments. We refer to Gioiosa et al. for an excellent empirical study of system call delegation [12].

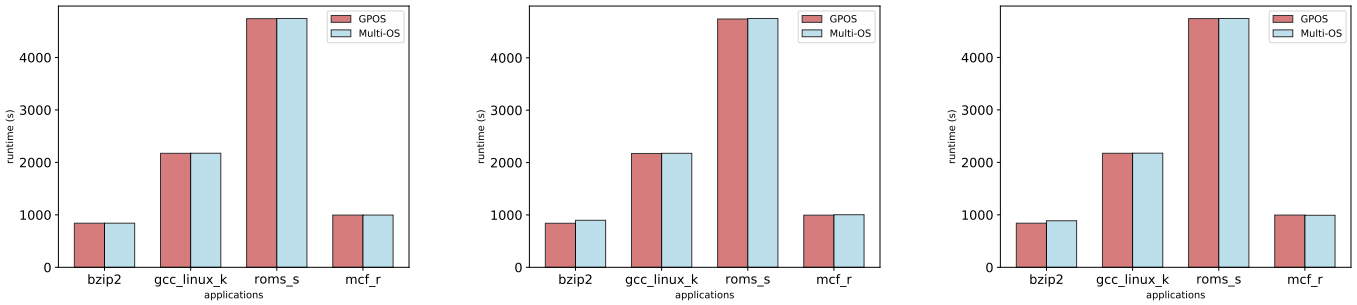
Our work was inspired by Amdahl's original formulation of parallel speedup [1], Gustafson's refinement [14], and Sun and Ni's extended model for incorporating memory-bound programs [36].

One might view a multi-OS setup as a general distributed system, where forwarded system calls are simply treated as RPCs. In its simplest case, such a system might suitably be modeled using a LogP model [4]. However, models like LogP primarily relate to the communication/computation ratio, and furthermore do not consider the asymmetry between execution times and system interfaces in different operating systems. Our model, in contrast, is designed to capture this asymmetry. As we extend our speedup model to include concurrently executing SOS and GPOS threads, we intend to draw on existing models of parallel computation.

Applications *limited* by system call usage can be viewed through a lens of "operational intensity," which others have visualized using roofline models [38], [22]. While roofline models are based on architectural characteristics, rather than properties inherent to the application and system software stack, we believe they could be adapted to provide insight for multi-OS systems, and we plan to explore this further in follow-up work.

VII. Conclusions and Future Work

We introduced two models to guide intuition on application speedup when a program's execution is split between two



(a) min. frequency (b) max. frequency (c) random

Fig. 11: Runtime of selected benchmarks running natively in the GPOS and running on top of our mktrace tool to approximate system call delegation overheads. We again show the results for three forwarding schemes.

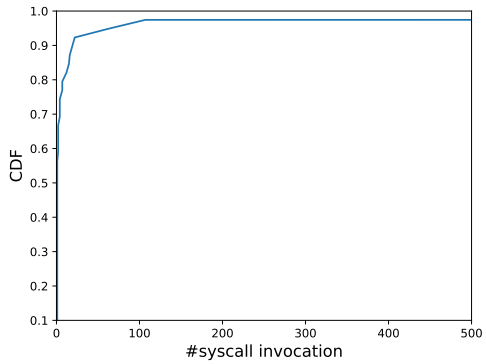


Fig. 12: System call profile for our synthetic benchmark.

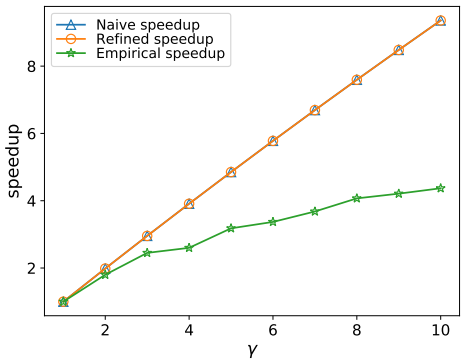


Fig. 13: Speedup for the synthetic benchmark when the gain factor (γ) is varied across the synthetic benchmark.

operating systems. We illustrated in detail how speedup gained by running in an environment provided by a heavily optimized OS can be curtailed by system call delegation overheads and by a poor choice of forwarding policy. We showed that applications with skewed system call distributions can tolerate higher forwarding overheads when a good policy is employed, but suffer from more dramatic effects when the wrong system calls are forwarded. To measure the effect of these overheads on real applications, we presented an open-source tool called `mktrace` which approximates forwarding overheads by delegating system calls to a remote thread. Using this tool, we showed that with μs-scale forwarding costs, very little overhead (max ~6%) is introduced for a variety of benchmarks. Finally, while our intent is to provide an intuitive guide for reasoning about speedup for multi-OS systems, we showed using a synthetic benchmark with a variable proxy for speedup from a specialized OS that our refined model predicts performance for a proxy delegation system within 9% of actual performance when a reasonable execution environment is considered.

In future work, we plan to extend our models to include multi-OS systems that do not assume a single delegator and delegate thread, but rather employ varying degrees of parallelism. We plan to extend `mktrace` to support multi-node systems. Finally, we plan on investigating other perspectives on multi-OS speedup, including roofline models.

VIII. Acknowledgments

We wish to thank the anonymous reviewers for their valuable feedback. We also thank Alexandru Iulian Orhean for providing access to hardware resources and his key insights on the speedup model. We thank Amal Rizvi for her valuable feedback on the speedup model.

References

[1] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference* (Apr. 1967), AFIPS '67 (Spring), pp. 483–485.

[2] Ammons, G., Appavoo, J., Butrico, M., Da Silva, D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B., Van Hensbergen, E., and Wisniewski, R. W. Libra: A library operating system for a JVM in a virtualized execution environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (June 2007), VEE '07, pp. 44–54.

[3] Baumann, A., Barham, P., Dagand, P. E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Oct. 2009), SOSP '09, pp. 29–44.

[4] CULLER, D., KARP, R., PATTERSON, D., SAHAY, A., SCHAUSER, K. E., SANTOS, E., SUBRAMONIAN, R., AND VON EICKEN, T. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (May 1993), PPOPP '93.

[5] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15^{th} ACM Conference on Computer and Communications Security (CCS 2008)* (Oct. 2008), pp. 51–62.

[6] ENGLER, D. R., AND KAASHOEK, M. F. Exterminate all operating system abstractions. In *Proceedings of the 5^{th} Workshop on Hot Topics in Operating Systems (HotOS 1995)* (May 1995), pp. 78–83.

[7] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15^{th} ACM Symposium on Operating Systems Principles* (Dec. 1995), SOSP '95, pp. 251–266.

[8] FERREIRA, K. B., BRIDGES, P., AND BRIGHTWELL, R. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of Supercomputing* (Nov. 2008), SC '08.

[9] GARA, A., BLUMRICH, M. A., CHEN, D., CHIU, G. L.-T., COTEUS, P., GIAMPAPA, M. E., HARING, R. A., HEIDELBERGER, P., HOENICKE, D., KOPCSAY, G. V., LIEBSCH, T. A., OHMACHT, M., STEINMACHER-BUROW, B. D., TAKKEN, T., AND VRANAS, P. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development 49*, 2 (Mar. 2005), 195–212.

[10] GEROFI, B., TAKAGI, M., HORI, A., NAKAMURA, G., SHIRASAWA, T., AND ISHIKAWA, Y. On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel. In *Proceedings of the 30^{th} IEEE International Parallel and Distributed Processing Symposium* (May 2016), IPDPS '16, pp. 1041–1050.

[11] GIAMPAPA, M., GOODING, T., INGLETT, T., AND WISNIEWSKI, R. W. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *Proceedings of Supercomputing* (Nov. 2010), SC '10.

[12] GIOIOSA, R., WISNIEWSKI, R. W., MURTY, R., AND INGLETT, T. Analyzing system calls in multi-OS hierarchical environments. In *Proceedings of the 5^{th} International Workshop on Runtime and Operating Systems for Supercomputers* (June 2015), ROSS '15.

[13] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6^{th} USENIX Security Symposium* (July 1996), SSYM '96.

[14] GUSTAFSON, J. L. Reevaluating amdahl's law. *Communications of the ACM 31*, 5 (May 1988), 532–533.

[15] HALE, K. C., AND DINDA, P. An evaluation of asynchronous events on modern hardware. In *Proceedings of the 26^{th} IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2018)* (Sept. 2018).

[16] HALE, K. C., AND DINDA, P. A. A case for transforming parallel runtimes into operating system kernels. In *Proceedings of the 24^{th} ACM Symposium on High-performance Parallel and Distributed Computing* (June 2015), HPDC '15.

[17] HALE, K. C., AND DINDA, P. A. Enabling hybrid parallel runtimes through kernel and virtualization support. In *Proceedings of the 12^{th} ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Apr. 2016), VEE'16, pp. 161–175.

[18] HALE, K. C., HETLAND, C., AND DINDA, P. A. Multiverse: Easy conversion of runtime systems into os kernels via automatic hybridization. In *Proceedings of the 14^{th} IEEE International Conference on Autonomic Computing* (July 2017), ICAC'17.

[19] HALE, K. C., XIA, L., AND DINDA, P. A. Shifting GEARS to enable guest-context virtual services. In *Proceedings of the 9^{th} International Conference on Autonomic Computing (ICAC 2012)* (Sept. 2012), pp. 23–32.

[20] HENNESSY, J. L., AND PATTERSON, D. A. A new golden age for computer architecture. *Communications of the ACM 62*, 2 (Jan. 2019), 48–60.

[21] HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. Intrusion detection using sequences of system calls. *Journal of Computer Security 6*, 3 (Aug. 1998), 151–180.

[22] IBRAHIM, K. Z., WILLIAMS, S., AND OLIKER, L. Roofline scaling trajectories: A method for parallel application and architectural performance analysis. In *Proceedings of the International Conference on High Performance Computing and Simulation* (July 2018), HPCS '18, pp. 350–358.

[23] JAIN, K., AND SEKAR, R. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *In Proceedings of the Network and Distributed System Security Symposium* (Feb. 2000), NDSS '00.

[24] KELLY, S. M., AND BRIGHTWELL, R. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Meeting* (May 2005), CUG'05.

[25] KIVITY, A., LAOR, D., COSTA, G., ENBERG, P., HAR'EL, N., MARTI, D., AND ZOLOTAROV, V. OSv—optimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX Annual Technical Conference* (June 2014), USENIX ATC'14.

[26] KOCOLOSKI, B., AND LANGE, J. XEMEM: Efficient shared memory for composed applications on multi-os/r exascale systems. In *Proceedings of the 24^{th} International Symposium on High-Performance Parallel and Distributed Computing* (June 2015), HPDC '15, pp. 89–100.

[27] LANGE, J., PEDRETTI, K., HUDSON, T., DINDA, P., CUI, Z., XIA, L., BRIDGES, P., GOCKE, A., JACONETTE, S., LEVENHAGEN, M., AND BRIGHTWELL, R. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24^{th} IEEE International Parallel and Distributed Processing Symposium* (Apr. 2010), IPDPS'10.

[28] LANKES, S., PICKARTZ, S., AND BREITBART, J. HermitCore: A unikernel for extreme scale computing. In *Proceedings of the 6^{th} International Workshop on Runtime and Operating Systems for Supercomputers* (June 2016), ROSS'16.

[29] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18^{th} International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2013), ASPLOS'13, pp. 461–472.

[30] MORARI, A., GIOIOSA, R., WISNIEWSKI, R. W., CAZORLA, F. J., AND VALERO, M. A quantitative analysis of os noise. In *Proceedings of the 25^{th} IEEE International Parallel and Distributed Processing Symposium* (May 2011), IPDPS '11, pp. 852–863.

[31] OUYANG, J., KOCOLOSKI, B., LANGE, J. R., AND PEDRETTI, K. Achieving performance isolation with lightweight co-kernels. In *Proceedings of the 24^{th} International Symposium on High-Performance Parallel and Distributed Computing* (June 2015), HPDC '15, pp. 149–160.

[32] PARK, Y., HENSBERGEN, E. V., HILLENBRAND, M., INGLETT, T., ROSENBURG, B., RYU, K. D., AND WISNIEWSKI, R. W. FusedOS: Fusing LWK performance with FWK functionality in a heterogeneous environment. In *Proceedings of the IEEE 24^{th} International Symposium on Computer Architecture and High Performance Computing* (Oct. 2012), SBAC-PAD '12, pp. 211–218.

[33] PETER, S., AND ANDERSON, T. Arrakis: A case for the end of the empire. In *Proceedings of the 14^{th} Workshop on Hot Topics in Operating Systems* (May 2013), HotOS '13.

[34] PROVOS, N. Improving host security with system call policies. In *Proceedings of the 12^{th} USENIX Security Symposium* (Aug. 2003), SSYM '03.

[35] SHARIF, M. I., LEE, W., CUI, W., AND LANZI, A. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16^{th} ACM Conference on Computer and Communications Security (CCS 2009)* (Nov. 2009), pp. 477–487.

[36] SUN, X.-H., AND NI, L. M. Another view on parallel speedup. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (Nov. 1990), SC '90, pp. 324–333.

[37] WALLACE, D. Compute Node Linux: Overview, progress to date & roadmap. In *Proceedings of the 2007 Cray User Group Meeting* (May 2007), CUG'07.

[38] WILLIAMS, S., WATERMAN, A., AND PATTERSON, D. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM 52*, 4 (Apr. 2009), 65–76.

[39] WISNIEWSKI, R. W., INGLETT, T., KEPPEL, P., MURTY, R., AND RIESEN, R. mOS: An architecture for extreme-scale operating systems. In *Proceedings of the 4^{th} International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2014)* (June 2014), pp. 2:1–2:8.