Shadow Kernels: A General Mechanism For Kernel Specialization in Existing Operating Systems

Oliver R. A. Chick Lucian Carata James Snee
Nikilesh Balakrishnan Ripduman Sohan
Univeristy of Cambridge
firstname.lastname@cl.cam.ac.uk

Abstract

Existing operating systems share a common kernel text section amongst all processes. It is not possible to perform kernel specialization or tuning such that different applications execute text optimized for their kernel use despite the benefits of kernel specialization for performance guided optimization, exokernels, kernel fastpaths, and cheaper hardware access. Current specialization primitives involve system wide changes to kernel text, which can have adverse effects on other processes sharing the kernel due to the global side-effects.

We present *shadow kernels*: a primitive that allows multiple kernel text sections to coexist in a contemporary operating system. By remapping kernel virtual memory on a context-switch, or for individual system calls, we specialize the kernel on a fine-grained basis. Our implementation of shadow kernels uses the Xen hypervisor so can be applied to any operating system that runs on Xen.

1. Introduction

Traditional monolithic operating system design has a shared kernel that is mapped into the top of the address space of every process. This design has numerous advantages: system calls are fast as they don't require a context switch, shared-code has a low memory footprint, there is a higher cache-hit rate, and the shared state eases kernel design and implementation.

At the same time, kernel specialization has been shown to be beneficial [6, 19]: profile-guided optimization of Linux can improve performance by up to 10% for some applications [20], exokernels eliminate kernel abstractions for applications so that applications communicate more directly with hardware thereby reducing kernel overheads [7], and kernel instrumentation can be added that only fires when the kernel is executing on behalf of certain processes.

Such kernel specialization is often process-specific, in that the specializations applied to one process may have an adverse effect on other processes. For instance, profile-guided optimization of the kernel improves the performance of some applications and diminishes the performance of others. Similarly, removal of security checks may be desirable for trusted processes, but undesirable for non-trusted processes.

Yet, current production operating systems do not provide a primitive for kernel specialization on a per-process level. The shared kernel means that any changes to the kernel text or data have global effects; there is no way to isolate kernel modifications to individual processes. As such, it is not currently possible to execute an individual process with different kernel optimizations or instrumenation to the rest of the processes executing on the system.

In order to provide a performant, useful and effective application augmentation primitive, it is important to have the ability to limit the scope of kernel specialization. A new low-level primitive is needed to support this: one that isolates kernel specialization for a single process and allows for quick changes to its scope. To this end we propose *shadow kernels*: kernel variants with specialized text sections that are modified with the specialization required, but share their data sections with the booted kernel. Non-specialized processes continue to run the original unmodified kernel instruction stream whereas those that require specialization are dynamically switched to execute the modified code of a shadow kernel.

When process-specific kernel specialization is required, a copy of an existing kernel instruction stream is made and remapped into the requesting process kernel address space.

APSys 2015,, July 27–28, 2015, Tokyo, Japan..
Copyright © 2015 ACM 978-1-4503-3554-6/15/07…$15.00.
http://dx.doi.org/10.1145/2797022.2797027

Because the specialized kernel instruction stream is limited to executing in the requesting process context, it has no performance side-effects on other processes in the system.

In the remainder of this paper we argue the benefits of per-process kernel text regions (§2); introduce a design for shadow kernels (§3); and evaluate our Xen-based implementation of shadow kernels (§5), showing that the performance overhead of shadow kernels is minimal, with the cost of using a shadow kernel consisting of ten pages of specialization to be $4.2\,\mu$s each time the process is scheduled.

2. Motivation

Shadow kernels allow kernel-text specialization without affecting processes other than those being specialized. We discuss such use cases, highlighting the main benefits of shadow kernels.

2.1 Per-process kernel profile-guided optimization

Recent work has considered applying profile-guided optimization to operating system kernels to improve performance. An unsolved issue with profile-guided optimization is that the optimization must be based on a representative workload. In particular, if the kernel is optimized based on one application then other applications executing on the same system often see a slowdown in performance. Yuan *et al.* show that profile-guided optimization of the Linux kernel can improve performance of some applications by 10%, and reduce performance of others [20], even when executing a single application on a machine.

Shadow kernels allow applications executing on the same machine to each execute with their own kernel that is optimized with profile-guided optimization specific to that program. This therefore allows a training-phase per-process that generates a shadow kernel per process. So-long as the profile-guided optimizations do not modify the data sections—which can be ensured through compiler flags—then each process can have its own shadow kernel. Each time that the scheduler schedules-in a process, it remaps the kernel to the appropriate shadow kernel. This can be further extended to allow *multiple* shadow kernels per-process by creating shadow kernels per process subsection.

2.2 Scoping probes

Modern operating system kernels provide a range of instrumentation primitives, for example Kprobes and DTrace. The basic mechanism for most of these approaches is identical: probe handlers are attached to kernel addresses by rewriting the instruction at the probed address with a software breakpoint, or jump instruction. CPUs executing the code raise an interrupt, or perform a jump, upon executing the instruction.

The Achilles' heel of this approach is that any process that executes the instrumented address or function calls into the instrumentation system regardless of whether it is required. It is impossible for users to restrict the scope of the instrumentation to a particular process. The unavoidable penalty of hitting the probe is incurred by *every* process *each* time it is executed, regardless of whether it is applicable to the executing process.

This is particularly problematic if the application being investigated consumes a minority of the system's CPU cycles, since if hot functions are probed—those that are an obvious cause of poor performance—every system call made by the rest of the applications on the system could become substantially slower.

With shadow kernels, probes can be set so they are only fired for an individual application, thereby leaving the performance of the well-behaving programs untouched. The overall probe effect of the added instrumentation is also reduced: setting kernel probes on hot functions such as `kmalloc` or `tcp_sendmsg` on a busy server no longer degrades overall system performance.

2.3 Kernel optimization and fastpaths

Kernel configurations options are often a tradeoff between performance and utility of features. For example, kernel options that provide debug modes for locks, schedulers and memory allocators add additional code to the instruction stream that is executed and causes a performance degradation. With shadow kernels, configuration options that only affect the text section can be applied to individual processes, without system wide effects.

Two such fastpaths that can be applied to individual processes are (i) removing security checks for trusted processes, and (ii) removing some concurrency operations when executing a process on a single core. (i) A key rôle of the operating system kernel is to perform security checks, however applying these checks can take a substantial amount of compute resource [13]. Often, some processes—such as system processes—are trusted whereas others ought to be subject to the usual kernel security checks. Moreover, applications such as debuggers often need so subvert the usual security checks to introspect the memory of another process. However, with current kernel models, the same checks are applied to all processes. With shadow kernels, priviliged processes can be mapped onto shadow kernels that contain exactly the security checks relevant to each application. (ii)Virtual machines operating in the cloud can be subject to vCPU hotplugging by the cloud provider. Should a virtual machine change between having multiple vCPUs to one vCPU there is scope for optimization by eliminating concurrency primitives from the kernel instruction stream. In particular, a key benefit of shadow kernels built using a hypervisor is that a priviliged domain—controlled by the cloud provider—can trigger switches of shadow kernel as part of the hotplugging routine.

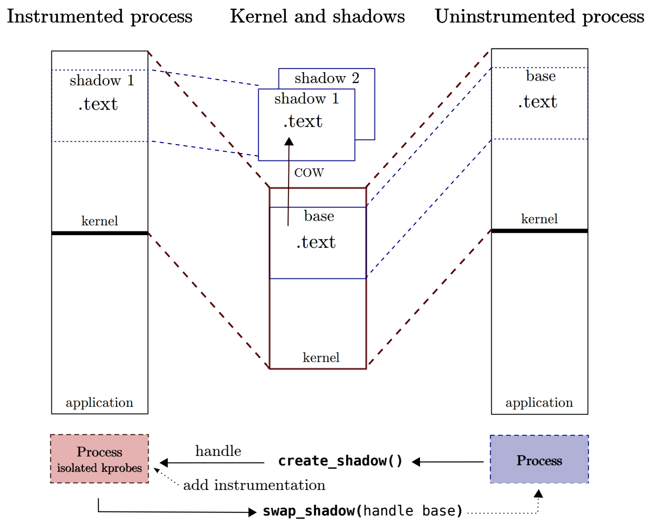

Figure 1. Overview of shadow kernel design

3. Design

An operating system with shadow kernels support boots in the traditional manner, with all programs initially sharing the kernel. An application can spawn a new shadow kernel through a call to a kernel module. This creates a copy-on-write version of the currently running kernel, which is mapped into the memory of the process that created it. As a process registers probes, the specialization mechanism makes modifications to the kernel's instruction stream. Due to the use of copy-on-write, every page that is modified is then physically copied, leaving the original kernel text untouched.

Where a function's definition is to be replaced by a definition that is the same length or shorter, we replace the function in place. However, where the specialized text is longer we allocate additional memory for the new function, and replace the first instruction of the function in the shadow kernel with a jump to the new function.

Figure 1 shows each time a shadow kernel is spawned, the shadow kernel API returns a `shadow_kernel_handle` that can be used to switch the process between the original and the shadow kernel. To perform this switch the application remaps the top of its address space to be the shadow kernel. This handle can also be shared amongst processes, applied to all processes in Linux container, or even used within a resource container [2]. For instance, a system may have a shadow kernel with complete instrumentation that any process can use to get a function call graph. On fork, shadow kernels are inherited by the child.

As shadow kernels only fork the text section of the kernel, and do not modify the address of function entry points, the benefits of having a shared memory kernel remain and the semantics of function pointers are preserved.

Shadow kernels are also compatible with kernel modules: each time a module is inserted, or removed, we iterate over the page tables, and perform a mapping, or unmapping for each shadow kernel.

3.1 Asynchronous tasks

One of the challenges of a shadow kernel design is dealing with code that runs in kernel mode outside particular processes. This includes code executed in kworkers, interrupts, timers or during scheduling. Asynchronous actions like these are routinely executed by the kernel for the benefit of multiple processes.

If a given application requires specialization (for example, for creating a hot path), it is difficult to obtain complete isolation of the specialization from the other applications: Take the example of a kworker that occasionally flushes data to disk. Some of these data might belong to specialized applications, some of it to regular un-specialized processes. The key challenge is to correctly identify the correct kernel text section for the kworker to execute.

Further complicating matters, two applications might want to set different probes for the code executed by that kworker (one application could be interested in measuring the number of bytes committed to disk, another in the time it takes to complete the same operation).

A solution to the issue is to give up isolation for these particular use cases and make the kworker run the code of a "union" shadow kernel that contains all the probes set by the various applications for their own shadow kernels. However, this may not provide correctness of results, if the specialization modifies the semantics of the original kernel. Also, in the worst-case it incurs the same overheads as current mechanisms.

For some cases, a better alternative exists: using hardware virtualization primitives. Single-Root I/O Virtualization support on recent network cards and flash storage makes it possible to assign per-process virtual devices (and reserve corresponding hardware resources—such as receive/send queues). This approach has already been shown to be viable in systems such as Arrakis [13, 14].

Once a process is given ownership of a particular virtualized PCI device, the problem of "routing" asynchronous tasks towards executing a given shadow kernel becomes solvable: All the code that is executed in kernel space when interacting with the device (interrupts, timers) uses the text section of the shadow kernel associated with the process that owns the device.

4. Implementation

Our current implementation of shadow kernels is a Linux kernel module, built on a Xen-paravirtualised Linux kernel. Whilst it is possible to build shadow kernels without a hypervisor, we chose to implement shadow kernels using Xen, as it uses a paravirtualised memory management unit (MMU), that forces all guests to update page tables by issuing hypercalls. As any operating system—including HVM containers—can issue hypercalls, this allows the core of our implementation to be used in any contemporary operating system with minimal engineering effort. Moreover, an existing criticism of kernel specialization has been that it requires invasive changes to the core memory-management of the operating system. As virtual machines have loose coupling between kernel virtual addresses and machine physical addresses our implementation is less invasive than modifying bare-metal kernel memory assumptions. Indeed, the core of our implementation of shadow kernels is 250 lines of code, entirely implemented as a Linux kernel module: no kernel modifications are necessary.

When a shadow kernel has been created, it initially uses the same memory as the booted kernel to reduce memory overheads and ensure high cache-hit rates. Each time a program modifies the kernel text, if that page has not previously been modified, we first allocate a new page and copy the original text section into this new page. After the modifications have been applied, we issue a hypercall that updates the machine-to-physical, physical-to-machine, and virtual-to-machine page tables. Whilst this design does invoke extra hypervisor-load, we note that the design of x86_64 protection rings require domains to trap into Xen on each system call anyway, therefore the additional load added by shadow overheads is minimal.

Our implementation has two methods of scoping shadow kernels. Firstly, per-process scoping interposes the operating system scheduler—using Kprobes for Linux—that swaps shadow kernels whenever a process is scheduled in-or-out. Secondly, manual-scoping allows processes to use a shadow kernel for a subset of their system calls.

4.1 Implementation safety

As shadow kernels are provided through a kernel module, users need root priviliges before enabling shadow kernels. It is possible to change the permissions such that non-root users can enable shadow kernels (for instance by using setuid) however given that enabling shadow kernels rewrites the kernel instruction stream, we feel this is unadvisable.

We apply suitable locking to the kernel to ensure safety from concurrency effects when switching shadow kernels.

5. Evaluation

5.1 Scoping probes

We initially show that the lack of per-process kernel specialization can be prohibitive by considering the case of inserting Kprobes on hot Linux kernel functions to profile system call latency of a specific process on a production system. This system is also running memcached—an application whose performance ought to be unaffected. We illustrate that Memcached's performance *is* substantially affected by the inserted probes by measuring its single-CPU single-threaded throughput degradation. This experiment runs on a Xeon E5-2660v2 serving a production workload at 10 Gbps

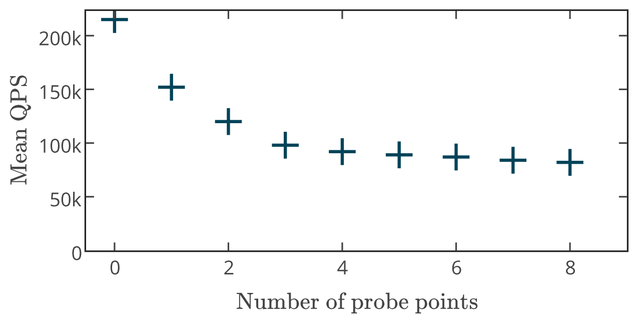

Figure 2. Probing hot functions in the Linux kernel causes a performance impact of up to 62% to applications that are not executing any instrumentation code. This is caused by the increased time to execute a system call, due to the probes. All results are the average of six runs. All samples were within 5% of the mean value.

on Linux 2.6.32.[1] We insert empty probe points into kernel hot functions (as determined by profiling the most commonly called functions across all CPUs in the server). Inserting probes on hot kernel functions is a common technique used to comprehend the interaction of kernel locks with applications semantics. We are therefore showing how the performance of Memcached is affected when instrumenting the interactions of other applications with the kernel.

Figure 2 shows that (without shadow kernels enabled) adding these probes reduces the throughput. Probing the most popular kernel function called across all CPUs in the system reduces single thread performance by 30%. Performance worsens with increasing numbers of probe points—with the top three functions being probed, performance is less than 50% of baseline performance. If the scope of the probes were isolated to only fire for *other processes* on the system the performance of memcached would not decrease substantially.

5.2 Overheads of shadow kernel creation

Having shown that the lack of per-process kernel specialization causes substantial performance degradations we now consider the costs of creating a shadow kernel, an action that we expect to be infrequent as it only occurs when an application requires new specialization. We execute all experiments on an Intel Xeon E3-1230 V2 @ 3.3 GHz, running Ubuntu 14.10, with a Linux 3.19 kernel compiled from the Linus branch.

To measure the cost of creating a shadow kernel we use a microbenchmark that repeatedly creates a shadow kernel, but does *not* switch to it. We measure this cost as $1.4\,\text{ms} \pm 0.1\,\text{ms}$. As this cost is only borne when spawning a new shadow kernel, which we expect most processes to do just once, such overheads are acceptable.

[1] This is the latest long-term kernel release of the 2.6 branch, used by RHEL6. It contains all major performance improvements between 2.6.32 and HEAD as of Nov 2014.

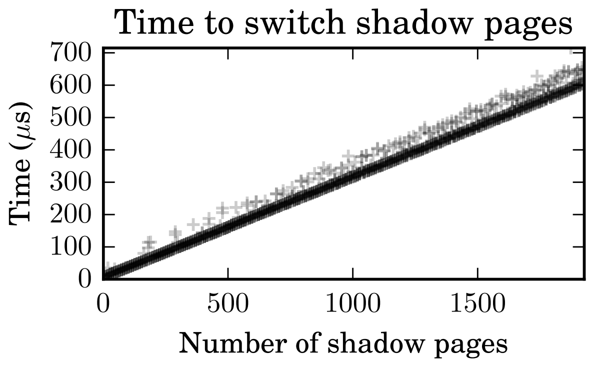

Figure 3. The time to switch to a shadow kernel is proportional to the number of pages that are modified in the shadow kernel.

5.3 Overheads of switching shadow kernel

We now consider the costs of switching to an existing shadow kernel, an action typically performed on each context switch.

5.3.1 Microbenchmark analysis

To measure the cost of switching shadow kernel we use a microbenchmark that repeatedly switches to a shadow kernel with a varying number of pages, and performs a single system call (`socket`). We issue the system call in order to trigger a mode change into the kernel, so that we can update the page tables.

Performing the `socket` system call, without switching to a shadow kernel, takes $5.19\,\mu\text{s} \pm 1.61\,\mu\text{s}$. Figure 3 shows the time to perform the microbenchmark, varying the number of specialized pages between 1 and 1927, the number of pages in the text section of our kernel. This time is directly proportional to the number of pages modified, since for each page Xen performs an unmap and remap operation. To specialize ten pages—in order to specialize the ten most-hot kernel functions—and execute the `socket` system call takes $9.4\,\mu\text{s}$. This additional $4\,\mu\text{s}$ only need be performed on a context switch, rather than being an additional overhead of every system call. Therefore, the overheads of performing such specialization on hot kernel functions are small. Whilst the overheads for remapping the *entire* kernel are higher, we envisage a hybrid scheme whereby we use a huge page to represent a shadow kernel that changes a large proportion of the kernel. Despite improving switching cost, using large pages does increase the memory footprint of shadow kernels, from n 4 KB pages (whereby n is the number of pages changed) to at least 2 MB.

An alternative approach is to assign virtual machines a vCPU dedicated to a particular process executing using a shadow kernel so that the hypervisor, rather than the OS scheduler performs scheduling decisions for the application.

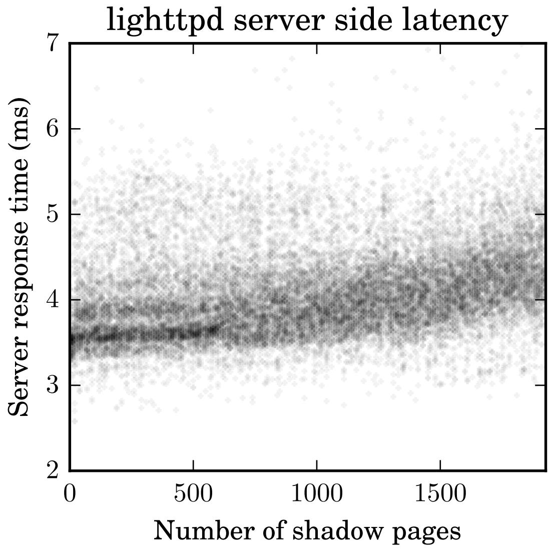

Figure 4. As the number of pages in the shadow kernel increases the server side latency of `lighttpd` increases.

This prevents the need to remap shadow kernels, as the OS scheduler will not deschedule the specialized application.

5.3.2 Application to web workload

Having shown a microbenchmark cost of switching to a shadow kernel, we now show the overheads when applied to a realistic workload. We modify `lighttpd` in order to switch to a shadow kernel whenever it is scheduled in by the operating system scheduler. This setup represents the case of adding probes to the kernel that only fire when one process is executing, as explained in Section 2.2. Lighttpd serves a 217 KB file.[2] We increase the number of pages in the shadow from 0 to 1920 in increments of 10, measuring the response time for 100 requests at each level, with a client concurrency level of 1.

Figure 4 shows that as the number of pages in the shadow kernel increases, the server response time increases as well. The server side median response time without using shadow kernels is $3.54\,\text{ms} \pm 5.50\,\text{ms}$. With a shadow kernel that remaps *every* page the server response time is $4.34\,\text{ms} \pm 5.32\,\text{ms}$. Of this increase, approximately 0.6 ms can be explained by the cost of switching to the shadow kernel when lighttpd is scheduled in due to the arrival of a request, as shown in Figure 3. The remaining slowdown, of 0.2 ms, is expected to be caused by an increased L1 i-cache miss rate. Whilst in low-latency setups an additional overhead of 0.2 ms may be undesirable, for kernel specializations that do not specialize all of the kernel text—the majority of which is for unused drivers—the overheads are lower. For in-

[2] The homepage of `https://edition.cnn.com/`.

stance, there is no significant increase in latency when fewer than 300 pages are specializied.

6. Related work

Shadow kernels isolate processes from unintended specialization overhead by only specializing a fork of the current kernel's text section. Since the 1980s, the benefits of kernel specialization for performance have been well known [15], however invasive changes have prevented uptake by mainstream kernels. Yet, the idea of kernel specialization has been reconsidered in research operating systems—such as Barrelfish—for many core computers whereby each core runs an entirely different kernel [16]. Furthermore, by implementing multiple services that provide competing implementations of a service, microkernels can offer kernel specialization [12].

In production operating systems, tools like Ksplice [1] enable the live patching of kernel code; once patched, the whole system is then switched to run the new modified kernel. Moreover, modern malware often uses similar page-table tricks to shadow kernels, for instance by desynchronizing the instruction and data TLBs [17]. Shadow kernels differ from these systems by allowing multiple kernel text variants to coexist. This is similar to systems that make use of multiple text sections to offer kernel hardening [11] or to run specific system calls in trusted kernels [18].

Virtualization techniques allow the running of multiple operating systems (kernel and user spaces) on a single machine [4]. Shadow kernels differ from virtualization however, by not requiring separate kernel data and user spaces; and instead isolate only the kernel text. Multikernels apply a similar technique by sharing a common user space but add support for running separate kernels on each CPU core with explicit communication between them [5].

A key feature of shadow kernels is that the user is able to switch a process to using one on demand. This is similar to *on-demand virtualization* [10], where a user migrates their operating system onto a virtual machine when they require isolation and checkpointing.

Instrumentation tools have been built to obtain performance data from an operating system kernel [3, 8]. Fowler *et. al* show that program performance can be improved by both kernel space and user space having access to kernel performance data [9]. However, what can be achieved with current instrumentation primitives is limited by the performance issues we show in Section 5. In that sense, our work is orthogonal to the instrumentation mechanism that an application uses, removing existing limitations and providing the isolation required to apply such instrumentation with a low performance overhead.

7. Conclusion

Current mainstream kernels share a common instruction stream amongst all processes. This prevents per-process kernel specialization, limiting the ability to do custom optimization, or isolation of probing.

To solve this problem we present shadow kernels. By switching between multiple kernel text sections, when processes are scheduled in and out, we allow processes to execute kernel variants that are optimized to their operation. Our Xen-based implementation adds an overhead of $4.2\,\mu s$ to each scheduler operation whilst specializing ten pages. Moreover, we show a negligible performance impact of shadow kernels when applied to a lighttpd web server.

8. Acknowledgements

We should like to thank Andrew Rice (`acr31@cl.cam.ac.uk`) and Andy Hopper (`ah12@cl.cam.ac.uk`) for their work in proof reading this paper and Gernot Heiser for shephering our paper.

This work was principally supported by internal funds from the Computer Laboratory at the University of Cambridge; and also by the Engineering and Physical Sciences Research Council [grant number EP/K503009/1].

The raw data and code required for reproducing the figures in this paper are available.[3] We aim to release the source code for shadow kernels along with a more complete technical description as part of a full-length paper in 2016.

References

[1] Arnold, J., and Kaashoek, M. F. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), EuroSys '09, ACM, pp. 187–198.

[2] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 45–58.

[3] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 18–18.

[4] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 164–177.

[5] 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 ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 29–44.

[3] `https://www.cl.cam.ac.uk/rscfl`

[6] BHATIA, S., CONSEL, C., LE MEUR, A., AND PU, C. Automatic specialization of protocol stacks in operating system kernels. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on* (Nov 2004), pp. 152–159.

[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 Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 251–266.

[8] ERLINGSSON, U., PEINADO, M., PETER, S., AND BUDIU, M. Fay: Extensible distributed tracing from kernels to clusters. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 311–326.

[9] FOWLER, R., COX, A., ELNIKETY, S., AND ZWAENEPOEL, W. Using performance reflection in systems software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems - Volume 9* (Berkeley, CA, USA, 2003), HOTOS'03, USENIX Association, pp. 17–17.

[10] KOOBURAT, T., AND SWIFT, M. The best of both worlds with on-demand virtualization. In *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2011), HotOS'13, USENIX Association, pp. 4–4.

[11] KURMUS, A., AND ZIPPEL, R. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 1366–1377.

[12] LIEDTKE, J. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 237–250.

[13] PETER, S., AND ANDERSON, T. Arrakis: A case for the end of the empire. In *Proceedings of the 14th USENIX Workshop on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2013), HotOS'13, USENIX Association, pp. 26–26.

[14] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 1–16.

[15] PU, C., MASSALIN, H., AND IOANNIDIS, J. The synthesis kernel. *Computing, Springer Verlag (Heidelberg, FRG and NewYork NY, USA)-Verlag Systems* (1988).

[16] SCHPBACH, A., PETER, S., BAUMANN, A., ROSCOE, T., BARHAM, P., HARRIS, T., AND ISAACS, R. Embracing diversity in the Barrelfish manycore operating system. In *In Proceedings of the Workshop on Managed Many-Core Systems* (2008).

[17] SPARKS, S., AND BUTLER, J. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan* (2005), 504–533.

[18] TA-MIN, R., LITTY, L., AND LIE, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 279–292.

[19] YOUSEFF, L. M., WOLSKI, R., AND KRINTZ, C. Linux kernel specialization for scientific application performance. Tech. rep., 2005.

[20] YUAN, P., GUO, Y., AND CHEN, X. Experiences in profile-guided operating system kernel optimization. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (New York, NY, USA, 2014), APSys '14, ACM, pp. 4:1–4:6.