Hypercallbacks: Decoupling Policy Decisions and Execution

Nadav Amit
VMware Research Group
namit@vmware.com

Michael Wei
VMware Research Group
mwei@vmware.com

Cheng-Chun Tu
VMware
tuc@vmware.com

1 INTRODUCTION

Hypervisors must make a wide range of policy decisions which overlap with decisions made by VMs, such as memory management, I/O scheduling and resource allocation. Coordinating these policy decisions for specific workloads can greatly improve the overall performance of the system. For example, during memory reclamation, a hypervisor can avoid unnecessary I/O by paging out only memory which is inactive in the VM. Privilege separation and the VM abstraction, however creates a *semantic gap* which makes it difficult for the hypervisor to know which VM pages are least likely to use. As a result, the hypervisor and VMs make policy decisions which are at odds with one another, greatly hampering performance.

Over time, several techniques have been developed to bridge the semantic gap. These mechanisms typically relegate policy to either the hypervisor or the VM, so that conflicting policies are not applied. Memory ballooning [1] is perhaps one of the most well known examples - ballooning gives the hypervisor the ability to reclaim memory by applying memory pressure to a VM when the hypervisor wishes to allocate memory to another VM. The decision on which memory to page out is delegated to the VM - but due to the semantic gap, the VM may spend considerable time and I/O paging out memory which is already on disk or deduplicated by the hypervisor. In addition, the hypervisor now depends on the VM in order to reallocate memory which can have serious implications (for example, if the VM is slow, reallocation will also be slow). This problem exists because the execution of the policy is coupled with deciding the policy.

In this paper, we present the design and implementation of the hypercallback in Linux KVM (Figure 1), an approach which bridges the semantic gap by enabling policy decision making to be decoupled from its execution. Hypercallbacks enable a hypervisor to *execute* a policy while permitting the VM to make the actual policy *decision*. In a hypercallback, VMs provide untrusted code for the hypervisor to both introspect the policy decisions a VM has made and to impact policy decisions made by the VM, bridging the semantic gap. Security and robustness are ensured by a safety checker, which verifies the runtime behavior of the hypercallback.

This paper makes the following contributions:

HotOS '17, Whistler, BC, Canada
 978-1-4503-5068-6/17/05. . . $15.00
DOI: 10.1145/3102980.3102987

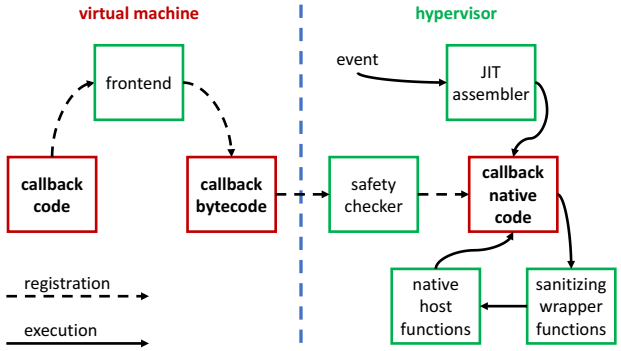

Figure 1: Using untrusted VM callbacks in the hypervisor through code verification.

- We present the design and implementation of the hypercallback, a hybrid mechanism which decouples policy from execution (§3). Hypercallbacks are inspired by existing work on safe OS extensions (i.e., BPF [2]) and allow the hypervisor to safely execute VM OS code.
- Our design is by no means complete and we identify challenges and opporunities which must be addressed to achieve maximum performance and usability (§5).
- We demonstrate a hypervisor using hypercallbacks to safely walk arbitrary VM page tables, and show a 3× performance improvement by automatically discarding free VM pages from the hypervisor (§6).

2 BRIDGING THE GAP

Privilege separation and the VM abstraction create a semantic gap between the hypervisor and VM. The hypervisor presents the VM with only an interface identical to real hardware. Without detailed knowledge of the internals of the VM's operating system, the hypervisor has no visibility into the VM. Likewise, without support and coordination with the hypervisor, the VM has no ability to introspect the hypervisor. The semantic gap is responsible for many of the overheads associated with virtualization, because the hypervisor and VM may make many resource allocation decisions which are at odds with one another.

Today, state-of-the-art techniques to bridge the semantic gap in virtualization fall into two categories: paravirtualization [3, 4] and virtual machine introspection (VMI) [5, 6]. We discuss both techniques below, but in general, they suffer from coupling policy decisions with execution.

Paravirtualization. Paravirtualization enables VMs to issue *hypercalls* to request services from the hypervisor or provide the hypervisor with information regarding the VM, for

example hinting the hypervisor about memory sharing opportunities [7]. Similarly, the hypervisor can issue *upcalls* to the VMs, requesting VM OS services. One of the key principles of paravirtualization is to delegate policy decisions to the side of the gap with the most knowledge to make a decision. The consequence of this design is that the hypervisor is now dependent on opaque logic within the VM in order to perform resource management.

Paravirtualization suffers from several drawbacks. First, paravirtualization is intrusive and requires modifications to the VM OS, typically in the form of modules to refrain from complicating OS maintainability [8, 9]. Second, paravirtualization operations require potentially long context switches (895 cycles [10]) to communicate information between the VM and the hypervisor. Paravirtual mechanisms also enlarge the hypervisor trusted computing base (TCB) and can introduce bugs that jeopardize both the hypervisor and VM robustness and security [11, 12]. Finally, there are no guarantees on the timeliness of upcalls from the hypervisor to the VM, since they are executed by the VM, which may be busy performing other tasks.

Several techniques have been employed to address the shortcomings of paravirtualization. Paravirtualization can be simplified by injecting code to the VM [13] or using binary patching of native OSes [14]. Communication between the VM and the hypervisor is possible without context-switches by running the hypervisor on a different core [15], or using shared rings between the VM and the hypervisor [7, 16]. In some situations, the hypervisor can infer the VM behavior without explicit communication with the VM [17, 18] or reduce the number of context switches [19]. In all these cases however, a hypervisor using paravirtualization to bridge the semantic gap depends on the VM to enforce policy.

Virtual machine introsepction (VMI).. VMI [5, 6] enables the hypervisor to directly inspect a VM's data structures. This enables the hypervisor to make some policy decisions without depending on the VM to execute requests, partially addressing a shortcoming of paravirtualization. VMI, however, is dependent on the hypervisor's knowledge of the VM, and minor updates of the VM OS can prevent VMI from working correctly and potentially cause failures of the VM OS. Complex interactions with a running VM involving locks and other concurrency mechanisms can further complicate VMI. As a result, VMI still couples execution and decision of policy in practice, with VMI introspecting VM decisions after they have been made.

Next, we discuss hypercallbacks, a hybrid mechanism which like VMI allows the hypervisor to execute policy and introspect the VM, but like paravirtualization, enables the VM to make policy decisions. By decoupling policy decisions from execution, hypercallbacks provide the best of both worlds.

3 HYPERCALLBACKS

Hypercallbacks enable the VM OS to provide untrusted code to the hypervisor (Figure 1). VM OSes register hypercallbacks to the kernel using a hypercall. In registering the hypercallback, the VM OS provides the bytecode for the callback, the memory regions the callback may need to use, and a hypercallback number, which specifies what the hypercallback should be used for. Upon registration, the hypervisor runs a safety checker, which verifies both the safety and running time of the bytecode. After verification, a native hypervisor function can call the hypercallback, and a JIT assembles the callback code in sanitized native code for maximum performance.

Operation	Description
`register (cb_num, callback, gpa_range)`	Register a callback number *cb_num* with bytecode *callback* and make the guest physical address range *gpa_range* available to the callback.
`callback (cb_num, args)`	Invoke the hypercallback numbered *cb_num* with arguments *args* and optionally return a value.

Table 1: Hypercallback Interface

Hypercallbacks provide several advantages over paravirtualization and VMI: their execution is efficient as they do not require context switches and only run when needed. They do not affect the hypervisor robustness since their safety is ensured. They are relatively non-intrusive to the kernel, as they are implemented separately from existing code-path. As part of their safety checks, the runtime of hypercallbacks is bounded to ensure their timely execution.

The interface to hypercallbacks is shown in Table 1. The two primary uses of hypercallbacks are *queries* from the hypervisor to the VM OS and *notifications* from the hypervisor to the VM OS.

The hypervisor uses queries to assist it in making informed resource allocation decisions. For example, the hypervisor can query the VM which memory pages should be reclaimed, migrated to a different NUMA node, promoted to hugepages or considered for deduplication. Notifications, on the other hand, are used by the hypervisor to notify the VM about operations that may affect the VM. For example the hypervisor can inform the VM when VCPUs are preempted or scheduled, which would allow the VM to avoid sending inter-processor interrupts to idle VCPUs or to rebalance the scheduler run-queues accordingly.

Hypercallbacks can also call hypervisor provided helper functions that can be used to assist tight cooperation of the hypervisor and the VM. For example, the hypervisor can provide an helper function that injects an exception or interrupt to the VM. The VM can use such a function to develop, for example, a similar feature an "asynchronous page-fault" mechanism, which informs the VM when it accesses unavailable memory [20]. Using hypercallbacks would provide more flexibility for the VM, allowing it to control in which situations such exceptions can be delivered.

In the next section, we describe some use cases where hypercallbacks offer significant benefits over paravirtalization and VMI.

4 USE CASES

Memory Management. As we described in the introduction, both the hypervisor and VM manage memory, and paravirtualization causes the latency of the hypervisor memory management operations to be controlled by the VM OS latency, which results in suboptimal reclamation decisions [18] and redundant I/O operations [21, 22]. While it is possible to address some of these inefficiencies [18, 21, 23, 24, 25, 22], hypercallbacks enable the hypervisor to inspect the state of VM pages, and conversely for the VM to be notified when memory is paged by the hypervisor, allowing memory reclamation to proceed in the hypervisor.

VCPU scheduling. In order to maximize the CPU utilization, hypervisors often abstain from co-scheduling virtual CPUs (VCPUs) [26]. However, OSes are unaware of VCPU scheduling, which results in synchronization latencies. For example, a preempted VCPU that holds a lock prevents other running VCPUs from progressing [27, 28]. Other mechanisms such as inter-processor interrupt (IPI) can induce overheads when the target CPU is not running [29]. Some paravirtual mechanisms exist [30, 31, 32] but they are limited by the cost of hypercalls or force the hypervisor into static CPU provisioning. Hypercallbacks can enable the VM OS to be notified of VCPU preemption and to handle locks, and give the hypervisor insight of the status of processes and threads scheduled within the VM.

Storage and I/O Scheduling. The VM's insight into the I/O policies of the hypervisor are limited, and typically the hypervisor enforces coarse-grained policies such as static IOPS limits [33] or relies on hardware via passthrough [34]. Hypercallbacks provide a high performance mechanism for hypervisors to introspect and prioritize VM I/O scheduling decisions as well as for the VM to react to I/O pressure in the hypervisor.

Profiling and tracing. Profiling the runtime behavior of a VM can be difficult without insight into the hypervisor. While paravirtual mechanisms can enable the VM to access data provided by the hypervisor (e.g., counters), tracing with hypervisor events is difficult unless the profiling engine is run from the VM. Hypercallbacks enable the VM to be notified of events when they occur, enabling fine-grained profiling not possible with paravirtualization.

5 IMPLEMENTATION

Hypercallbacks draw on ideas from safe OS kernel extensions [35, 36, 37, 38] and specifically are inspired by the Berkeley Packet Filter (BPF) [2]. Originally designed for performant filtering of packets from kernel to userspace, BPF has been extended and is now thought of as the "universal in-kernel virtual machine" and used in the kernel outside of networking for features such as tracing and `seccomp` [39].

BPF generates bytecode for the BPF VM, which is translated into native code that is checked for safety by a verifier. Given BPF's use throughout the Linux kernel, we expected that BPF would be a good fit for hypercallbacks, and used it in our implementation. Our design treats the memory range given during registration as a packet using the direct packet access (DPA) function, and relied on the BPF verifier to make sure that the hypercallback never tried to access memory outside of the registered range for safety. In the next sections, we describe our experience and the challenges we faced in our implementation of hypercallbacks using BPF.

5.1 Code Reuse and Verification

One of our early goals was to be able to directly translate kernel functions to BPF bytecode with limited modification using the LLVM BPF backend. We use the BPF verifier to check several safety properties: (1) All memory accesses are to the VM state and to the callback arguments; (2) The callback runtime or number of instructions is bounded; (3) No privileged instruction are run; and (4) Only whitelisted helper functions are called.

Even for a relatively simple function (`is_free_buddy_page`), with about 10 SLOC, we needed to make several modifications immediately to pass the verifier: directing the compiler to unroll loops, adding some redundant checks as the verifier is too restrictive, removing the use of native CPU instructions and and removing branches to avoid exhausting the verifier resources. For all of these reasons, the hypercallback code is suboptimal. We believe verifier enhancements could have allowed fewer modifications and better performance.

BPF also bounds the execution time, which limits the code a hypercallback can use. For instance, hypercallbacks cannot wait for locks, but only acquire them if they are not taken (e.g., using `try_lock` primitives). Similarly, waking cores that wait for a lock must be performed using an helper function, since the hypercallback is disallowed to send IPI directly. In addition, the hypercallback cannot traverse reverse mapping data structures since a frame may be mapped in multiple, potentially many, address spaces.

Some attention is required to prevent other potential attack vectors. The hypervisor helper functions must perform input checks that are carefully selected to introduce new side-channels that may potentially leak sensitive data to malicious VMs. Potential attacks using JIT spraying should be prevented by limiting the number of callbacks and their length as well as randomizing their location [40, 41].

5.2 Accessing VM Memory

To further the goal of code reuse, hypercallbacks need to be able to use memory pointers in the guest virtual address (GVA) space. The hypervisor uses a different address space, the host virtual address space (HVA), in which these pointers are invalid. Unfortunately, the overheads of either switching address spaces or tracking GVA changes is too high [42, 43]. To overcome this challenge, we limit hypercallbacks only to access VM kernel data structures that are permanently mapped. In Linux, for example, this memory includes the direct mappings section and the virtual memory and in Windows—the nonpaged pool. Once the VM OS loads it registers the GVA range that the VM is allowed to use.

Nevertheless, the GVA range may already be used in the HVA. The hypervisor therefore maps the GVA range contiguously, but within an offset from the GVA range base address. When the callbacks are registered, the hypervisor modifies memory accesses to the VM to use the correct address. In the x86-64 architecture such modification can be done rather efficiently by using a segment base.

Yet another problem may occur if the memory that the hypercallback accesses is paged out, since it is not expected that the hypervisor would withstand the latency of paging-in. If the memory area is fixed and limited, for example the Linux virtual memory map, it can be pinned upon registration, by the hypervisor. Otherwise, hypercallbacks should assume accesses to this memory may fail and provide code for recovery, which the hypervisor would initiate when they access paged-out memory.

6 EVALUATION

Our evaluation of hypercallbacks explores two questions: first, in isolation, how do hypercallbacks perform compared to paravirtualization with hypercalls? Second, can we use hypercallbacks to improve performance on a real workload?

We first examine the performance of hypercallbacks in isolation by using microbenchmarks to measure the runtime of the callbacks. Callbacks take on average 58 cycles for a page that cannot be discarded, 81 cycles for unmapped page-cache, and 157 cycles for a free page when most VM memory is free. This is a considerable improvement from a hypercall, which cost at least 835 cycles [10], which the additional penalty of a context switch. Admittedly, our prototype can only acquire locks in the hypercallback, and may not release them, causing additional exits when the VM OS accesses the lock. However, we do not have to pay the penalty of context switching. If we had the ability to execute native atomic machine instructions in a hypercallback, then this penalty could be avoided by releasing locks in the callback as well.

Next, we evaluate hypercallbacks in the context of memory reclamation. In the introduction we showed that memory ballooning depends on the VM OS, which may not be responsive if the VM OS is low on resources. When this occurs the hypervisor resorts to host swapping, reclaiming memory directly. Due to the semantic gap the hypervisor does not know how the VM uses its memory (whether a page already resides on disk) and the VM is unaware of physical resource limits. Consequently, the hypervisor and the VM make suboptimal decisions and perform or cause redundant paging activity. Some solutions include tracking VM I/O operations to find the VM page-cache or page rewrite operations to find free pages [18, 21]. These solutions, however, require tracking that may cause overheads when no memory pressure exists and do not prevent free VM pages from being written to disk. In contrast to these solutions, hypercallbacks can eliminate such inefficiencies in a secure, fast and robust way.

We implement our prototype using the Linux hypervisor [44]. Before the hypervisor reclaims a page, it needs to find which VM owns the page and the guest frame number (GFN) to update the nested page table. The hypercallback system uses this information to call the page reclamation callback of the respected VM, providing it with the reclaimed GFN and the preregistered memory. This callback can then be used to notify the VM OS about the upcoming reclamation as well as inform the hypervisor whether the page can be discarded as its content is not needed. If the VM chooses to discard the page, refaulting the page would be faster, making it in the best interest of the VM to report truthfully.

	baseline	hypercallbacks
runtime (s)	31.49	9.48
reclaim time (s)	8.18	1.30

Table 2: Runtime of file reading and the following reclamation time when memory is overcommitted, with and without hypercallbacks.

In our prototype, we implemented a callback for Linux, which discards, according to the VM OS page metadata (`struct page`), free and unmapped page-cache memory. During startup, the Linux VM registers its callback and the memory regions that are needed for reclamation decisions. When this callback decides to allow the host to discard unmapped page-cache memory, it marks the page as discarded in its metadata. The Linux VM is modified to check this flag later when it fetches a page from the page-cache and frees it accordingly.

We validate our implementation using the SysBench benchmark [45] to sequentially read files (1GB in total) after a transient memory pressure. To do so we artificially limit the VM memory, causing it to drop from 2GB to 512MB. We measure the memory reclamation time and the benchmark runtime. Table 2 shows the results, comparing the baseline swapping mechanism to one that also employs hypercallbacks. As shown, hypercallbacks reduce the runtime by 70% and the reclamation time by 86%.

It is noteworthy that these results were obtained after the Linux page-frame reclamation mechanism already scanned the VM memory following a previous memory pressure. The overheads of this mechanism are significant, suggesting it is a suitable candidate for hypercallback optimization.

7 CONCLUSION

While hardware assisted virtualization provides relatively high security, its performance lags behind alternative approaches, such as containers. Existing paravirtualization techniques do not suffice as they require context switches that induce high overheads. We have shown hypercallbacks, which can run efficiently in the hypervisor, may reduce virtualization overheads, and have demonstrated a prototype using memory management. We have also shown that the requirements for hypercallbacks differ from previous work on safe OS extensions, and defining those requirements will be necessary to extract the maximum performance from hypercallbacks.

REFERENCES

[1] Carl A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review (OSR)*, 36:181–194, 2002.

[2] Jonathan Corbet. BPF: the universal in-kernel virtual machine. LWN.nethttps://lwn.net/Articles/599755/, 2014.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[4] Andrew Whitaker, Marianne Shaw, and Steven D Gribble. Scale and performance in the Denali isolation kernel. *ACM SIGOPS Operating Systems Review (OSR)*, 36(SI):195–209, 2002.

[5] Zach Amsden, Daniel Arai, Daniel Hecht, Anne Holler, and Pratap Subrahmanyam. Vmi: An interface for paravirtualization. In *Ottawa Linux Symposium (OLS)*, pages 371–386, 2006.

[6] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Network & Distributed System Security Symposium (NDSS)*, volume 3, pages 191–206, 2003.

[7] Grzegorz Miłós, Derek G Murray, Steven Hand, and Michael A Fetterman. Satori: Enlightened page sharing. In *USENIX Annual Technical Conference (ATC)*, 2009.

[8] Justin M Forbes. Why virtualization fragmentation sucks. In *Ottawa Linux Symposium (OLS)*, volume 1, pages 125–130, 2007.

[9] Linus Torvalds. Re: Xen is a feature. http://yarchive.net/comp/linux/xen.html, 2009.

[10] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazieres, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *OSDI*, pages 335–348, 2012.

[11] Aleksandar Milenkoski, Marco Vieira, Bryan D Payne, Nuno Antunes, and Samuel Kounev. Technical information on vulnerabilities of hypercall handlers. Technical Report SPEC-RG-2014-001, Spec Research, 2014.

[12] Aaron Adams. Some notes about the Xen XSA-122 bug. https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2015/may/some-notes-about-the-xen-xsa-122-bug/, 2015.

[13] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005–30, Fakultät für Informatik, Universität Karlsruhe (TH), 2005.

[14] XenParavirtOps. https://wiki.xenproject.org/wiki/XenParavirtOps, 2016.

[15] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*, 2007.

[16] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR)*, 42(5):95–103, 2008.

[17] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference (ATC)*, page 1, 2006.

[18] Nadav Amit, Dan Tsafrir, and Assaf Schuster. VSwapper: A memory swapper for virtualized environments. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 349–366, 2014.

[19] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *USENIX Annual Technical Conference (ATC)*, 2011.

[20] Gleb Natapov. Asynchronous page faults - AIX did it. www.linux-kvm.org/wiki/images/a/ac/2010-forum-Async-page-faults.pdf.

[21] Kapil Arya, Yury Baskakov, and Alex Garthwaite. Tesseract: reconciling guest I/O and hypervisor swapping in a VM. In *ACM SIGPLAN Notices*, volume 49, pages 15–28. ACM, 2014.

[22] Balbir Singh. Page/slab cache control in a virtualized environment. In *Ottawa Linux Symposium (OLS)*, volume 1, pages 252–262, 2010.

[23] Dan Magen heimer, Chris Mason, Dave McCracken, and Kurt Hackel. Transcendent memory and Linux. In *Ottawa Linux Symposium (OLS)*, pages 191–200, 2009.

[24] Martin Schwidefsky, Hubertus Franke, Ray Mansell, Himanshu Raj, Damian Osisek, and JongHyuk Choi. Collaborative memory management in hosted Linux environments. In *Ottawa Linux Symposium (OLS)*, volume 2, pages 313–328, 2006.

[25] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *International Symposium on High Performance Distributed Computer (HPDC)*, pages 15–26, 2012.

[26] Orathai Sukwong and Hyong S. Kim. Is co-scheduling too expensive for SMP VMs? In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 257–272, 2011.

[27] Thomas Friebel. How to deal with lock-holder preemption. Xen Summit http://www.amd64.org/fileadmin/user_upload/pub/LHP-slides.pdf, 2008.

[28] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Opportunistic spinlocks: Achieving virtual machine scalability in the clouds. *ACM SIGOPS Operating Systems Review (OSR)*, 50(1):9–16, 2016.

[29] Jiannan Ouyang, John R Lange, and Haoqiang Zheng. Shoot4U: Using VMM assists to optimize TLB operations on preempted vCPUs. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, 2016.

[30] Joergen Gross. Patch: x86: reduce paravirtualized spinlock overhead. Xen mailing list, https://lists.gt.net/xen/devel/379478, 2015.

[31] Chuliang Weng, Zhigang Wang, Minglu Li, and Xinda Lu. The hybrid scheduling framework for virtual machine systems. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 111–120, 2009.

[32] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. Schedule processes, not VCPUs. In *ACM Asia-Pacific Workshop on Systems (APSys)*, 2013.

[33] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.

[34] Jiuxing Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[35] George C Necula and Peter Lee. Safe kernel extensions without run-time checking. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, volume 96, pages 229–243, 1996.

[36] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 45–60, 2006.

[37] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 75–88, 2006.

[38] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2009.

[39] Jonathan Corbet. Extending extended BPF. LWN.nethttps://lwn.net/Articles/603983/, 2014.

[40] Keegan McAllister.

[41] Elena Reshetova, Filippo Bonazzi, and N. Asokan. Randomization cant stop BPF JIT spray. In *Black Hat Europe*, 2016.

[42] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. Selective hardware/software memory virtualization. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 217–226, 2011.

[43] Henry Wong. TLB and pagewalk coherence in x86 processors. http://blog.stuffedcow.net/2015/08/pagewalk-coherence/, 2016.

[44] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux Virtual Machine Monitor. *Ottawa Linux Symposium (OLS)*, 2007.

[45] Alexey Kopytov. SysBench: a system performance benchmark. sysbench.sourceforge.net.