Antfarm: Tracking Processes in a Virtual Machine Environment

Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
Department of Computer Sciences
University of Wisconsin, Madison
{*stjones,dusseau,remzi*}@*cs.wisc.edu*

Abstract

In a virtualized environment, the VMM is the system's primary resource manager. Some services usually implemented at the OS layer, like I/O scheduling or certain kinds of security monitoring, are therefore more naturally implemented inside the VMM. Implementing such services at the VMM layer can be complicated by the lack of OS and application-level knowledge within a VMM. This paper describes techniques that can be used by a VMM to independently overcome part of the "semantic gap" separating it from the guest operating systems it supports. These techniques enable the VMM to track the existence and activities of *operating system processes*. Antfarm is an implementation of these techniques that works without detailed knowledge of a guest's internal architecture or implementation. An evaluation of Antfarm for two virtualization environments and two operating systems shows that it can accurately infer process events while incurring only a small 2.5% runtime overhead in the worst case. To demonstrate the practical benefits of process information in a VMM we implement an anticipatory disk scheduler at the VMM level. This case study shows that significant disk throughput improvements are possible in a virtualized environment by exploiting process information within a VMM.

1 Introduction

Virtual machine technology is increasingly being deployed on a range of platforms from high-end servers [4, 24, 25] to desktop PCs [22]. There is a large and growing list of reasons to use virtualization in these diverse computing environments, including server consolidation [25], support for multiple operating systems (including legacy systems) [11], sandboxing and other security features [9, 16], fault tolerance [3], and optimization for specialized architectures [4]. As both software [6] and hardware support [12, 13] for zero-overhead virtualization develops, and as virtualization is included in dominant commercial operating systems [2], we expect virtualized computing environments to become nearly ubiquitous.

As virtualization becomes prevalent, the *virtual machine monitor* (VMM) naturally supplants the operating system as the primary resource manager for a machine. Where one used to consider the OS the main target for innovation in system services, one should now consider how to implement some of those services within a VMM [5].

The transition of some functionality from the OS into the VMM has many potential benefits. For example, by implementing a feature a single time within a VMM, it becomes available to *all* operating systems running above. Further, the VMM may be the only place where new features can be introduced into a system, as the operating system above is legacy or closed-source or both. Finally, the VMM is the *only* locale in the system that has total control over system resources and hence can likely make the most informed resource management decisions.

However, pushing functionality down one layer in the software stack into the VMM has its drawbacks as well. One significant problem is the lack of higher-level knowledge within the VMM, sometimes referred to as a *semantic gap* [5]. Previous work in virtualized environments has partially recognized this dilemma, and researchers have thus developed techniques to infer higher-level *hardware resource utilization* [4, 20, 24]. These techniques are useful because they allow a VMM to better manage the resources of the system, (*e.g.*, by reallocating an otherwise idle page in one virtual machine to a different virtual machine that could use it [24]).

In addition, some recently proposed VMM-based services use explicit information about the *software abstractions* of the operating systems running above them to bridge the semantic gap [10, 15]. However, previous work has not thoroughly explored how a VMM can *learn* about the software abstractions of the operating systems running above without the information being given explicitly to it. Being able to implicitly learn about operating systems from within a VMM is important if a guest OS is proprietary, untrusted, or is managed by a different entity than the one managing the VMM. In these cases, explicit information about the details of the guest's memory layout or implementation will be unavailable or unreliable.

In this paper, we develop a set of techniques that enable a virtual machine monitor to implicitly discover and exploit information about one of the most important operating system abstractions, the *process*. By monitoring low-level interactions between guest operating systems and the memory management structures on which they depend, we show that a VMM can accurately determine when a guest operating system creates processes, destroys them, or context-switches between them. These techniques operate without any explicit information about the guest operating system vendor, version, or implementation details.

We demonstrate the utility and efficacy of VMM-level process awareness by building an anticipatory disk scheduler [14] within a VMM. In a virtual machine environment, an anticipatory disk scheduler requires information from both the VMM and OS layers and so cannot be implemented exclusively in either. Making a VMM process aware overcomes this limitation and allows an OS-neutral implementation at the VMM layer without any modifications or detailed knowledge of the OS above. Our implementation within the VMM is able to improve throughput among competing sequential streams from processes across different virtual machines or within a single guest operating system by a factor of two or more.

In addition to I/O scheduling, process information within the VMM has several other immediate applications, especially in the security domain. For example, it can be used to detect that processes have been hidden from system monitoring tools by malicious software or to identify code and data from sensitive processes that should be monitored for runtime modification [10]. Patterns of system calls associated with a process can be used to recognize when a process has been compromised [8, 19]. In addition to detection, techniques exist to slow or thwart intrusions at the process level by affecting process scheduling [21]. Finally, process information can be used as the basis for discovering other high-level OS abstractions. For example, the parent-child relationship between processes can be used to identify groups of related processes associated with a *user*. All of these applications are feasible within a VMM only when process information is available.

Antfarm is the implementation of our process identification techniques for two different virtualization environments, Xen and Simics. Antfarm has been evaluated as applied to x86/Linux, x86/Windows, and SPARC/Linux guest operating systems. This range of environments spans two processor families with significantly different virtual memory management interfaces and two operating systems with very different process management semantics. Antfarm imposes only a small runtime overhead of about 2.4% in a worst case scenario and about 0.6% in a more common, process-intensive compilation environment.

The rest of the paper is organized as follows. In Section 2 we place Antfarm in context with related work. Then in Section 3, we cover some required background material relating to our implementation architectures and virtual machines in general. This is followed in Section 4 by a discussion of the techniques underlying Antfarm. Section 5 covers the implementation details of Antfarm. We evaluate the accuracy and overhead imposed by Antfarm in Section 6. In Section 7, we present our anticipatory scheduling case study and then conclude in Section 8.

2 Related Work

Antfarm informs a VMM about one important operating system abstraction, the process, about which it would otherwise have no information. Other research has recognized that information not explicitly available to a VMM is nevertheless useful when implementing VMM features and services.

In some cases the information relates to hardware. Disco [4], for example, determines when the guest is executing in its idle loop by detecting when it enters a low-power processor mode. VMWare's ESX Server [24] uses page sampling to determine the utilization of physical memory assigned to each of its virtual machines. Antfarm differs from these efforts in that it focuses on inferring information about processes, a software construct.

Other projects have also recognized the value of OS-level information in a VMM. In some cases, detailed version-specific memory layout information as well as the semantic knowledge to make use of that information has been exported directly to the VMM. VMI [10] does this to implement security techniques like detecting malicious, hidden processes within a guest. IntroVirt [15] uses memory layout and implementation details to enable novel host-based intrusion detection features in a VMM. Antfarm, in contrast, enables a more limited and inexact level of information to be inferred by a VMM. It does this, however, without any explicit information about memory layout or implementation of affected guests and so can be deployed in a broader set of environments.

Work by Uhlig *et al.* [23] is more similar to our own. It shows how to infer guest-level information to do processor management more intelligently in a multiprocessor environment. Specifically, they deduce when no kernel locks are held by observing when the OS above is executing in user versus kernel mode. Antfarm is complementary. It observes a different virtual resource, the MMU, to infer information about operating system processes.

Finally, as an alternative to inferring OS-level information, such knowledge could be passed explicitly from the OS to the VMM, as is done, (to some extent), in paravirtualized architectures [6, 25]. Explicit information supplied by a paravirtualized OS is guaranteed to match what is available inside the OS. By this metric, paravirtual information should be considered the gold standard of OS information within a VMM. In some important environments, however, the explicit approach is less valuable. For example, paravirtualization requires OS-level modification, which implies that functionality cannot be deployed in VMM's running beneath legacy or closed-source operating systems. For the same reasons, dependence on explicit interfaces forces innovation in the VMM that requires OS-level information to be coupled with changes to supported operating systems. Inferring guest information allows a VMM to innovate independent of the OS implementation. Finally, in the case of security applications, a guest OS cannot be trusted to report on its own activities using a paravirtualized interface because it may have been compromised and intentionally mislead the VMM.

3 Background

The techniques we describe in this paper are based on the observations that a VMM can make of the interactions between a guest OS and virtual hardware. Specifically, Antfarm monitors how a guest uses a virtual MMU to implement virtual address spaces. In this section we review some of the pertinent details of the Intel x86 and the SPARC architectures used by Antfarm. We also discuss some basic features of virtual machine monitors and the runtime information available to them.

3.1 x86 Virtual Memory Architecture

Our first implementation platform is the Intel x86 family of microprocessors. We chose the x86 because it is the most frequently virtualized processor architecture in use today. This section reviews the features of the x86 virtual memory architecture that are important for our inference techniques.

The x86 architecture uses a two-level, in-memory, architecturally-defined page table. The page table is organized as a tree with a single 4 KB memory page called the *page directory* at its root. Each 4-byte entry in the page directory can point to a 4 KB page of the *page table* for a process.

Each page table entry (PTE) that is in active use contains the address of a physical page for which a virtual mapping exists. Various page protection and status bits are also available in each PTE that indicate, for example, whether a page is writable or whether access to a page is restricted to privileged software.

A single address space is active per processor at any given time. System software informs the processor's MMU that a new address space should become active by writing the physical address of the page directory for the new address space into a processor control register (CR3). Since access to this register is privileged the VMM must virtualize it on behalf of guest operating systems.

TLB entries are loaded on-demand from the currently active page tables by the processor itself. The operating system does not participate in handling TLB misses.

An operating system can explicitly remove entries from a TLB in one of two ways. A single entry can be removed with the `INVLPG` instruction. All non-persistent entries (those entries whose corresponding page table entries are not marked "global") can be flushed from the TLB by writing a new value to CR3. Since no address space or process ID tag is maintained in the TLB, all non-shared entries must be flushed on context switch.

3.2 SPARC Virtual Memory Architecture

In this section we review the key aspects of the SPARC MMU, especially how it differs from the x86. We chose the SPARC as our second implementation architecture because it provides a significantly different memory management interface to system software than the x86.

Instead of architecturally-defined, hardware-walked page tables as on the x86, SPARC uses a software managed TLB, *i.e.*, system software implements virtual address spaces by explicitly managing the contents of the hardware TLB. When a memory reference is made for which no TLB entry contains a translation, the processor raises an exception, which gives the operating system the opportunity to supply a valid translation or deliver an error to the offending process. The CPU is not aware of the operating system's page table organization.

In order to avoid flushing the entire TLB on process context switches, SPARC supplies a tag for each TLB entry, called a *context ID*, that associates the entry with a specific virtual address space. For each memory reference, the current context is supplied to the MMU along with the desired virtual address. In order to match, both the virtual page number and context in a TLB entry must be identical to the supplied values. This allows entries from distinct address spaces to exist in the TLB simultaneously.

An operating system can explicitly remove entries from the TLB at the granularity of a single page or at the granularity of an entire address space. These operations are called `page demap` and `context demap` respectively.

3.3 Virtual Machines

A VMM implements a hardware interface in software. The interface includes the privileged, or system, portions of the microprocessor architecture as well as peripherals like disk, network, and user interface devices. Note that the non-privileged, or user, portion of the microprocessor instruction set is not virtualized; when running unprivileged instructions, the guest directly executes on the processor with no additional overhead.

A key feature of a virtualized system environment is that guest operating systems execute using the unprivileged mode of the processor, while the VMM runs with full privilege. All guest OS accesses to sensitive system components, like the MMU or I/O peripherals, cause the processor to trap to the VMM. This allows the VMM to virtualize sensitive system features by mediating access to the feature or emulating it entirely. For example, because the MMU is virtualized, all attempts by a guest operating system to establish a virtual-to-physical memory mapping are trapped by the VMM; hence, the VMM can observe all such attempts. Similarly, each request to a virtual disk device is available for a VMM to examine. The VMM can choose to service a request made via a virtualized interface in any way it sees fit. For example, requests for virtual mappings can be altered or disk requests can be reordered.

4 Process Identification

The key to our process inference techniques is the logical correspondence between the abstraction *process*, which is not directly visible to a VMM, and the *virtual address space*, which is. This correspondence is due to the traditional single address space per process paradigm shared by all modern operating systems.

There are three major process events we seek to observe: creation, exit, and context switch. To the extent address spaces correspond to processes, these events are approximated by address space creation, destruction, and context switch. Hence, our techniques track processes by tracking address spaces.

Our approach to tracking address spaces on both x86 and SPARC is to identify a VMM-visible value with which we can associate a specific address space. We call this value an address space identifier (ASID). Tracking address space creation and context switch then becomes simply observing the use of a particular piece of VMM-visible operating system state, the ASID.

For example, when an ASID is observed that has not been seen before, we can infer that a new address space has been created. When one ASID is replaced by another ASID, we can conclude that an address space context switch has occurred. The technique we use to identify address space deallocation consists of detecting when an ASID is available for reuse. We assume that the address space, to which an ASID refers, has been deallocated if its associated ASID is available for reuse.

4.1 Techniques for x86

On the x86 architecture we use the physical address of the page directory as the ASID. A page directory serves as the root of the page table tree that describes each address space. The address of the page directory is therefore characteristic of a single address space.

4.1.1 Process Creation and Context Switch

To detect address space creation on x86 we observe how page directories are used. A page directory is in use when its physical address resides in CR3. The VMM is notified whenever a guest writes a new value to CR3 because it is a privileged register. If we observe an ASID value being used that has not been seen before, we can infer that a new address space has been created. When an ASID is seen for the first time, the VMM adds it to an ASID registry, akin to an operating system process list, for tracking purposes.

Writes to CR3 also imply address space context switch. By monitoring these events, the VMM always knows which ASID is currently “active”.

4.1.2 Process Exit

To detect address space deallocation, we use knowledge about the generic responsibilities of an operating system to maintain address space isolation. These requirements lead to distinctive OS behavior that can be observed and exploited by a VMM to infer when an address space has been destroyed.

Operating systems must strictly control the contents of page tables being used to implement virtual address spaces. Process isolation could be breached if a page directory or page table page were reused for distinct processes without first being cleared of their previous entries. To ensure this invariant holds, Windows and Linux systematically clear the non-privileged portions of page table pages used by a process prior to reusing them. Privileged portions of the page tables used to implement the protected kernel address space need not be cleared because they are shared between processes and map memory not accessible to untrusted software.

An OS must also ensure that no stale entries remain in any TLB once an address space has been deallocated. Since the x86 architecture does not provide a way for entries from multiple address spaces to coexist in a TLB, a TLB must be completely flushed prior to reusing address space structures like the page directory. On x86, the TLB is flushed by writing a value to CR3, an event the VMM can observe.

Hence, to detect user address space deallocation, a VMM can keep a count of the number of user virtual mappings present in the page tables describing an address space. When this count drops to zero, the VMM can infer that one requirement for address space reuse has been met. It is simple for a VMM to maintain such a counter because the VMM *must* be informed of all updates to a process's page tables in order for those updates to be effective. This requirement follows from the VMM's role in virtualizing the MMU. Multi-threading does not introduce additional complexity, because updates to a process's page tables must always be synchronized within the VMM for correctness.

By monitoring TLB flushes on all processors, a VMM can detect when the second requirement for address space deallocation has been met. Once both events have been observed for a particular ASID, the VMM can consider the corresponding address space dead and its entry in the ASID registry can be removed. A subsequent use of the same ASID implies the creation of a new and distinct process address space.

4.2 Techniques for SPARC

The key aspect that was used to enable process awareness on x86 is still present on SPARC. Namely, there is a VMM-visible identifier associated with each virtual address space. On x86 this was the physical address of the page directory. On SPARC we use the virtual address space context ID as an ASID. Making the obvious substitution leads to a process detection technique for SPARC similar to that for x86.

4.2.1 Creation and Context Switch

On SPARC, installing a new context ID is a privileged operation and so it is always visible to a VMM. By observing this operation, a VMM can maintain a registry of known ASIDs. When a new ASID is observed that is not in the ASID registry, the VMM infers the creation of a new address space. Context switch is detected on SPARC when a new context ID is installed on a processor.

4.2.2 Exit

The only requirement for the reuse of a context ID on SPARC is that all stale entries from the previously associated address space be removed from each processor's TLBs. SPARC provides the context demap operation for this purpose. Instead of monitoring page table contents, as on x86, a VMM can observe context demap operations. If all entries for a context ID have been flushed from every processor it implies that the associated address space is no longer valid.

	x86	**SPARC**
ASID	Page directory PA	Context ID
Creation	New ASID	New ASID
Exit	No user mappings and TLB flushed	Context demap
Context switch	CR3 change	Context ID change

Table 1: **Process identification techniques.** *The table lists the techniques used by Antfarm to detect each process event on the x86 and SPARC architectures.*

5 Implementation

Antfarm has been implemented for two virtualization environments. The first, Xen [6], is a true VMM. The other is a low-level system simulator called Simics [17] which we use to explore process awareness for operating systems and architectures not supported by Xen.

5.1 Antfarm for Xen

Xen is an open source virtual machine monitor for the Intel x86 architecture. Xen provides a paravirtualized [25] processor interface, which enables lower overhead virtualization at the expense of porting system software. We explicitly do *not* make use of this feature of Xen; hence, the mechanisms we describe are equally applicable to a more conventional virtual machine monitor such as VMWare [22, 24]. Because operating systems must be ported to run on Xen, proprietary commercial operating systems like Microsoft Windows are not currently supported.

Antfarm for Xen is implemented as a set of patches to the Xen hypervisor. Changes are concentrated in the handlers for events like page faults, page table updates, and privileged register access. Additional hooks were added to Xen's back-end block device driver. The Antfarm patches to Xen, including debugging and measurement infrastructure, total approximately 1200 lines across eight files.

5.2 Antfarm for Simics

Simics [17] is a full system simulator capable of executing unmodified, commercial operating systems and applications for a variety of processor architectures. While Simics is not a virtual machine monitor in the strict sense of direct execution of user instructions [18], it can play the role of a VMM by allowing Antfarm to observe and interpose on operating system and application hardware requests in the same way a VMM does. Simics allows us to explore process awareness techniques for SPARC/Linux and x86/Windows which would not be possible with a Xen-only implementation.

Antfarm for Simics is implemented as a Simics extension module. Simics extension modules are shared libraries dynamically linked with the main Simics executable. Extension modules can read or write OS and application memory and registers in the same way as a VMM.

Simics provides hooks called "haps" for various hardware events for which extension modules can register callback functions. Antfarm for Simics/x86 uses a hap to detect writes to CR3 and Antfarm for Simics/SPARC uses a hap to detect when the processor context ID is changed. Invocation of a callback is akin to the exception raised when a guest OS accesses privileged processor registers on a true VMM. A memory write breakpoint is installed by Antfarm for Simics/x86 on all pages used as page tables so that page table updates can be detected. A VMM like Xen marks page tables read-only to detect the same event.

Antfarm for Simics/x86 consists of about 800 lines of C code. For Simics/SPARC the total is approximately 450 lines.

6 Process Awareness Evaluation

In this section we explore the accuracy of Antfarm in each of our implementation environments. We also characterize the runtime overhead of Antfarm for Xen.

The analysis of accuracy can be decomposed into two components. The first is the ability to correctly detect process creations, exits, and context switches. We call this aspect *completeness*. The second component is the time difference or *lag* between process events as they occur within the operating system and when they are detected by the VMM.

6.1 x86 Evaluation

Our evaluation on x86 uses Xen version 2.0.6. Version 2.6.11 of the Linux kernel was used in Xen's privileged control VM. Linux kernel version 2.4.30 and 2.6.11 are used in unprivileged VMs as noted. Our evaluation hardware consists of a 2.4 GHz Pentium IV PC with 512 MB of RAM. Virtual machines are each allocated 128 MB of RAM in this environment.

We also evaluate our techniques as applied to Microsoft Windows NT4. Since Windows does not currently run on Xen, Simics/x86 is used for this purpose. Our Simics/x86 virtual machines were configured with a 2.4 GHz Pentium IV and 256 MB of RAM.

6.1.1 Completeness

To quantify completeness, each guest operating system was instrumented to report process creation, exit, and context switch. Event records include the appropriate ASID, as well as the time of the event obtained from the processor's cycle counter. These OS traces were compared to similar traces generated by Antfarm. Guest OS traces are functionally equivalent to the information that would be provided by a paravirtualized OS that included a process event interface. Hence, our evaluation implicitly compares the accuracy of Antfarm to the ideal represented by a paravirtual interface.

In addition to process creation, exit, and context switch, guests report address space creation and destruction events so that we can discriminate between errors caused by a mismatch between processes and address spaces and errors induced by inaccurate address space inferences made by Antfarm.

We categorize incorrect inferences as either false negatives or false positives. A false negative occurs when a true process event is missed by Antfarm. A false positive occurs when Antfarm incorrectly infers events that do not exist.

To determine if false negatives occurred, one-to-one matches were found for every OS-reported event in each pair of traces. We required that the matching event have the same ASID, and that it occur within the range for which the event was plausible. For example, to match an OS process-creation event, the corresponding inferred event must occur after any previous OS-reported process exit events with the same ASID and before any subsequent OS-reported process creation events with the same ASID.

Table 2 reports the process and address space event counts gathered by our guest OSes and by Antfarm during an experiment utilizing two process intensive workloads. The first workload is synthetic. It creates 1000 processes, each of which runs for 10 seconds then exits. The process creation rate is 10 processes/second. On Linux, this synthetic workload has three variants. The first creates processes using fork only; the second uses fork followed by exec; the third employs vfork followed by exec. Under Windows, processes are created using the CreateProcess API.

The second workload is a parallel compile of the bash shell sources using the command "make -j 20" in a clean object directory. A compilation workload was chosen because it creates a large number of short-lived processes, stressing Antfarm's ability to track many concurrent processes that have varying runtimes.

Antfarm incurs no false negatives in any of the tested

	Process Create	Addr Spc Create	Inferred Create	Process Exit	Addr Spc Exit	Inferred Exit	Context Switch	CS Inferred
Linux 2.4 x86								
Fork Only	1000	1000	1000	1000	1000	1000	3331	3331
Fork + Exec	1000	1000	1000	1000	1000	1000	3332	3332
Vfork + Exec	1000	1000	1000	1000	1000	1000	3937	3937
Compile	815	815	815	815	815	815	4447	4447
Linux 2.6 x86								
Fork Only	1000	1000	1000	1000	1000	1000	3939	3939
Fork+Exec	1000	**2000**	**2000**	1000	**2000**	**2000**	4938	4938
Vfork + Exec	1000	1000	1000	1000	1000	1000	3957	3957
Compile	748	**1191**	**1191**	748	**1191**	**1191**	2550	2550
Windows								
Create	1000	1000	1000	1000	1000	1000	74431	74431
Compile	2602	2602	2602	2602	2602	2602	835248	835248

Table 2: **Completeness.** *The table shows the total number of creations and exits for processes and address spaces reported by the operating system. The total number of process creations and exits inferred by Antfarm are shown in comparison. Antfarm detects all process creates and exits without false positives or false negatives on both Linux 2.4 and Windows. Fork and exec, however, lead to false positives under Linux 2.6 (***bold face values***). All false positives are due to the mismatch between address spaces and processes indicated by matching counts for address space creates and inferred creates. Actual and inferred context switch counts are also shown for completeness and are accurate as expected.*

cases, *i.e.*, all process-related events reported by our instrumented OSes are detected by the VMM. The fact that inferred counts are always greater than or equal to the reported counts suggests this, but we also verified that each OS-reported event is properly matched by at least one VMM-inferred event.

Under Linux 2.4 and Windows, no false positives occur, indicating Antfarm can precisely detect address space events and that there is a one-to-one match between address spaces and processes for these operating systems. Under Linux 2.6, however, false positives do occur, indicated in Table 2 by the inferred event counts that are larger than the OS-reported counts. This discrepancy is due to the implementation of the Linux 2.6 fork and exec system calls.

UNIX programs create new user processes by invoking the fork system call which, among other things, constructs a new address space for the child process. The child's address space is a copy of the parent's address space. In most cases, the newly created child process immediately invokes the exec system call which replaces the child's virtual memory image with that of another program read from disk.

In Linux 2.4, when exec is invoked the existing process address space is cleared and reused for the newly loaded program. In contrast, Linux 2.6 destroys and releases the address space of a process invoking exec. A new address space is allocated for the newly exec'd program. Hence, under Linux 2.6, a process that invokes exec has two distinct address spaces associated with it, which do not overlap in time. In other words, the runtime of the process is *partitioned* into two segments. One segment corresponds to the period between fork and exec and the other corresponds to the period between exec and process exit. Antfarm, because it is based on address space tracking, concludes that two different processes are created leading to twice as many inferred process creations and exits as actually occurred.

Due to the idiomatic use of fork and exec, however, a process is partitioned in a distinctive way. The Linux 2.6/x86 case in Figure 1 depicts the temporal relationship between the two inferred pseudo-processes. The duration of the first pseudo-process will nearly always be small. For example, in the case of our compilation workload, the average time between fork and exec is less than 1 ms, compared to the average lifetime of the second pseudo-process, which is more than 2 seconds, a difference of three orders of magnitude.

The two pseudo-processes are separated by a short time period where neither is active. This interval corresponds to the time after the original address space is destroyed and before the new address space is created. During the compilation workload this interval averaged less than 0.1 ms and was never larger than 2.3 ms. Since no user instructions can be executed in the absence of a user address space, the combination of the two pseudo-processes detected by Antfarm encompasses all user activity of the true process. Conventional use of fork and exec imply

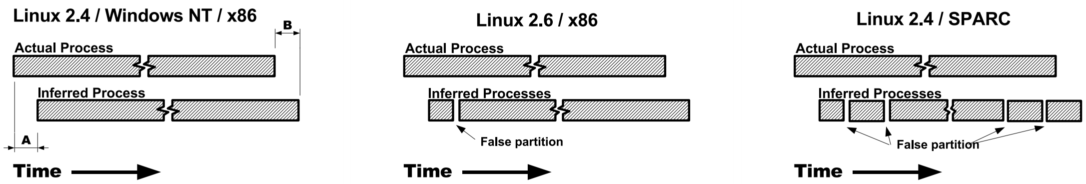

Figure 1: **Effects of error.** *The figure shows where each type of process identification error occurs for each tested platform. Error is either lag between when the true event occurs and when the VMM detects it, (e.g., A and B in the figure) or consists of falsely partitioning a single OS process into multiple inferred processes. In Linux 2.6/x86, this only occurs on* `exec`*, which typically happens immediately after fork. On SPARC this partitioning happens whenever a process calls either* `fork` *or* `exec`*.*

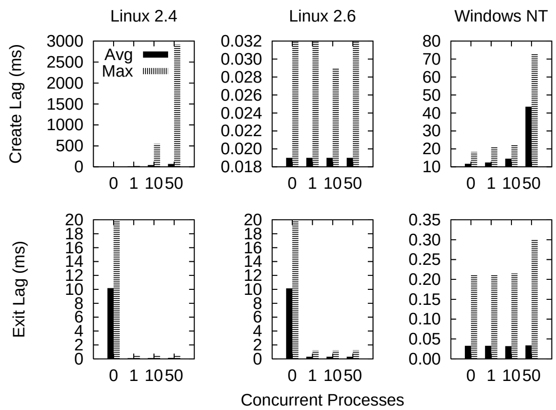

Figure 2: **Lag vs. System Load.** *The figure shows average and maximum create and exit lag time measurements for a variety of system load levels in each of our x86 evaluation environments. Average and worst case create lag are affected by system load in Linux 2.4 and Windows, but are small and nearly constant under Linux 2.6. Except for a large exit lag with no competing processes on Linux, exit lag does not appear to be sensitive to system load.*

that nearly all substantive activity of the true user process is captured within the second pseudo-process.

6.1.2 Lag

The second aspect of process identification accuracy that we consider is the time difference between a process event and when the same event is detected by the VMM. We define a process to exist at the instant the fork (or its equivalent) system call is invoked. Exit is defined as the start of the exit system call. These definitions are maximally conservative. In Figure 1 create lag is labeled `A` and exit lag is labeled `B`.

Lag is similar in nature to response time, so we expect it to be sensitive to system load. To evaluate this sensitivity, we conduct an experiment that measures lag times for various levels of system load on Linux 2.4, Linux 2.6, and Windows. In each experiment, 0, 1, 10, or 50 CPU-bound processes were created. 100 additional test processes were then created and the create and exit lag time of each were computed. Test process creations were separated by 10 ms and each test process slept for one second before exiting.

The results of these experiments are presented in Figure 2. For each graph, the x-axis shows the number of concurrent CPU-bound processes and the y-axis shows lag time. Create lag is sensitive to system load on both Linux 2.4 and Windows, as indicated by the steadily increasing lag time for increasing system load. This result is intuitive since a call to the scheduler is likely to occur between the invocation of the create process API in the parent (when a process begins) and when the child process actually runs (when the VMM detects it). Linux 2.6, however, exhibits a different process creation policy that leads to relatively small and constant creation lag. Since Antfarm detects a process creation when a process first runs, the VMM will always be informed of a process's existence before any user instructions are executed.

Exit lag is typically small for each of the platforms. The exception is for an otherwise idle Linux which shows a relatively large exit lag average of 10 ms. The reason for this anomaly is that most Linux kernel tasks, including the idle task, do not need an associated user address space and therefore borrow the previously active user address space when they need to run. This mechanism allows a kernel task to run without incurring the expense of a TLB flush. In the case of this experiment, test processes were started at intervals of 10 ms and each process sleeps for one second; hence, when no other processes are ready to run, approximately 10 ms elapse between process exit and when another process begins. During this interval, the Linux idle task is active and prevents the previous address space from being released, which leads to the observed delay.

6.1.3 The Big Picture

Figure 3 shows a set of timelines depicting how Antfarm tracks process activity over time for a parallel compilation

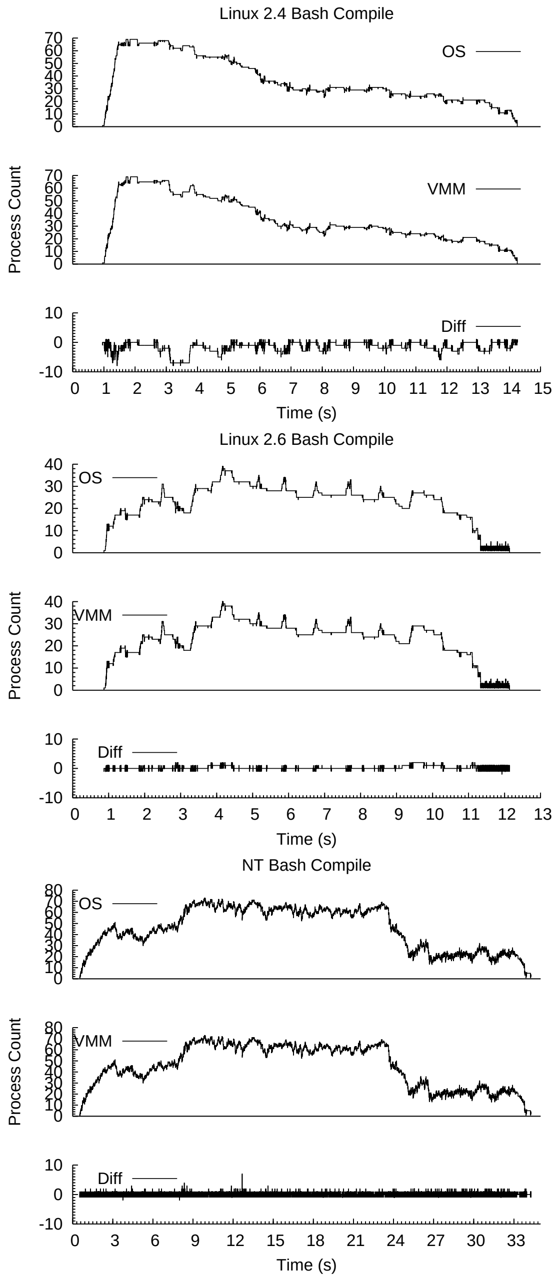

Figure 3: **Compilation Workload Timelines.** *For x86/Linux 2.4, x86/Linux 2.6 and x86/Windows a process count timeline is shown. Each timeline depicts the OS-reported process count, the VMM-inferred process count and the difference between the two versus time. Lag has a larger impact on accuracy than false positives. x86/Linux 2.6, which exhibits significantly smaller lag than x86/Linux 2.4 is able to track process counts more accurately.*

workload on each of our x86 platforms. The top curve in each graph shows the true, current process count over time as reported by the operating system. The middle curve shows the current process count as inferred by Antfarm. The bottom curve shows the difference between the two curves calculated as $Inferred - Actual$.

The result of the relatively large creation lag under Linux 2.4 is apparent in the larger negative process count differences compared to Linux 2.6. For this workload and metric combination, creation lag is of greater concern than the false positives experienced by Linux 2.6. In another environment such as a more lightly loaded system, which would tend to reduce lag, or for a metric like total cumulative process count, the false positives incurred by Linux 2.6 could be more problematic.

Exit lag is not prominent in any of the graphs. Large, persistent exit lag effects would show up as significant positive deviations in the difference curves. The fact that errors due to fork and exec do not accumulate over time under Linux 2.6 is also apparent because no increasing inaccuracy trend is present.

6.2 Overhead

To evaluate the overhead of our process awareness techniques we measure and compare the runtime of two workloads under Antfarm and under a pristine build of Xen. The first workload is a microbenchmark that represents a worst case performance scenario for Antfarm. Experiments were performed using Linux 2.4 guests.

Since our VMM extensions only affect code paths where page tables are updated, our first microbenchmark focuses execution on those paths. The program allocates 100 MB of memory, touches each page once to ensure a page table entry for every allocated page is created and then exits, causing all of the page tables to be cleared and released. This program is run 100 times and the total elapsed time is computed. The experiment was repeated five times and the average duration is reported. There was negligible variance between experiments. Under an unmodified version of Xen this experiment required an average of 24.75 seconds to complete. Under Antfarm for Xen the experiment took an average of 25.35 seconds to complete. The average slowdown is 2.4% for this worst case example.

The runtime for configuring and building bash was also compared between our modified and unmodified versions of Xen. In the unmodified case the average measured runtime of five trials was 44.49 s. The average runtime of the same experiment under our modified Xen was 44.74 s. The variance between experiments was negligible yielding a slowdown of about 0.6% for this process-intensive application workload.

	Process Create	Addr Spc Create	Inferred Create	Process Exit	Addr Spc Exit	Inferred Exit	Context Switch	CS Inferred
SPARC/Linux								
Fork Only	1000	1000	**2000**	1000	1000	**2000**	3419	3419
Fork & Exec	1000	1000	**3000**	1000	1000	**3000**	3426	3426
Vfork	1000	1000	1000	1000	1000	1000	4133	4133
Compile	603	603	**1396**	603	603	**1396**	1678	1678

Table 3: **Completeness for SPARC.** *The table shows the results for the same experiments reported for x86 in Table 2, but for SPARC/Linux 2.4. False positives occur for each fork due to an implementation which uses copy-on-write. Antfarm also infers an additional, non-existent exit/create event pair for each exec. This error is not due to multiple address spaces per process as on x86, but rather stems from the flush that occurs to clear the caller's address space upon exec.*

6.3 SPARC Evaluation

Our implementation of process tracking on SPARC uses Simics. Each virtual machine is configured with a 168 MHz UltraSPARC II processor and 256 MB of RAM. We use SPARC/Linux version 2.4.14 as the guest operating system for all tests. The guest operating system is instrumented to report the same information as described for x86.

6.3.1 Completeness

We use the same criteria to evaluate process awareness under SPARC as under x86. Table 3 lists the total event counts for our process creation micro-benchmark and for the bash compilation workload.

As on x86, no false negatives occur. In contrast to x86, the fork-only variant of the microbenchmark incurs false positives. The reason for this is the copy-on-write implementation of fork under Linux. During fork all of the writable portions of the parent's address space are marked read-only so that they can be copy-on-write shared with the child. Many entries in the parent's page tables are updated and all of the corresponding TLB entries must be flushed. SPARC/Linux accomplishes this efficiently by flushing all of the parent's current TLB entries using a context demap operation. The context demap is incorrectly interpreted by Antfarm as a process exit. As soon as the parent is scheduled to run again, we detect the use of the address space and signal a matching spurious process creation.

The false positives caused by the use of fork under SPARC are different in character than those caused by exec under x86. These errors are not limited (by convention) to the usually tiny time interval between fork and exec. They will appear whenever fork is invoked, which for processes like a user shell can occur repeatedly throughout the process's lifetime. The Linux 2.4/SPARC case in Figure 1 depicts how a process that repeatedly invokes fork might be partitioned into many inferred pseudo-processes by Antfarm.

When exec is used we see additional false positives, but for a different reason than under x86/Linux 2.6. In this case the process inference technique falsely reports the creation of new address spaces that don't really exist. The cause of this behavior is a TLB demap operation that occurs when a process address space is cleared on exec. This error mode is different than under x86 where observed errors were due to a faulty assumption of a single address space per process. On SPARC, the error occurs because our chosen indicator, context demap, can happen without the corresponding address space being deallocated.

Given these two sources of false positives, one would expect our compilation workload to experience approximately the same multiple of false positives as seen for the fork+exec synthetic benchmark. We see, however, fewer false positives than we expect, due to the use of vfork by both GNU make and gcc. Vfork creates a new process but does not duplicate the parent's address space. Since no parent page tables are changed, no flush is required. When exec is invoked we detect the creation of the *single* new address space. Hence, when vfork and exec are used to create new processes under SPARC/Linux, Antfarm experiences no false positives. The build process, however, consists of more than processes created by make and gcc. Many processes are created by calls to an external shell and these process creations induce the false positives we observe.

6.3.2 Lag

Lag between OS-recorded and VMM-inferred process events under SPARC/Linux is comparable to Linux on x86. The average and maximum lag values for SPARC/Linux under various system loads are shown in Figure 4. Create lag is sensitive to system load. Exit lag is unaffected by load as on x86.

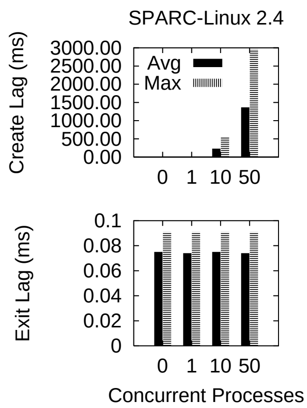

Figure 4: **Lag vs. System Load, SPARC.** *The figure shows average and maximum create and exit lag time measurements for the same experiments described in Figure 2. Create lag grows with system load. Exit lag is small and nearly constant, independent of load.*

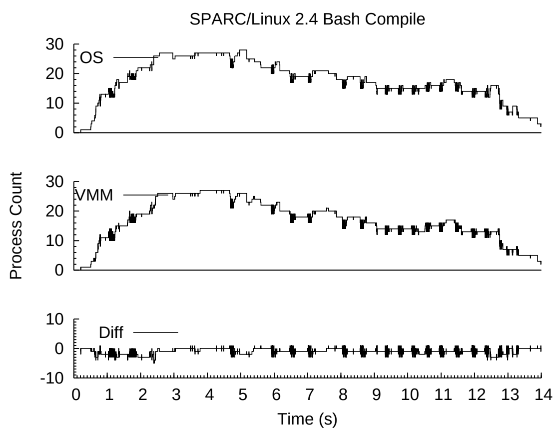

Figure 5: **Compilation Workload Timeline.** *Compilation timeline comparable to Figure 3 for SPARC/Linux.*

6.3.3 Limitations

While the SPARC inference technique is simple, it suffers drawbacks relative to x86. As shown, the technique incurs more false positives than the x86 techniques. In spite of the additional false positives, Figure 5 shows that the technique can track process events during a parallel compilation workload at least as accurately as x86/Linux 2.4.

Unlike the x86, where one can reasonably assume that a page directory page would not be shared by multiple runnable processes, one cannot make such an assumption for context IDs on SPARC. The reason is the vastly smaller space of unique context IDs. The SPARC provides only 13 bits for this field which allows up to 8192 distinct contexts to be represented concurrently. If a system exceeds this number of active processes, context IDs must necessarily be recycled. In some cases, system software will further limit the number of concurrent contexts it supports. For example, Linux on SPARC architectures uses only 10 of the available 13 context bits, so only 1024 concurrent address spaces are supported without recycling.

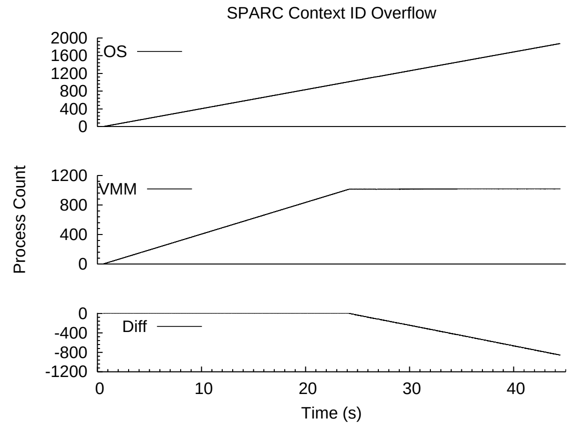

Figure 6: **Context ID Overflow.** *When more processes exist than can be represented by the available SPARC context IDs our techniques fail to detect context ID reuse.*

Figure 6 shows the behavior of our SPARC process detection techniques when more processes exist than can be distinguished by the available context IDs. Once the limit is reached at 1024, the technique fails to detect additional process creations.

The importance of this second limitation is somewhat reduced because even very busy servers rarely have more than 1000 active processes, a fact which no doubt influenced the selection of the context ID field's size.

6.4 Discussion

The process event detection techniques used by Antfarm are based on the mechanisms provided by a CPU architecture to implement and manage virtual address spaces, and on the responsibilities of general-purpose operating systems to maintain process isolation. The techniques assume an OS will follow the address space conventions suggested by the MMU features available in an architecture. If an OS deviates from the convention, detection accuracy will likely differ from what we have reported here. Our evaluation shows that two widely used operating systems adhere to our assumptions. Antfarm precisely identifies the desired process events on x86/Windows and x86/Linux 2.4. Some false positives occur under x86/Linux 2.6 and SPARC/Linux. However, the false positives are stylized and affect the ability of Antfarm to keep an accurate process count very little.

New architectures devoted to hardware-assisted virtualization [1, 13] will, in some configurations, reduce or eliminate the need for a VMM to track guest page table updates and context switches. For example, AMD plans to include two levels of address translation and a private guest-CR3 as options in its Secure Virtual Machine (SVM) architecture. This fact does not prevent a VMM from observing its guest operating systems; shadow page tables are explicitly supported by these architectures. It will, however, likely increase the performance penalty exacted by the techniques used in Antfarm.

7 Case Study: Anticipatory Scheduling

The order in which disk requests are serviced can make a large difference to disk I/O performance. If requests to adjacent locations on disk are serviced consecutively, the time spent moving the disk head unproductively is minimized. This is the primary performance goal of most disk scheduling algorithms. This case study explores the application of one innovative scheduling algorithm called *anticipatory scheduling* [14] in a virtual machine environment. The implementation makes use of Antfarm for Xen.

7.1 Background

Iyer *et al.*[14] have demonstrated a phenomenon they call *deceptive idleness* for disk access patterns generated by competing processes performing synchronous, sequential reads. Deceptive idleness leads to excessive seeking between locations on disk. Their solution, called anticipatory scheduling, introduces a small amount of waiting time between the completion of one request and the initiation of the next if the process whose disk request just completed is likely to issue another request for a nearby location. This strategy leads to substantial seek savings and throughput gains for concurrent disk access streams that each exhibit spatial locality.

Anticipatory scheduling makes use of process-specific information. It decides whether to wait for a process to issue a new read request and how long to wait based on statistics the disk scheduler keeps for all processes about their recent disk accesses. For example, the average distance from one request to the next is stored as an estimate of how far away the process's next access will be. If this distance is large, there is little sense waiting for the process to issue a request nearby. Statistics about how long a process waits after one request completes before it issues another are also kept in order to determine how long it make sense to wait for the next request to be issued.

Anticipatory scheduling does not work well in a virtual machine environment. System-wide information about disk requests is required to estimate where the disk head is located, which is essential in deciding if a request is nearby. Information about individual process's I/O behavior is required to determine whether and how long to wait. This information is not completely available to either a single guest, which only knows about its own requests, or to the VMM, which cannot distinguish between guest-level processes. While guests and the VMM could cooperate to implement anticipatory scheduling, this requires the introduction of additional, specialized VMM-to-guest interfaces. New interfaces may not be possible in the case of legacy or binary-only components. In any case, such interfaces do not exist today.

7.2 Information

To implement anticipatory scheduling effectively in a VMM, the VMM must be able to distinguish between guest processes. Additionally, it must be able to associate disk read requests with specific guest processes. Given those two pieces of information, a VMM implementation of anticipatory scheduling can maintain average seek distance and inter-request waiting time for processes across all guests. We use Antfarm to inform an implementation of anticipatory scheduling inside of Xen.

To associate disk read requests to processes, we employ a simple *context association* strategy that associates a read request with whatever process is currently active. This simple strategy does not take potential asynchrony within the operating system into account. For example, due to request queuing inside the OS, a read may be issued to the VMM after the process in which it originated has blocked and context switched off the processor. This leads to association error. We have researched more accurate ways of associating reads to their true originating process by tracking the movement of data from the disk through memory towards the requesting process. These methods have proven effective in overcoming association error due to queuing. Because of limited space, however, we do not present these techniques here. The implementation of anticipatory scheduling described in this paper uses simple context association.

7.3 Implementation

Xen implements I/O using device driver virtual machines (DDVM) [7]. A DDVM is a virtual machine that is allowed unrestricted access to one or more physical devices. DDVMs are logically part of the Xen VMM. Operationally, guests running in normal virtual machines make disk requests to a DDVM via an idealized disk device interface and the DDVM carries out the I/O on their behalf. In current versions of Xen, these driver VMs run Linux to take advantage of the broad device support it offers. A device back-end in the driver VM services requests submitted by an instance of a front-end driver located in all

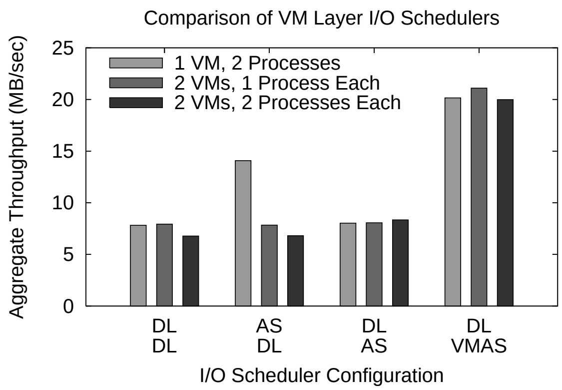

Figure 7: **Benefit of process awareness for anticipatory scheduling.** *The graph shows the aggregate throughput for various configurations of I/O scheduler, number of virtual machines and number of processes per virtual machine. The experiment uses the Linux deadline scheduler (DL), the standard anticipatory scheduler (AS), and our VMM-level anticipatory scheduler (VMAS). Adding process awareness enables VMAS to achieve single process sequential read performance in aggregate among competing sequential streams. AS running at the guest layer is somewhat effective in the 1 VM / 2 process case since it has global disk request information.*

normal VMs.

The standard Linux kernel includes an implementation of anticipatory scheduling. We implement anticipatory scheduling at the VMM layer by enabling the Linux anticipatory scheduler within a Xen DDVM that manages a disk drive. To make this existing implementation process-aware, we introduce a foreign process abstraction that represents processes running in other VMs. When a disk request arrives from a foreign virtual machine, the Xen back-end queries our process-aware Xen hypervisor about which process is currently active in the foreign virtual machine. Given the ability to distinguish between processes we expect that our VMM-level anticipatory scheduler (VMAS) will improve synchronous read performance for competing processes whether they exist in the same or different VMs.

7.4 Evaluation

To demonstrate the effectiveness of our implementation of VMAS, we repeat one of the experiments from the original anticipatory scheduling paper in a virtual machine environment. Our experiment consists of running multiple instances of a program that sequentially reads a 200 MB segment of a private 1 GB file. We vary the number of processes, the assignment of processes to virtual machines, and the disk scheduler used by guests and by the VMM to explore how process awareness influences the effectiveness of anticipatory scheduling in a VMM. We make use of the Linux deadline I/O scheduler as our non-anticipatory baseline. Results for each of four scheduler configurations combined with three workloads are shown in Figure 7. The workloads are: (1) one virtual machine with two processes, (2) two virtual machines with one process each, and (3) two virtual machines with two processes each.

The first experiment shows the results from a configuration without anticipatory scheduling. It demonstrates the expected performance when anticipation is not in use for each of the three workloads. On our test system this results in an aggregate throughput of about 8 MB/sec.

The second configuration enables anticipatory scheduling in the guest while the deadline scheduler is used by Xen. In the one virtual machine/two process case, where the guest has complete information about all processes actively reading the disk, we expect that an anticipatory scheduler at the guest level will be effective. The figure shows that this is in fact the case. Anticipatory scheduling is able to improve aggregate throughput by 75% from about 8 MB/sec to about 14 MB/sec. In the other cases guest-level anticipatory scheduling performs about as well as the deadline scheduler due to its lack of information about processes in other virtual machines.

Our third experiment demonstrates the performance of unmodified anticipatory scheduling at the VMM layer. Similar to the case of anticipatory scheduling running at the guest layer we would expect performance improvement for the two-virtual-machine/one-process-each case to be good because a VMM can distinguish between virtual machines just as an operating system can distinguish between processes. The improvement does not occur, however, because of an implementation detail of the Xen DDVM back-end driver. The back-end services all foreign requests in the context of a single dedicated task so the anticipatory scheduler interprets the presented I/O stream as a single process making alternating requests to different parts of the disk. The performance is comparable to the configuration without anticipation for all workloads.

The final configuration shows the benefit of process awareness to anticipatory scheduling implemented at the VMM layer. In each of the workload configurations anticipatory scheduling works well, improving aggregate throughput by more than a factor of two, from about 8 MB/sec to about 20 MB/sec. Because it is implemented at the VMM layer, anticipatory scheduling in this configuration has complete information about all requests reaching the disk. Our process awareness extensions allow it to track statistics for each individual process enabling it to make effective anticipation decisions.

8 Conclusion

The widespread adoption of virtual machines brings with it interesting research opportunities to reevaluate where and how certain operating system services are implemented. Implementing OS-like services in a VMM is made more challenging by the lack of high-level OS and application information.

The techniques developed in this paper and their implementation in Antfarm are an explicit example of how information about one important operating system abstraction, the process, can be accurately and efficiently inferred inside a VMM by observing the interaction of a guest OS with its virtual hardware. This method is a useful alternative to explicitly exporting the required information to the VMM directly. By enabling a VMM to independently infer the information it needs, the VMM is decoupled from the specific vendor, version, and even correctness of the guests it supports.

Acknowledgments

This work is sponsored by the Sandia National Laboratories Doctoral Studies Program, by NSF CCR-0133456, ITR-0325267, CNS-0509474, and by generous donations from Network Appliance and EMC.

References

[1] AMD. *AMD64 Programmer's Manual, Volume 2: System Programming*. December 2005.

[2] S. Ballmer. Keynote address. Microsoft Management Summit, April 2005.

[3] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 1–11, Copper Mountain Resort, Colorado, December 1995.

[4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[5] P. M. Chen and B. D. Noble. When virtual is better than real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133. IEEE Computer Society, 2001.

[6] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[7] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *OASIS ASPLOS 2004 Workshop*, 2004.

[8] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the USENIX Security Symposium*, pages 103–118, San Diego, CA, USA, August 2004.

[9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[11] R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[12] P. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.

[13] Intel. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*. April 2005.

[14] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 117–130, Banff, Canada, October 2001.

[15] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 91–104, Brighton, United Kingdom, October 2005.

[16] S. T. King and P. M. Chen. Backtracking Intrusions. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[18] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[19] R. Sekar, T. F. Bowen, and M. E. Segal. On preventing intrusions by process behavior monitoring. In *Proc. Workshop on Intrusion Detection and Network Monitoring*, pages 29–40, Berkeley, CA, USA, 1999. USENIX Association.

[20] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.

[21] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, San Diego, California, June 2000.

[22] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

[23] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM '04)*, pages 43–56, San Jose, California, May 2004.

[24] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[25] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.