Shifting GEARS to Enable Guest-context Virtual Services

Kyle C. Hale
Dept. of EECS
Northwestern University
Evanston, IL 60208
kh@u.northwestern.edu

Lei Xia
Dept. of EECS
Northwestern University
Evanston, IL 60208
lxia@northwestern.edu

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

ABSTRACT

We argue that the implementation of VMM-based virtual services for a guest should extend into the guest itself, even without its cooperation. Placing service components directly into the guest OS or application can reduce implementation complexity and increase performance. In this paper we show that the set of tools in a VMM required to enable a broad range of such guest-context services is fairly small. Further, we outline and evaluate these tools and describe their design and implementation in the context of Guest Examination and Revision Services (GEARS), a new framework within the Palacios VMM. We then describe two example GEARS-based services—an MPI communication accelerator and an overlay networking accelerator—that illustrate the benefits of allowing virtual service implementations to span across the VMM, guest, and application. Other VMMs could employ the ideas and tools in GEARS.

Categories and Subject Descriptors

D.4.4 [**Software**]: OPERATING SYSTEMS

Keywords

virtual machines, services, code transformation

1. INTRODUCTION

The virtualization layer is commonplace in cloud/data center computing, well attested to in adaptive/autonomic computing, and increasingly common even in high performance computing. In each of these areas, the capability to augment guests with VMM-based virtual services that can enhance their functionality, security, or performance, and can better match a user's VMs with a provider's hardware resources, is one of the key benefits of virtualization.

This project is made possible by support from the United States National Science Foundation (NSF) via grant CNS-0709168 and the Department of Energy (DOE) via grant DE-SC0005343.

To expedite the creation and deployment of virtual services, it would be beneficial to provision the VMM with the ability to implant parts of the service implementations directly in the guest. VMM code *running directly within the context of the guest OS or application without its cooperation* has the potential to considerably simplify the design and implementation of services because the services could then directly manipulate aspects of the guest from within the guest itself. Furthermore, these kinds of services could eliminate many overheads associated with costly exits to the VMM, improving their performance. Finally, extending a service into the guest would enable new classes of services that are not possible, or are extremely difficult to implement solely within the VMM context. We refer to virtual services that can span the VMM, the guest kernel, and the guest application, as *guest-context virtual services.*

We present the Guest Examination and Revision Services (GEARS), a novel framework that aims to enable guest-context virtual services. The tools in GEARS have been carefully selected as those sufficient to implement a wide range of such services, and their abstractions are independent of our implementation. We also present the design, implementation, and performance evaluation of the GEARS tools in the context of the Palacios VMM, an effort that demonstrates that the tools can be made available in a VMM with only a modest increase in its code size and with very low overhead (and zero overhead when not used). Finally, we demonstrate how we built two guest-context virtual services using the tools, and how they can enhance overlay networking and MPI performance without significant implementation complexity. The overlay service improves latency by 3–20%, while the MPI service approaches the native memory copy throughput limit for co-located VMs.

The GEARS implementation also attempts to broaden the community of virtual service developers by eliminating the need for programmers to be intimately familiar with VMM internals. While basic knowledge of the guest is required, the GEARS framework provides the necessary tools to transform standard code into a guest-context service. Guest-specific information can then remain mostly opaque to the service developer, and how these services are used to affect guest operation is entirely at his or her discretion. GEARS also allows the VMM to transparently *mutate* guest applications and OSes into VMM-aware entities. Thus, we believe that frameworks like GEARS have the potential to beget adaptive applications that reap the benefits of virtualization while retaining the performance available to those aware of the underlying software and hardware stack.

2. RELATED WORK

Adaptive or autonomic computing in virtualized computing environments is an area of considerable current interest and promise (e.g., [21, 24, 19]). Autonomic computing in this context extends new services to the guest, typically without guest knowledge. This has much in common with the virtual service model in other contexts.

Virtual services need to be cognizant of the guest, but there exists a *semantic gap* [4] between the VMM and the guest. Some have shown that the VMM can infer a great deal of information about the guest (both the applications and the kernel) through indirect means [8, 7]. These techniques have particularly useful applications in security, where trust can be placed in the VMM to detect attacks and malicious processes [5, 9, 18, 15].

While these methods glean useful information about the guest, the information only flows in one direction. Further, the guest cannot assist in providing the information, limiting the VMM to coarse-grained observations and decisions. In contrast, *Paravirtualization* makes the guest OS aware of the underlying VMM, allowing it to provide information directly to the VMM [2]. However, the paravirtualized interface can only be utilized directly by the guest. The VMM sees this interface as an extension to its existing event-driven model.

Symbiotic virtualization allows information to flow both ways between the VMM and guest [11], where the guest OS optionally exposes a software interface directly to the VMM. The VMM can then call into guest-context code. This approach also requires that the guest be aware of the VMM and implies modifications to the guest OS. It would certainly be beneficial to have a two-way interface that requires no *a priori* modification to the guest. With GEARS, we aspire to extend service implementations into the guest itself, even without guest knowledge or cooperation.

Secure in-VM monitoring, or SIM [20], also utilizes guest-context code execution, but it is aimed primarily at security applications, not virtual services. Further, SIM requires some degree of guest modification. Namely, it assumes that a paravirtualized driver is present in the guest. This driver is necessary for running VMM-trusted code in guest context. GEARS requires no such preconditions in order to forcefully and transparently affect guest execution.

One of the tools that GEARS employs is system call interposition. This technique has been explored in the context of several hypervisors, including Xen [3], KVM [17], and QEMU [13]. Onoe et al. present a method to filter system calls based on security policies [16]. MAVMM, a custom, lightweight VMM designed for malware detection [15] also utilizes system call interception. Ether [6] uses the same interception mechanism, but focuses on keeping its presence undetectable by malware. While system call interposition is an important technique in the GEARS tool set, it alone does not afford the ability to spread virtual services across virtualization layers. Ether and VAMPiRE [22] have the interesting capability to insert breakpoints into guest processes to implement instruction stepping, but they do not enable the broad range of services offered by the code injection facilities in the GEARS framework.

3. EXAMPLE SERVICES

When a VMM utilizes its higher privilege level to enable or enhance functionality, optimize performance, or otherwise modify the behavior of the guest in a favorable manner, it is said to provide a service to the guest. VMM-based services are usually provided transparently to the guest without its knowledge (e.g. via a virtual device). We now consider several example services that could profit from GEARS.

Overlay networking acceleration An important service for many virtualized computing environments is an overlay networking system that provides fast, efficient network connectivity among a group of VMs and the outside world, regardless of where the VMs are currently located. Such an overlay can also form the basis of an adaptive/autonomic environment, as described in Section 2. A prominent challenge for overlay networks in this context is achieving low latency and high throughput, even in high performance settings, such as supercomputers and next-generation data centers. We show in Section 7.3 how GEARS can enhance an existing overlay networking system with a guest-context component.

MPI acceleration MPI is the most widely used communication interface for distributed memory parallel computing. In an adaptive virtualized environment, two VMs running an application communicating using MPI may be co-located on a single host. Because the MPI library has no way of knowing this, it will use a sub-optimal communication path between them. An MPI acceleration service would detect such cases and automatically convert message passing into memory copies and/or memory ownership transfers. Section 7.4 outlines the design of this service.

Procrustean services While administrators can install services or programs on guests already, this task must be repeated many times. Furthermore, because the administrators of guests and those of provider hosts may not be the same people, providers may execute guests that are not secure. GEARS functionality would permit the creation of services that would automatically deploy security patches and software updates on a provider's guests.

4. GUEST-CONTEXT VIRTUAL SERVICES

Services that reside within the core of the VMM have the disadvantage of relying on the mechanism by which control is transferred to the VMM. A VMM typically does not run until an exceptional situation arises, such as the execution of a privileged instruction (a direct call to the VMM in the case of paravirtualization) or the triggering of external or software interrupts. Much like in an operating system, the transition to the higher privilege level, called an *exit*, introduces substantial overhead. Costly exits remain one of the most prohibitive obstacles to achieving high-performance virtualization.

Eliminating these exits can, thus, improve performance considerably. The motivation is similar to minimizing costly system calls to OS code in user-space processes. Modern Linux implementations, for example, provide a mechanism called *virtual system calls*, in which the OS maps a read-only page into every process's address space on start-up. This page contains code that implements commonly used services and obviates the need to switch into kernel space. If the implementation of a VMM service could be pushed up into the guest in a similar manner, more time would be spent in direct execution of guest code rather than costly invocations of the VMM. This is precisely what GEARS seeks to achieve, and this ability to eliminate exits will, perhaps, become most clear as we discuss our fast system call exiting utility in Sections 6 and 7.

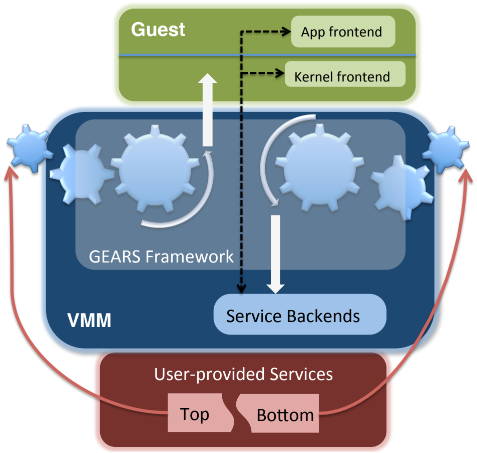

Figure 1: GEARS services are broken into two parts; one exists in the guest and the other in the VMM. GEARS takes both parts provided as source code and uses several utilities to register and manage execution of the service.

Moving components of a service implementation into the guest can not only improve performance, but also enable services that would otherwise not be feasible. In particular, guest-context services have a comprehensive view of the state of the guest kernel and application. These services can make more informed decisions than those implemented in a VMM core, which must make many indirect inferences about guest state. The VMM must reconstruct high-level operations based on the limited information that the guest exposes architecturally. Moreover, in order to manipulate the state of a guest kernel or application, the VMM must make many transformations from the low-level operations that the guest exposes to high-level operations that affect guest execution. While services certainly exist that can accomplish this transformation, their implementation would, perhaps, become more elegant operating at the same semantic level as the guest components they intend to support.

These services are also easier for the developer to design. Rather than having to effectively reverse engineer a particular execution path in the guest from the VMM's limited perspective, the developer can reason about the service at a high level, avoiding the intricacies introduced by the semantic gap.

GEARS employs a tiered approach, which involves both the host and the VMM, to inject and run services in guest context. The process is outlined in Figure 1. Users (service developers) provide standard C code for the VMM without needing extensive knowledge of VMM internals. This makes the procedure of implementing a service straight-forward, enabling rapid development. The code provided is a service implementation, split into two clearly distinguishable parts. We refer to these as the *top-half* and the *bottom-half*. The top-half is the portion of the service that will run in the guest-context. The bottom-half, which may not always be present, resides within a host kernel module readily accessible to the VMM. The top-half will call into its respective bottom-half if it requires its functionality. The bottom-half can similarly invoke the top-half, allowing for a two-way interaction between the service components.

The code for the top-half must adhere to a guest-specific format. However, GEARS provides host-resident utilities that transform the code appropriately. Hence, from the user's perspective, writing the top-half of a guest-context

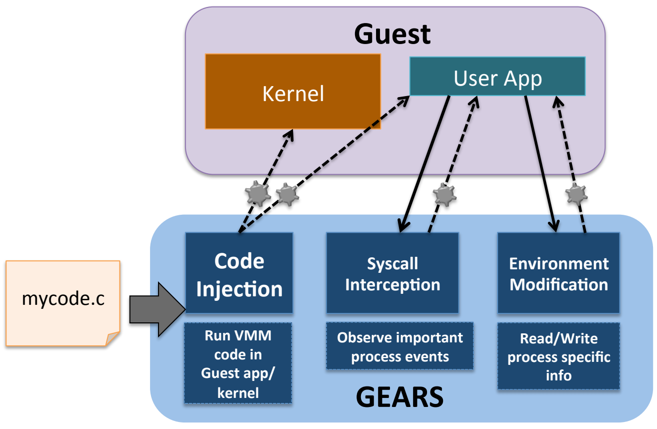
Guest
Kernel
User App
mycode.c
Code Injection
Run VMM code in Guest app/ kernel
Syscall Interception
Observe important process events
Environment Modification
Read/Write process specific info
GEARS

Figure 2: GEARS tools used to implement guest-context services. Each tool is shown with a box indicating its primary function. Dashed lines with gears indicate the ability to modify control flow in the guest. Solid lines show that a tool is used for passively extracting information from the guest. Note that all of these tools can be used in different ways to alter the guest execution path.

service implies little more requisite knowledge than the ability to write a normal program or kernel module for the guest in question. GEARS must simply provide the transformation utilities appropriate for that guest. If the service requires access to, or assistance from the VMM core, the developer can design a bottom half by writing a host kernel module that implements the relevant interface. We outline how this module connects with the VMM in Section 6.5.

The notion of leaving service implementations entirely up to the user allows a clean separation between the framework and the services that it enables. GEARS provides the necessary tools to create cross-layer services, and users are responsible for using this platform to design innovative guest-VMM interactions.

5. GEARS

We now provide a high-level description of the specific tools used in the GEARS framework. These tools allow for a broad range of feasible services, some of which we illustrated in Section 3.

To enable guest-context services, the VMM must provide a mechanism that can place components directly within the guest. GEARS implements this mechanism with the code injection tool. Further, the VMM must have a way to select an appropriate time at which to perform this placement. The GEARS system call interception utility provides one way of accomplishing this. Finally, in order for the VMM to control guest execution paths at a higher level, e.g. for library calls, it must have the ability to modify the environment passed to the process. GEARS does this with process environment modification. Because these three conditions alone can facilitate the creation of guest-context services, we claim that GEARS provides the necessary and sufficient tools to accomplish this task.

5.1 System call interception

System call interception allows a VMM to monitor the activity of the guest at a fine granularity. Normally, system calls are not exceptional events from the standpoint of the VMM. However, system calls are commonly triggered using software interrupts, which modern hardware allows VMM interception of. Once the VMM can track the execution of system calls, it can provide a wide range of services to the guest, such as sanity checking arguments to sensitive kernel code or matching system call patterns as in [13].

5.2 Process environment modification

System call interception enables the VMM to essentially see the creation of every process in the guest through calls to `execve`, for example. The interception is done before the system call even starts, so the VMM has the option to modify the guest's memory at this point. One useful thing it can do is modify the environment variables that the parent process passes on to its child.

There are certain environment variables that are particularly useful. One is the `LD_PRELOAD` variable, which indicates that a custom shared library should be given precedence over the one originally indicated. This variable gives the VMM an opportunity to directly modify the control flow of a guest application. Other interesting environment variables affecting control flow include `LD_BIND_NOW` and `LD_LIBRARY_PATH`. The very fact that the Linux kernel itself uses the environment to pass information to the process (e.g. with the `AT_SYSINFO` variable) opens up a broad range of interesting possibilities.

Environment variables can not only be modified, but also, with careful treatment of memory, added or removed. This introduces the potential for the VMM to provide information directly to the guest application without the need for paravirtualization. The guest OS would not need to be aware of the underlying VMM. Instead, VMM-awareness could vary on an application by application basis. This would allow developers to make rapid optimizations to utilize VMM services. As we will later discuss in detail, these developers could even implement their own VMM service with minimal effort. Notice that this is a marked divergence from the usual reliance of user space applications on the operating system's ABI.

5.3 Code injection

Code injection is perhaps the most unique mechanism in the GEARS framework. Because it allows the VMM to run arbitrary code in the context of the guest without any cooperation or knowledge on the part of the guest OS or application, it is the core tool enabling guest-context services.

We employ two types of code injection—user-space and kernel-space. User-space injection allows the VMM to map a piece of trusted code into the address space of a user-space process. On exits, the VMM can invoke this code manually or redirect user-space function calls to it by patching the process binary image. The latter requires more complex techniques that we are currently developing.

GEARS can also inject code into a guest that will dynamically link with the libraries mapped into the applications' address spaces. Our current example services do not utilize this GEARS feature and we leave its detailed description to future work.

The other type of injection is kernel-space code injection, which relies on the ability to inject code into a user-space process. Injected kernel code must currently be implemented in a kernel module compiled for the guest. We use user-space code injection to write the module into the guest file system and subsequently insert it into the guest kernel.

Figure 2 shows the GEARS tools and outlines how they are used together to control and modify guest execution. These particular tools are the fundamental components that allow a VMM to push service implementations into the guest and affect its operation using guest semantics.

Component	**Lines of Code**
System Call Interception	833
Environment Modification	683
Code Injection	915
Total	2431

Figure 3: Implementation complexity for GEARS and its constituent components.

6. IMPLEMENTATION

We now outline the implementation of the tools comprising the GEARS framework. GEARS is implemented within the Palacios Virtual Machine Monitor, an open-source, embeddable VMM actively developed and maintained by researchers at several institutions [10, 12]. Its source code is available for free online at `v3vee.org`, and will soon include the latest GEARS framework.

While the current GEARS implementation is targeted at Linux guests, it consists of relatively few components, each of which rely on features provided almost universally by modern OSes and architectures. This means that porting GEARS for other kinds of guests entails no great effort. Figure 3 shows the size of the GEARS codebase. Each component is relatively compact. GEARS is currently implemented as a set of extensions to the Palacios VMM, and would likely become even more compact if integrated into the hypervisor core.

GEARS currently focuses on AMD hardware, and the port to Intel hardware is a work in progress. There are no fundamental limitations that will make the implementation of GEARS on Intel any more challenging since the hardware virtualization extensions that we utilize are provided by both vendors. While the current GEARS framework has many simplifications to ease implementation, we note that the purpose of this paper is to demonstrate its ability to enable transparent guest-context services—there is no loss of general applicability of GEARS to hypervisor software.

6.1 Hooking system calls

Both Intel VT and AMD-V hardware virtualization support hypervisor interception of software interrupts (`INTn` instructions). This provides a fairly simple way to catch the execution of system calls in guests by looking specifically for `INT 0x80` instructions. Once this instruction is intercepted and handled in the VMM, the original software interrupt can be injected back into the guest.

Most modern 64-bit software uses AMD's more recently introduced `SYSCALL` instruction to invoke system routines, but hardware support for its interception is not yet provided. Several have worked around this issue by intercepting the write to the model-specific register (MSR) containing the system call target address (named LSTAR in the case of AMD) [3, 13, 16, 15, 6].

GEARS supports both of these implementations. However, there are some limitations because of the simplicity of our current implementation. In the case of `INT 0x80` on AMD hardware, situations can arise where the system call invoked with this instruction causes another faulting condition that triggers an exit to the VMM—namely, a page fault. This is particularly noticeable with `fork()`. Normally, the system call is restarted by the kernel, but the VMM implementation will have already incremented the instruction pointer, so it will return to the next instruction, and behave as if the system call were skipped. Thus, without hardware support, the VMM must support full emulation of the software interrupt instruction. However, this problem can be avoided if the hardware supports nested paging, as these page faults will not trap to the VMM. We therefore only support the `INT 0x80` instruction on machines supporting nested paging. Without hardware support for `INTn` interception, the VMM can overwrite the appropriate entry in the IDT with either an illegal instruction or a `VMMCALL`. When intercepting `INT 0x80`, GEARS currently assumes the necessary hardware support.

Interception of the `SYSCALL` instruction can be achieved by guaranteeing that the register containing the system call entry point (LSTAR on AMD hardware) points to an unmapped memory address, causing a trap to the VMM by page fault. However, finding an address that will trap under nested paging is more difficult, as the VMM is invoked much less frequently on page faults. We currently only intercept `SYSCALL` instructions when the machine is using shadow paging. The VMM can also unset the SCE bit in the EFER MSR, causing an illegal instruction exception when a `SYSCALL` instruction is executed. This exception will cause a trap to the VMM, at which point the instruction at the RIP is checked against the `SYSCALL` opcode. GEARS currently only supports the method using page faults.

While these methods of interception are certainly viable, they introduce substantial performance overhead. These techniques require that *every* system call trigger an exit to the VMM. We have developed a mechanism using GEARS that can virtually eliminate this overhead. We call this the Fast System Call Exiting Utility. This mechanism bears some resemblance to the *exit-gates* presented in SIM [20], but requires no guest cooperation. During guest boot, GEARS records the address written to the appropriate STAR register by the guest kernel and subsequently emulates the write. After the guest has booted, GEARS can use code injection to insert a kernel module into the guest. This kernel module contains a compact stub intended to compare every system call number to a bit vector mapped into the kernel address space by the VMM. During module initialization, the VMM is provided the address of this stub, which it writes to the STAR register. All system calls invoked by the guest are then redirected to this stub. GEARS can dynamically update the system call bit-vector, indicating which system calls should trap to the VMM. At any time, this module can be forcefully removed from the guest kernel, completely disabling system call exiting when not required.

6.2 Process monitoring

With the system call interception facility, we can easily track the creation of new processes by looking for calls to `fork` and `execve`. For our purposes in this paper, `execve` is of particular interest because it represents the moment when a process receives a new address space and begins executing code independent of that in its parent's address space. In Linux, this call takes an executable object as an argument and runs it within the new process. We allow the VMM to hook the execution of specified binaries by comparing the arguments to `execve` to binaries registered dynamically within the VMM.

6.3 Modifying process environments

Along with the binary to execute, and arguments passed to the new process, `execve` passes a pointer to an array of environment variables. Since the system call is intercepted before process creation, the VMM sees the state of the parent process. This allows the VMM to change the environment that the parent passes to the child process.

We analyze environment variables by tracing the environment pointer, given as an argument, back to all of the strings that it points to. The VMM can then modify these strings at will. However, more work must be done if a modification results in a string larger than the original. The VMM shifts the preceding components up past the stack pointer to make room for the overlap and repairs pointers by the appropriate offsets. This is essentially a form of code relocation, and also allows for the addition of an arbitrary number of environment variables to a child process's environment.

In this paper, we utilize this facility to redirect library calls in targeted binaries to custom library wrappers using the `LD_PRELOAD` environment variable.

6.4 Code injection

GEARS uses two modes of code injection—immediate injection and *exec-hooked* injection. When using immediate injection, the VMM, after receiving code to inject at run-time, will inject the code into any user-space process as soon as it can. There are many ways of tracking user-space events, but GEARS currently uses system calls for this purpose. Thus, the code will be injected at the subsequent system call intercepted by the VMM.

Exec-hooked injection allows a means by which to inject at more specific points. With this method, a user specifies a binary file in the guest, which, upon execution, will trigger the injection of the provided code. While this provides more control, the injection actually happens in the parent process, as the interception of the system call happens before the new process is created. One potential way of addressing this issue is to correlate CR3 values with calls to `execve`, and looking for future user-space events matching the recorded value.

The VMM injects code into a process using several steps of binary patching. We outline the steps comprising immediate code injection below:

1. Intercept next system call
2. Inject `mmap()` call into process. This will allocate space for code in a mapped region marked as `rwx`.
3. Touch every page in the region to ensure guest kernel allocates pages
4. Inject a page fault for a page in the region
5. Subsequently copy a page of code into the faulted page
6. Continue steps 4 and 5 until all code is copied
7. Finally, set RIP to start of injected code

Once the code has been copied, the VMM can immediately set the instruction pointer to the code's entry point so that it begins executing on guest entry. The common way that code is compiled in Linux directs the loader to map the ELF binary into memory twice, once read-only for code, and once read-write for data. The linked code will take into account the fixed offset between the mapped locations of code and data. However, we cannot guarantee the address at which `mmap` will allocate memory, so the presence of two mappings would require dynamic relocation of references from the code segment to data segment. To avoid this complication, we currently use custom linker scripts for injected code to ensure that it is mapped into memory only once. For the same reason, the injected code must be compiled to be position-independent.

We can use GEARS to modify the behavior of the guest kernel as well. Given user-space code injection, we can inject code that will forcefully write out a file to the guest file system. One kind of file that we can write out is a kernel module. This presents an opportunity to affect the operation of the guest kernel. Once GEARS writes out a module, it can subsequently insert it into the kernel. The entire process ensues without any cooperation from the guest. While GEARS could also potentially inject code into the kernel address space directly using a technique similar to that employed by user-space injection, kernel modules represent the simplest and safest way to inject service components into the guest kernel. Note that we do not claim the guest cannot *detect* the effects of the injection. We are concerned with the case in which the VMM is simply providing a beneficial service to the guest. While creating invisible guest-context services is paramount to preventing exploits, it is outside the scope of this paper. We direct the interested reader to the security-related projects outlined in Section 2.

6.5 Bottom-half interface

The bottom half of a GEARS-based VMM service can be implemented either directly in the VMM itself or as a separate kernel module for the Linux host OS. In the latter case, the kernel module can be implemented without detailed knowledge of Palacios. The module hooks to Palacios through a new *host hypercall* interface that allows hypercall implementations to be created outside of the Palacios codebase. This host interface provides the hypercall implementations with a *constrained guest access* interface that enables them to inspect and modify guest state, such as general purpose and control registers, guest address translation, and read/write access to guest physical and virtual memory. Since Palacios is host OS-independent, the implementation of these two interfaces is split between the general logic within Palacios, and a host-specific component. The host-independent logic in Palacios comprises 274 lines of C, while the host-specific component, for Linux, consists of 165 lines of C.

7. EVALUATION

In this section we evaluate the performance-sensitive component of GEARS and the prototype example services we have built.

7.1 Experimental setup

We perform all experiments on AMD 64-bit hardware. We primarily use two physical machines for our testbed:

- 2GHz quad-core AMD Opteron 2350 with 2GB memory, 256KB L1, 2MB L2, and 2MB L3 caches. We refer to this machine as *vtest*.
- 2.3GHz 2-socket, quad-core (8 cores total) AMD Opteron 2376 with 32GB of RAM, 512KB L1, 2MB L2, and 6MB L3 caches, called *lewinsky*.

Both of these machines have Fedora 15 installed, with Linux kernel versions 2.6.40 and 2.6.42, respectively. The guests we use in our testbed are Linux kernel versions 2.6.38. All experiments were run using the Palacios VMM configured for nested paging.

Strategy	**Latency (μs)**
guest	4.83
guest+intercept	10.24

Figure 4: Average system call latency for getpid system call using INT 80 exiting. Because getpid is such a simple system call, the latency difference represents the fixed cost of system call exiting. This form of exiting roughly doubles the fixed cost of all system calls.

7.2 System call interception

The primary performance issue associated with GEARS is the cost of intercepting system calls. As we mentioned in Section 6, we must use either `INT 0x80` interception or `SYSCALL` interception until the machine is booted. GEARS can then forcibly inject the fast system call exiting utility into the guest to all but eliminate the overhead of system call exiting. Since our main concern is the effect of this overhead on applications, we present the performance of the fast system call exiting utility.

Our system call micro-benchmark suite consists of two timing programs that measure the time to completion for a single system call. Each timing run essentially consists of the following sequence of instructions:

```
time:
    cpuid
    rdtsc
    mov $SYSCALL_NR, %rax
    syscall
    cpuid
    rdtsc
```

The `cpuid` instructions enforce serialized execution, avoiding situations in which `rdtsc` instructions might be rearranged by the instruction scheduler. Each experiment consists of 1000 trial runs. We take the minimum of these runs to ignore any intermittent influences such as context switches.

Figures 4 and 5 show the latency associated with the `getpid` system call for software interrupt interception and selective system call exiting, respectively. This particular call is one of the simpler system calls in Linux, so these numbers represent the fixed cost introduced by system call interception. The row labeled *guest* represents a standard guest with no GEARS extensions. The *guest+intercept* row indicates a GEARS-enabled guest using system call exiting. For `INT 0x80` exiting, every system call results in a VMM exit, so this technique has a significant effect on the latency of the system call path. In the case of selective exiting, all system calls are routed through the injected kernel module, but do not cause an exit to the VMM unless marked to do so. In this experiment, no system calls were marked for exiting. Thus, this figure shows the overhead introduced to the system call path only by this rerouting process. Whether or not any particular system call is marked to trigger a VMM invocation, this shows the bare minimum cost that must be paid for system call exiting. Selective system call exiting adds only about 6% latency overhead to a standard guest—less than one microsecond.

Figure 6 shows the *bandwidth* cost of selective system call exiting. For this experiment, we chose a system call that varies in performance according to the amount of data handled. We measured the bandwidth of the `write` system call with varying buffer sizes. Notice that the difference in bandwidth is virtually negligible. Figure 7 displays this difference

Strategy	Latency (μs)
guest	4.26
guest+intercept	4.51

Figure 5: Average system call latency for getpid system call using selective exiting. The overhead is significantly smaller than INT 80 exiting.

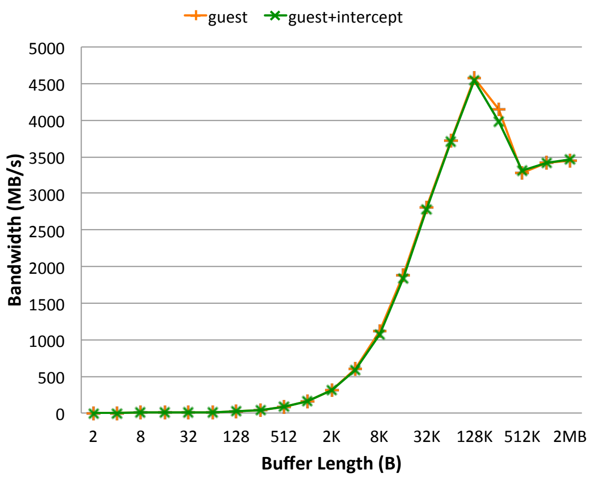

Figure 6: Bandwidth (MB/s) vs. bytes transferred for a write system call. These lines track very closely, demonstrating how little selective system call exiting affects the bandwidth of a data-intensive system call.

more clearly with the ratio of bandwidth with a standard guest over the bandwidth of a GEARS-provisioned guest. The ratio approaches one as the fixed overhead cost of system call exiting is amortized over the amount of data being written.

7.3 VNET/P accelerator

VNET/P [23] is an overlay networking system with a layer 2 abstraction implemented inside the Palacios VMM. It currently achieves near-native performance in the 1 Gbps and 10 Gbps switched networks common in clusters today, and we intend for it to work at native speeds on even faster networks, such as InfiniBand, in the future. At the time of this writing, VNET/P can achieve, with a fully encapsulated data path, 75% of the native throughput with 3-5x the native latency between directly connected 10 Gbps machines. We use GEARS tools to further improve this performance.

The throughput and latency overheads of VNET/P are mostly due to guest/VMM context switches, and data copies or data ownership transfers. We can potentially reduce the number of context switches, and the volume of copies or transfers, by shifting more of the VNET/P data path into the guest itself. In the limit, the entire VNET/P data path could execute in the guest with guarded privileged access to the underlying hardware. In this paper, we explore an initial step towards this goal that does not involve privileged access and aims at reducing the latency overhead as a proof-of-concept.

Figure 8 illustrates this initial proof-of-concept implementation of the VNET/P Accelerator Service. In the baseline VNET/P data path, shown on the left, raw Ethernet packets sent from a VM through a virtual NIC are encapsulated and forwarded within the VMM and sent via a physical NIC. In the VNET/P Accelerator data path, shown on the right, the

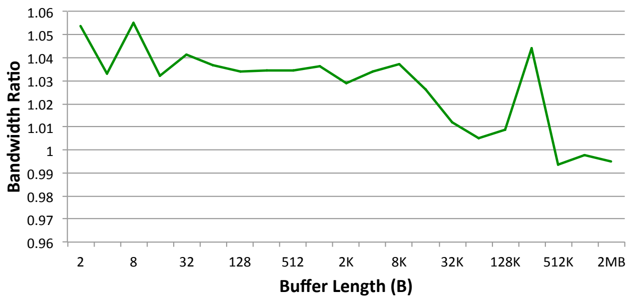

Figure 7: Bandwidth ratio of standard guest to guest with GEARS. This ratio approaches unity as the fixed system call exiting cost is amortized.

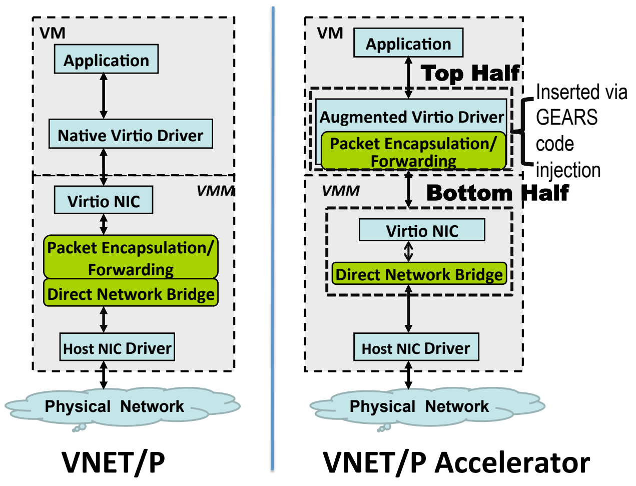

Figure 8: Implementation of the prototype VNET/P Accelerator.

encapsulation and forwarding functionality of VNET/P resides within the guest as part of the guest's device driver for the virtual NIC. This augmented device driver kernel module is uncooperatively inserted into the guest kernel using the GEARS code injection tool. The augmented driver then delivers Ethernet frames containing the encapsulated packets to the virtual NIC. In our implementation, the driver that has been augmented is the Linux virtio NIC driver. The backend virtio NIC implementation in Palacios has no changes; it is simply bridged to the physical NIC.

The implementation complexity of the proof-of-concept VNET/P accelerator is shown in Figure 9, which illustrates that few changes are needed to split VNET/P functionality into a top half and a bottom half. The control plane of VNET/P remains in the bottom half in the VMM; only the encapsulation and forwarding elements move into the top half that GEARS injects into the guest.

Figure 10 depicts the performance of the initial, proof-of-concept VNET/P Accelerator. Here, the round-trip latency and throughput is measured between a VM running on the *vtest* machine and a VM running on an adjacent machine that does not use the accelerator. We measure latency using ping with 1000 round-trips. The throughputs are measured using ttcp, where both TCP and UDP throughput are reported. We run ttcp with a 6400 byte buffer, 10000 packets sent, and a standard 1500 byte MTU. We compare accelerated VNET/P with standard VNET/P and native performance between the two host machines without virtualization or overlays.

Component	Lines of Code
vnet-virtio kernel module (Top Half)	329
vnet bridge (Bottom Half)	150
Total	479

Figure 9: Implementation complexity of prototype VNET/P Accelerator Service. The complexity given is the total number of lines of code that were changed. The numbers indicate that few changes are necessary to port VNET/P functionality into a Linux virtio driver module

Benchmark	Native	VNET/P	VNET/P Accel
Latency			
min	0.082 ms	0.255 ms	0.205 ms
avg	0.204 ms	0.475 ms	0.459 ms
max	0.403 ms	2.787 ms	2.571 ms
Throughput			
UDP	922 Mbps	901 Mbps	905 Mbps
TCP	920 Mbps	890 Mbps	898 Mbps

Figure 10: VNET/P accelerator results.

The VNET/P Accelerator achieves the same bandwidth as VNET/P, and both are as close to native as possible given that encapsulation is used. The VNET/P Accelerator achieves a modest improvement in latency compared to VNET/P (20% minimum, 3% average, 8% maximum). It is important to note that this accelerator is a proof-of-concept of using GEARS to build a guest-context service. Our next step will require guarded privileged execution of injected code, which is in progress. We should note, however, that the implementation complexity (Figure 9) will not change significantly, since guarded privileged execution is GEARS-level functionality, not service-level.

7.4 MPI accelerator

Consider an MPI application executing within a collection of VMs that may migrate due to decisions made by an administrator, an adaptive computing system, or for other reasons. The result of such migrations, or even initial allocation, may be that two VMs are co-located on the same host machine. However, the MPI application and the MPI implementation itself are oblivious to this, and will thus employ regular network communication primitives when an MPI process located in one VM communicates with an MPI process in the other. VNET/P will happily carry this communication, but performance will be sub-optimal.

Fundamentally, the communication performance in such cases is limited to the main memory copy bandwidth. Ideally, matching MPI send and receive calls on the two VMs would operate at this bandwidth. We assume here that the receiver touches all of the data. If that is not the case, the performance limit could be even higher because copy-on-write techniques might apply. The goal of the MPI Accelerator service is to do precisely this transformation of MPI sends and receives between co-located VMs into memory copy operations.

Building such an MPI Accelerator purely within the VMM would be extremely challenging because MPI send and receive calls are *library routines* that indirectly generate system calls and ultimately cause guest device driver interactions with the virtual hardware the VMM provides. It is these virtual hardware interactions that the VMM sees. In order to implement an MPI Accelerator service, it would be necessary to reconstruct the lost semantics of MPI operation. The ability to discern the MPI semantics *from the guest application* is the key enabler of our MPI Accelerator implementation.

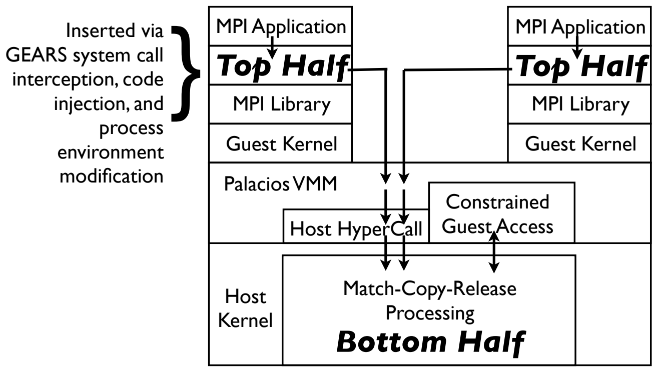

Figure 11: Implementation of the MPI Accelerator service for co-located VMs. This illustrates the fast path between an MPI_Send and its matching MPI_Recv.

GEARS provides two essential tools that the MPI Accelerator service leverages: (a) user space code injection, and (b) process environment modification. At any point during VM execution, the service uses (a) to inject and run a program that creates a file in the VM. The file contains a shared library that is an `LD_PRELOAD` wrapper for MPI. The system then uses (b) to force `exec()`s of processes to use the `LD_PRELOAD` wrapper. This can be limited to specific executables by name if desired. The wrapper installs itself between the processes and the MPI shared library such that MPI calls bind to the wrapper. The wrapper, which constitutes the top half of the service can then decide how to process each MPI call in coordination with the bottom half of the service that resides in the VMM. The top and bottom halves communicate using service-specific hypercalls.

In our prototype implementation, illustrated in Figure 11, we focus on the blocking MPI_Send and MPI_Recv calls. The top half intercepts the appropriate MPI calls as follows:

- MPI_Init() : After normal initialization processing in MPI, this call also notifies the bottom half of this MPI process, including its name, arguments, and other parameters. It registers the process for consideration with the service.
- MPI_Finalize(): Before normal de-initialization in MPI, this call notifies the bottom half that the process can be unregistered.
- MPI_Comm_rank(): After a normal ranking in MPI, this call notifies the bottom half of the process's rank.
- MPI_Send(): The wrapper checks to see if this is an MPI_Send() that the bottom half can implement. If it is not, it hands it to the MPI library. If it is, it touches each page of the data to assure it is faulted in, and then hands the send request to the bottom half and waits for it to complete the work. If the bottom half asserts that it cannot, the wrapper defaults to the MPI library call.
- MPI_Recv(): This is symmetric to MPI_Send().

The hypercalls also implicitly carry a pointer to Palacios's structures that represent the VM and the current virtual core's state, allowing ready access to the CR3 register, which contains a pointer to the current page table. This information is used to identify a guest process. It should be clear that this model can readily be extended to support a wider range of MPI functionality.

The bottom half of the service is implemented as a Linux kernel module that supplies the hypercall implementations. This module is relatively straightforward. It maintains a table of registered processes, as well as their current rank,

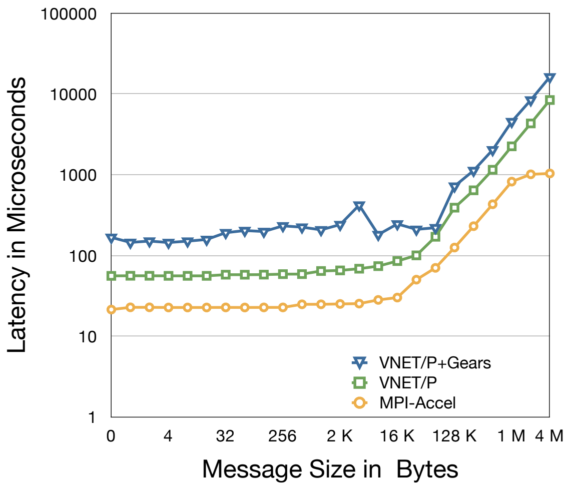

Figure 12: Performance of MPI Accelerator Service on OSU MPI Latency Benchmark running in two co-located VMs on *lewinsky* test machine. For small messages, it achieves a 22 μs latency, limited by system call interception and hypercall overheads. For large messages, the MPI accelerator approaches the maximum possible performance given the memory copy bandwidth of the machine (4.6 GB/s).

send state, and receive state. The table can be queried by the VM/virtual core/CR3/executable name combination that uniquely identifies registered MPI processes across all of the co-located VMs on the host. When an MPI process initiates an applicable MPI_Send, the guest exists due to the top half's hypercall, and Palacios redirects execution into the kernel module's handler. The handler attempts to find a matching pending receive. If it can, it copies the data from the guest virtual addresses in the sending VM to the guest virtual addresses in the receiving VM. It then writes the appropriate return code to the receiving thread, which waits in the kernel module as well, and subsequently releases it. Both hypercalls return to their respective guests, and the transfer is seen as complete by both of them. If the receive is not pending or no matching receive is yet available, the sender saves the send side state into its own entry in the table, marks itself as pending, and waits for the receiver. In this instance, the receiver will perform the copy operation when it arrives and then release the sender. Note that if the copy operation fails, for example, due to a guest virtual address that is not currently paged in on the guest, the copy simply stops and the error is signaled to the top half, which falls back on the MPI library to complete the transfer.

Our implementation is intended as a proof of concept demonstrating the utility of GEARS tools and advancing the overall argument of this paper. Nonetheless, it also performs quite well. We have run the OSU MPI Latency benchmark [1] (osu_latency) between two co-located VMs using VNET/P, VNET/P with the GEARS tools enabled in Palacios, and with the MPI Accelerator active. We use the MPICH2 MPI library [14] for our measurements. Our performance measurements are taken on the *lewinsky* machine, previously described. The results are shown in Figure 12. It is important to note that for larger message sizes, the message transfer time is dominated by the machine's memory bandwidth. According to the STREAM benchmark, the machine has a main memory copy bandwidth of 4.6 GB/s. Our results suggest that we approach this—specifically, the MPI latency for 4 MB messages implies a bandwidth of 4

Component	Lines of Code
Preload Wrapper (Top Half)	345
Kernel Module (Bottom Half)	676
Total	1021

Figure 13: Implementation complexity of MPI Accelerator.

GB/s has been achieved. For small messages the MPI latency is approximately 22 μs (about 50,000 cycles). The small message latency is limited by system call interception, exit/entry, and hypercall overheads. Here, GEARS selective system call interception is not enabled. Using it would further reduce the overhead for small messages.

The figure also shows the performance of using VNET/P for this co-located VM scenario. The "VNET/P" curve illustrates the performance of VNET/P without any GEARS features enabled. Without system call interception overheads, we see that VNET/P achieves a 56 μs latency for small messages and the large message latency is limited due to a transfer bandwidth of a respectable 500 MB/s. The "VNET/P+Gears" curve depicts VNET/P with the GEARS features enabled and illustrates the costs of non-selective system call interception. The small message latency grows to 150 μs, while the large message latency is limited due to a transfer bandwidth of 250 MB/s. In contrast to these, the MPI Accelerator Service, based on GEARS, is achieving 1/3 the latency and 8 times the bandwidth, approaching the latency limits expected due to the hypercall processing and the bandwidth limits expected due to the system's memory copy bandwidth. Note that the impact of GEARS system call interception on the MPI Accelerator's small message latency is much smaller than its impact on VNET/P. This is not a discrepancy. With the MPI Accelerator, far fewer system calls are made per byte transfered because the injected top-half intercepts each MPI library call before it can turn into multiple system calls.

Figure 13 illustrates that the service implementation is quite compact. The GEARS tools are the primary reason for the service's feasibility and compactness.

8. CONCLUSIONS AND FUTURE WORK

GEARS is a set of tools that enable the creation of *guest-context services* which span the VMM, the guest kernel and the guest application. We have shown through an implementation within the Palacios VMM that the complexity of these tools is tractable, suggesting that they could be implemented in other VMMs without great effort. GEARS in Palacios allows developers to write VMM services with relatively little knowledge of VMM internals. Further, we have shown that the implementations of the services themselves can remain relatively compact while still delivering substantial performance or functionality improvements.

We are currently investigating several aspects of guest-context services, including boundaries between guest code and code injected by the VMM. This includes the ability to safely run trusted components of a service in the guest while providing them with fully privileged hardware access. Finally, we intend to explore interfaces that provide a direct connection between the VMM and guest application.

The source code for GEARS is available as part of the open-source Palacios VMM from `v3vee.org`.

9. REFERENCES

[1] Osu micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[2] 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 *ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[3] Beck, F., and Festor, O. Syscall interception in xen hypervisor. Technical report, Institut National Polytechnique de Lorraine (INPL), 2009.

[4] Chen, P. M., and Noble, B. D. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS 2001)* (May 2001).

[5] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. 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 2008)* (March 2008).

[6] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS 2008)* (October 2008).

[7] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Antfarm: tracking processes in a virtual machine environment. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 2006)* (June 2006).

[8] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS 2006)* (October 2006).

[9] King, S. T., Chen, P. M., Wang, Y.-M., Verbowski, C., Wang, H. J., and Lorch, J. R. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (May 2006).

[10] 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).

[11] Lange, J. R., and Dinda, P. Symcall: symbiotic virtualization through vmm-to-guest upcalls. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE 2011)* (March 2011).

[12] Lange, J. R., Dinda, P., Hale, K. C., and Xia, L. An introduction to the palacios virtual machine monitor—version 1.3. Tech. Rep. NWU-EECS-11-10, Department of Electrical Engineering and Computer Science, Northwestern University, November 2011.

[13] Li, B., Li, J., Wo, T., Hu, C., and Zhong, L. A vmm-based system call interposition framework for program monitoring. In *16th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2010)* (December 2010).

[14] Lusk, E., Doss, N., and Skjellum, A. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing 22* (1996), 789–828.

[15] Nguyen, A., Schear, N., Jung, H., Godiyal, A., King, S., and Nguyen, H. Mavmm: Lightweight and purpose built vmm for malware analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2009)* (December 2009).

[16] Onoue, K., Oyama, Y., and Yonezawa, A. Control of system calls from outside of virtual machines. In *Proceedings of the 2008 ACM symposium on Applied computing (SAC 2008)* (March 2008).

[17] Pfoh, J., Schneider, C., and Eckert, C. Nitro: Hardware-based system call tracing for virtual machines. In *Proceedings of the International Workshop on Security (IWSEC 2011)* (November 2011).

[18] Riley, R., Jiang, X., and Xu, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection (RAID 2008)* (September 2008).

[19] Ruth, P., Rhee, J., Xu, D., Kennell, R., and Goasguen, S. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In *Proceedings of the 3rd International Conference on Autonomic Computing (ICAC 2006)* (June 2006).

[20] Sharif, M. I., Lee, W., Cui, W., and Lanzi, A. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS 2009)* (November 2009).

[21] Sundararaj, A., Gupta, A., and Dinda, P. Increasing application performance in virtual environments through run-time inference and adaptation. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (July 2005).

[22] Vasudevan, A., and Yerraballi, R. Stealth breakpoints. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)* (December 2005).

[23] Xia, L., Cui, Z., Lange, J., Tang, Y., Dinda, P., and Bridges, P. VNET/P: Bridging the cloud and high performance computing through fast overlay networking. In *Proceedings of 21st International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2012)* (June 2012).

[24] Xu, J., and Fortes, J. A multi-objective approach to virtual machine management in datacenters. In *Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2011)* (June 2011).