SymCall: Symbiotic Virtualization Through VMM-to-Guest Upcalls

John R. Lange
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
jacklange@cs.pitt.edu

Peter A. Dinda
Department of EECS
Northwestern University
Evanston, IL 60208
pdinda@northwestern.edu

Abstract

Symbiotic virtualization is a new approach to system virtualization in which a guest OS targets the native hardware interface as in full system virtualization, but also optionally exposes a software interface that can be used by a VMM, if present, to increase performance and functionality. Neither the VMM nor the OS needs to support the symbiotic virtualization interface to function together, but if both do, both benefit. We describe the design and implementation of the *SymCall* symbiotic virtualization interface in our publicly available Palacios VMM for modern x86 machines. SymCall makes it possible for Palacios to make clean upcalls into a symbiotic guest, much like system calls. One use of symcalls is to collect semantically rich guest data to enable new VMM features. We describe the implementation of *SwapBypass*, a VMM-based service based on SymCall that reconsiders swap decisions made by a symbiotic Linux guest, adapting to guest memory pressure given current constraints. Finally, we present a detailed performance evaluation of both SwapBypass and SymCall.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

General Terms Design, Experimentation, Measurement, Performance

Keywords virtual machine monitors, operating systems

1. Introduction

Virtualization is rapidly becoming ubiquitous, especially in large-scale data centers. Significant inroads have also been made into high performance computing and adaptive systems [14]. The rapid adoption of virtualization in all of these areas is in no small part due to the ability of virtualization to adapt existing OSes to virtual environments with few or no OS implementation changes.

A consequence of this compatibility is that current virtualization interfaces are largely designed to be purely unidirectional. A guest OS is only able to interact with a VMM through a virtualized hardware interface (e.g., VMware [26], KVM [21]) or via paravirtualized hypercalls (e.g. Xen [3]). The only way a VMM can signal a guest OS is through interrupts or interfaces built on top of a hardware device abstraction. Despite being easy to implement and widely compatible, such interfaces can be very restrictive.

This has led to considerable effort being put into bridging the *semantic gap* [4] of the VMM/OS interface [7, 10, 11]. The advantage of the approaches proposed in previous efforts is that they operate without guest modifications and do not require guest cooperation. However, the information gleaned from such black-box or grey-box approaches is semantically poor, which restricts the kinds of decision making that the VMM can do, or services it can offer. Furthermore, even when these tools can reconstruct needed guest state, the reconstruction effort can be a significant burden on the VMM or service developer, especially given that it is probably already readily available inside the guest environment.

We propose *symbiotic virtualization*, a new approach to designing VMMs and guest OSes to work better together. In symbiotic virtualization, a cooperative symbiotic guest kernel can detect that it is running on a symbiotic VMM, synchronize with it, and then offer it access to semantically rich information, and, more generally, to what amounts to a system call interface that the VMM can use to interact with the guest kernel. This interface can then be used by the VMM and its associated services. If no symbiotic VMM, or no VMM, is available, a symbiotic guest kernel behaves normally. Symbiotic virtualization preserves the benefits of full system virtualization, while providing a standard interface for VMM/OS interactions. A symbiotic interface is not a required mechanism for OS functionality, nor is a VMM required to use it if it is present.

In this paper, we focus on the SymCall symbiotic interface. SymCall allows the VMM to make upcalls into the guest kernel, making it possible for a guest to easily provide an efficient and safe system call interface to the VMM. These calls can then be used during the handling of a guest exit. That is, the VMM can invoke the guest *during the handling of a guest exit.* We describe the design and implementation of SymCall in Palacios in considerable detail, and we evaluate the latency of SymCall.

Using the SymCall interface, we designed, implemented, and evaluated a proof-of-concept symbiotic service in Palacios. This service, *SwapBypass*, uses shadow paging to reconsider swapping decisions made by a symbiotic Linux guest running in a VM. If the guest is experiencing high memory pressure relative to its memory partition, it may decide to swap a page out. However, if the VMM has available physical memory, this is unnecessary. Although a page may be swapped out and marked unavailable in the guest page table, SwapBypass can also keep the page in memory and mark it available in the shadow page table. The effect is that access to the

This effort was funded by the United States National Science Foundation (NSF) via grant CNS-0709168, and the Department of Energy (DOE) via grant DE-SC0005343.

VEE'11, March 9–11, 2011, Newport Beach, California, USA.

"swapped" page is at main memory speeds, and that the guest is using more physical memory than initially allotted, even if it is incapable of dynamically adapting to changing physical memory size.

Implementing SwapBypass requires information about the mapping of swap IDs to swap devices, which is readily provided via a SymCall, but extremely challenging to glean otherwise. We evaluate SwapBypass both through performance benchmarks, and through an examination of its implementation complexity.

The contributions of this paper include:

- The definition of symbiotic virtualization.
- The design and implementation of the SymCall framework to support symbiotic virtualization in the Palacios VMM.
- An evaluation of the performance of SymCall.
- The design and implementation of a proof-of-concept VMM feature that leverages SymCall: SwapBypass, a service that reconsiders Linux kernel swap decisions.
- An evaluation of SwapBypass, considering both performance and implementation complexity.

Palacios VMM: Our implementation is in the context of the Palacios VMM. Palacios is an OS-independent, open source, BSD-licensed, publicly available type-I VMM whose details can be found elsewhere [13, 14]. Palacios achieves full system virtualization for x86 and x86_64 hosts and guests using either the AMD SVM or Intel VT hardware virtualization extensions and either shadow or nested paging. The entire VMM, including the default set of virtual devices is on the order of 47 thousand lines of C and assembly. When embedded into Kitten, a lightweight kernel available from Sandia National Labs, as done in this work, the total code size is on the order of 108 thousand lines. Palacios is capable of running on environments ranging from commodity Ethernet-based servers to large scale supercomputers, specifically the Red Storm Cray XT supercomputer located at Sandia National Labs.

2. Symbiotic virtualization

Symbiotic virtualization is an approach to designing VMMs and OSes such that both support, but neither requires, the other. A symbiotic OS targets a native hardware interface, but also exposes a software interface, usable by a symbiotic VMM, if present, to optimize performance and increase functionality. The goal of symbiotic virtualization is to introduce a virtualization interface that provides access to high level semantic information while still retaining the universal compatibility of a virtual hardware interface. Symbiotic virtualization is neither full system virtualization nor paravirtualization, however it can be used with either approach.

A symbiotic OS exposes two types of interfaces. The first is a passive interface, called SymSpy, that allows a symbiotic VMM to simply read out structured information that the OS places in memory. This interface has extremely low overhead, as the VMM can readily read guest memory during an exit or from a different core. However, the information is necessarily provided asynchronously with respect to exits or other VMM events. Because of this, guest information that may be useful in handling the exit may not be available at the time of the exit.

The second is a functional interface, SymCall, that allows a symbiotic VMM to invoke the guest synchronously, during exit handling or from a separate core. However, these invocations have considerably higher costs compared to the passive interface. Furthermore, the implementation complexity may be much higher for two reasons. First, the VMM must be able to correctly support reentry into the guest *in the process of handling a guest exit*. Second, from the guest's perspective, the functional interface provides an additional source of concurrency *that is not under guest control.* The VMM and guest must be carefully designed so this concurrency does not cause surprise race conditions or deadlocks.

In addition to the functional and passive interfaces, symbiotic virtualization requires a discovery protocol that the guest and VMM can run to determine which, if any, of the interfaces are available, and what data forms and entry points are available.

3. Discovery and configuration

One of the principal goals of symbiotic virtualization is to provide an enhanced interface between a VMM and an OS while still allowing compatibility with real hardware. In contrast to paravirtualization, symbiotic virtualization is designed to be enabled and configured at run time without requiring any changes to the OS. As such, symbiotic upcalls are implemented using existing hardware features, such as CPUID values and Model Specific Registers (MSRs). A guest is able to detect a symbiotic VMM at boot time and selectively enable symbiotic features that it supports. The discovery and configuration process is shown in Figure 1.

In order to indicate the presence of a symbiotic VMM we have created a virtualized CPUID value. The virtualized CPUID returns a value denoting a symbiotic VMM, an interface version number, as well as machine specific interface values to specify hypercall parameters. This maintains hardware compatibility because on real hardware the CPUID instruction simply returns an empty value indicating the non-presence of a symbiotic OS which will cause the OS to abort further symbiotic configurations[1]. If the guest does detect a symbiotic VMM then it proceeds to configure the symbiotic environment using a set of virtualized MSRs.

4. SymSpy passive interface

The SymSpy interface provides a mechanism for the sharing of structured information between the VMM and the guest OS. This information is stored in a memory region that is mapped into the guest's physical address space. The guest indicates to the VMM a guest physical address at which to map the SymSpy page. After this is completed the guest can read/write to this memory location without an exit. The VMM can also directly access the page during its execution.

SymSpy is used to enumerate what structured data types are available as well as which symbiotic services, such as specific symcalls, are supported by the guest OS. The SymSpy interface is also useful for sharing VMM state information with the guest OS. For example, Palacios uses the interface to expose the identities of PCI devices that the guest has direct access to. This allows the guest to explicitly modify its DMA addresses to account for the location of guest memory inside physical memory.

Configuring SymSpy is the second step in the symbiotic configuration process shown in Figure 1. After a guest has detected the presence of a symbiotic VMM it chooses an available guest physical memory address that is not currently in use. This address does not have to be inside the guest's currently allocated memory, and can instead be mapped into any guest physical address range that the guest OS has available. Once an address has been found the guest writes it to the SymSpy MSR, which is a special virtual MSR implemented by the VMM. The symbiotic VMM intercepts this operation, allocates a new page, and maps it into the guest at the location specified in the MSR.

[1] We use CPUID instead of a virtual MSR because accesses to non-present MSRs generate a General Protection Fault

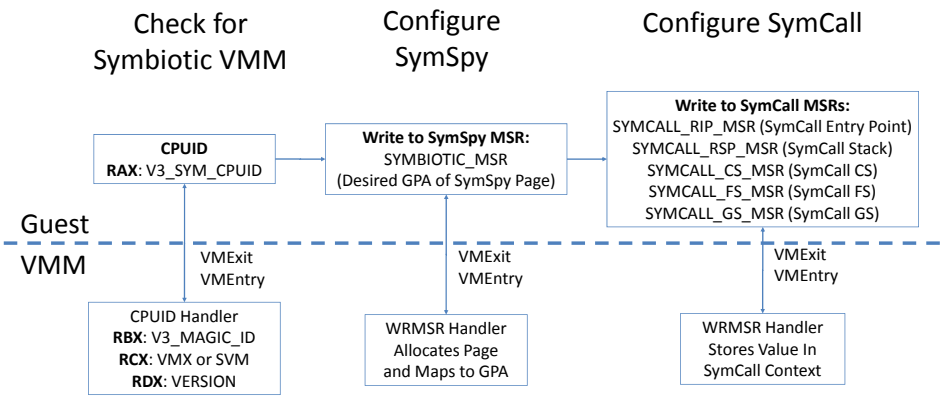

Figure 1. Symbiotic VMM discovery/configuration.

5. SymCall functional interface

SymCalls are a new VMM/guest interface by which a VMM can make synchronous upcalls into a running guest OS. In a guest OS, this interface is designed to resemble the existing system call interface as much as possible, both in terms of the hardware interface presented to the guest, as well as the internal upcall implementations. Based on the similarity to system calls we refer to symbiotic upcalls as symcalls.

5.1 Alternative upcall mechanisms

The utility of upcalls in layered architectures is well understood [6], and most existing virtualization architectures include interfaces that allow guest environments to react to requests made from the VMM layer. For example, balloon drivers are a common interface that allows a VMM to request that a guest release memory pages. These interfaces are built on top of the virtual device abstraction which is implemented using standard device I/O mechanisms: interrupts and DMA. As such, these interfaces are fundamentally different from upcalls as defined in [6], and instead rely on asynchronous signals (interrupts) to request that a VM take a guest defined action (I/O to/from a virtual device). In contrast, SymCall provides an interface that enables a VMM to directly execute a specific upcall synchronously as part of an exit handling operation.

This synchronous behavior allows a VMM to easily modify exit handler behavior based on complex information gleaned from internal guest state queries that are executed as upcalls inside the guest's context. This functionality is not possible when using a virtual device abstraction due to the asynchronous nature of the interface. Even if an exit handler were to raise an interrupt inside the guest context it would have no guarantee of when the associated interrupt handler would actually be executed. This prevents the virtual device interface from being used inside exit handlers or anywhere else where synchronous behavior is required.

5.2 SymCall architecture

The x86 architecture has a several well defined frameworks for supporting OS system calls. These interfaces allow a system call to be executed via a special instruction that instantiates a system call context defined at initialization time by the OS. When a System call instruction is executed, the context variables are copied out of a set of MSRs and instantiated on the hardware. When execution resumes the CPU is running a special OS code path that dispatches to the correct system call handler. When a system call returns it executes the corresponding exit instructions that reverse this procedure.

Due to the conceptual similarity between symcalls and system calls we designed our implementation to be architecturally similar as well. Just as with system calls, the guest OS is responsible for enabling and configuring the environment which the symcalls will execute in. It does this using a set of virtualized MSRs that are based on the actual MSRs used for the SYSCALL and SYSRET interface. When the VMM makes a symbiotic upcall, it configures the guest environment according to the values given by the guest OS. The next time the guest executes it will be running in the SymCall dispatch routine that invokes the handler for the specific symcall.

5.3 Virtual hardware support

The SymCall virtual hardware interface consists of a set of MSRs that are a union of the MSRs used for the SYSENTER and SYSCALL frameworks[2]. We combine both the MSR sets to provide a single interface that is compatible for both the Protected (32 bit) and Long (64 bit) operating modes. The set of symbiotic MSRs are:

- *SYMCALL_RIP:* The value to be loaded into the guest's RIP/EIP register. This is the address of the entry point for symcalls in the guest kernel
- *SYMCALL_RSP:* The value to be loaded into the guest's RSP/ESP register. This is the address of the top of the stack that will be used when entering a symcall.
- *SYMCALL_CS:* The location of the code segment to be loaded during a symcall. This is the code segment that will be used during the symcall. The stack segment is required to immediately follow the code segment, and so can be referenced via this MSR.
- *SYMCALL_GS:* The GS segment base address to be loaded during a symcall.
- *SYMCALL_FS:* The FS segment base address to be loaded during a symcall. The GS or FS segments are used to point to kernel-level context for the symcall.

The RIP, RSP, and CS(+SS) MSRs are needed to create the execution context for the symbiotic upcall. The FS and GS MSRs typically hold the address of the local storage on a given CPU core. FS or GS is typically used based on the operating mode of the processor.

[2] The execution model however more closely resembles the SYSCALL behavior

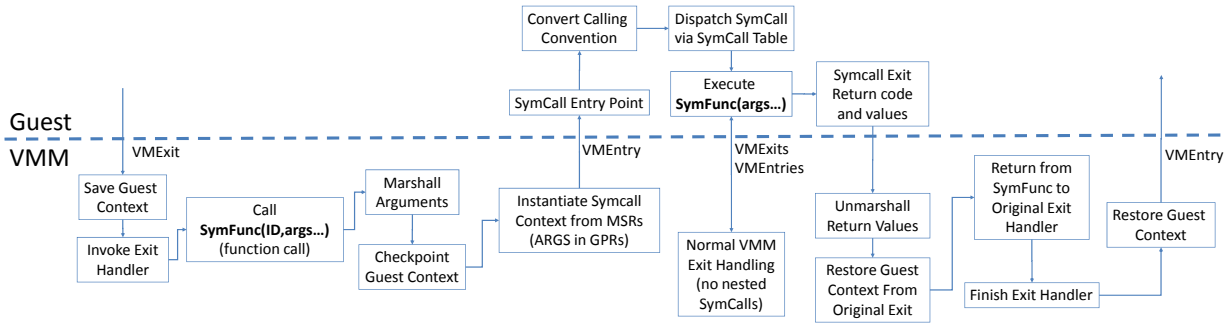

Figure 2. The execution path of the SymCall functional interface in the Palacios VMM. Symcalls execute synchronously with respect to VM exits, and allow exit handlers to optimize their behavior based on complex state information collected from queries executed inside the guest context.

Component	Lines of code
VMM infrastructure	300(C)
Guest infrastructure	211(C) + 129(ASM)
Total	511(C) + 129(ASM)

Figure 3. Lines of code needed to implement the SymCall infrastructure as measured by SLOCcount

As we stated earlier the execution model for a symbiotic upcall is based on system calls. The one notable difference is that symbiotic upcalls always store the guest state before the call is executed and reload it when the symcall returns. Furthermore the state is saved inside the VMM's address space and so is inaccessible to the guest OS. This is largely a safety precaution due to the fact that the guest OS has much less control over when a symbiotic call is executed. For example, a system call can only be executed when a process is running, but a symcall can also occur when the guest is executing in the kernel.

As we mentioned earlier, the system call return process copies back the context that existed before the system call was made (but possibly modified afterward). Returning from a symbiotic upcall is the same with the exception being that the symbiotic call always returns to the context immediately before the symcall was made. This is because the calling state is not saved in the guest environment, but instead stored by the VMM. Because there is no special instruction to return from a symcall the guest instead executes a special hypercall indicating a return from a symcall.

The virtual hardware interface we have developed follows the system call design to minimize the behavioral changes of a guest OS. Our other objective was to create an interface that would be implementable in physical hardware. Existing hardware implementations could be extended to provide hardware versions of the MSRs that would only be accessible while the CPU is executing in a VM context. A second type of VM entry could be defined which launches into the state defined by the MSRs and automatically saves the previous guest state in the virtual machine control structures. And finally a new instruction could be implemented to return from a symbiotic upcall and reload the saved guest state. In our implementation we use a special hypercall to return from a symcall.

5.4 Symbiotic upcall interface

Using the virtual hardware support, we have implemented a symbiotic upcall facility in the Palacios VMM. Furthermore we have implemented symbiotic upcall support for two guest OSes: 32 bit Linux and the 64 bit Kitten OS. Our SymCall framework supports both the Intel VMX and AMD SVM virtualization architectures. The symcalls are designed to resemble the Linux system call interface as closely as possible. We will focus our description on the Linux implementation.

Implementing the SymCall interface required modifications to both the Palacios VMM as well as the Linux kernel running as a guest. The scale of the changes is shown in Figure 3. The modifications to the guest OS consisted of 211 lines of C and 129 lines of assembly as measured by SLOCcount. This code consisted of the generic SymCall infrastructure and did not include the implementation of any symcall handlers. The VMM infrastructure consisted of an additional 300 lines of C implemented as a compile time module.

5.4.1 Guest OS support

The Linux guest implementation of the symbiotic upcall interface shares much commonality with the system call infrastructure. Symbiotic upcalls are designed to be implemented in much the same manner as a normal system call. Each symbiotic upcall is associated with a given call index number that is used to look up the appropriate call handler inside a global array. The OS loads the *SYMCALL_RIP* MSR with a pointer to the SymCall handler, which uses the value of the *RAX* General Purpose Register (GPR) as the call number. The arguments to the symcall are supplied in the remaining GPRs, which limits each symbiotic upcall to at most 5 arguments. Our current implementation does not support any form of argument overflow, though there is no inherent reason why this would not be possible. The arguments are passed by value. Return values are passed in the same way, with the error code passed in *RAX* and additional return values in the remaining GPRs. Any kernel component can register a symbiotic upcall in exactly the same way as it would register a system call.

One notable difference between symcalls and normal system calls is the location of the stack during execution. Normal system calls execute on what is known as the kernel mode stack. Every process on the system has its own copy of a kernel mode stack to handle its own system calls and possibly also interrupts. Among other things this allows context switching and kernel preemption, because each execution path running in the kernel is guaranteed to have its own dedicated stack space. This assurance is possible because processes are unable to make multiple simultaneous system calls. Symbiotic upcalls on the other hand can occur at any time, and so cannot use the current process' kernel stack. In our implementation the guest OS allocates a symbiotic stack at initialization. Every symbiotic upcall that is made then begins its execution with *RSP* loaded with the last address of the stack frame. Furthermore we mandate that symbiotic upcalls cannot nest, that is the VMM cannot perform a symcall while another symcall is running. This also means that symbiotic upcalls are an independent thread of ex-

ecution inside the OS. This decision has ramifications that place a number of restrictions on symcall behavior, which we will elaborate on in Section 5.5.

5.4.2 VMM support

From the VMM perspective symbiotic upcalls are accessed as standard function calls, but are executed inside the guest context. This requires modifications to the standard behavior of a conventional VMM. The modifications to the Palacios VMM required not only additional functionality but also changes and new requirements to the low level guest entry/exit implementation.

As we stated earlier the VMM is responsible for saving and restoring the guest execution state before and after a symbiotic upcall is executed. Only a single instance of the guest state is saved, which means that only one symcall can be active at any given time. This means that symbiotic upcalls cannot nest. Our design does not perform a full checkpoint of the guest state but rather only saves the minimal amount of state needed. This allows symbiotic upcalls some leeway in modifying the current guest context. For example the guest OS is not prevented from modifying the contents of the control registers. In general the saved state corresponds to the state that is overwritten by values specified in the symcall MSRs.

The guest state that is saved by the VMM includes:

- *RIP:* The instruction pointer that the guest was executing before the exit that led to the symbiotic upcall.
- *Flags Register:* The system flags register
- *GPRs:* The full set of available General Purpose registers, including the Stack Pointer (*RSP*) used for argument passing.
- *Code Segment Descriptor/Selector:* The selector and cached descriptor of the code segment
- *Stack Segment Descriptor/Selector:* The selector and cached descriptor of the Stack segment
- *FS and GS Segment Bases:* The base addresses for both the FS and GS segments. These are used by the guest OS to store the address of the local processor data area.
- *CPU Privilege Level:* The AMD virtualization architecture requires the CPU Privilege level be saved as a separate entity, even though it is specified by the lower bits of the CS and SS segment selectors. For simplicity we save it separately when running on SVM.

Because symbiotic upcalls are executed in guest context we had to modify the VMM to perform a nested VM entry when a symcall is executed. VMM architectures are based on an event model. The VMM executes a guest in a special CPU operating mode until an exceptional event occurs, a special action is taken or an external event occurs. This causes the CPU to perform a VM exit that resumes inside the VMM context at a given instruction address. The VMM is then responsible for determining what caused the exit event and taking the appropriate action. This generally entails either emulating a certain instruction, handling an interrupt, modifying the guest state to address the exception, or servicing a request. This leads most VMMs to be implemented as event-dispatch loops where VM entries are made implicitly. That is a VM entry occurs automatically as part of a loop, and exit handlers do not need to be written to explicitly re-enter the guest.

For symbiotic upcalls we had to make VM entries available as an explicit function while also retaining their implicit nature. To do this we had to make the main event loop as well as the exit handlers reentrant. Reentrancy is necessary because it is not only possible but entirely likely that the guest will generate additional exits in the course of executing a symbiotic upcall. We found that it was fairly straightforward to modify the exit handlers to be reentrant, however the dispatch function was considerably more complicated.

Implementing reentrancy centered around ensuring safe access to two global data structures: The guest state structure which contains the state needed by the VMM to operate on a given guest environment and the virtualization control structures that store the hardware representation of the guest context. The guest state needed by the VMM is unserialized and serialized atomically before and after a VM entry/exit. This structure is reentrant because the VMM checkpoints the necessary state before and after a symbiotic call is made. This ensures that the guest will safely be able to re-enter the guest after the symbiotic upcall returns, because the guest state is copied back to the hardware structures before every entry. However it does not store the hardware state containing the exit information. In practice the exit information is small enough to store on the stack and pass as arguments to the dispatch function.

5.5 Current restrictions

In our design, symbiotic upcalls are meant to be used for relatively short synchronous state queries. Using symcalls to modify internal guest state is much more complicated and potentially dangerous. Since our current implementation is based on this fairly narrow focus, we made a number of design choices that limit the behavior of the symcall handler in the guest OS. These requirements ensure that only a single symcall will be executed at any given time and it will run to completion with no interruptions, i.e. it will not block.

The reasoning behind restricting the symcall behavior is to allow a simplified implementation as well as provide behavioral guarantees to the VMM executing a symbiotic upcall. If symbiotic upcalls were permitted to block the synchronous model would essentially be broken, because a guest OS would be able to defer the upcall's execution indefinitely. Furthermore it would increase the likelihood that when a symbiotic upcall did return, the original reasons for making the upcall would no longer be valid. This is in contrast to system calls where blocking is a necessary feature that allows the appearance of synchrony to applications.

In order to ensure this behavior, a symcall handler in the guest OS is not allowed to sleep, invoke the OS scheduler, or take any other action that results in a context switch. Furthermore while the guest is executing a symbiotic upcall the VMM actively prevents the injection of any external interrupts such as those generated by hardware clocks. Our implementation also blocks the injection of hardware exceptions, and mandates that symcall handlers do not take any action that generates a processor exception that must be handled by the guest OS. While this might seem restrictive, we note that, in general, exceptions generated in a kernel code path are considered fatal.

The requirement that symcall handlers not block has further ramifications in how they deal with atomic data structures. This is particularly true because, as we stated earlier, a VMM can execute a symbiotic upcall at any point in the guest's execution. This means that it is possible for a symcall to occur while other kernel code paths are holding locks. This, and the fact that symcalls cannot block, mean that symcalls must be very careful to avoid deadlocks. For example, if a kernel control path is holding a spinlock while it modifies internal state it can be pre-empted by a symbiotic upcall that tries to read that same state. If the symcall ignores the lock it will end up reading inconsistent state, however if it tries to acquire the spinlock it will deadlock the system. This is because the symcall will never complete which in turn means the process holding the lock will never run because symcalls must run to completion and cannot be interrupted.

In order to avoid deadlock scenarios while still ensuring data integrity, special care must be taken when dealing with protected data structures. Currently our implementation allows symbiotic

upcalls to acquire locks, however they cannot wait on that lock if it is not available. If a symcall attempts to acquire a lock and detects that it is unavailable, it must immediately return an error code similar to the POSIX error *EWOULDBLOCK*. In multiprocessor environments we relax the locking requirements in that symbiotic upcall handlers can wait for a lock as long as it is held by a thread on another CPU.

6. SwapBypass example service

We will now show how symcalls make possible optimizations that would be intractable given existing approaches, by examining SwapBypass, a VMM extension designed to bypass the Linux swap subsystem. SwapBypass allows a VMM to give a VM direct access to memory that has been swapped out to disk, without requiring it be swapped in by the guest OS.

SwapBypass uses a modified disk cache that intercepts the I/O operations to a swap disk and caches swapped out pages in the VMM. SwapBypass then leverages a VM's shadow page tables to redirect swapped out guest virtual addresses to the versions in the VMM's cache. SwapBypass uses a single symcall to determine the internal state and permissions of a virtual memory address. The information returned by the symcall is necessary to correctly map the page and would be extremely difficult to gather with existing approaches.

We will now give a brief overview of the Linux swap architecture, and describe the SwapBypass architecture.

6.1 Swap operation

The Linux swap subsystem is responsible for reducing memory pressure by moving memory pages out of main memory and onto secondary storage that is generally on disk. The swap architecture is only designed to handle pages that are assigned to anonymous memory regions in the process address space, as opposed to memory used for memory mapped files. The swap architecture consists of a number of components such as the collection of swap disks, the swap cache, and a special page fault handler that is invoked by faults to a swapped out memory page. The swap subsystem is driven by two scenarios: low memory conditions that drive the system to swap out pages, and page faults that force pages to be swapped back into main memory.

6.1.1 Swap storage

The components that make up the swap storage architecture include the collection of swap devices as well as the swap cache. The swap devices consist of storage locations that are segmented into an array of page sized storage locations. This allows them to be accessed using a simple index value that specifies the location in the storage array where a given page is located. In Linux this index is called the *Swap Offset*. The swap devices themselves are registered as members of a global array, and are themselves identified by another index value, which Linux calls the *Swap Type*. This means that a tuple consisting of the *Swap Offset* and *Swap Type* is sufficient for determining the storage location for any swapped out page.

As pages are swapped out, the kernel writes them to available swap locations and records their location. As a side effect of swapping out the page, any virtual address that refers to that page is no longer valid and furthermore the physical memory location is most likely being used by something else. To prevent accesses to the old virtual address from operating on incorrect data, Linux marks the page table entries pointing to the swapped out page as not present. This is accomplished by clearing the *Present* bit in the page table entry (PTE). Because marking a page invalid only requires a single bit, the rest of the page table entry is ignored by the hardware. Linux takes advantage of this fact and stores the swap location tuple into the available PTE bits. We refer to PTEs that are marked not present and store the swap location tuple as *Swapped PTEs*.

As a performance optimization Linux also incorporates a special cache that stores memory pages while they are waiting to be swapped out. Because anonymous memory is capable of being shared between processes and thus referenced by multiple virtual addresses, Linux must wait until all the PTEs that refer to the page are marked as swapped PTEs before it can safely move the page out of main memory and onto the appropriate swap device. Tracking down all the references to the page and changing them to Swapped PTEs is typically done in the background to minimize the impact swapping has on overall system performance. Thus it is possible for pages to remain resident in the cache for a relatively long period of time, and furthermore it is possible that one set of PTEs will point to a page in the swap cache while another set will be marked as Swapped PTEs. This means that just because a PTE is a Swapped PTE does not mean the page it refers to is actually located at the location indicated by the Swapped PTE. It is important to note that every page in the swap cache has a reserved location on a swap device, this means that every page in the swap cache can be referenced by its Swapped PTE. The swap cache itself is implemented as a special substructure in the kernel's general page cache, this is a complex internal kernel data structure organized as a radix tree.

6.1.2 Swapped page faults

As we mentioned earlier Linux marks the page table entries of swapped out pages as invalid and stores the swap location of the page into the remaining bits. This causes any attempted access to a swapped out virtual address to result in a page fault. Linux uses these page faults to determine when to swap pages back into main memory. When a page fault occurs the kernel exception handler checks if the faulting virtual address corresponds to a Swapped PTE. If so it first checks if the page is resident in the swap cache. If the page is found in the page cache then the handler simply updates the PTE with the physical memory address of the page in the swap cache and indicates that there is a new reference to the page. If the page is not found in the swap cache then the Swapped PTE contains the location of the page in the collection of swap devices. This triggers a swap in event, where the swap subsystem reads the page from the swap device and copies it to an available physical memory location. This operation could itself trigger additional swap out events in order to make a location in main memory available for the swapped in page. Once the page is copied into main memory it is added to the swap cache, because its possible that other Swapped PTEs reference that page and have not been updated with its new physical address. Once all references have been updated the page is removed from the swap cache.

Finally it should be noted that after a page has been swapped in a copy of the page remains on the swap device. This means that if a process swaps in a page only in order to read it, the page can simply be deleted from the swap cache without writing it to the swap device. The next time the page is referenced it will simply be copied back into the swap cache. Also note that a page can be swapped in from disk and written to while still in the swap cache, and then swapped out again. In this case the version in the swap cache must be written back to the swap device. This makes it possible for a swapped out page to be de-synchronized from its copy on the swap device. This behavior is important and has ramifications for SwapBypass that we will discuss later.

6.2 SwapBypass implementation

SwapBypass uses shadow page tables to redirect the swapped PTEs in a guest's page table to pages that are stored in a special cache located in the VMM. This allows a guest application to directly reference memory that has been swapped out to disk by its OS. An

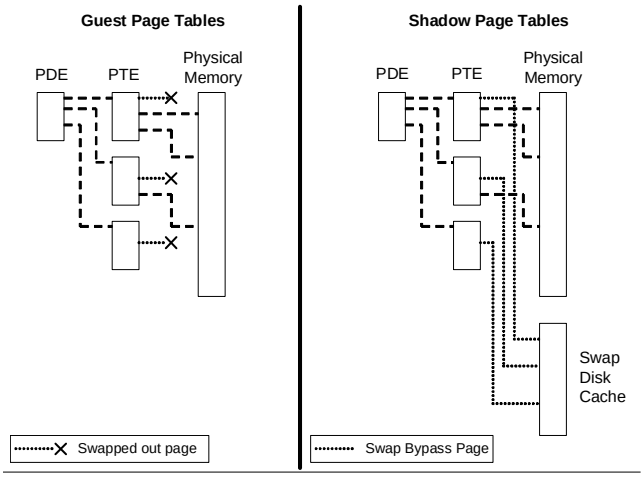

Figure 4. The guest and shadow page table configuration needed to provide a guest access to memory it has swapped out.

Component	Lines of Code
Swap disk cache	373(C)
Page fault handler	47(C)
SwapBypass core	182(C)
Guest SymCall functions	53(C)
Total	655(C)

Figure 5. Lines of code needed to implement SwapBypass as measured by SLOCcount

example set of page table hierarchies are shown in Figure 4. In this case the guest OS has swapped out 3 pages that are referenced by the current set of page tables. As we described earlier it has marked the Swapped PTEs as not present in order to force a page fault when they are accessed. However, when a VMM is using shadow paging all page faults cause VMExits, which allows a VMM to handle page faults before the guest OS. In many cases the VMM updates its shadow page tables to reflect the guest page tables and continues execution in the guest, other times the VMM must forward the page fault exception to the VM so it can be handled by the guest OS. Without SwapBypass the VMM would only see that the guest marked its page table entries as invalid, and thus forward the page fault to the guest OS. However when SwapBypass is active it is able to detect that the guest's PTE is in fact a Swapped PTE[3], and set the shadow PTE to point at the page in the cache. SwapBypass uses a special symcall to inspect the internal swap state of the guest Linux kernel as well as to determine the access permissions of the virtual address containing the swapped PTE.

SwapBypass is implemented with several components. A single symcall that returns the state of a guest virtual address, a special swap device cache that intercepts I/O operations to a swap disk, a new edge case that is added to the shadow page fault handler, and the SwapBypass core that provides the interface between the symcall, swap disk cache, and shadow page table hierarchy.

Figure 5 shows the implementation complexity of the different components in lines of code as measured by SLOCCount. All together SwapBypass consists of 655 lines of code.

[3] A Swapped PTE contains a set of flags to indicate a swapped page without any additional information

6.2.1 Swap disk cache

The first component of SwapBypass is a cache that is located inside the VMM between a guest and its swap disk. This cache intercepts all I/O operations and caches swapped out pages as they are written to disk. As swapped pages are written to disk they are first inserted into the cache, if the cache is full then victim pages are chosen and flushed to disk according to a Least Recently Used (LRU) policy. When pages are read from disk they are copied from cache if found, otherwise the pages are read directly from disk and not inserted into the cache.

During initialization the swap disk cache registers itself with SwapBypass, and supplies its *Swap Type* identifier as well as a special function that SwapBypass uses to query the cache contents. This function takes as an argument the *Swap Offset* of a page located on disk and returns the physical address of the page if it is present in the cache. In order for the swap disk cache to determine its *Swap Type* identifier we had to modify Linux to add the *Swap Type* to the swap header that is written to the first page entry of every swap device. The swap disk cache intercepts this write and parses the header to determine its *Swap Type*.

The swap disk cache is also responsible for notifying SwapBypass of disk reads which correspond to swap in events. These events drive invalidations that we will discuss in more detail later.

6.2.2 SwapBypass symcall

Implementing SwapBypass requires knowledge of the state of a swapped out page and the permissions that the current process has on the virtual address referring to that page. This information is extremely difficult to determine from outside the guest context. Furthermore this information cannot be collected asynchronously. The reason for this is that if the VMM does not immediately modify the shadow page tables to point to a page in the cache then it *must* inject a page fault into the guest. The page fault would then cause the guest to swap the page back into its memory space, and modify the Swapped PTE to point at the new location. By the time the asynchronous upcall completed the reason for calling it would no longer exist. Furthermore, because symcalls are executed synchronously they execute in the process context that existed when the exit leading to the symcall occurred. In the case of SwapBypass this means that the symcall executes as the process that generated the page fault on the swapped PTE. An asynchronous approach could not provide this guarantee, which would greatly complicate the implementation.

The symcall takes two arguments: a *guest virtual address* and a *Swapped PTE*. The guest virtual address is the virtual address that caused the page fault while the Swapped PTE is the guest PTE for that virtual address. The symcall returns a set of three flags that mirror the permission bits in the hardware PTEs (Present, Read/Write, and User/System). These bits indicate whether the virtual address is valid, whether the page is writable, and finally whether it can be accessed by user processes.

The first action taken by the symcall is to find the task descriptor of the process which generated the page fault. Linux stores the current task descriptor in a per-CPU data area whose location is stored in the FS segment selector. This means that the task descriptor is found by simply calling `get_current()`, because the FS segment is loaded as part of the symcall entry.

Next, the symcall determines if the page referenced by the Swapped PTE is in fact swapped out or if it is present in the kernel's swap cache. As we stated before, Linux does not immediately update all the Swapped PTEs referencing a given page, so it is possible for the PTEs to be out of date. In this case the guest's page fault handler would simply redirect the PTE to the page's location in the swap cache and return. Therefore, if the symcall detects that the page is present in the swap cache, it immediately returns with

a value indicating that the VMM should not use the on disk version of the page. This will cause SwapBypass to abort and continue the normal VMM execution path by injecting a page fault into the guest, thus invoking the guest's swap subsystem. SwapBypass cannot operate on pages in the swap cache, even if they are available in the SwapBypass cache because of the synchronization issues mentioned earlier.

If the swapped PTE does not refer to a page in the swap cache, then it can be redirected by SwapBypass. In this case it is necessary to determine what access permissions the current process has for the virtual address used to access the page. Linux does not cache the page table access permissions for swapped out pages, so it is necessary to query the process' virtual memory map. The memory map is stored as a list of virtual memory areas that make up the process' address space. The symcall scans the memory map searching for a virtual memory area that contains the virtual address passed as an argument to the symcall. Once the region is located, it checks if the region is writable and if so sets the writable flag in the return value.

Finally the symcall checks if the virtual address is below the 3GB boundary, and if so sets the user flag in the return value.

6.2.3 Shadow page fault handler

Similar to the Linux swap subsystem, SwapBypass is driven by page faults that occur when a guest tries to access a swapped out page. When operating normally, the shadow page fault handler parses the guest page tables in order to create a shadow page table hierarchy. If the shadow handler determines that the guest page tables are invalid, then it simply injects a page fault into the guest.

For SwapBypass to function correctly the shadow page fault handler must be able to detect when a guest page fault was generated by a swapped PTE. This can be determined by simply checking several bits in the swapped PTE. If this check succeeds, then the shadow page fault handler invokes SwapBypass. Otherwise it continues normally and injects a page fault. The important take away here is that the shadow page fault handler can determine if a fault is caused by a swapped PTE by simply checking a couple of bits that are already available to it. This means that there is *essentially no additional overhead* added to the shadow paging system in the normal case.

When the shadow page fault handler invokes SwapBypass it supplies the virtual address and the swapped PTE from the guest page tables. SwapBypass returns to the shadow page fault handler the physical address where the swapped page is located and a set of page permissions. The shadow page fault handler then uses this information to construct a shadow PTE that points to the swapped out page. This allows the guest to continue execution and operate on the swapped out page as if it was resident in the guest's address space. If the swapped page is unavailable to SwapBypass then the shadow page fault handler falls back to the default operation and injects a page fault into the guest.

6.2.4 SwapBypass core

The SwapBypass core interfaces with the swap disk cache, the symcall, and the Shadow page fault handler and tracks the swapped PTEs that have been successfully redirected to the swap disk cache. SwapBypass is driven by two guest events: page faults to swapped PTEs and I/O read operations to the swap disk cache. Page faults create mappings of swapped pages in the shadow page tables, while read operations drive the invalidation of those mappings. The execution path resulting from a page fault is shown in Figure 6, and the execution path for disk reads is shown in Figure 7.

Page faults When a guest page fault occurs and the shadow page fault handler determines that it was caused by an access to a swapped PTE, SwapBypass is invoked and passed the faulting virtual address and guest PTE. First SwapBypass determines which swap device the swapped PTE refers to and the location of the page on that device. Next, it queries the swap disk cache to determine if that page is present in the memory cache. If the page is present, SwapBypass makes a symcall into the guest passing in the virtual address and swapped PTE value. The symcall returns whether the swapped page is in fact located on disk, and the permissions of the virtual address.

If the page is present in the swap disk cache and the symcall indicates that the page on disk is valid, then SwapBypass adds the virtual address onto a linked list that is stored in a hash table keyed to the swapped PTE value. This allows SwapBypass to quickly determine all the shadow page table mappings currently active for a swapped page. Finally SwapBypass returns the permissions and physical address of the swapped page to the shadow page fault handler.

Disk reads Read operations from a swap disk result in the guest OS copying a page from the swap device and storing it in the swap cache. When this operation completes the OS will begin updating the swapped PTEs to reference the page in memory. When this occurs SwapBypass must remove any existing shadow page table entries that reference the page. If the shadow page table entries were not invalidated, then the guest could see two different versions of the same memory page. One version would be in the guest's swap cache and be referenced by any new page table entries created by the guest, while any old swapped PTEs would still only see the version on disk.

When the swap disk cache detects a read operation occurring, it combines its *Swap Type* with the page index being read to generate the swapped PTE that would be used to reference that page. The swap disk cache then notifies SwapBypass that the page referenced by the swapped PTE has been read. SwapBypass then locates the list of shadow page table mappings for that swapped PTE in the previously mentioned hash table. Each shadow page table entry is invalidated and the swapped PTE is deleted from the hash table. SwapBypass then returns to the swap disk cache which completes the I/O operation.

6.3 Alternatives

We believe that SwapBypass is a compelling example that argues for symbiotic virtualization in general, and SymCall in particular, when considered in comparison with the two current alternatives. The first alternative, the graybox/introspection approach, would require that the VMM read and parse the internal guest state to determine whether a page was capable of being remapped by SwapBypass. Even if the guest was modified to include the read/write and user/system bits in the swapped PTE format, the VMM would still have to access the swap cache directly. This would be a very complex procedure that would need to locate and access a number of nested data structures.

The second alternative approach would be to use the current upcall implementations that are based on hardware interrupts and guest device drivers. This approach has two problems: interrupts are asynchronous by nature and Linux uses a return from an interrupt handler as an opportunity to reschedule the current task. Asynchrony could potentially be handled within the VMM by first ensuring that the guest context was configured to immediately handle the interrupt if it was injected. However, this would be complex and might result in some upcalls being aborted. It would also require changes to the Linux interrupt handling architecture to forbid context switches for certain classes of interrupts.

Finally, a simple disk cache might be used in place of a SwapBypass-like service in order to speed up accesses to swapped pages. While this would indeed benefit performance, SwapBypass is further capable of *completely eliminating the overhead of the*

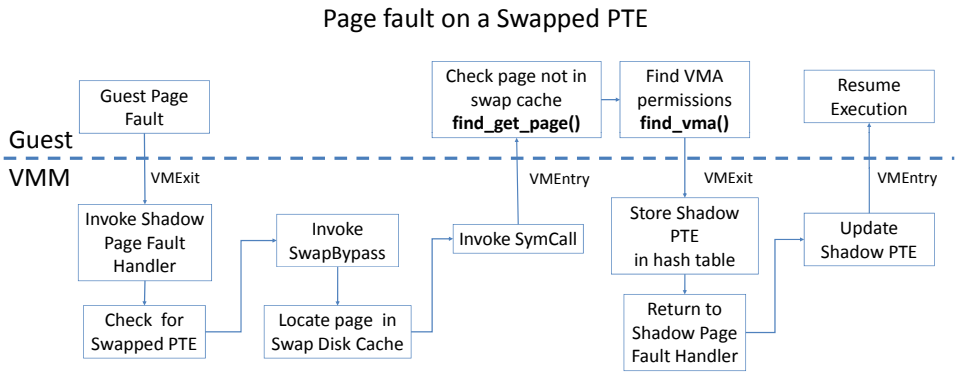

Figure 6. The execution path of SwapBypass in response to a guest page fault on a swapped PTE

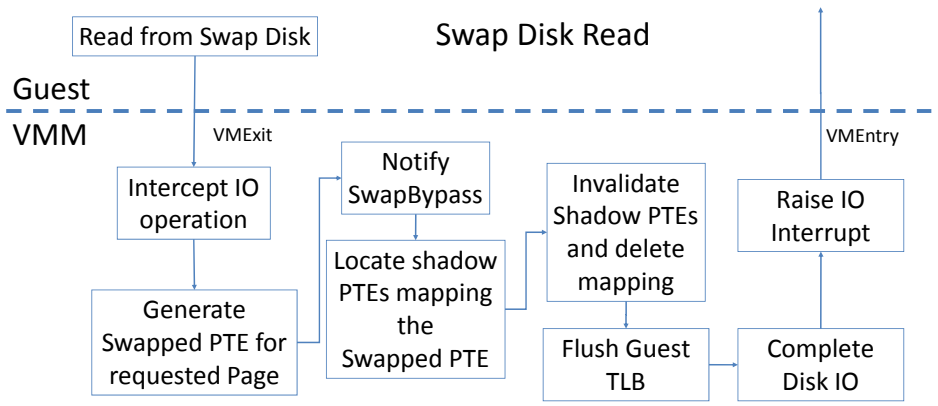

Figure 7. The execution path of SwapBypass in response to an I/O read operation to a swap device

Latency for echo() SymCall			
First ("cold")	5 VMExits	63455 cycles	35 μs
Next ("warm")	0 VMExits	15771 cycles	9 μs

Figure 8. SymCall latency for a simple echo() symcall

swap system in the Linux kernel. As our evaluation shows, this dramatically improves performance, even over an ideal swap device with no I/O penalty.

7. Evaluation

We evaluated both the performance of our SymCall implementation as well as our implementation of SwapBypass. These tests were run on a Dell SC440 server with a 1.8GHz Intel Core 2 Duo Processor and 4GB of RAM. The guest OS implementation was based on Linux 2.6.30.4.

7.1 SymCall latency

The first test we ran measured the latency in making a symcall. For this test we implemented an `echo()` symcall, that simply returned the arguments as return values. First, we measured the latency of a symcall made for the first time. When a symcall is first executed, or "cold", it will access a number of locations in kernel memory that are not present in the shadow page tables. The guest will generate shadow page faults until all the memory locations are accessible. For a simple symcall with no external references this requires 5 shadow page faults. We also ran a second test of a symcall after its memory regions have been added to the shadow page tables. In this "warm" case the symcall generated no exits. The results shown in Figure 8 are an average of 10 test calls. The latency for a "cold" symcall is 64 thousand CPU cycles, which on our test machine equates to around 35 microseconds. The "warm" symcall completed in $\sim$16 thousand cycles or 9 microseconds.

7.2 SwapBypass performance

We have evaluated the effectiveness of SwapBypass using a set of memory benchmarks that are implemented to use anonymous memory regions. These benchmarks include the microbenchmarks Stream [18] (small vector kernel) configured to use 300MB of memory and GUPS [20] (random access) configured to use 256MB of memory. Stream and GUPS are part of the HPC Challenge benchmark suite. We also used the ECT memperf benchmark [23] configured to use 256MB of memory. ECT memperf is designed to characterize a memory system as a function of working set size, and spatial and temporal locality. Each benchmark was run in a guest configured with 256MB of memory and a 512MB swap disk combined with a swap disk cache in the Palacios VMM. We measured the performance of each benchmark as a function of the size of the swap disk cache. The benchmarks were timed using an external time source.

For the Stream benchmark we ran tests using a hardware swap disk, a virtual swap disk implemented using a simple disk performance model, and a pure RAM disk implemented in VMM memory. The hardware disk was a 7200RPM SATA disk partitioned with a 512MB swap partition. Our hard disk model used a simple aver-

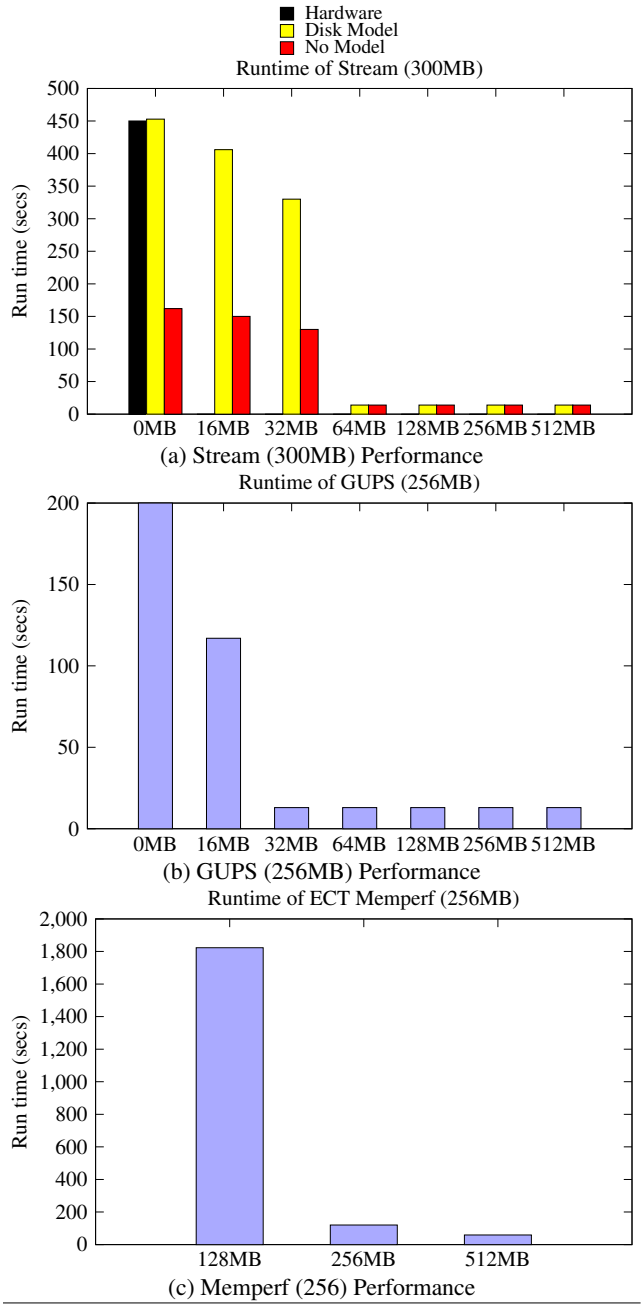

Figure 9. Performance results for Stream, GUPS, and ECT Memperf benchmarks. The benchmark run time was measured for variable sized swap disk caches.

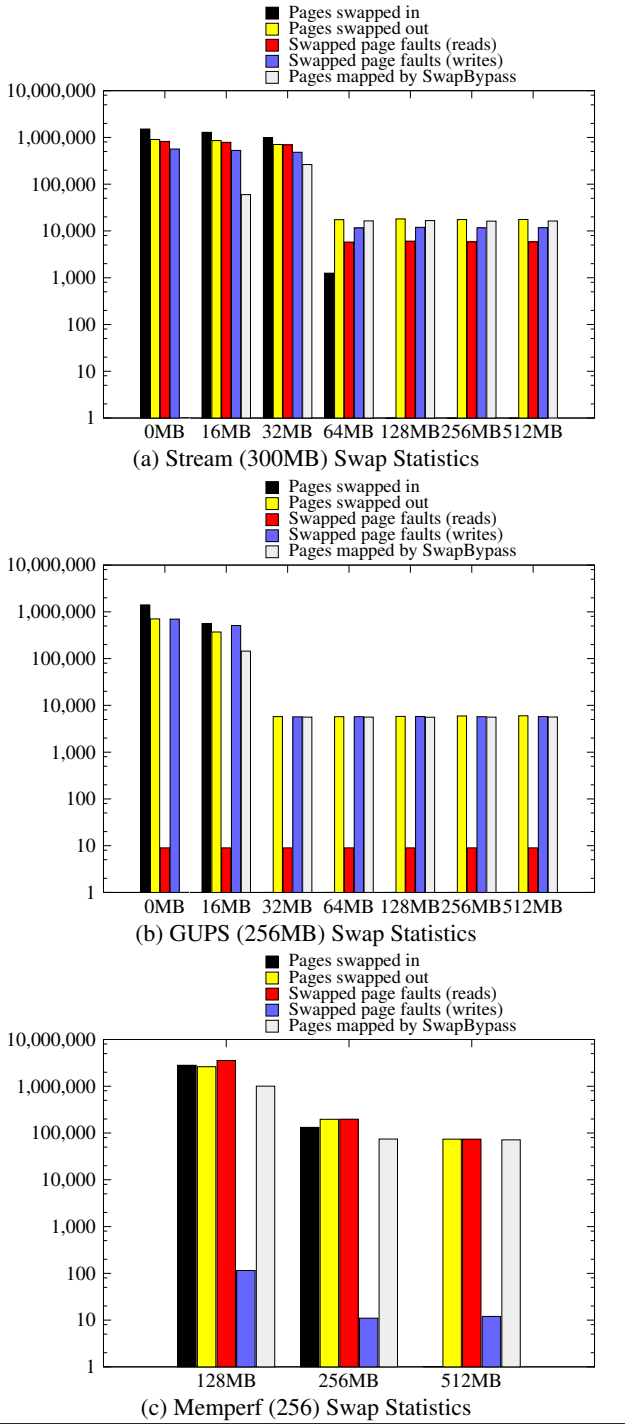

Figure 10. Hardware event statistics for Stream, GUPS, and ECT Memperf benchmarks. Events were counted for each benchmark run using variable sized swap disk caches.

age seek delay based on the hardware specifications of the SATA disk. For the other benchmarks we only used the RAM disk without a disk model. Our reason for concentrating our evaluation on a RAM-based swap disk is to generate ideal I/O conditions for the Linux swap system. With the RAM-based swap disk all disk I/O is eliminated and data is transferred at the speed of system memory. This means that the Linux swap architecture is the *sole* source of overhead, assuring that any performance benefits gained by SwapBypass are *not* simply the result of implementing a disk cache in RAM. Our inclusion of the hardware swap disk and disk model evaluations is to help illustrate the effects of SwapBypass with non-ideal I/O.

Stream Our initial benchmark is Stream, configured to use a 300MB region of memory. Figure 9(a) shows the run time of the Stream benchmark for exponentially increasing swap disk cache sizes. We first ran the benchmark using the hardware swap disk as well as with SwapBypass configured with no swap disk cache. Without a cache, SwapBypass flushes all swapped pages to disk and so cannot access them, meaning that SwapBypass will have no beneficial effect on the performance. As the figures show both the hardware and disk model configurations performed comparably with run times of around 450 seconds or 7.5 minutes. The configuration using the RAM disk swap device completed in only around 150 seconds or 2.5 minutes due to the lack of disk I/O.

We then began to scale up the size of the swap disk cache exponentially to determine the impact SwapBypass would have on performance. As the cache size increases the run time begins to decrease until the combined size of the cache and the VM's physical memory partition exceeds the working set size of the benchmark. As soon as this threshold is reached the run time drops off dramatically and the performance of both disk model and RAM disk configurations are essentially identical at 14 seconds. At this point and beyond, SwapBypass is able to satisfy every swapped page fault by mapping the shadow page tables to the page in the swap disk cache—the Linux swap system is completely bypassed and is essentially cut out of the guest's execution path.

The effectiveness of SwapBypass at bypassing the swap system is demonstrated in Figure 10(a), which provides a hardware level view of Linux swap system. For each benchmark run we collected the number of pages transferred to and from the swap device (*Pages swapped in* and *Pages swapped out*), the number of page faults generated by swapped out pages (*Swapped page faults (reads)* and *Swapped page faults (writes)*), and also the number of pages that SwapBypass was able to map into the guest from the swap disk cache (*Pages mapped by SwapBypass*). As the swap disk cache size initially increases the number of page faults and swap I/O operations does not change much but the number of pages mapped by SwapBypass increases substantially. However, when the cache size plus the guest memory partition size reaches the benchmark's working set size, all the measurements decrease dramatically. Also, the number of pages swapped in by the guest OS goes to 0.

GUPS GUPS exhibits behavior similar to that of Stream. The GUPS results are shown in Figures 9(b) & 10(b)

ECT Memperf ECT Memperf results are shown in Figures 9(c) & 10(c). The memperf results are limited to swap disk cache sizes of 128MB and greater because the execution time for lower cache sizes was too large to measure. The execution time for the 128MB cache size was around 1800 seconds or 30 minutes, and the test run for the 64MB cache size was terminated after 6 hours.

Summary of results

A summary of the speedups that SwapBypass can provide for the different benchmarks is shown in Figure 11. The reason for the dramatic increase in performance once the working set size threshold is reached is due to a compounding factor. When an OS is in a low memory situation, swapping in a page necessitates swapping another page out, which will need to be swapped in at a later time, which will in turn force a page to be swapped out, and so on. Therefore when SwapBypass is able to avoid a swap in operation, it is also avoiding a swap out that would be needed to make memory available for the swapped in page. SwapBypass is therefore able to prevent the guest from trashing, which not surprisingly improves performance dramatically.

Benchmark	speedup
Stream (No model)	11.5
Stream (disk model)	32.4
GUPS	15.4
ECT Memperf	30.9

Figure 11. Performance Speedup factors of SwapBypass. The speedup of ECT memperf is measured over the 128MB swap disk cache configuration

Our results show that it is possible to artificially and transparently expand a guest's physical memory space using a VMM service. Furthermore, the availability of a symbiotic interface makes implementing this feature relatively easy, while existing approaches would require deep guest introspection or substantial modifications to the guest OS. Being able to easily implement SwapBypass suggests that there are other extensions and optimizations that could be built using symbiotic interfaces.

8. Related work

There are currently two approaches taken by existing virtualization tools: full system virtualization [19, 25, 26] and paravirtualization [1, 3, 17, 27]. In fact it is quickly becoming the case that these approaches are no longer mutually exclusive. Despite the blurring of the boundaries between both of these methods there has not been a significant departure from either. Symbiotic virtualization is a new virtualization interface that introduces a high level guest interface accessible by a VMM. Furthermore, a guest OS can support a symbiotic interface without sacrificing hardware compatibility.

There has also been considerable effort put into better bridging the *semantic gap* of the VMM↔OS interface and leveraging the information that flows across it [8–10, 15, 16, 24]. One of the most compelling uses for this approach is virtual machine introspection, most commonly used in security applications [2, 5, 7, 11, 12, 22, 28]. However, the information gleaned from such black-box and gray-box approaches is still semantically poor, and thus constrains the decision making that the VMM can do. Further, it goes one way; the OS learns nothing from the VMM. Symbiotic virtualization allows explicit two way communication at a high semantic level. Using symbiotic interfaces a VMM can determine guest state by simply asking the guest for it, instead of reverse engineering the running OS.

While SymCall is a new interface for invoking upcalls into a running guest environment, providing upcall support for a guest OS is not a new concept. However the standard approaches are generally based on notification signals as opposed to true upcall interfaces [6]. These notifications usually take the form of hardware interrupts that are assigned to special vectors and injected by the VMM. Because interrupts can be masked by a guest OS, these upcall interfaces are typically asynchronous. Furthermore, existing upcalls consist of only a notification signal and rely on a virtual device or event queue to supply any arguments. Symcalls in contrast are always synchronous and do not need to be disabled with the same frequency as interrupts. Furthermore they allow argument

passing directly into the upcall, which enables the VMM to expose them as normal function calls.

9. Conclusion

We have introduced symbiotic virtualization, a new approach to designing VMMs and OSes such that both support, but neither requires, the other. Furthermore we presented the design and implementation of a symbiotic framework consisting of the SymCall functional upcall interface. This framework was implemented in the Palacios VMM and a Linux guest kernel. Using the symbiotic interfaces we implemented SwapBypass, a new method of decreasing memory pressure on a guest OS. Furthermore we showed how SwapBypass is only possible when using a symbiotic interface. Finally, we evaluated SwapBypass showing it improved swap performance by avoiding thrashing scenarios resulting in 11–32x benchmark speedups.

References

[1] KVM: Kernel-based virtualization driver. White Paper.

[2] BAIARDI, F., AND SGANDURRA, D. Building trustworthy intrusion detection through vm introspection. In *IAS '07: Proceedings of the Third International Symposium on Information Assurance and Security* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 209–214.

[3] 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 *19th ACM Symposium on Operating Systems Principles (SOSP)* (October 2003).

[4] CHEN, P. M., AND NOBLE, B. D. When virtual is better than real. In *The 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (2001).

[5] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. K. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)* (Seattle, WA, USA, Mar. 2008).

[6] CLARK, D. D. The structuring of systems using upcalls. In *Proceedings of the tenth ACM symposium on Operating systems principles (SOSP)* (1985).

[7] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium* (2003), pp. 191–206.

[8] GUPTA, A. *Black Box Methods for Inferring Parallel Applications Properties in Virtual Environments*. PhD thesis, Northwestern University, Department of Electrical Engineering and Computer Science, March 2008.

[9] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Antfarm: tracking processes in a virtual machine environment. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), USENIX Association, pp. 1–1.

[10] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (2006), pp. 14–24.

[11] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Vmm-based hidden process detection and identification using lycosid. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2008), pp. 91–100.

[12] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM, pp. 91–104.

[13] LANGE, J., PEDRETTI, K., DINDA, P., BRIDGES, P., BAE, C., SOLTERO, P., AND MERRITT, A. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011)* (March 2011).

[14] LANGE, J., PEDRETTI, K., HUDSON, T., DINDA, P., CUI, Z., XIA, L., BRIDGES, P., GOCKE, A., JACONETTE, S., LEVENHAGEN, M., AND BRIGHTWELL, R. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)* (April 2010).

[15] LANGE, J. R., AND DINDA, P. A. Transparent network services via a virtual traffic layer for virtual machines. In *In Proceedings of the 16th International Symposium on High Performance Distributed Computing (HPDC)* (2007).

[16] LANGE, J. R., SUNDARARAJ, A. I., AND DINDA, P. A. Automatic dynamic run-time optical network reservations. In *In Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (2005), pp. 255–264.

[17] LEVASSEUR, J., UHLIG, V., CHAPMAN, M., CHUBB, P., LESLIE, B., AND HEISER, G. Pre-virtualization: soft layering for virtual machines. Technical Report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.

[18] MCCALPIN, J. D. A survey of memory bandwidth and machine balance in current high performance computers. In *Newsletter of the IEEE Technical Committee on Computer Architecture (TCCA)* (December 1995).

[19] PARALLELS CORPORATION. http://www.parallels.com.

[20] PLIMPTON, S. J., BRIGHTWELL, R., VAUGHAN, C., UNDERWOOD, K., AND DAVIS, M. A simple synchronous distributed-memory algorithm for the hpcc randomaccess benchmark. In *Proceedngs of the IEEE International Conference on Cluster Computing (CLUSTER)* (September 2006).

[21] QUMRANET CORPORATION. Kvm - kernel-based virtual machine. Tech. rep., 2006. KVM has been incorporated into the mainline Linux kernel codebase.

[22] QUYNH, N. A., AND TAKEFUJI, Y. Towards a tamper-resistant kernel rootkit detector. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing* (New York, NY, USA, 2007), ACM, pp. 276–283.

[23] STRICKER, T., AND GROSS, T. Optimizing memory system performance for communication in parallel computers. In *Proceedings of the 22nd annual international symposium on Computer architecture (ISCA)* (1995).

[24] SUNDARARAJ, A. I., GUPTA, A., AND DINDA, P. A. Increasing application performance in virtual environments through run-time inference and adaptation. In *In Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (2005).

[25] VIRTUALBOX. http://www.virtualbox.org.

[26] WALDSBURGER, C. Memory resource management in vmware esx server. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)* (2002).

[27] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 195–209.

[28] YU, Y., GUO, F., NANDA, S., LAM, L.-C., AND CHIUEH, T.-C. A feather-weight virtual machine for windows applications. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments* (New York, NY, USA, 2006), ACM, pp. 24–34.