Leveraging Hyperupcalls To Bridge The Semantic Gap: An Application Perspective

Michael Wei
VMware Research

Nadav Amit
VMware Research

Abstract

Hyperupcalls are a mechanism which we recently proposed to bridge the semantic gap between a hypervisor and its guest virtual machines (VMs) by allowing the guest VM to provide the hypervisor safe, verifiable code to transfer information. With a hyperupcall, a hypervisor can safely read and update data structures inside the guest, such as page tables. A hypervisor could use such a hyperupcall, for example, to determine which pages are free and can be reclaimed in the VM without invoking it.

In this paper, we describe hyperupcalls and how they have been used to improve and gain additional insight on virtualized workloads. We also observe that complex applications such as databases hold a wealth of semantic information which the systems they run on top of are unaware of. For example, a database may store records, but the operating system can only observe bytes written into a file, and the hypervisor beneath it blocks written to a disk, limiting the optimizations the system may make: for instance, if the operating system understood the database wished to fetch a database record, it could prefetch related records. We explore how mechanisms like hyperupcalls could be used from the perspective of an application, and demonstrate two use cases from an application perspective: one to trace events in both the guest and hypervisor simultaneously and another simple use case where a database installs a hyperupcall so the hypervisor can prioritize certain traffic and improve response latency.

1 Introduction

Previously, we have described Hyperupcalls [4], a tool used to bridge the *semantic gap* in virtualization, where one side of an abstraction (virtual machines) must be oblivious to the other (hypervisors). The abstraction of a virtual machine (VM), enables hosts known as *hypervisors* to run multiple operating systems (OSs) known as *guests* simultaneously, each under the illusion that they are running in their own physical machine. This is achieved by exposing a hardware interface which mimics that of true, physical hardware. The introduction of this simple abstraction has led to the rise of the modern data center and the cloud as we know it today. Unfortunately, virtualization is not without drawbacks. Although the goal of virtualization is for VMs and hypervisors to be oblivious from each other, this separation renders both sides unable to understand decisions made on the other side, a problem known as the semantic gap.

The semantic gap is not limited to virtualization - it exists whenever an abstraction is used, since the purpose of the abstraction is to hide implementation details. For example, in databases, a SQL query has limited insight into how the query planner may execute it, and the query planner has limited insight into the application that sent the SQL query. Insight into the application may help the database execute the query optimally. If the database could see that an application was overloaded, it might be able to delay execution of the query and use the resources for other applications. Likewise, if the application could see that a column was being reindexed, it may perform the query on another column, or defer that query for later.

In virtualization, addressing the semantic gap is critical for performance. Without information about decisions made in guests, hypervisors may suboptimally allocate resources. For example, the hypervisor cannot know what memory is free in guests without understanding their internal OS state, breaking the VM abstraction. As a result, in virtualized environments, many mechanisms have been developed to bridge the semantic gap. State-of-the-art hypervisors today typically bridge the semantic gap with *paravirtualization* [10, 35], which makes the guest aware of the hypervisor. Several paravirtual mechanisms exist and are summarized in Table 1. Another mechanism hypervisors may leverage is *introspection*, which enables the hypervisor to observe the behavior of virtual machines without prior coordination.

Each one of these mechanisms used in virtualization may be used elsewhere where the semantic gap exists. For example, a paravirtual mechanism such as a hypercall might be similar to an application informing a database using an RPC call, whereas an upcall may involve a database informing an application about it's internal state. While introspection would be more complicated to implement in the context of applications such as databases, one can imagine a database attempting to infer the workload of an application by measuring its query rate, or an application trying to determine if a database is overloaded by measuring response latency.

In this paper, we describe the design and implementation of hyperupcalls [1], a technique which enables a hypervisor to communicate with a guest, like an upcall, but without a context switch, like VMI. This is achieved through the use of verified code, which enables a guest to communicate to the hypervisor in a flexible manner while ensuring that the guest cannot provide misbehaving or malicious code. Once a guest registers a hyperupcall, the hypervisor can execute it to perform actions such as locating free guest pages or running guest interrupt handlers without switching into the guest. We believe that hyperupcalls are useful from the perspective of an application - both as a tool for an application such as a database to gain insight about its clients, and for applications to use in a virtualized environment, breaking the semantic gap between the application, guest operating system and hypervisor.

Hyperupcalls are easy to build: they are written in a high level language such as C, and we provide a framework which allows hyperupcalls to share the same codebase and build system as the Linux kernel that may be generalized to other operating systems. When the kernel is compiled, a toolchain translates the hyperupcall into verifiable bytecode. This enables hyperupcalls to be easily maintained. Upon boot, the guest registers the hyperupcalls with the hypervisor, which verifies the bytecode and compiles it back into native code for performance. Once recompiled, the hypervisor may invoke the hyperupcall at any time.

We have previously shown that hyperupcalls enable a hypervisor to be proactive about resource allocation when memory is overcommitted, enhance performance when interprocessor interrupts (IPIs) are used, and enhance the security and debuggability of systems in virtual environments [4]. In this paper, we show an application use case: we design a hyperupcall which is installed by memcached to prioritize the handling and reduce the latency of certain requests.

2 Communication Mechanisms

It is now widely accepted that in order to extract the most performance and utility from virtualization, hypervisors and their guests need to be aware of one another. To that end, a number of mechanisms exist to facilitate communication between hypervisors and guests. Table 1 summarizes these mechanisms, which can be broadly characterized by the requestor, the executor, and whether the mechanism requires that the hypervisor and the

[1]Hyperupcalls were previously published as "hypercallbacks" [5].

		Paravirtual, Executed by:		**Uncoordinated**
Requestor		Hypervisor	Guest	Introspection
	Guest	*Hypercalls*	Pre-Virt [24]	HVI [42]
	HV	**Hyperupcalls**	*Upcalls*	VMI [18]

Table 1: *Hypervisor-Guest Communication Mechanisms.* Hypervisors (HV) and guests may communicate through a variety of mechanisms, which are characterized by who initiates the communication, who executes and whether the channel for communication is coordinated (paravirtual). *Italicized* cells represent channels which require context switches.

guest coordinate ahead of time. We note that many of these mechanisms have analogs in other places where the semantic gap may exist. For example, a hypercall might be similar to a query notification in a database, and an upcall might resemble an out-of-band request to a service.

In the next section, we discuss these mechanisms and describe how hyperupcalls fulfill a need for a communication mechanism where the hypervisor makes and executes its own requests without context switching. We begin by introducing state-of-the-art paravirtual mechanisms in use today.

2.1 Paravirtualization

Hypercalls and upcalls. Most hypervisors today leverage paravirtualization to communicate across the semantic gap. Two mechanisms in widespread use today are *hypercalls*, which allow guests to invoke services provided by the hypervisor, and *upcalls*, which enable the hypervisor to make requests to guests. Paravirtualization means that the interface for these mechanisms are coordinated ahead of time between hypervisor and guest [10].

One of the main drawbacks of upcalls and hypercalls is that they require a context switch as both mechanisms are executed on the opposite side of the request. As a result, these mechanisms must be invoked with care. Invoking a hypercall or upcall too frequently can result in high latencies and computing resource waste [3].

Another drawback of upcalls in particular that the requests are handled by the guest, which could be busy handling other tasks. If the guest is busy or if a guest is idle, upcalls incur the additional penalty of waiting for the guest to be free or for the guest or woken up. This can take an unbounded amount of time, and hypervisors may have to rely on a penalty system to ensure guests respond in a reasonable amount of time.

Finally, by increasing the coupling between the hypervisor and its guests, paravirtual mechanisms can be difficult to maintain over time. Each hypervisor have their own paravirtual interfaces, and each guest must implement the interface of each hypervisor. The paravirtual interface is not thin: Microsoft's paravirtual interface specification is almost 300 pages long [26]. Linux provides a variety of paravirtual hooks, which hypervisors can use to communicate with the VM [46]. Despite the effort to standardize the paravirtualization interfaces they are incompatible with each other, and evolve over time, adding features or even removing some (e.g., Microsoft hypervisor event tracing). As a result, most hypervisors do not fully support efforts to standardize interfaces and specialized OSs look for alternative solutions [25, 31].

Pre-virtualization. Pre-Virtualization [24] is another mechanism in which the guest requests services from the hypervisor, but the requests are served in the context of the guest itself. This is achieved by code injection: the guest leaves stubs, which the hypervisor fills with hypervisor code. Pre-virtualization offers an improvement over hypercalls, as they provide more flexible interface between the guest and the hypervisor. Arguably, pre-virtualization suffers from a fundamental limitation: code that runs in the guest is deprivileged and cannot perform sensitive operations, for example, accessing shared I/O devices. As a result, in pre-virtualization, the hypervisor code that runs in the guest still needs to communicate with the privileged hypervisor code using hypercalls.

2.2 Introspection

Introspection occurs when a hypervisor or guest attempts to infer information from the other context without directly communicating with it. With introspection, no interface or coordination is required. For instance, a hypervisor may attempt to infer the state of completely unknown guests simply by their memory access patterns. Another difference between introspection and paravirtualization is that no context switch occurs: all the code to perform introspection is executed in the requestor.

Virtual machine introspection (VMI). When a hypervisor introspects a guest, it is known as VMI [18]. VMI was first introduced to enhance VM security by providing intrusion detection (IDS) and kernel integrity checks from a privileged host [9, 17, 18]. VMI has also been applied to checkpointing and deduplicating VM state [1], as well as monitoring and enforcing hypervisor policies [32]. These mechanisms range from simply observing a VM's memory and I/O access patterns [22] to accessing VM OS data structures [14], and at the extreme end they may modify VM state and even directly inject processes into it [19, 15]. The primary benefits of VMI are that the hypervisor can directly invoke VMI without a context switch, and the guest does not need to be "aware" that it is inspected for VMI to function. However, VMI is fragile: an innocuous change in the VM OS, such as a hotfix which adds an additional field to a data structure could render VMI non-functional [8]. As a result, VMI tends to be a "best effort" mechanism.

HVI. Used to a lesser extent, a guest may introspect the hypervisor it is running on, known as hypervisor introspection (HVI) [42, 37]. HVI is typically employed either to secure a VM from untrusted hypervisors [38] or by malware to circumvent hypervisor security [36, 28].

2.3 Extensible OSes

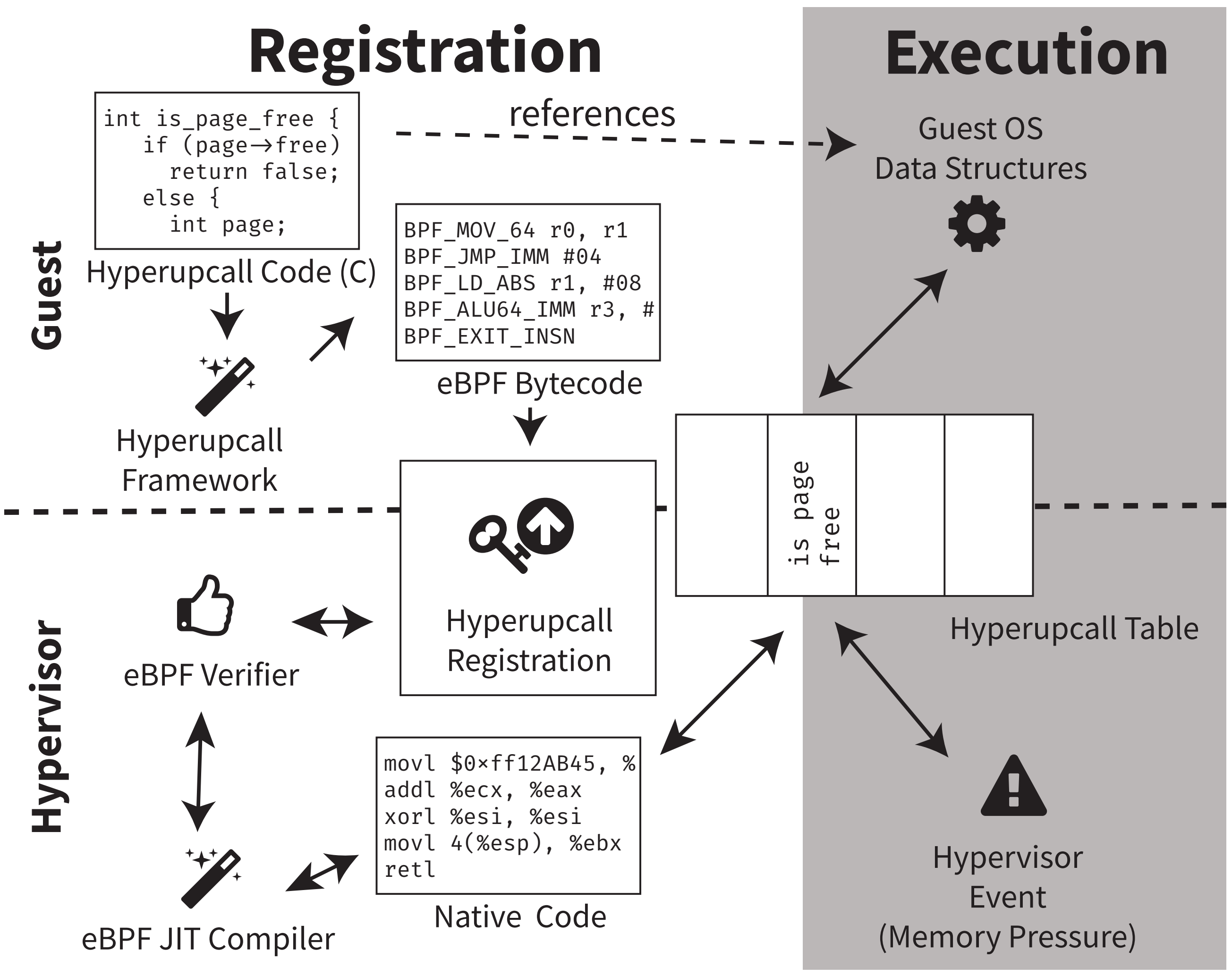
While hypervisors provide a fixed interface, OS research suggested along the years that flexible OS interfaces can improve performance without sacrificing security. The Exokernel provided low level primitives, and allowed applications to implement high-level abstractions, for example for memory management [16]. SPIN allowed to extend kernel functionality to provide application-specific services, such as specialized interprocess communication [11]. The key feature that enables these extensions to perform well without compromising security, is the use of a simple byte-code to express application needs, and running this code at the same protection ring as the kernel. Our work is inspired by these studies, and we aim to design a flexible interface between the hypervisor and guests to bridge the semantic gap.

2.4 Hyperupcalls

This paper introduces hyperupcalls, which fulfill a need for a mechanism for the hypervisor to communicate to the guest which is coordinated (unlike VMI), executed by the hypervisor itself (unlike upcalls) and does not require context switches (unlike hypercalls). With hyperupcalls, the VM coordinates with the hypervisor by registering verifiable code. This code is then executed by the hypervisor in response to events (such as memory pressure, or VM entry/exit). In a way, hyperupcalls can be thought of as upcalls executed by the hypervisor.

In contrast to VMI, the code to access VM state is provided by the guest so the hyperupcalls are fully aware of guest internal data structures— in fact, hyperupcalls are built with the guest OS codebase and share the same code, thereby simplifying maintenance while providing the OS with an expressive mechanism to describe its state to underlying hypervisors.

Compared to upcalls, where the hypervisor makes asynchronous requests to the guest, the hypervisor can execute a hyperupcall at any time, even when the guest is not running. With an upcall, the hypervisor is at the mercy of the guest, which may delay the upcall [6]. Furthermore, because upcalls operate like remote requests,



```
int is_page_free {
   if (page→free)
     return false;
   else {
     int page;
```

```
BPF_MOV_64 r0, r1
BPF_JMP_IMM #04
BPF_LD_ABS r1, #08
BPF_ALU64_IMM r3, #
BPF_EXIT_INSN
```

```
movl $0×ff12AB45, %
addl %ecx, %eax
xorl %esi, %esi
movl 4(%esp), %ebx
retl
```

Figure 1: *System Architecture.* Hyperupcall registration (left) consists of compiling C code, which may reference guest data structures, into verifiable bytecode. The guest registers the generated bytecode with the hypervisor, which verifies its safety, compiles it into native code and sets it in the VM hyperupcall table. When the hypervisor encounters an event (right), such as a memory pressure, it executes the respective hyperupcall, which can access and update data structures of the guest.

upcalls may be forced to implement OS functionality in a different manner. For example, when flushing remote pages in memory ballooning [41], the canonical technique for identifying free guest memory, the guest increases memory pressure using a dummy process to free pages. With a hyperupcall, the hypervisor can act as if it were a guest kernel thread and scan the guest for free pages directly.

Hyperupcalls resemble pre-virtualization, in that code is transferred across the semantic gap. Transferring code not only allows for more expressive communication, but it also moves the execution of the request to the other side of the gap, enhancing performance and functionality. Unlike pre-virtualization, the hypervisor cannot trust the code being provided by the virtual machine, and the hypervisor must ensure that execution environment for the hyperupcall is consistent across invocations.

In some ways, hyperupcalls may resemble SQL stored procedures: they are a block of code installed across the semantic gap, and code is more expressive than a simple request. One important distinction between hyperupcalls and stored procedures is that hyperupcalls have access to the guest state and data structures, whereas stored procedures do not. This allows hyperupcalls to be much more expressive and dynamic compared to simple code transfer mechanisms.

3 Architecture

Hyperupcalls are short verifiable programs provided by guests to the hypervisor to improve performance or provide additional functionality. Guests provide hyperupcalls to the hypervisor through a *registration* process at boot, allowing the hypervisor to access the guest OS state and provide services by *executing* them after verification. The hypervisor runs hyperupcalls in response to events or when it needs to query guest state. The architecture of hyperupcalls and the system we have built for utilizing them is depicted in Figure 1.

We aim to make hyperupcalls as simple as possible to build. To that end, we provide a complete *framework* which allows a programmer to write hyperupcalls using the guest OS codebase. This greatly simplifies the development and maintenance of hyperupcalls. The framework compiles this code into verifiable code which the guest registers with the hypervisor. In the next section, we describe how an OS developer writes a hyperupcall using our framework. Some of the implementation details, especially in regards to how we verify that the code is safe and execute the hyperupcall, are out of the scope of this paper. For more details, see [4].

3.1 Building Hyperupcalls

Guest OS developers write hyperupcalls for each hypervisor event they wish to handle. Hypervisors and guests agree on these events, for example VM entry/exit, page mapping or virtual CPU (VCPU) preemption. Each hyperupcall is identified by a predefined identifier, much like the UNIX system call interface [33].

3.1.1 Providing Safe Code

One of the key properties of hyperupcalls is that the code must be guaranteed to not compromise the hypervisor. In order for a hyperupcall to be safe, it must only be able to access a restricted memory region dictated by the hypervisor, run for a limited period of time without blocking, sleeping or taking locks, and only use hypervisor services that are explicitly permitted.

Since the guest is untrusted, hypervisors must rely on a security mechanism which guarantees these safety properties. There are many solutions that we could have chosen: software fault isolation (SFI) [40], proof-carrying code [30] or safe languages such as Rust. To implement hyperupcalls, we chose the enhanced Berkeley Packet Filter (eBPF) VM.

We chose eBPF for several reasons. First, eBPF is relatively mature: BPF was introduced over 20 years ago and is used extensively throughout the Linux kernel, originally for packet filtering but extended to support additional use cases such as sandboxing system calls (`seccomp`) and tracing of kernel events [21]. eBPF enjoys wide adoption and is supported by various runtimes [12, 29]. Second, eBPF can be provably verified to have the safety properties we require, and Linux ships with a verifier and JIT which verifies and efficiently executes eBPF code [43]. Finally, eBPF has a LLVM compiler backend, which enables eBPF bytecode to be generated from a high level language using a compiler frontend (Clang). Since OSes are typically written in C, the eBPF LLVM backend provides us with a straightforward mechanism to convert unsafe guest OS source code into verifiably safe eBPF bytecode.

3.1.2 From C to eBPF — the *Framework*

Unfortunately, writing a hyperupcall is not as simple recompiling OS code into eBPF bytecode. However, our framework aims to make the process of writing a hyperupcalls simple and maintainable as possible. The framework provides three key features that simplify the writing of hyperupcalls. First, the framework takes care of dealing with guest address translation issues so guest OS symbols are available to the hyperupcall. Second, the framework addresses limitations of eBPF, which places significant constraints on C code. Finally, the framework defines a simple interface which provides the hyperupcall with data so it can execute efficiently and safely.

Guest OS symbols and memory. Even though hyperupcalls have access to the entire physical memory of the guest, accessing guest OS data structures requires knowing where they reside. OSes commonly use kernel address space layout randomization (KASLR) to randomize the virtual offsets for OS symbols, rendering them unknown during compilation time. Our framework enables OS symbol offsets to be resolved at runtime by associating pointers using address space attributes and injecting code to adjust the pointers. When a hyperupcall is registered, the guest provides the actual symbol offsets enabling a hyperupcall developer to reference OS symbols (variables and data structures) in C code as if they were accessed by a kernel thread.

Global / Local Hyperupcalls. Not all hyperupcalls need to be executed in a timely manner. For example, notifications informing the guest of hypervisor events such as a VM-entry/exit or interrupt injection only affect the guest and not the hypervisor. We refer to hyperupcalls that only affect the guest that registered it as local, and hyperupcalls that affect the hypervisor as a whole as global. If a hyperupcall is registered as local, we relax the timing requirement and allow the hyperupcall to block and sleep. Local hyperupcalls are accounted in the vCPU time of the guest similar to a trap, so a misbehaving hyperupcall penalizes itself.

Global hyperupcalls, however, must complete their execution in a timely manner. We ensure that for the guest OS pages requested by global hyperupcalls are pinned during the hyperupcall, and restrict the memory that can be accessed to 2% (configurable) of the guest's total physical memory. Since local hyperupcalls may block, the memory they use does not need to be pinned, allowing local hyperupcalls to address all of guest memory.

Addressing eBPF limitations. While eBPF is expressive, the safety guarantees of eBPF bytecode mean that it is not Turing-complete and limited, so only a subset of C code can be compiled into eBPF. The major limitations of eBPF are that it does not support loops, the ISA does not contain atomics, cannot use self-modifying code, function pointers, static variables, native assembly code, and cannot be too long and complex to be verified.

One of the consequences of these limitations is that hyperupcall developers must be aware of the code complexity of the hyperupcall, as complex code will fail the verifier. While this may appear to be an unintuitive restriction, other Linux developers using BPF face the same restriction, and we provide a helper functions in our framework to reduce complexity, such as `memset` and `memcpy`, as well as functions that perform native atomic operations such as `cmpxchg`. A selection of these helper functions is shown in Table 2. In addition, our framework masks memory accesses (§3.4), which greatly reduces the complexity of verification. In practice, as long as we were careful to unroll loops, we did not encounter verifier issues while developing the use cases in (§4) using a setting of 4096 instructions and a stack depth of 1024.

Hyperupcall interface. When a hypervisor invokes a hyperupcall, it populates a context data structure, shown in Table 3. The hyperupcall receives an `event` data structure which indicates the reason the callback was called, and a pointer to the guest (in the address space of the hypervisor, which is executing the hyperupcall). When the hyperupcall completes, it may return a value, which can be used by the hypervisor.

Writing the hyperupcall. With our framework, OS developers write C code which can access OS variables and data structures, assisted by the helper functions of the framework. A typical hyperupcall will read the `event` field, read or update OS data structures and potentially return data to the hypervisor. Since the hyperupcall is part of the OS, the developers can reference the same data structures used by the OS itself—for example, through header files. This greatly increases the maintainability of hyperupcalls, since data layout changes are synchronized between the OS source and the hyperupcall source.

Helper Name	Function
`send_vcpu_ipi`	Send an interrupt to VCPU
`get_vcpu_register`	Read a VCPU register
`set_vcpu_register`	Read a VCPU register
`memcpy`	`memcpy` helper function
`memset`	`memset` helper function
`cmpxchg`	compare-and-swap
`flush_tlb_vcpu`	Flush VCPU's TLB
`get_exit_info`	Get info on an `VM_EXIT` event

Table 2: *Selected hyperupcall helper functions.* The hyperupcall may call these functions implemented in the hypervisor, as they cannot be verified using eBPF.

Input field	Function
`event`	Event specific data including event ID.
`hva`	Host virtual address (HVA) in which the guest memory is mapped.
`guest_mask`	Guest address mask to mask bits which are higher than the guest memory address-width. Used for verification (§3.4).
`vcpus`	Pointers to the hypervisor VCPU data structure, if the event is associated with a certain VCPU, or a pointer to the guest OS data structure. Inaccessible to the hyperupcall, but used by helper functions.
`vcpu_reg`	Frequently accessed VCPU registers: instruction pointer and VCPU ID.
`env`	Environment variables, provided by the guest during hyperupcallregistration. Used to set address randomization offsets.

Table 3: *Hyperupcall context data.* These fields are populated by the hypervisor when a hyperupcall is called.

It is important to note that a hyperupcall cannot invoke guest OS functions directly, since that code has not been secured by the framework. However, OS functions can be compiled into hyperupcalls and be integrated in the verified code.

3.2 Compilation

Once the hyperupcall has been written, it needs to be compiled into eBPF bytecode before the guest can register it with the hypervisor. Our framework generates this bytecode as part of the guest OS build process by running the hyperupcall C code through Clang and the eBPF LLVM backend, with some modifications to assist with address translation and verification:

Guest memory access. To access guest memory, we use eBPF's direct packet access (DPA) feature, which was designed to allow programs to access network packets safely and efficiently without the use of helper functions. Instead of passing network packets, we utilize this feature by treating the guest as a "packet". Using DPA in this manner required a bug fix [2] to the eBPF LLVM backend, as it was written with the assumption that packet sizes are $\leq$64KB.

Address translations. Hyperupcalls allow the hypervisor to seamlessly use guest virtual addresses (GVAs), which makes it appear as if the hyperupcall was running in the guest. However, the code is actually executed by the hypervisor, where host virtual address (HVAs) are used, rendering guest pointers invalid. To allow the use of guest pointers transparently in the host context, these pointers therefore need to be translated from GVAs into HVAs. We use the compiler to make these translations.

To make this translation simple, the hypervisor maps the GVA range contiguously in the HVA space, so address translations can easily be done by adjusting the base address. As the guest might need the hyperupcall to access multiple contiguous GVA ranges—for example, one for the guest 1:1 direct mapping and of the OS text section [23]—our framework annotates each pointer with its respective “address space” attribute. We extend the LLVM compiler to use this information to inject eBPF code that converts each of the pointer from GVA to HVA by a simple subtraction operation. It should be noted that the generated code safety is not assumed by the hypervisor and is verified when the hyperupcall is registered.

Bound Checks. The verifier rejects code with direct memory accesses unless it can ensure the memory accesses are within the “packet” (in our case, guest memory) bounds. We cannot expect the hyperupcall programmer to perform the required checks, as the burden of adding them is substantial. We therefore enhance the compiler to automatically add code that performs bound checks prior to each memory access, allowing verification to pass. As we note in Section 3.4, the bounds checking is done using masking and not branches to ease verification.

Context caching. Our compiler extension introduces intrinsics to get a pointer to the context or to read its data. The context is frequently needed along the callback for calling helper functions and for translating GVAs. Delivering the context as a function parameter requires intrusive changes and can prevent sharing code between the guest and its hyperupcall. Instead, we use the compiler to cache the context pointer in one of the registers and retrieve it when needed.

3.3 Registration

After a hyperupcall is compiled into eBPF bytecode, it is ready to be registered. Guests can register hyperupcalls at any time, but most hyperupcalls are registered when the guest boots. The guest provides the hyperupcall event ID, hyperupcall bytecode and the virtual memory the hyperupcall will use.

3.4 Verification

The hypervisor verifies that each hyperupcall is safe to execute at registration time. Our verifier is based on the Linux eBPF verifier and checks three properties of the hyperupcall: memory accesses, number of runtime instructions, and helper functions used.

3.5 Execution

Once the hyperupcall is compiled, registered and verified, it may be executed by the hypervisor in response to an event. There are some complexities to executing hyperupcalls, for accessing remote CPU states and dealing with locks. In general, the hypervisor can run the hyperupcall to obtain information about the guest without waiting on a response from the guest.

4 Use Cases and Evaluation

Previously, we presented several use cases for hyperupcalls which primarily involved the guest operating system: enabling a hypervisor to be proactive about resource allocation when memory is overcommitted, enhancing performance when interprocessor interrupts (IPIs) are used, and increasing the security and debuggability of systems in virtual environments [4]. From the application perspective, we also provided a modified version of the `ftrace` utility to help application developers see both hypervisor and guest events in a unified view. In this section, we focus on the application use cases of hyperupcalls: first we present both the frace utility previously presented and a new use case where we enable memcached to install a hyperupcall to prioritize certain traffic. This enables the hypervisor to safely understand application traffic and perform actions based on application state without coupling the hypervisor with application code.

4.1 Unified Event Tracing

Event tracing is an important tool for debugging correctness and performance issues. However, collecting traces for virtualized workloads is somewhat limited. Traces collected inside a guest do not show hypervisor events, such as when a VM-exit is forced, which can have significant effect on performance. For traces that are collected in the hypervisor to be informative, they require knowledge about guest OS symbols [13]. Such traces cannot be collected in cloud environments. In addition, each trace collects only part of the events and does not show how guest and hypervisor events interleave.

To address this issue, we run the Linux kernel tracing tool, `ftrace` [34], inside a hyperupcall. `Ftrace` is well suited to run in a hyperupcall. It is simple, lockless, and built to enable concurrent tracing in multiple contexts: non-maskable interrupt (NMI), hard and soft interrupt handlers and user processes. As a result, it was easily be adapted to trace hypervisor events concurrently with guest events. Using the `ftrace` hyperupcall, the guest can trace both hypervisor and guest events in one unified log, easing debugging. Since tracing all events use only guest logic, new OS versions can change the tracing logic, without requiring hypervisor changes.

Tracing is efficient, despite the hyperupcallcomplexity (3308 eBPF instructions), as most of the code deals with infrequent events that handles situations in which trace pages fill up. Tracing using hyperupcalls is slower than using native code by 232 cycles, which is still considerably shorter time than the time a context switch between the hypervisor and the guest takes.

Tracing is a useful tool for performance debugging, which can expose various overheads [47]. For example, by registering the `ftrace` on the VM-exit event, we see that many processes, including short-lived ones, trigger multiple VM exits due to the execution of the `CPUID` instruction, which enumerates the CPU features and must be emulated by the hypervisor. We found that the GNU C Library, which is used by most Linux applications, uses `CPUID` to determine the supported CPU features. This overhead could be prevented by extending Linux virtual dynamic shared object (vDSO) for applications to query the supported CPU features without triggering an exit.

4.2 Scheduler Activation

Our scheduler activation hyperupcall prototype performs scheduler activation by increasing the virtual machine priority for a short time when a packet which matches a condition arrives to the guest. memcached registers a hyperupcall with the guest, which in turn registers a hyperupcall with the hypervisor on a guest packet receiving event. In our prototype implemetation, this hyperupcall simply boosts the VM priority using a helper function, but we could have also performed other actions such as inspect the packet contents or access guest data structures. We increase the VM priority (`cpu.weight`) from a default value of 10 to 100 for 100ms.

Figure 2 shows the results. We used a memcached server in a guest with a single VCPU and increased the level of overcommittment of the physical CPU the machine was running on. The guest and multiple background tasks (iperf) were executed in different `cgroups` with equal priority. We assume in our experiment that the same user owns all the different guests. A workload generator (memcslap) was dispatched every second to issue 100 requests to the VM. Each experiment as conducted 50 times and the average execution time and standard deviation are shown.

Overall, this use case demonstrates that an application can use hyperupcalls to prioritize requests to the guest in an application-specific manner, greatly reducing latency and variability. We believe that we have only scratched the surface, and there are many other use cases of hyperupcalls which can be used to enhance applications. Since hyperupcalls can seamlessly integrate into the codebase of the application and are able to access guest state, developing new use cases is greatly simplified. We are currently working on making it easier for the hyperupcall to access application state, as well as methods for the guest OS to safely register multiple hyperupcalls on the same event.

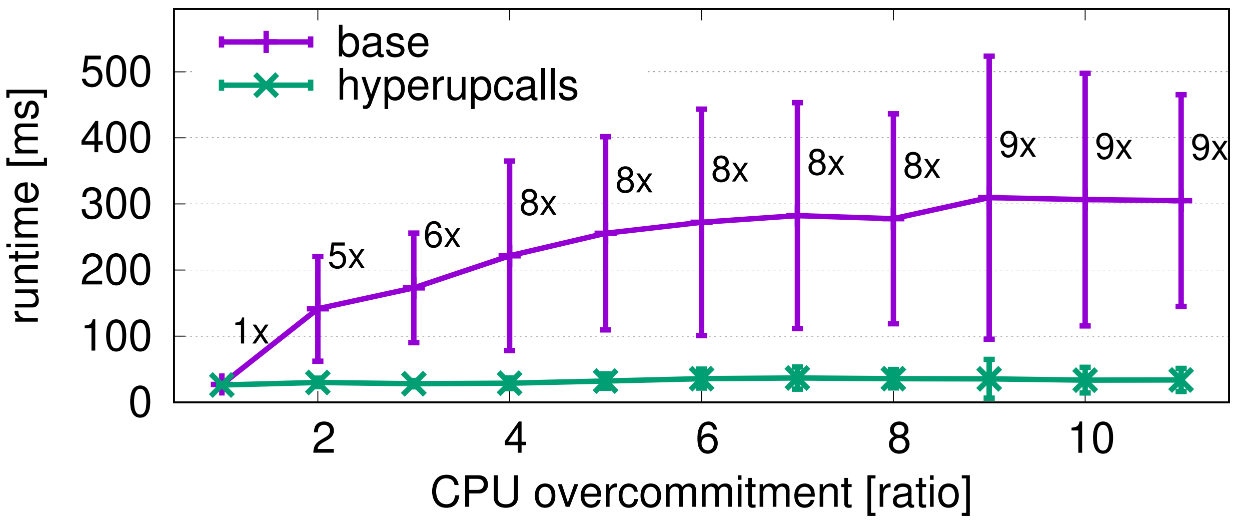


Figure 2: The runtime of 100 memcached requests with varying level of CPU overcommitment, with and without scheduler activation hyperupcalls. The slowdown ratio is presented above the lines.

5 Availability

We are in the process of open sourcing the hyperupcalls infrastructure and intend to make it available to the public soon. In the meantime, interested parties may contact the authors to obtain a pre-release version. Currently, the hyperupcalls infrastructure we intend to release is built for Linux virtual machines, and is integrated into the source code of the Linux kernel. Additional work is necessary to fully expose this interface to other applications.

There are several contexts which the hyperupcalls infrastructure could be used in other applications. An application wishing to install its own hyperupcalls, as in the example use cases we developed, could do so using an interface provided by the operating system. However, the operating system would need to have a mechanism for multiplexing hyperupcalls registered to the same event, or not support multiplexing at all. Another context that hyperupcalls could be used by applications is for applications to use the hyperupcalls concept of running verified trusted code. Our infrastructure does not yet support this, but could serve as an example system for leveraging eBPF in applications to run code to bridge the semantic gap.

6 Conclusion

Bridging the semantic gap is critical for performance and for the hypervisor to provide advanced services to guests. Hypercalls and upcalls are currently used to bridge the gap, but they have several drawbacks: hypercalls cannot be initiated by the hypervisor, upcalls do not have a bounded runtime, and both incur the penalty of context switches. Introspection, an alternative which avoids context switches can be unreliable as it relies on observations instead of an explicit interface. Hyperupcalls overcome these limitations by allowing the guest to expose its *logic* to the hypervisor, avoiding a context switch by enabling the hyperupcall to safely execute guest logic directly.

We have built a complete infrastructure for developing hyperupcalls which allow developers to easily add new paravirtual features using the codebase of the OS. This infrastructure could easily be extended to be used by applications as well, enabling applications to provide the hypervisor insight into their internal state. They can also be used to bridge the semantic gap outside of virtualization, for example, for services such as databases to gain more insight about the applications which use them.

References

[1] Ferrol Aderholdt, Fang Han, Stephen L Scott, and Thomas Naughton. Efficient checkpointing of virtual machines using virtual machine introspection. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 414–423, 2014.

[2] Nadav Amit. Patch: Wrong size-extension check in BPFDAGToDAGISel::SelectAddr. `https://lists.iovisor.org/pipermail/iovisor-dev/2017-April/000723.html`, 2017.

[3] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, 2011.

[4] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (ATC)*, pages 97–112, 2018.

[5] Nadav Amit, Michael Wei, and Cheng-Chun Tu. Hypercallbacks: Decoupling policy decisions and execution. In *ACM Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 37–41, 2017.

[6] 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, 2014.

[7] Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. System performance evaluation of para virtualization, container virtualization, and full virtualization using Xen, OpenVX, and Xenserver. In *IEEE International Conference on Advances in Computing and Communications (ICACC)*, pages 247–250, 2014.

[8] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *IEEE Symposium on Reliable Distributed Systems*, pages 82–91, 2010.

[9] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(5):670–684, 2011.

[10] 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.

[11] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the spin operating system. *ACM SIGOPS Operating Systems Review (OSR)*, 29(5):267–283, 1995.

[12] Big Switch Networks. Userspace eBPF VM. `https://github.com/iovisor/ubpf`, 2015.

[13] Martim Carbone, Alok Kataria, Radu Rugina, and Vivek Thampi. VProbes: Deep observability into the ESXi hypervisor. VMware Technical Journal `https://labs.vmware.com/vmtj/vprobes-deep-observability-into-the-esxi-hypervisor`, 2014.

[14] Andrew Case, Lodovico Marziale, and Golden G Richard. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7:S32–S40, 2010.

[15] Tzi-cker Chiueh, Matthew Conover, and Bruce Montague. Surreptitious deployment and execution of kernel agents in windows guests. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 507–514, 2012.

[16] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. *Exokernel: An operating system architecture for application-level resource management*, volume 29. 1995.

[17] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *IEEE Symposium on Security and Privacy (SP)*, pages 586–600, 2012.

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

[19] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process implanting: A new active introspection framework for virtualization. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 147–156, 2011.

[20] Nur Hussein. Randomizing structure layout. `https://lwn.net/Articles/722293/`, 2017.

[21] IO Visor Project. BCC - tools for BPF-based Linux IO analysis, networking, monitoring, and more. `https://github.com/iovisor/bcc`, 2015.

[22] 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)*, pages 1–14, 2006.

[23] Andi Kleen. Linux virtual memory map. Linux-4.8:Documentation/x86/x86_64/mm.txt, 2004.

[24] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. *Pre-virtualization: Slashing the cost of virtualization*. Universität Karlsruhe, Fakultät für Informatik, 2005.

[25] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, volume 48, pages 461–472, 2013.

[26] Microsoft. Hypervisor top level functional specification v5.0b, 2017.

[27] Aleksandar Milenkoski, Bryan D Payne, Nuno Antunes, Marco Vieira, and Samuel Kounev. Experience report: an analysis of hypercall handler vulnerabilities. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111, 2014.

[28] Preeti Mishra, Emmanuel S Pilli, Vijay Varadharajan, and Udaya Tupakula. Intrusion detection techniques in cloud environment: A survey. *Journal of Network and Computer Applications*, 77:18–47, 2017.

[29] Quentin Monnet. Rust virtual machine and JIT compiler for eBPF programs. `https://github.com/qmonnet/rbpf`, 2017.

[30] George C Necula. Proof-carrying code. In *ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL)*, pages 106–119, 1997.

[31] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library OS from the top down. In *ACM SIGPLAN Notices*, volume 46, pages 291–304, 2011.

[32] Adit Ranadive, Ada Gavrilovska, and Karsten Schwan. Ibmon: monitoring VMM-bypass capable infiniband devices using memory introspection. In *Workshop on System-level Virtualization for HPC (HPCVirt)*, pages 25–32, 2009.

[33] Dannies M. Ritchie and Ken Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1929, 1978.

[34] Steven Rostedt. Debugging the kernel using Ftrace. LWN.net, `http://lwn.net/Articles/365835/`, 2009.

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

[36] Joanna Rutkowska and Alexander Tereshkin. Bluepilling the Xen hypervisor. *BlackHat USA*, 2008.

[37] Jiangyong Shi, Yuexiang Yang, and Chuan Tang. Hardware assisted hypervisor introspection. *SpringerPlus*, 5(1):647, 2016.

[38] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing NFV states by using SGX. In *ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV)*, pages 45–48, 2016.

[39] VMware. open-vm-tools. `https://github.com/vmware/open-vm-tools`, 2017.

[40] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review (OSR)*, volume 27, pages 203–216, 1994.

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

[42] Gary Wang, Zachary John Estrada, Cuong Manh Pham, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Hypervisor introspection: A technique for evading passive virtual machine monitoring. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[43] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 33–47, 2014.

[44] Rafal Wojtczuk. Analysis of the attack surface of Windows 10 virtualization-based security. BlackHat USA, 2016.

[45] Britta Wuelfing. Kroah-Hartman: Remove Hyper-V driver from kernel? Linux Magazine `http://www.linux-magazine.com/Online/News/Kroah-Hartman-Remove-Hyper-V-Driver-from-Kernel`, 2009.

[46] Xen Project. XenParavirtOps. `https://wiki.xenproject.org/wiki/XenParavirtOps`, 2016.

[47] Haoqiang Zheng and Jason Nieh. WARP: Enabling fast CPU scheduler development and evaluation. In *IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS)*, pages 101–112, 2009.