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 control 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 model. We present an analysis of the model for a diverse set of benchmarks, as well as a prototype tool to project multi-OS speedups for applications on existing systems. Finally, we validate our model on state-of-the-art multi-OS systems, demonstrating that it reliably predicts speedup with 96% average accuracy.

Index Terms—operating systems, multi-kernels, speedup models, performance modeling

1 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 [1], [2], system interference and jitter from "OS noise" [3], [4], or unnecessary complexity introduced by a general-purpose OS design [5].

For decades the HPC community has considered—and in several cases deployed in production [6]–[10]—*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 [11]–[13]. Unikernels (and their intellectual predecessor, Exokernels [14], [15]), take advantage of this fact to provide an OS tailored to a specific application. The tailoring process occurs primarily with two design strategies; starting from scratch with a "clean slate" design [1], [2], [12], [16], [17] or starting with an existing kernel and paring it down to a minimal state. The latter approach, termed "rump kernels," [18]–[21], shares many similarities with the light-weight kernel design philosophy from the HPC OS community. 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 [22]. 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 [1], [13], [22], [23] 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 or co-kernel [24] setup, partitions the machine (either virtually [2], [25], [26] or physically [8], [24], [27]–[29]) such that different OSes control different resources. Usually this means that a GPOS (such as Linux) acts as the *control plane*—

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

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

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 2. As hardware designers employ increasing levels of specialization [30], 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 increasingly important. Underlining this importance, since our original publication of this work, a real multi-kernel system (IHK/McKernel [29]) has been deployed on the world's currently top-ranked supercomputer system: the Fugaku machine at RIKEN.

While others have presented empirical analysis of delegation [31], and several multi-OS designs exist [24], [28], [29], [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 [33] and its successors [34], [35] provided keen insight on the limitations of parallel program performance at a time when parallel systems and algorithms were emerging. Roofline models [36] 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 with a multi-OS emulator which we hope will give insight into the limitations of program performance in multi-OS environments without users having to port their application. In Section 3, we present and analyze a naïve yet intuitive model to represent application speedup in a multi-OS setup and subsequently refine it in Section 3.2 to present a more accurate picture of reality. We use our model to project the performance of real-world benchmarks on multi-OS systems using our multi-OS emulator (`mktrace`) in Section 4. We validate our model using real multi-OS systems in Section 5, discuss its limitations, potential uses, and further refinements in Section 6, and draw conclusions in Section 8.

1.1 Extension of Prior Work

This paper is an extension to work previously published in the proceedings of MASCOTS 2019 [37]. Since that paper's publication, we have made several research contributions and gained new insights which we summarize here.

New Contributions: Our prior paper did not include an experimental validation of our models on real multi-kernel systems. We added that here, using IHK/McKernel [29] and Intel mOS [28], both of which have been deployed in production (the former on the world's top supercomputer). We made significant improvements to `mktrace`, our multi-OS emulator, including increasing its accuracy, its efficiency, and correctness. We also added more representative HPC benchmarks to further evaluate our model, namely LAMMPS (molecular dynamics), CCS QCD (a lattice QCD miniapp), and the NAS benchmark suite [38]. We refined our measurement framework to produce more accurate results, and now give several examples of model parameters that others can use to estimate multi-OS speedup. While there have been minor changes and enhancements throughout the paper, Sections 3, 4, and 6 have undergone major changes, and Section 5 is entirely new.

New Insights: We have gleaned several new insights with our extended work. First, we discovered that our prior assumptions about the cost of system call delegation being fixed can result in significant model inaccuracies. We address that in this paper. We see that real implementations have significant differences in delegation mechanisms, leading to surprisingly divergent performance characteristics. We found that many scientific applications, being compute- and memory-intensive, are quite resilient to delegation on real systems, even when a large portion of the system call interface is delegated. This aligns with experimental results from others [31]. Since our original paper, we found that for many scientific workloads, our simpler model suffices and the model parameters we use in the paper can be applied for other applications.

2 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: which calls to forward; and *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 [31].

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 [29], Pisces [24], FusedOS [32], and mOS [28] leverage this feature. In this case, because the two OS kernels run on the same node, delegation can occur between an SOS and the GPOS using shared memory. If the SOS and GPOS are running on separate nodes, however, delegation must occur over the network, which involves marshalling arguments

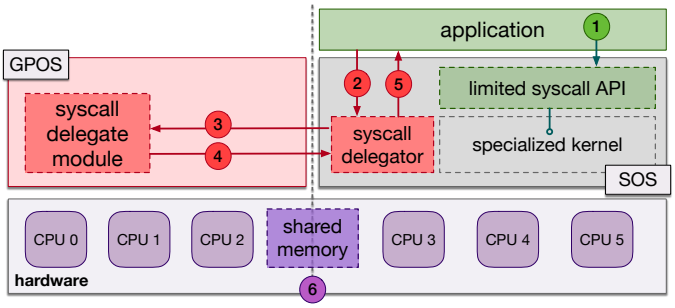


Fig. 1: High-level architecture of a multi-OS system leveraging the delegate model.

and initiating remote procedure calls (RPC) between nodes, unless the system supports distributed shared memory. Remote delegation is necessary for the I/O offloading example mentioned previously. In cases where the machine is partitioned using virtualization, as in Libra [26] and Hybrid Virtual Machines [2], 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 (6). 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 [28]. 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.

3 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 6) whose computation portion runs in a specialized (compute plane) OS, and whose system portion (namely, system calls) runs on a general-purpose (data plane) OS. Thus, *all* system calls are initially assumed to be delegated to the GPOS.

3.1 Naïve Model

We begin by outlining the simplest and most 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:

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

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 user space than in kernel space. In this case, the application receives a significant benefit[4] from the SOS, and the overall speedup equation 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 $\frac{1}{p}$.

This succinctly captures the bounded speedup of the multi-OS environment, and echoes the insight provided by

3. This would include transparent system events such as page fault exceptions and interrupt handling.

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.

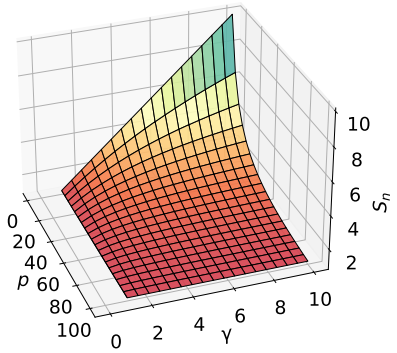


Fig. 2: Speedup in a multi-OS environment (S_n) given the proportion of time an application spends on system calls (p) and the gain factor (γ) from running in the specialized OS.

TABLE 1: Empirically determined system call portion (p) for SPEC and NAS benchmarks (class C).

Benchmark	Description	p
`gcc_linux_k`	Linux kernel compilation	35.05%
`bzip2`	bzip2	3.74%
`cam4_r`	Atmospheric modeling	0.107%
`deepsjeng_r`	Deep Sjeng chess engine (tree search)	0.0054%
`mcf_r`	Combinatorial optimization	0.00264%
`BT`	Block Tri-diagonal solver	0.000067%
`CG`	Conjugate Gradient	0.00026%
`EP`	Embarrassingly Parallel	0.000152%

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 < 2), but the interplay between γ and p are still significant for application performance. We show speedups on the order of 10× here to show the general behavior of the model.

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 of course echoes the oft-stated design 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 both from the SPEC CPU 2017 and NAS Serial suite, and projected real application speedup given a fixed gain (γ) factor.

To determine p empirically, we used `strace`[5] and the `time` command for a reference run of the individual benchmark.[6] We calculate p by summing the system call times (measured with `strace`) spent in the application and computing its ratio to the total execution time (measured with `time`).

5. `strace -c -f -w -D`

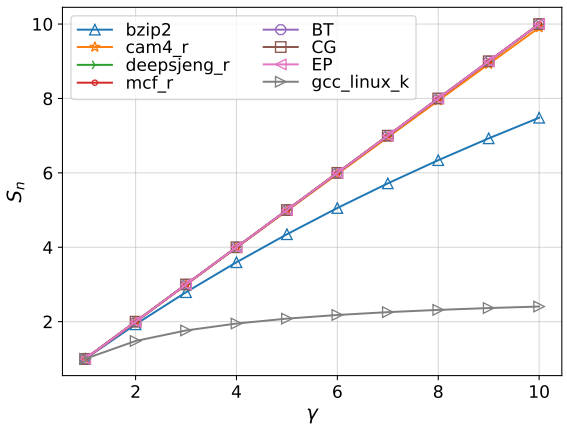


Fig. 3: Speedup projected by our simple model for SPEC and NAS benchmarks with varying gain factor (γ).

Furthermore, p may vary for a particular benchmark when its inputs are changed. Table 1 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 skewed towards computation, and thus achieve sub-linear speedup as γ increases.

3.2 Refined 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

6. Note that in the prior version of this paper we used a simplified measurement scheme, which resulted in overestimation of the system call portion.

stores, but integer/floating point instructions will be unaffected unless they involve memory references.[7]

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[8] when run on a general-purpose OS. This function also includes time spent in the kernel due to, e.g., blocking system calls.

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 vary widely depending on the software mechanisms which implement forwarding and the underlying interconnect over which system calls are forwarded.[9] For example, for delegation over shared memory (Section 2), f_c would likely be in the ns to μs range For delegation over a network, this number might be closer to several μs or several ms, depending on the network characteristics. Note that delegation over the network may involve more complex and asymmetric forwarding costs. For example, the forward trip may involve marshalling system call arguments, and the return trip from the GPOS might only involve a single integer return value. We do not currently take these complexities into account in our model.

The second function, $b(s)$ is a boolean predicate function[10] which 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 $g(s) = g'(s)$, so that both implementations take the same amount of time.

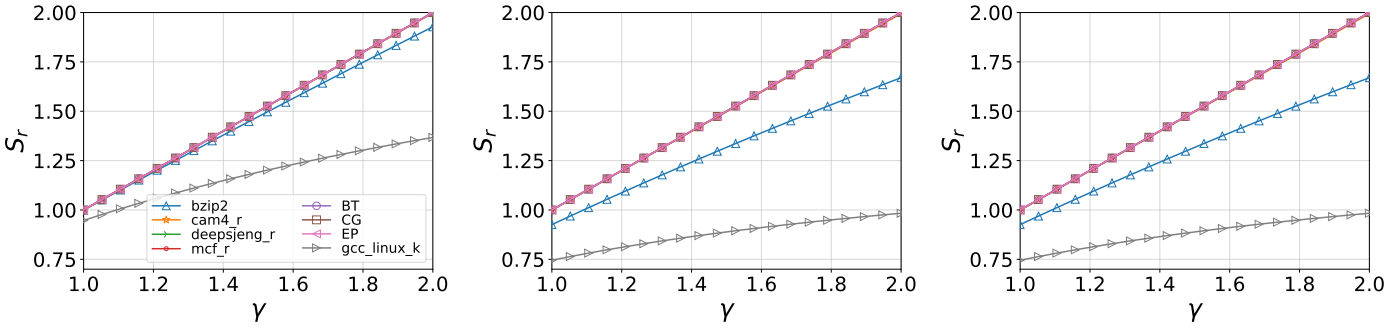
Figure 4 illustrates speedup projections (represented by S_r) using our refined model for a subset of the benchmarks

7. This is unless, of course, the instructions cause an exception or involve addressing modes that necessitate a memory reference.

8. That is, the sum of the execution times for all `mmap()` invocations.

9. 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.

10. We could also refer to this as a characteristic function or an indicator function.



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

Fig. 4: Projected speedup when the gain (γ) varies for SPEC CPU 2017/NAS benchmarks. This assumes a fixed forwarding cost (f_c) of 10 μs and a fixed percentage (90%) 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).

shown in Table 1. We initially fix the forwarding cost, f_c, at 10 μs. This is representative of a scenario where forwarding occurs over shared memory. We vary γ to illustrate the effects of running the application in the SOS.

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 90% of system calls are forwarded. This proportion reflects what we have seen on mOS, a real multi-kernel system. In each graph we vary *which* system calls constitute that fixed proportion. 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. Note that in IHK/McKernel, another multi-kernel system we studied, roughly 30% of system calls are forwarded. If that proportion were used here, we would not see much effect on projected speedup since most of these calls are infrequently invoked (see Figure 6 and related discussion). Figure 4b shows the speedup when we choose system calls which are invoked *most frequently*. This is not a forwarding policy one should choose, but is shown here for comparison. Finally, Figure 4c depicts the results when we make a random choice. Note how different benchmarks are affected differently by the forwarding schemes. Both SPEC CPU and NAS benchmarks achieve a high speedup no matter the scheme. This is because of the generally low system call activity in these benchmarks (as shown in Table 1). However, `bzip2` shows significant 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. In this experiment we traced all system calls invoked by each benchmark, and counted the number of invocations for each individual system call. We report these counts using a CDF. A point on this figure thus represents the percentage of system calls that have been invoked fewer times than the value on the horizontal axis. Looking at the CDF for `bzip2`, it is clear that its speedup is curtailed because it uses a small set of system calls 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 profiles with more statistical structure will be more amenable to selective forwarding schemes. This aligns with intuition and prior experimental results from Gioiosa et al. [31].

In Figure 5, we choose three benchmarks `bzip2` (skewed system call distribution), Linux kernel compilation with `gcc` (uniform distribution of system calls) and one from the NAS suite, `BT` class C (compute intensive with very low system call activity), and show with surface plots how their projected speedups change as we vary both the forwarding cost (f_c) and the gain factor from execution in the SOS. Here we forward 90% of system calls (those invoked most *infrequently*). Note again 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. The `BT` benchmark achieves the highest speedup, both because it has a small system call portion, and because that portion involves very few system calls that are invoked in general. 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, resulting in a flat surface. For `gcc` as the forwarding cost increases, we observe negative speedup (i.e., GPOS performs better than SOS), when speedup values go below 1 we trim the values in Figure 5 for better visualization.

Forwarding cost only becomes a significant factor in the `bzip2` and `BT` cases when it becomes greater than tens of milliseconds. This is more tolerance to forwarding overheads than we expected, and indicates that in many

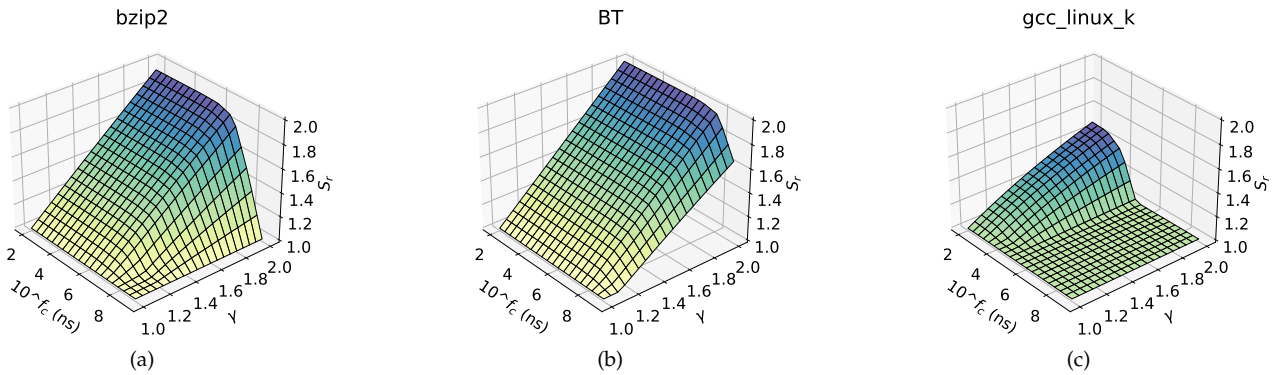


Fig. 5: Projected speedup as the gain (γ) and forwarding cost (f_c) are varied for SPEC benchmarks. The proportion of forwarded system calls is fixed at 90%, and which calls to forward is determined according to those least frequently invoked (min. frequency). γ is fixed at 2.

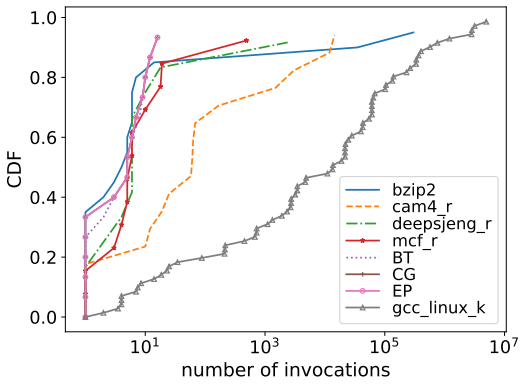


Fig. 6: System call profile for selected benchmarks. Note the log scale on horizontal axis.

practical cases our naïve model may be sufficient.

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 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 when a relatively large portion of the system call interface is forwarded. The second observation is that when these workloads are *not* properly accounted for (i.e. when the *wrong* calls are forwarded), the performance degradation is dramatic, as 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 surfaces has a steeper drop-offs when an increasing number of system calls are forwarded, indicating a heavy skew in the system call distribution (cf. Figure 6). The smoother "rolling hill" of the BT benchmark indicates a small amount of syscall activity, and the more pronounced drop-off of `gcc` indicates higher, but more uniform syscall activity.

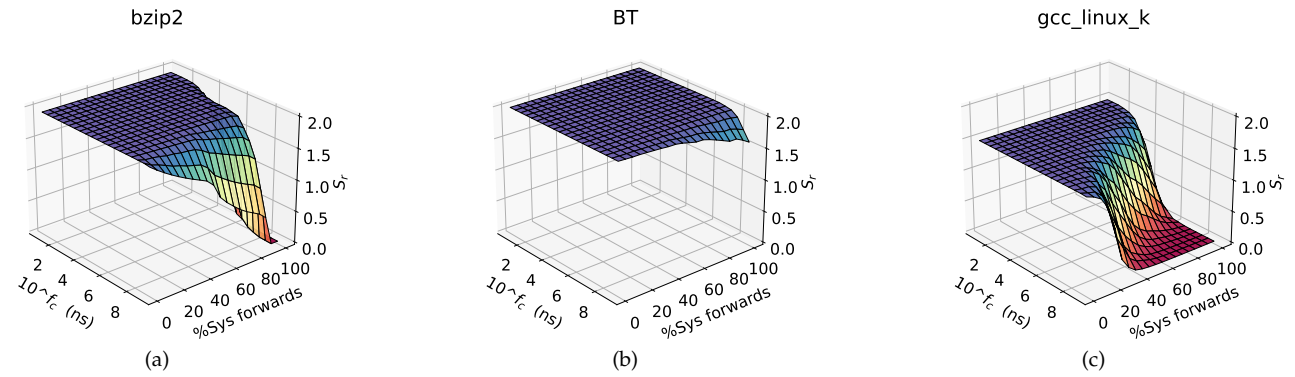
Figure 7 shows us that compute-intensive benchmarks (like `BT`) are most amenable to multi-kernel environments, since they are mostly unaffected by forwarding overheads. In Figure 8, we vary the gain and the proportion of forwarded calls. The interesting point here is that with the fixed forwarding cost 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 `BT` 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`, `deepsjeng_r` and `bzip2` have steep drop-offs 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 several hundreds of milliseconds before making a significant impact.

4 Approximate Speedup Projections

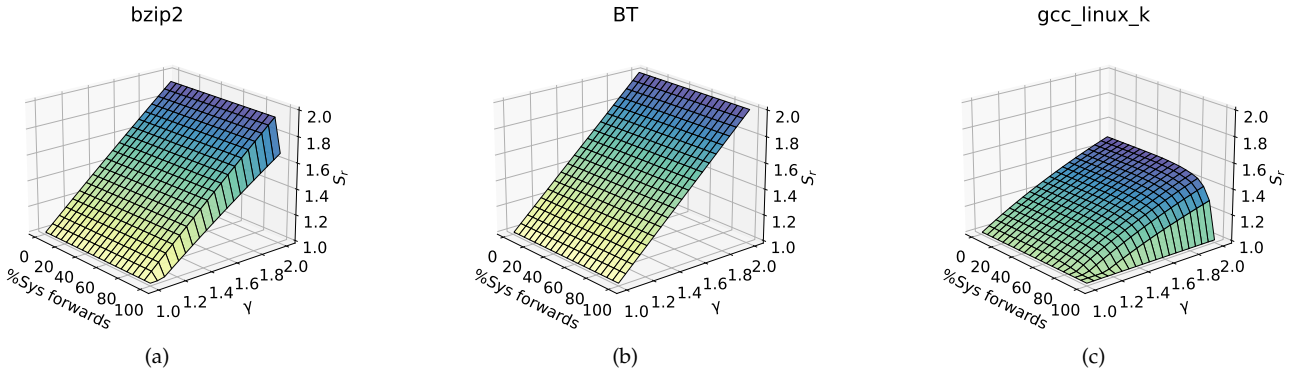
The primary obstacle in measuring application performance directly in a multi-OS environment is the OS development burden required to implement functionality in the SOS. The simplest case is when the *entire* system interface is delegated to a GPOS. In this case, the application benefits solely from the properties of the execution environment provided by the SOS (e.g., simplified, deterministic paging and fine-grained interrupt control), 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 run an application directly on a multi-OS system. In this section, we describe our multi-OS emulator, called `mktrace`, that enables this projection, allowing users interested in multi-OS setups to perform a kind of "what-if" analysis. Users can run their unmodified programs using our tool to project performance in a multi-OS setup without investing in a porting effort. We believe our tool can provide key insights to developers, in particular those who have no prior experience with multi-OS systems.

Note that this tool does not actually leverage two separate operating systems. Instead, we leverage a Linux kernel



(a) (b) (c)

Fig. 7: Projected speedup as the forwarding cost (f_c) and the proportion of forwarded calls vary, with gain (γ) fixed at 2.



(a) (b) (c)

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

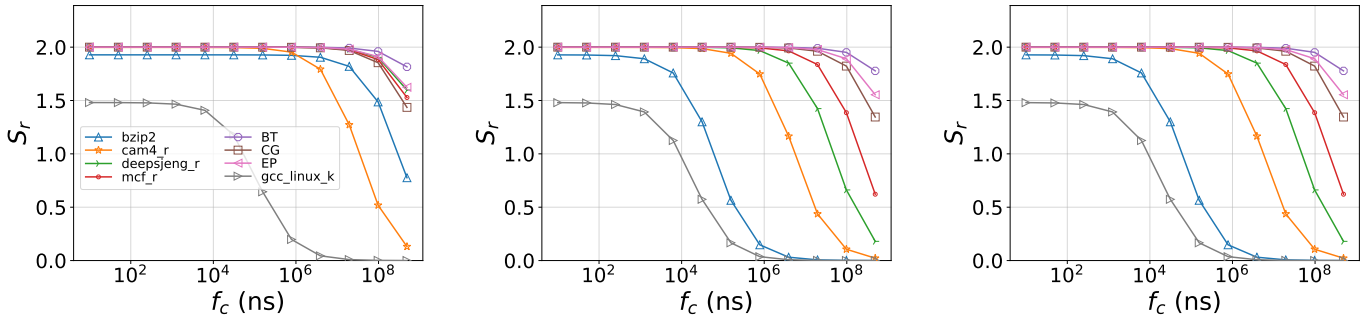
module that employs a kernel thread on the *same OS* to serve as a delegate (standing in for the control-plane OS); this delegate fields system call requests from a running process. However, the delegation mechanism in `mktrace` is similar to real multi-OS systems. For example, in the case of IHK/McKernel [29], for every process running on light weight kernel (LWK), a proxy process is created on the Linux side to handle delegated system calls. At a high level, the proxy process acts in a similar capacity to our service threads in `mktrace`. One difference is that a proxy process in a real multi-kernel setup runs in user-space, which requires an additional context switch to handle delegated system calls. 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 emulator tool, called `mktrace`, is freely available online,[11] 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 [39]–[42] and out-of-kernel by leveraging an underlying virtual machine monitor (VMM) [43]–[45]. 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 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 of the system call execution on the delegate kernel thread are sent back to the calling process and execution continues normally.

4.1 Experimental Setup

We conducted our experiments on a system called *tinker*, which has a 2.2GHz Xeon E5-2630 CPU with 10 cores and 48 GB RAM. It runs Centos 7.7 with stock Linux kernel

11. https://github.com/hexsa-lab/mktrace



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

Fig. 9: Projected speedup as the forwarding cost (f_c) varies. We fix the gain factor (γ) to 2 and assume that 90% of system calls are forwarded.

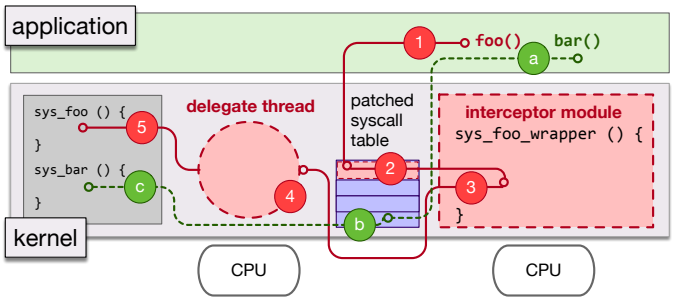


Fig. 10: High-level overview of `mktrace`.

version 3.10.0-1062. Hyperthreading is disabled, and the BIOS is configured for a static power profile (maximum performance) to mitigate measurement noise from DVFS.

For these experiments, we selected several benchmarks from the initial set: `bzip2`, Linux kernel compilation (kernel version 5.1.4) with `gcc`, the SPEC CPU suite, and the NAS benchmarks (currently the serial versions).

In this section we are not actually running applications in a real multi-OS system (cf. Section 5), so 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, 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. The benchmark uses a synthetic system call profile derived from traces gathered from `bzip2` (skewed system call profile, I/O intensive) and `BT` (uniform distribution of system calls, compute intensive) using `strace`.

4.2 Experiments

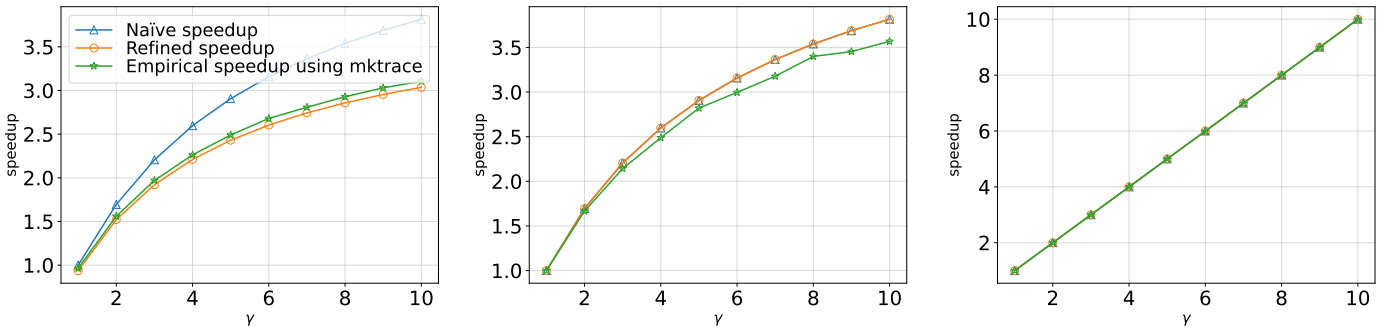
We first run the benchmarks described above in a standard Linux environment and then using our `mktrace` tool, which approximates delegation. We measured the minimum overhead of forwarding using this mechanism (represented by f_c in previous sections) to be 6.3 μs, measured with 1000 trials. This is quite close to the forwarding costs incurred by real multi-kernels, as we will see in Section 5. In all cases, because of the small forwarding overhead, there is very little impact on performance. The largest overhead we observed for `mktrace` was less than 1% across all benchmarks when using the "min. frequency" syscall profile.

As described above, we now attempt to incorporate the gain factor (γ) by approximating gain using a variable amount of computation. Figure 11 shows the projected speedup from both the naïve model and the refined model for the synthetic benchmark compared to the speedup empirically determined with `mktrace`. In this figure, we see how the measured speedup of the benchmark changes with a varying gain factor. The curve labeled "empirical speedup using mktrace" represents speedup relative to the default setup on Linux without any system call delegation.

In Figure 11a we delegate >75% (similar to mOS) system calls with a system call profile derived from a `bzip2` trace. Here the refined model matches the measured speedup closely. In Figure 11b we only delegate 30% of system calls (similar to IHK/McKernel). We see that for higher γ values the gap between refined speedup and measured speedup increases. In Figure 11c, the system call profile is derived from BT, a compute-intensive benchmark. The predicted speedup from both models converge in this case because forwarding costs are negligible.

The small gap we see between the predicted speedup and the measured speedup in Figure 11b we discovered was due to the fixed forwarding cost assumption our model makes. Depending on how the delegation mechanism is implemented, there is actually some non-determinism in this cost (for example due to queueing and thread wake-up latencies).

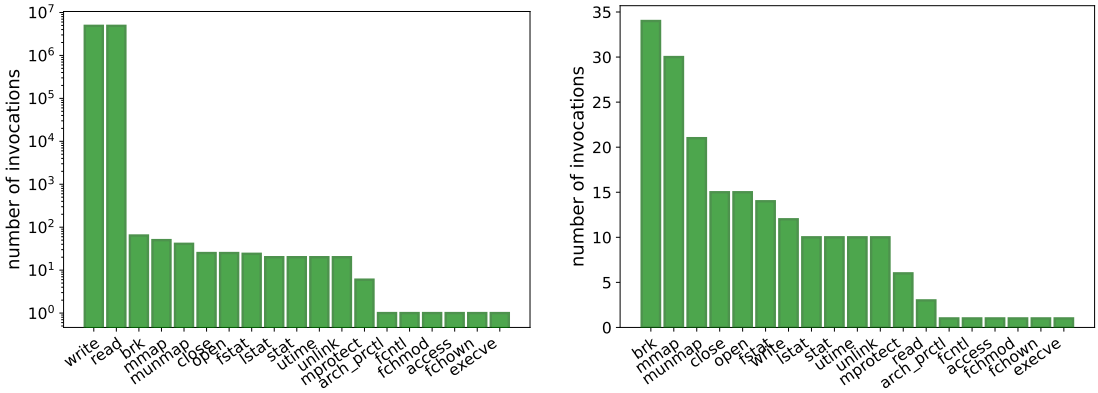
In order to understand this further, we run the synthetic benchmark (with the bzip2 system call profile) with a forwarding cost derived using three summary statistics: mean, min, and max. The forwarding cost is estimated by taking the time difference between a system call running with and without `mktrace`. Figure 13 shows us that using the minimum measured forwarding cost gives us the best speedup prediction, suggesting a skewed distribution. This



(a) bzip2 profile; 90% forwarded (b) bzip2 profile; 30% forwarded (c) BT profile; 90% forwarded

Fig. 11: Projected speedup when the gain (γ) varies for the synthetic benchmark.

.



(a) bzip2 (b) BT

Fig. 12: System call profiles for bzip2 and BT, derived with `strace`. Note the log scale on vertical axis of the first figure.

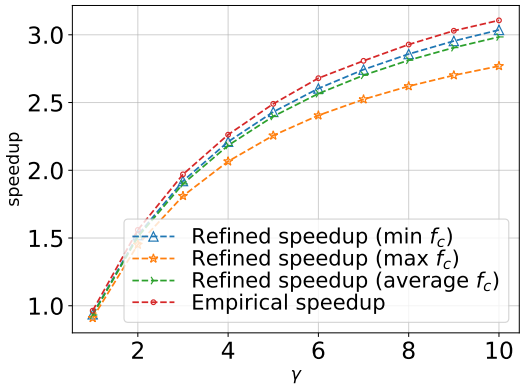


Fig. 13: Model sensitivity to different summary statistics for derived forwarding cost (f_c).

is not too surprising, since very rarely kernel events like interrupts and context switches will inflate the forwarding cost.

5 Experimental Validation

In this section, we validate our model by comparing its speedup projections with actual execution time on real multi-OS systems. We study two multi-kernel systems that have been deployed in production and that have quite different designs: IHK/McKernel[12] from RIKEN and Intel's mOS.[13] We configure both mOS[14] and IHK/McKernel[15] such that one core (the LWK core) and 15GB RAM are dedicated to the SOS. We select 15 GB as this was the maximum memory we could allocate for a single core in IHK/McKernel on our *tinker* testbed.

5.1 IHK/McKernel

IHK/McKernel runs HPC applications on a light-weight kernel (LWK) to achieve scalable execution [29], but notably the complete Linux API is available via system call delegation. McKernel acts as the LWK and is primarily designed for HPC; it is launched from IHK, the shim on the Linux side. McKernel retains a binary compatible ABI with Linux and it implements only a small set of performance-sensitive system calls, delegating the remainder to Linux. For every process running on McKernel there is a process spawned on Linux called the *proxy process*. The proxy process facilitates

12. commit hash 62153.
13. mOS v0.7.
14. `lwkctl -c lwkcpus=0.1 lwkmem=15g`
15. `mcreboot.sh -t -c 1 -m 15`

system call delegation. It provides an execution context on behalf of the application so that delegated calls can be directly invoked in Linux. The list of system calls handled by IHK/McKernel, can be found in a kernel header.[16] We use this information in our refined model to estimate the forwarding cost for delegated system calls.

5.2 mOS

mOS uses a different design [28]. While the fundamental concepts of mOS remain similar, mOS incorporates the LWK code directly in a Linux kernel image. The mOS system call delegation mechanism is quite different from the proxy process approach. mOS retains Linux kernel compatibility at the level of its internal kernel data structures, which enables it to migrate threads directly into Linux. mOS implements system call delegation by migrating the issuer thread into Linux, executing the system call, and migrating the thread back to the LWK component. The list of system calls handled by mOS, can be found in a kernel header.[17]

5.3 Experiments

To validate our models experimentally, we selected benchmarks from the SPEC CPU 2017 [46], NAS Class C [38], and LAMMPS [47] benchmark suites, in addition to the CCS QCD miniapp (Class 1) from RIKEN's FiBER Miniapp Suite [48]. We only use benchmarks that were able to run on both multi-OS systems. These are listed in Table 2.

We compare the performance of the benchmarks running purely on Linux with multi-OS performance, for both mOS and IHK/McKernel. Both mOS and IHK/McKernel are designed to improve the performance of large-scale parallel applications, not sequential benchmarks. Since our model currently only captures sequential setups, we are not looking for a performance improvement on the benchmarks relative to Linux. Rather, we are looking to validate the projections of our model using real systems. We direct readers interested in multi-OS performance benefits and scaling studies to work by Gerofi et al. [49].

We measure the absolute execution time of the benchmarks on the various platforms for ten trials and report the medians in Figure 14. We can see that for most applications, IHK/McKernel and mOS have similar performance to Linux, which again is not surprising given that these benchmarks run sequentially on a single core. We observe that I/O-intensive applications such as `bzip2`, which involves frequent system calls, has an advantage on Linux since the multi-OS systems suffer from forwarding costs on system call-intensive workloads.

Our goal here is to compare these execution times to our models' predictions. In general, one seeking to use our models could derive estimates for model parameters (γ, f_c, and the syscall profile) with microbenchmarking, profiling, or by using reported performance numbers from relevant multi-OS papers or developers. We provide example parameters here. To do so, we use both microbenchmarking and profiling with the benchmark suites to arrive at suitable parameters, which could be used by others evaluating speedup for similar workloads.

16. See `syscall_list.h`, available at https://github.com/RIKEN-SysSoft/mckernel

17. See `include/linux/syscalls.h`, available at https://github.com/intel/mOS

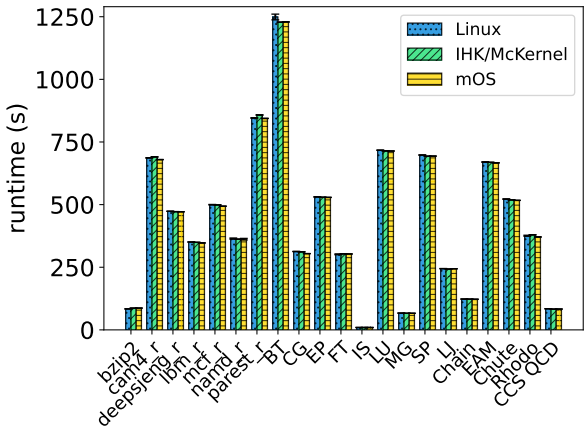


Fig. 14: Runtime of selected SPEC, NAS Class C, LAMMPS, and CCS QCD benchmarks on Linux, IHK/McKernel, and mOS (lower is better).

We calculate the forwarding cost parameter (f_c) for IHK/McKernel and mOS separately by running a delegated system call natively on Linux and subtracting its execution time from the time it takes to run the same system call in the relevant multi-OS system. We take the minimum measured value over 1000 trials. Table 3 shows the results (measured in μs) of four delegated system calls. Based on these results, we set the round-trip forwarding cost ($2f_c$) to the smallest measurement observed for each system (6.28μs on McKernel and 6.46μs on mOS). Note that our measurement tool invokes these system calls *directly* using assembly wrappers and the `syscall` instruction to avoid including the costs of user-space system call code (i.e., the syscall wrappers in `libc`).

For the gain parameter (γ), we first determine a *benchmark-specific* gain factor for each of the benchmarks in our suites, then aggregate these using a summary statistic which we will use for the actual γ parameter in the model. For brevity, we only show results using the median in this paper. To determine a *benchmark-specific* γ value, we plugged in the forwarding cost and the overall speedup measured from benchmark runs into our refined model and solved for γ for each benchmark. Table 4 shows our derived, benchmark-specific γ values. For IHK/McKernel most benchmarks have γ in the range of ~0.96–1.1. For mOS, the gain factor ranges between ~0.98 and 1.1. If we exclude `bzip2` (not representative for HPC workloads), γ ranges from ~0.99–1.1 for both multi-OS systems. This is in line with what we expect from single-core, compute-intensive benchmarks.

Figure 15 compares our model predictions with measured speedups. Figure 15a shows the SPEC benchmarks, Figure 15b shows the NAS benchmark suite, and Figure 15c shows the LAMMPS and CCS QCD benchmarks. The left halves of the figures show projections and measurements for IHK/McKernel, and the right side depicts mOS. We show speedup projections from our naïve model (every syscall

TABLE 2: Selected benchmarks and parameters from the SPEC CPU 2017, NAS Class C, LAMMPS, and CCS QCD benchmarks for model validation.

Benchmark	Description	Invocation	Extra Parameters
bzip2	File compression	bzip2 bigfile.txt	-
cam4_r	Atmospheric modeling	./cam4_r	-
deepsjeng_r	Deep Sjeng chess engine (tree search)	./deepsjeng_r ref.txt	-
lbm_r	Fluid dynamics	./lbm_r 3000 reference.dat 0 0 100_100_130_ldc.of	-
mcf_r	Combinatorial optimization	./mcf_r_base.sys_call_fw_1-m64 inp.in	-
namd_r	Molecular dynamics	./namd_r --input apoa1.input --output apoa1.ref.output --iterations 65	-
parest_r	Optical tomography with finite elements	./parest_r ref.prm	-
BT	Non-linear PDE solver (using block tri-diagonal)	./bt.C	-
CG	Estimates minimal Eigenvalue of a sparse matrix	./cg.C	-
EP	Generates independent Gaussian deviates	./ep.C	-
FT	Solves a 3D PDE using fast Fourier transform	./ft.C	-
IS	Sorts small integers using bucket sort	./is.C	-
LU	Non-linear PDE solver (using Gauss-Seidel)	./lu.C	-
MG	Multi-grid on a sequence of meshes	./mg.C	-
SP	Non-linear PDE solver (using scalar penta-diagonal)	./sp.C	-
LJ	Lennard-Jones atomic fluid simulation	./lmp_serial -in in.lj	10^4 time steps
Chain	Simulation of bead-spring polymer chain melt	./lmp_serial -in in.chain	10^4 time steps
EAM	Simulation of Cu metallic solid using EAM method	./lmp_serial -in in.eam	10^4 time steps
Chute	Chute flow simulation for packed granular particles	./lmp_serial -in in.chute	10^5 time steps
Rhodo	Simulation of Rhodopsin protein	./lmp_serial -in in.rhodo	10^3 time steps
CCS QCD	Lattice quantum chromodynamics miniapp	./ccs_qcd_solver_bench_class1	tolerance=0, kappa=0.124d0, Lws=1.0d0

TABLE 3: Measured round-trip forwarding costs for delegated system calls in IHK/McKernel and mOS.

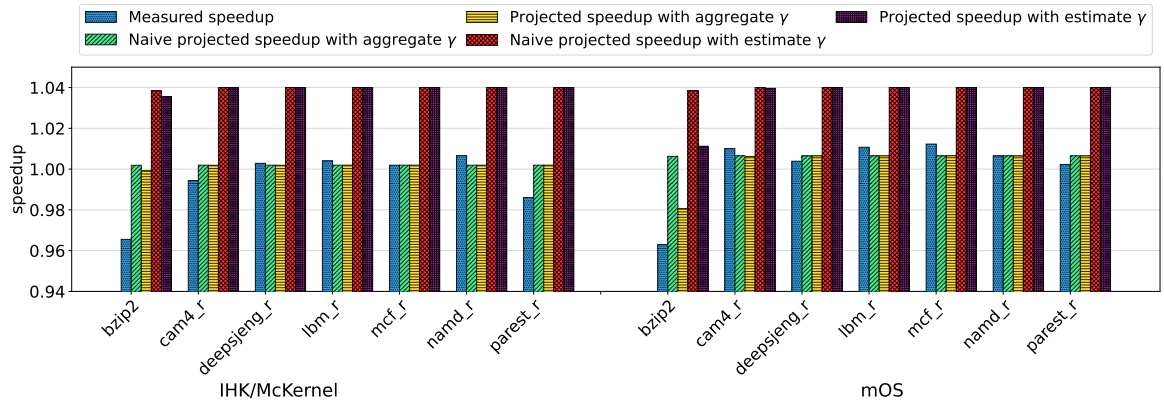
Benchmark	Measured $2f_c$ (min., in μs)	
	IHK/McKernel	mOS
write_to_file	6.28	7.56
fstat	7.05	7.06
uname	8.05	7.60
write_to_console	10.36	6.46

Benchmark	Derived γ		Syscalls Forwarded	
	IHK/McKernel	mOS	IHK/McKernel	mOS
bzip2	0.96	0.98	25%	90%
cam4_r	0.99	1.01	35%	88%
deepsjeng_r	1.00	1.00	25%	83%
lbm_r	1.00	1.01	31%	85%
mcf_r	1.00	1.01	23%	77%
namd_r	1.01	1.00	25%	83%
parest_r	0.98	1.00	36%	79%
BT	1.01	1.01	33%	87%
CG	1.00	1.02	33%	87%
EP	1.00	1.00	33%	87%
FT	0.99	0.99	33%	87%
IS	1.00	1.00	25%	83%
LU	1.00	1.00	33%	87%
MG	1.01	1.01	33%	87%
SP	1.00	1.00	33%	87%
LJ	1.00	1.00	27%	80%
Chain	1.00	1.00	27%	80%
EAM	1.00	1.00	31%	81%
Chute	1.00	1.01	27%	80%
Rhodo	0.99	1.01	27%	80%
CCS QCD	1.01	1.01	21%	91%

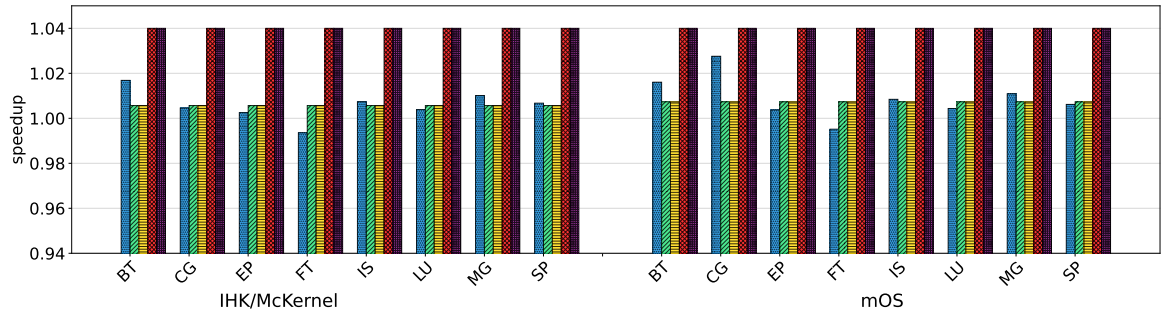
TABLE 4: Derived γ values and percentage of system calls forwarded for both multi-kernels.

delegated) and our refined model. For both, we use two methods to supply a γ parameter to our model. In the first, we use an estimated gain factor by using empirical measurements from others. In particular, we referred to speedup measurements from Gerofi et al. [49], where the improvements from running in a multi-OS system vary from 4% to 280%. We pick the lowest of these (4% improvement observed in LAMMPS, HACC, and QBOX benchmarks on a single node) as a conservative estimate for γ, as the larger improvements arise from these systems running at scale. We also show projections using a γ value computed as an aggregate (median) of the benchmark-specific values reported in Table 4.

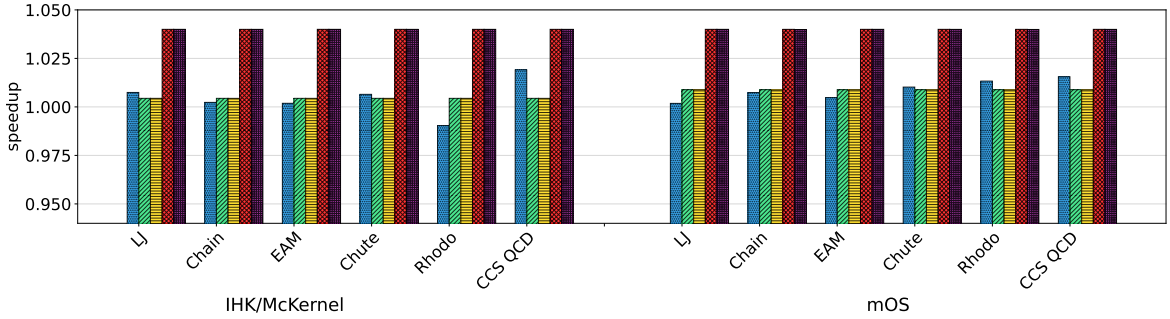
In Table 4 we also report the percentage of system calls delegated in IHK/McKernel and mOS; we observe that IHK/McKernel delegates a small set of system calls while mOS delegates >70% system calls. In Figures 15a and 15b the projected speedup values using an aggregate γ parameter are closest to the mark. This is unsurprising, since this value was derived from the same benchmarks whose speedup is being projected here. The naïve model and the refined model project similar speedups for most benchmarks, since on the whole they invoke system calls infrequently. To determine the accuracy of our models, we compute the mean absolute percentage error (MAPE) of its predictions relative to the measured overall speedups on mOS and IHK/McKernel. The refined model achieves 99.30% accuracy and 99.43% accuracy for IHK/McKernel and mOS, respectively (using the median aggregate γ value). Using parameter estimates from prior work, the refined model achieves 96.18% and 96.79% accuracy, respectively. In practice, a user of this model may not be able to directly determine γ, so the estimated parameters represent a more realistic scenario. One thing to note here is that, provided a fairly close estimate of γ is given, our



(a) SPEC CPU benchmarks and bzip2.



(b) NAS benchmarks (class C).



(c) LAMMPS and CCS QCD benchmarks.

Fig. 15: Empirical speedup on IHK/McKernel (left) and mOS (right) compared to model predictions. We use an aggregate γ value of 1.003 for IHK/McKernel and 1.007 for mOS. We use an estimated γ of 1.04 for both multi-OS systems.

naïve model will likely suffice to quickly project speedup for compute- and memory-intensive workloads. It can thus be helpful in guiding developers in deciding whether or not to use a multi-OS system.

6 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. The models can guide development efforts when designing a custom OS for a multi-OS environment. Notably in the case of compute intensive benchmarks, with forwarding costs up to the microsecond range, the naïve model matches the refined model quite well, indicating that it would be sufficient for single-node setups, as well as multi-node setups with low-latency interconnects. Thus for the single node case, our intuitive model is both simple and accurate. While we determine the parameters for our model in this paper using manual experiments, a good estimate for them would suffice for coarse speedup projections, a likely use case for our models. Parameters could also be determined automatically by profiling application code or by running a suite of microbenchmarks.

There remain several limitations with these models, however. The primary limitation is that they 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 (application threads) or several machines. In this case, a single delegate (GPOS) thread servicing these requests would be inadequate unless the workloads collec-

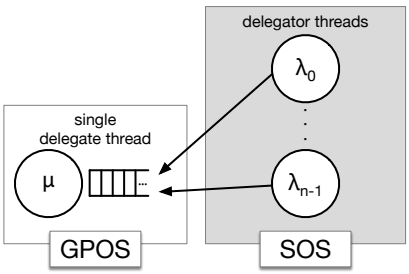


Fig. 16: Simple $M/M/1$ queueing model for n delegator threads and a single delegate thread.

tively involve few system calls, thus minimizing queueing delays. As we have seen, this is actually not an unreasonable expectation for HPC applications, so an enhancement involving simple queueing models is possible. For example, Figure 16 shows a single GPOS delegate thread modeled as an $M/M/1$ queue (Poisson arrivals and exponential service times). Here we assume each of n application threads makes system call delegation requests following a distinct Poisson arrival process, λ_k. By the merging property, these arrivals sum as follows.

$$\lambda_{total} = \sum_{k=0}^{n-1} \lambda_k \tag{12}$$

The delegate thread handles system call requests following exponential service times, with average service rate μ. The λ_k and μ parameters can be easily determined with application traces. Using Little's Law we can then determine the average queueing delay W for system call delegation requests (subtracting out service times):

$$W = \frac{1}{\mu - \lambda_{total}} - \frac{1}{\mu} \tag{13}$$

We can then estimate speedup by incorporating this into our refined model in Equation 11, adding $W \times n(s)$ to the system call service times $g(s)$, where $n(s)$ is the number of invocations of syscall s and W is the average queueing delay determined above. While a careful analysis of such queueing models is outside the scope of this paper, it is worth discussing their limitations. In workload scenarios that require more than one delegate thread, an initial treatment might extend the model to an $M/M/c$ (multiple server) model. However, the primary issue here is that treating the service rate as a memoryless process ignores confounding performance factors caused by concurrent request streams. Concurrent system call requests coming from two delegator threads may actually both touch the same shared state in the GPOS. For example, in Linux, two concurrent `mmap()` calls will both mutate the region tree in the parent process's task struct. The locking overheads caused by such shared accesses will not be included in a simple queueing model, so it would likely overestimate speedup. Similarly, such a model would not account for low-level hardware overheads sensitive to thread placement, e.g., those due cache coherence traffic and NUMA. We plan to explore more sophisticated queueing models in future work, building on prior work from Pan et al. on analytical models for lock contention and cache performance [50]–[52].

The fixed forwarding cost assumptions we make are also a limitation. This assumption will amplify inaccuracies when we move to multi-node systems.

7 Related Work

As far as we are aware, we are the first to model speedup analytically for multi-OS environments. We refer readers to Gioiosa et al. for an excellent empirical study of system call delegation [31]. Our work was inspired by Amdahl's original formulation of parallel speedup [33], Gustafson's refinement [34], and Sun and Ni's extended model for incorporating memory-bound programs [35]. 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 [53]. 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 [36], [54]. 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.

8 Conclusion

We introduced two models to guide intuition on application speedup when a program's execution is split between two 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 poor choice of forwarding policies. We showed that applications with skewed system call distributions can tolerate higher forwarding overheads when a good forwarding policy is selected, but suffer from more dramatic effects when we forward the wrong system calls. To measure the effect of these overheads on real applications, we presented an open-source tool called `mktrace` which approximates forwarding overheads on existing systems. Using this tool, we demonstrated how to make speedup projections for applications without actually deploying them on a multi-OS system. We also provide reasonable parameters for our model that others can use to project speedups. Finally, we validated our model using two real multi-OS systems, one of which is deployed on the world's top-ranked supercomputer, Fugaku. Our model achieves 96.18% and 96.79% accuracy in these settings.

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, for example using multi-server queueing models. We also plan to extend `mktrace` to support multi-node

systems. Finally, we plan on investigating other perspectives on multi-OS speedup, including roofline models.

ACKNOWLEDGMENTS

We thank Rolf Riesen and Balazs Gerofi for their valuable input, without which our mOS and McKernel experiments would not have been possible. We also thank Peter Dinda for insightful discussions, and the anonymous reviewers for their helpful feedback for improving the manuscript. This work is supported by the United States National Science Foundation (NSF) via grants CNS-1718252, CNS-1730689, CNS-1763612, CCF-2028958, and CCF-2029014.

AVAILABILITY

Our speedup projection tool, `mktrace`, is freely available online at https://github.com/hexsa-lab/mktrace. Our data and experimental scripts are also freely available.

REFERENCES

[1] K. C. Hale and P. A. Dinda, "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*, ser. HPDC '15, Jun. 2015.

[2] ——, "Enabling hybrid parallel runtimes through kernel and virtualization support," in *Proceedings of the* 12^{th} *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '16, Apr. 2016, pp. 161–175.

[3] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Proceedings of Supercomputing*, ser. SC '08, Nov. 2008.

[4] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of OS noise," in *Proceedings of the* 25^{th} *IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '11, May 2011, pp. 852–863.

[5] K. C. Hale and P. A. Dinda, "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*, ser. MASCOTS '18, Sep. 2018.

[6] S. M. Kelly and R. Brightwell, "Software architecture of the light weight kernel, Catamount," in *Proceedings of the 2005 Cray User Group Meeting*, ser. CUG'05, May 2005.

[7] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski, "Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK," in *Proceedings of Supercomputing*, ser. SC '10, Nov. 2010.

[8] A. Gara, M. A. Blumrich, D. Chen, G. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. M. Vranas, "Overview of the Blue Gene/L system architecture," *IBM Journal of Research and Development*, vol. 49, no. 2, pp. 195–212, Mar. 2005.

[9] D. Wallace, "Compute Node Linux: Overview, progress to date & roadmap," in *Proceedings of the 2007 Cray User Group Meeting*, ser. CUG'07, May 2007.

[10] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell, "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*, ser. IPDPS'10, Apr. 2010.

[11] T. E. Anderson, "The case for application-specific operating systems," in *Proceedings of the* 3^{rd} *Workshop on Workstation Operating Systems*, Apr. 1992.

[12] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "OSv—optimizing the operating system for virtual machines," in *Proceedings of the 2014 USENIX Annual Technical Conference*, ser. USENIX ATC'14, Jun. 2014.

[13] S. Peter and T. Anderson, "Arrakis: A case for the end of the empire," in *Proceedings of the* 14^{th} *Workshop on Hot Topics in Operating Systems*, ser. HotOS XIII, May 2013.

[14] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *Proceedings of the* 5^{th} *Workshop on Hot Topics in Operating Systems*, ser. HotOS V, May 1995, pp. 78–83.

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles*, ser. SOSP '95, Dec. 1995, pp. 251–266.

[16] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "IncludeOS: A minimal, resource efficient unikernel for cloud services," in *Proceedings of the* 7^{th} *IEEE International Conference on Cloud Computing Technology and Science*, ser. CloudCom '15, Nov. 2015, pp. 250–257. [Online]. Available: https://doi.org/10.1109/CloudCom.2015.89

[17] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo, "EbbRT: A framework for building per-application library operating systems," in *Proceedings of the* 12^{th} *USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '16, Oct. 2016, pp. 671–688.

[18] A. Kantee, "The design and implementation of the anykernel and rump kernels," Ph.D. dissertation, Aalto University, Helsinki, Finland, 2012.

[19] A. K. , "The rise and fall of the operating system," *USENIX ;login*, vol. 40, no. 5, pp. 6–9, Oct. 2015.

[20] A. Raza, P. Sohal, J. Cadden, J. Appavoo, U. Drepper, R. Jones, O. Krieger, R. Mancuso, and L. Woodman, "Unikernels: The next stage of linux's dominance," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS XVII, May 2019, pp. 7–13. [Online]. Available: https://doi.org/10.1145/3317550.3321445

[21] R. Nikolaev, M. Sung, and B. Ravindran, "LibrettOS: A dynamically adaptable multiserver-library OS," in *Proceedings of the* 16^{th} *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20, Mar. 2020, pp. 114–128. [Online]. Available: https://doi.org/10.1145/3381052.3381316

[22] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *Proceedings of the* 18^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'13, Mar. 2013, pp. 461–472.

[23] S. Lankes, S. Pickartz, and J. Breitbart, "HermitCore: A unikernel for extreme scale computing," in *Proceedings of the* 6^{th} *International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS'16, Jun. 2016.

[24] J. Ouyang, B. Kocoloski, J. R. Lange, and K. Pedretti, "Achieving performance isolation with lightweight co-kernels," in *Proceedings of the* 24^{th} *International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15, Jun. 2015, pp. 149–160.

[25] K. C. Hale, C. Hetland, and P. A. Dinda, "Multiverse: Easy conversion of runtime systems into os kernels via automatic hybridization," in *Proceedings of the* 14^{th} *IEEE International Conference on Autonomic Computing*, ser. ICAC'17, Jul. 2017.

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

[27] B. Kocoloski and J. Lange, "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*, ser. HPDC '15, Jun. 2015, pp. 89–100.

[28] R. W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen, "mOS: An architecture for extreme-scale operating systems," in *Proceedings of the* 4^{th} *International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '14, Jun. 2014, pp. 2:1–2:8.

[29] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa, "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*, ser. IPDPS '16, May 2016, pp. 1041–1050.

[30] J. L. Hennessy and D. A. Patterson, "A new golden age for

computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.
[31] R. Gioiosa, R. W. Wisniewski, R. Murty, and T. Inglett, "Analyzing system calls in multi-OS hierarchical environments," in *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '15, Jun. 2015.
[32] Y. Park, E. V. Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. D. Ryu, and R. W. Wisniewski, "FusedOS: Fusing LWK performance with FWK functionality in a heterogeneous environment," in *Proceedings of the IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, ser. SBAC-PAD '12, Oct. 2012, pp. 211–218.
[33] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the Spring Joint Computer Conference*, ser. AFIPS '67 (Spring), Apr. 1967, pp. 483–485.
[34] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, May 1988.
[35] X.-H. Sun and L. M. Ni, "Another view on parallel speedup," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, ser. SC '90, Nov. 1990, pp. 324–333.
[36] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.
[37] B. Tauro, C. Liu, and K. C. Hale, "Modeling speedup in multi-OS environments," in *Proceedings of the 27th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '19, Oct. 2019.
[38] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks-summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. SC '91. New York, NY, USA: Association for Computing Machinery, Aug. 1991, pp. 158–165. [Online]. Available: https://doi.org/10.1145/125826.125925
[39] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications confining the wily hacker," in *Proceedings of the 6th USENIX Security Symposium*, ser. SSYM '96, Jul. 1996.
[40] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, Aug. 1998.
[41] K. Jain and R. Sekar, "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*, ser. NDSS '00, Feb. 2000.
[42] N. Provos, "Improving host security with system call policies," in *Proceedings of the 12th USENIX Security Symposium*, ser. SSYM '03, Aug. 2003.
[43] K. C. Hale, L. Xia, and P. A. Dinda, "Shifting GEARS to enable guest-context virtual services," in *Proceedings of the 9th International Conference on Autonomic Computing*, ser. ICAC '12, Sep. 2012, pp. 23–32.
[44] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08, Oct. 2008, pp. 51–62.
[45] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, Nov. 2009, pp. 477–487.
[46] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771
[47] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995. [Online]. Available: https://www.lammps.org/
[48] K.-I. Ishikawa, Y. Kuramashi, A. Ukawa, and T. Boku, "CCS QCD miniapp," Mar. 2017. [Online]. Available: https://github.com/fiber-miniapp/ccs-qcd
[49] B. Gerofi, R. Riesen, M. Takagi, T. Boku, K. Nakajima, Y. Ishikawa, and R. W. Wisniewski, "Performance and scalability of lightweight multi-kernel based operating systems," in *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '18, May 2018, pp. 116–125.
[50] X. Pan, J. Lindén, and B. Jonsson, "Predicting the cost of lock contention in parallel applications on multicores using analytic modeling," in *Proceedings of the 5th Swedish Workshop on Multi-Core Computing*, ser. MCC '12, 2012.
[51] X. Pan and B. Jonsson, "Modeling cache coherence misses on multicores," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS '14, Mar. 2014, pp. 96–105.
[52] ——, "A modeling framework for reuse distance-based estimation of cache performance," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS '15, Mar. 2015, pp. 62–71.
[53] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a realistic model of parallel computation," in *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '93, May 1993.
[54] K. Z. Ibrahim, S. Williams, and L. Oliker, "Roofline scaling trajectories: A method for parallel application and architectural performance analysis," in *Proceedings of the International Conference on High Performance Computing and Simulation*, ser. HPCS '18, Jul. 2018, pp. 350–358.

Brian R. Tauro received his B.Tech in Computer Science from Karunya University, and his MS in Computer Science from Illinois Institute of Technology. He is currently a PhD student in the HExSA Lab at Illinois Institute of Technology, advised by Dr. Kyle C. Hale. His research interests include low-level systems software, the intersection of compilers and operating systems, and operating systems security.

Conghao Liu received his BS degree in software engineering from East China Normal University, and his MS in computer science from Illinois Institute of Technology. He is currently a PhD student in the HExSA Lab at Illinois Institute of Technology, advised by Dr. Kyle C. Hale. His research interests include virtualization, high-performance computing, computer architecture, and operating systems for fog computing.

Kyle C. Hale received his BS degree in computer science from the University of Texas at Austin, and his MS and PhD degrees in computer science from Northwestern University. He is an assistant professor of Computer Science at Illinois Institute of Technology. His research interests include operating systems, virtualization, high-performance computing, computer architecture, and systems security. His research group creates software and hardware artifacts that are freely available online (https://github.com/HExSA-Lab).