FoxyTechnique: Tricking Operating System Policies with a Virtual Machine Monitor

Hiroshi Yamada Kenji Kono

Department of Information and Computer Science,
Keio University
3-14-1 Hiyoshi Kohoku-ku Yokohama, Japan
yamada@sslab.ics.keio.ac.jp, kono@ics.keio.ac.jp

Abstract

Integrating new resource management policies into operating systems (OSes) is an ongoing process. Despite innovative policy proposals being developed, it is quite difficult to deploy a new one widely because it is difficult, costly and often impractical endeavor to modify existing OSes to integrate a new policy. To address this problem, we explore the possibility of using virtual machine technology to incorporate a new policy into an existing OS without the need to make any changes to it. This paper describes *FoxyTechnique*, which virtualizes physical devices *differently* from real ones and tricks a guest OS into producing a behavior similar to a desired policy. FoxyTechnique offers several advantages. First, it allows us to implement a new policy without the need to make any changes to OS kernels. Second, Foxy-based policies are expected to be portable across different operating systems because they are isolated from guest OSes by stable virtual hardware interfaces. Finally, Foxy-based policies sometimes outperform guest OS policies because they can measure performance indicators more accurately than guest OSes. To demonstrate the usefulness of FoxyTechnique, we conducted two case studies, FoxyVegas and FoxyIdle, on the Xen virtual machine monitor. FoxyVegas and FoxyIdle tricked the original Linux and successfully mimicked TCP Vegas and Idletime scheduling, respectively.

Categories and Subject Descriptors D.4.1 [*Process Management*]: Scheduling; K.6.2 [*Installation Management*]: Pricing and resource allocation

General Terms Management, Design

Keywords Resource Management, Virtual Machine, Interference

1. Introduction

Integrating new resource management policies into operating systems (OSes) is an ongoing process, even though resource management has been extensively explored for the past decades. Because an appropriate resource management policy depends largely on the type of applications and their computing environments, OS researchers must continue to develop innovative resource management policies to satisfy the needs of emerging applications and everchanging computing environments.

Despite the numerous sophisticated, innovative policy proposals being presented, it is quite difficult to widely deploy a single innovation. The traditional approach to integrating an innovation is to modify an OS kernel, which is the primary layer of software for resource management. However, modifying the OS kernel is a difficult, costly, and often impractical endeavor because modern OSes consist of large and complex bodies of code. Changes to even a single line of OS code can make the deployment of an innovative policy much less likely because it is almost impossible to modify proprietary and/or closed-source OSes. Even if supported by a single vendor, the new policy is unlikely to become widely used since cross-platform applications would not be able to use the features only available in specific OSes.

To address this modification issue, many researchers have investigated how the operating system should be *restructured* to reduce the efforts required to change the OS. Microkernels [1, 14, 19], extensible OSes [5, 12, 21], and infokernels [3] have all been used in an attempt to provide a base on which resource management policies can easily be built. Unfortunately, to benefit from these approaches, users are forced to replace their favorite OSes with something less familiar. To eliminate the need to replace OS, many techniques have been developed that enable us to build resource management policies on "as is" operating systems. Newhouse *et al*. [18] have shown that a user-level CPU scheduler for CPU intensive jobs can be implemented on unmodified FreeBSD. Graybox techniques [2] facilitate the development of OS-like services at the user-level by inferring an OS internal state.

In this paper, we explore the possibility of using virtual machine technology [13] to incorporate an innovative policy into an existing OS without the need to make any changes to it. Inserting a new policy into a virtual machine monitor (VMM) offers many potential benefits. First, we can incorporate innovative policies without the need to make any changes to guest OSes because the guest OSes are isolated from the VMM by stable virtual hardware interfaces. Second, the fact that the VMM is isolated from guest OSes results in another noteworthy feature that new policies within a VMM can be portable across many guest OSes. These advantages are widely recognized, and some researchers are developing mechanisms [15, 16] that allow us to implement innovative policies within a VMM.

FoxyTechnique, presented in this paper, enables us to incorporate new resource management policies into a guest OS without the need to make any changes to it. FoxyTechnique virtualizes physical devices differently from real ones so that a guest OS policy is *tricked* into producing a similar policy to the one we want to incorporate. For example, a Foxy-enabled VMM can change the rate of timer interrupts to alter the guest OS policy of CPU scheduling. By

VEE'07, June 13–15, 2007, San Diego, California, USA.

changing the rate of timer interrupts, we can control the length of time slices allocated to each process. To trick guest OS policies, we extensively use our *graybox* knowledge about guest OSes; graybox knowledge means we can predict how a guest OS reacts to virtual devices whose behavior has been transformed. In the above example, we used graybox knowledge that time slices are measured by counting timer interrupts in the guest OS. FoxyTechnique has the following advantages:

- **No changes to OS kernels required**
  FoxyTechnique enables us to incorporate new resource management policies into an existing OS without the need to make any changes to it. Foxy-based resource management policies are completely implemented within a VMM and thus are isolated from guest OSes by stable virtual hardware interfaces. Hence, even a proprietary OS can benefit from innovative resource management policies; we do not have to wait until an OS vendor officially supports the innovations.
- **Increased portability**
  Foxy-based resource management policies are expected to be portable across different guest OSes. Although a *Foxy-based module*, which controls virtual hardware devices within the VMM, uses graybox knowledge about guest OSes, many resource management policies can be built without using the knowledge specific to a single guest OS. In this paper, we demonstrate that a single Foxy-based module can trick many guest OS policies. For example, our Foxy-based module can trick binary increase congestion (BIC) control, NewReno, CUBIC TCP, and TCP-Hybla into TCP Vegas. However, FoxyTechnique does not work well in some situations. The limitation of FoxyTechnique is discussed later in this paper.
- **Better performance estimates**
  Resource management policies are often required to measure performance indicators to estimate the cost of following possible operations. The accuracy of these indicators affects how efficiently resources are managed. For example, TCP Vegas measures round trip times (RTTs) of network packets to detect congestion in the network. If the RTT contains large errors, TCP Vegas does not work well. Since a Foxy-based module runs directly on physical hardware, it can measure performance indicators more accurately than a guest OS running in a virtualized environment. Therefore, Foxy-based modules may result in better overall performance than guest OS policies. In fact, our Foxy-based module for TCP Vegas outperforms TCP Vegas running in the guest Linux.

We conducted two case studies to embody the concept of FoxyTechnique, *FoxyVegas* and *FoxyIdle*, on the Xen virtual machine monitor [4]. FoxyVegas is a Foxy-based module for TCP Vegas, a TCP/IP congestion control algorithm for determining the congestion window size. FoxyVegas is based on our graybox knowledge that TCP/IP adjusts its window size for flow and congestion control. When network congestion occurs, FoxyVegas creates the illusion that the receiver is overloaded to force a guest OS to reduce its window size. By doing this, FoxyVegas brings the benefits of TCP Vegas; many congestion control algorithms are tricked into mimicking TCP Vegas. Furthermore, FoxyVegas performs better than Linux TCP Vegas running on Xen. This is because FoxyVegas can measure the RTTs of network packets more accurately than TCP Vegas in guest OSes.

FoxyIdle is a Foxy-based module for regulating disk I/O contention and implements the idletime scheduling [11]. This scheduling prevents background processes from degrading the performance of foreground processes. The idletime scheduling stalls disk read requests from background processes if they interfere with requests from foreground processes. To stall the disk read requests, FoxyIdle pretends that a virtual disk device has been seeking data for a long time. By carefully controlling this seek time, FoxyIdle mimics the idletime scheduling. Experimental results reveals that FoxyIdle can trick various disk scheduling algorithms (Noop, Anticipatory, Deadline, and CFQ) and avoid performance degradation caused by background processes.

The rest of the paper is organized as follows. Section 2 describes FoxyTechnique, and Section 3 presents its implementation. Sections 4 and 5 report the design, implementation and experiments in case studies of FoxyVegas and FoxyIdle. Section 6 describes the work related to ours, and Section 7 concludes the paper.

2. FoxyTechnique

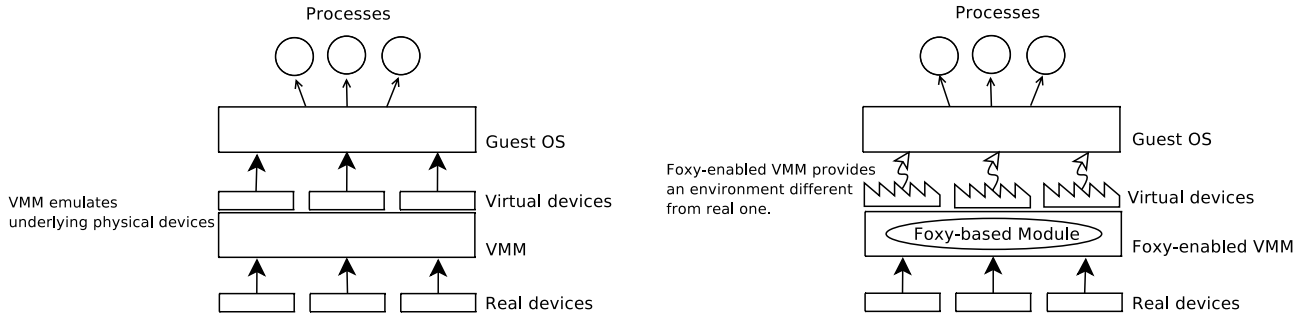
To incorporate new resource management policies into an existing kernel without the need to make any changes to it, FoxyTechnique exploits VMMs. Figure 1 outlines the difference between conventional virtualization and Foxy-enabled virtualization. The major difference lies in the way in which the Foxy-enabled VMM interacts with the guest OSes. In a conventional VMM, virtual hardware devices exposed to guest OSes *emulate* the underlying physical hardware. In contrast, the Foxy-enabled VMM virtualizes physical devices *differently* from the real ones; the behavior of physical devices is *transformed* by the Foxy-enabled VMM. A guest OS runs on the transformed hardware and thus behaves differently on the Foxy-enabled VMM from the conventional VMM. By taking guest policies into consideration, Foxy-enabled VMM can control virtual devices so that the guest policy is tricked into producing a behavior similar to an expected policy.

2.1 Foxy-enabled VMM

In a virtualized environment, a VMM runs in the privileged mode to manage and multiplex the underlying physical hardware devices, whereas guest OSes run in the non-privileged mode. When a guest OS executes a privileged instruction, such as access to MMU or I/O peripherals, software interrupts occur, and control is transfered to the VMM. At this point, the VMM can capture and regulate all resources because it processes the interrupts before delivering them to the guest OS.

By changing the behavior of virtual devices, Foxy-enabled VMM can control guest OSes. FoxyTechnique takes the guest policy into consideration to achieve the expected behavior of a guest OS. In other words, we make use of *graybox knowledge* [2] on the guest OS. Here, graybox knowledge means we can predict how the guest OS reacts to virtual devices whose behavior has been transformed. For example, we can change the interrupt rate of the virtual timer device to alter the policy for CPU scheduling. Because we have graybox knowledge that the CPU scheduler counts down ticks every timer interrupt, we can control the length of time slices. To provide a tricked environment with guest OSes, the following techniques are combined in FoxyTechnique.

- **Changing rate of interrupts**
  A Foxy-based module controls the rate of periodic interrupts, such as the timer. As explained earlier, if the Foxy-based module raises more timer interrupts than the real one, the guest OS clock advances more quickly. If the Foxy-based module raises fewer timer interrupts, the clock advances more slowly.
- **Delaying or discarding interrupts**
  Even if a physical device causes an interrupt, it is not sent immediately to the guest OS. A Foxy-based module delays the interrupt notification or completely revokes the interrupt. For example, we can throttle network bandwidth by delaying or revoking network interrupts. Since network packets do not arrive



(a) Conventional virtual machine monitor

(b) Foxy-enabled virtual machine monitor

Figure 1. Difference between conventional virtualization and Foxy virtualization. *These figures show virtualization of a conventional virtual machine monitor (VMM) and Foxy-enabled VMM. The conventional VMM emulates underlying physical hardware. In contrast, the Foxy-enabled VMM virtualizes physical hardware so that a guest OS is forced to change its own behavior differently from the real one.*

at the guest OS, it considers congestion in the network has occurred and reduces the congestion window size. As a result, the TCP window size is reduced and the effective bandwidth is throttled.

- **Rewriting content of device registers**
  A Foxy-based module rewrites the value of data registers. When a guest OS writes data in virtual device registers, a Foxy-based module may rewrite the value before sending it to the physical device. When the physical device assigns data into registers, the Foxy-based module rewrites the values of the registers, and then sends interrupts to the guest OS. Rewriting the content of data registers enables us to control the window size of packets in TCP/IP. By rewriting server receivable data size included in the packets, we can make the guest OS believe that the state of the receiver has changed. As a result, the guest OS starts regulating the window size.

To build a new policy, a Foxy-based module needs to recognize an OS-level abstraction, such as processes and files. For example, to implement priority-based disk I/O scheduling, a Foxy-based module must recognize the 'process' abstraction to discern which process issues each disk I/O request. Unfortunately, the VMM lacks this knowledge of OS-level abstractions; this problem is often referred to as a *semantic gap* [10].

To bridge OS-VMM semantic gaps, FoxyTechnique uses techniques already proposed. Antfarm [15] infers 'process' abstraction, and Geiger [16] infers 'buffer cache' abstraction from the VMM layer. Unfortunately, the techniques for inferring other OS-level abstractions have not been established yet. To obtain information about these abstractions, FoxyTechnique assumes that a user-level process (running on the guest OS) informs the Foxy-enabled VMM of such information. There is one thing to be noted. Currently, we assume the information arriving from the user-level could be trusted. If the user-level information is incorrect, the Foxy-enabled VMM would not work as expected. In the worst-case scenario, a single virtual machine would monopolize all resources. Defending against this kind of attack is beyond the scope of this paper but would be an interesting research topic that bears further investigation.

2.2 Limitations

Incorporating new policies at the VMM layer is not always successful. If the policy to be implemented at the VMM layer conflicts with that of the guest OS, we cannot trick the guest OS. For example, FoxyTechnique cannot implement the deadline disk I/O scheduling [6] if the guest OS employs the idletime scheduling [11]. In the idletime scheduling, the guest OS retains background I/O requests as long as there are foreground I/O requests. Thus, the Foxy-enabled VMM cannot capture the requests from background processes. Even if the deadline for an I/O request from a background process has passed, the Foxy-enabled VMM cannot schedule the request because no such request has been made to the VMM.

Foxy-based modules are not always portable across different OSes if we use specific graybox knowledge for a single OS. Imagine that we are implementing a Foxy-based module that reads and writes the kernel data structures of Linux. In this case, this Foxy-based module would only be effective on Linux. However, there is an interesting tradeoff between accuracy and portability. If a Foxy-based module uses detailed knowledge about a single guest OS, the policy forged by this Foxy-based module would behave very closely to the expected one, but portability would be lost. If a Foxy-based module only uses the common features of ordinary OSes, portability is high, but the policy implemented by the Foxy-based module would be slightly different from the expected one because the Foxy-based module interferes with the guest policy. Recall that our goal is *not* to completely emulate the behavior of the expected policy; FoxyTechnique aims to mimic an expected policy without losing portability. As shown in Section 4 and 5, our case studies demonstrate FoxyTechnique can incorporate a new policy without losing portability. In fact, the tricked guest OS exhibits behavior similar to the expected policy, even though the guest policies employed in the guest OS are completely different.

3. Implementation

We have built two Foxy-based modules, FoxyVegas and FoxyIdle, to demonstrate the usefulness of FoxyTechnique. The target resources for these Foxy-based modules are different. FoxyVegas targets TCP/IP congestion control whereas FoxyIdle targets disk I/O scheduling. Section 4 and 5 describe FoxyIdle and FoxyVegas, respectively.

FoxyTechnique has been applied to Xen [4] VMM version 3.0.2-2. Xen is an open source VMM for the Intel x86 architecture. Xen provides a paravirtualized [27] processor interface, which reduces the virtualization overhead at the expense of porting guest OSes. We carefully avoided making use of this feature of Xen when implementing our Foxy-based modules. Thus FoxyTechnique can be applied to a more conventional VMM such as VMware [23, 26].

Our case studies consist of a set of patches to the Xen hypervisor and Xen's backend drivers. FoxyVegas changes the handlers related to sending and receiving network packets. FoxyVegas patches consist of about 800 lines of code. FoxyIdle changes the VMM

handlers for events like page table updates, accesses to privileged registers and disk reads. FoxyIdle patches consist of about 1500 lines of code.

4. Case Study: FoxyVegas

FoxyVegas is a Foxy-based module for TCP Vegas [7], a TCP congestion control algorithm. TCP congestion control algorithms adjust the size of congestion window to reduce packet loss when the network congested. Numerous congestion control algorithms have been proposed, and each is superior under different circumstances and workloads. TCP Vegas detects network congestion more sensitively and maintains high throughput in congested networks.

4.1 TCP Vegas

TCP Vegas utilizes RTT values of network packets to determine the size of the congestion window (`cwnd`). It detects fluctuations in RTTs to detect network congestion. If RTT values are increasing, TCP Vegas considers network congestion has occurred and reduces `cwnd`. If RTT values are decreasing, `cwnd` is increased. If RTT values are similar to those previous, TCP Vegas does not change `cwnd`.

4.2 Implementation

FoxyVegas controls the virtual network interface card (NIC) to mimic TCP Vegas. Since `cwnd` is kernel data and is not visible to VMM, FoxyVegas adjusts the window size of the guest OS. To control the window size, FoxyVegas creates the illusion that the receiver is loaded heavily. Tricked by this illusion, the guest OS starts reducing the window size. If this behavior is sufficiently similar to TCP Vegas, we can say FoxyVegas tricks guest OSes successfully.

FoxyVegas makes use of graybox knowledge about TCP/IP. TCP/IP controls the transmission rate by adjusting the window size. To adjust the window size, TCP/IP uses two variables: the receiver's advertised window size (`rwnd`) and `cwnd`. TCP/IP chooses the minimum of `cwnd` and `rwnd` as the window size. Thus, we can adjust the window size by controlling either `rwnd` or `cwnd`. As previously explained, FoxyVegas controls `rwnd` since `cwnd` is not visible to VMM, whereas the value of `rwnd` is included in ACKs and thus visible to VMM.

To create the illusion that a receiver is heavily-loaded, the virtual NIC rewrites the `rwnd` included in an ACK packet (Fig. 2). When the virtual NIC receives the ACK packet, it rewrites `rwnd` to `cwnd` calculated by FoxyVegas, if and only if the calculated `cwnd` is smaller than the real `rwnd`. Then, the rewritten ACK is delivered to the guest OS. The guest OS believes that the receiver is overloaded, and sets the `rwnd` rewritten by FoxyVegas to the window size. As a result, the guest OS starts using the value calculated by FoxyVegas as the window size.

4.3 Experiments

To demonstrate FoxyVegas works successfully, we conducted experiments on three machines. Each machine is equipped with a 2.4 GHz Pentium 4 PC with 512 of RAM and an IDE hard disk drive. These machines run as a sender, a router and a receiver, respectively. The sender and receiver machines are connected with the router via a Gigabit Ethernet. FoxyVegas runs on the sender machine. We used Linux kernel version 2.6.16 in both the Xen control and guest domains. The Xen control domain is configured with 256 MB of memory and the guest domain is assigned 128 MB of memory. The router and the receiver execute Linux 2.6.16.

To confirm whether FoxyVegas can trick guest OS policies, we configured the guest Linux to use all congestion control algorithms supported by Linux: 1) Binary Increase Congestion control (`bic`), 2) CUBIC TCP (`cubic`), 3) NewReno (`newreno`),

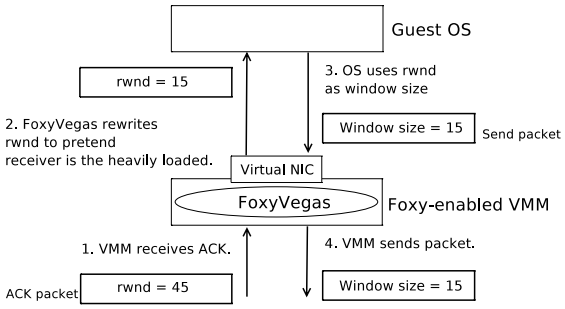


Figure 2. FoxyVegas. *When a receiver's acknowledgment packet (ACK) arrives, FoxyVegas rewrites the advertised window size (rwnd) included in the ACK. The guest OS considers the receiver is loaded heavily, and changes the window size conforming to the rewritten rwnd.*

4) H-TCP (`htcp`), 5) High Speed TCP (`highspeed`), 6) TCP-Hybla (`hybla`), 7) TCP Westwood+ (`westwood`), 8) Scalable TCP (`scalable`) and 9) TCP Vegas (`vegas`). By showing that a single FoxyVegas module can trick various guest policies, we demonstrate that a Foxy-based module is portable across different OSes. The sender and receiver perform the discard communication through the router, which emulates a bottleneck of 200 MB/s, delay of 10 ms, and a maximum queue size of 10500 byte. To emulate a bottleneck, the router used a token bucket filter [22].

The experimental results are shown in Figure 3. The x-axis represents the elapsed time, and the y-axis is the window size. In addition to the nine guest OS policies above, we also plotted the window sizes for Linux TCP Vegas running on physical hardware (*Native Vegas*) and those of Linux TCP Vegas running in the Xen guest domain (*Guest Vegas*). We can see that FoxyVegas tricks the all guest OS policies, except for `vegas`, into behaving like TCP Vegas. In the all policies except for `vegas`, the window size with FoxyVegas is more similar to one of Native Vegas than without it. FoxyVegas does not work well for the guest OS running TCP Vegas. Since the guest OS calculates a smaller `cwnd` than FoxyVegas, the window size is set to the smaller `cwnd`.

Note that FoxyVegas behaves more closely to native Vegas than guest Vegas. Since FoxyVegas is running on physical hardware, it can measure RTTs as accurately as native Vegas. Figure 4 shows the distribution for RTTs observed by native Vegas, guest Vegas, and FoxyVegas. The distribution for guest Vegas is much larger due to the virtualization overhead. Hence, guest Vegas regards the overhead as network congestion, and reduces `cwnd`. In contrast, because FoxyVegas can measure RTTs without the virtualization overhead, FoxyVegas maintains a larger window size than guest Vegas. Therefore, in the experiment where we measured throughput, FoxyVegas outperforms guest Vegas (Tab. 1).

To demonstrate that FoxyVegas provides the benefits of TCP Vegas, we measured throughput when the network is congested. The sender transmits 400 MB data to the receiver. The router emulates a bottleneck of 2000 KB/s, a delay of 10 ms, and a maximum queue size of 6000 byte.

Table 2 lists the experimental results, which suggest that FoxyVegas provides the benefits of TCP Vegas. We can see that throughput with FoxyVegas is higher than that without FoxyVegas except for TCP Vegas. The reason FoxyVegas decreases throughput for TCP Vegas is that it does not work well for TCP Vegas. This reason is described in the previous experiment.

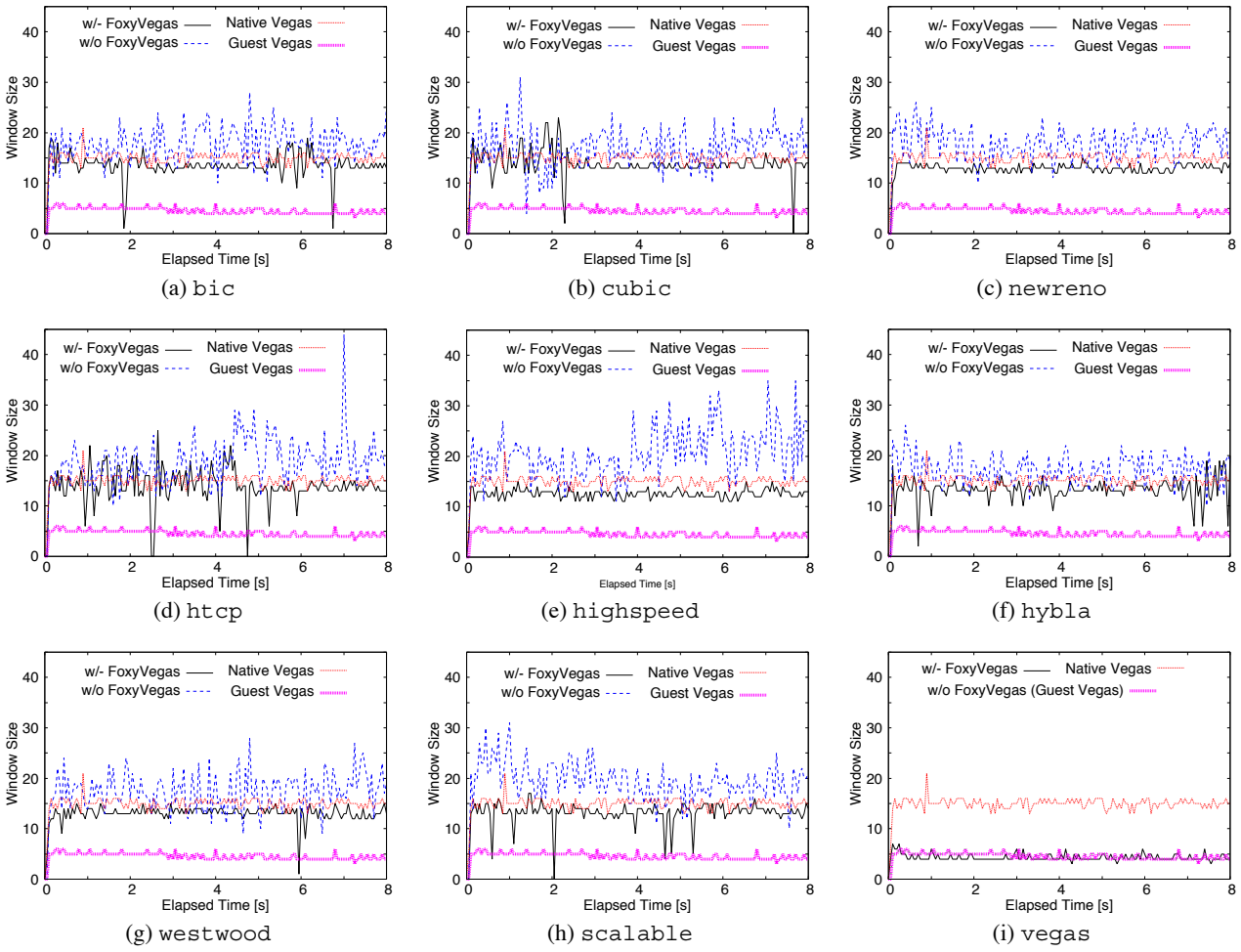


Figure 3. Comparison of the window size. *The figures show the aggregate window size for various TCP congestion control algorithms when FoxyVegas is used or not. Nine congestion control algorithms are used in the experiment.*

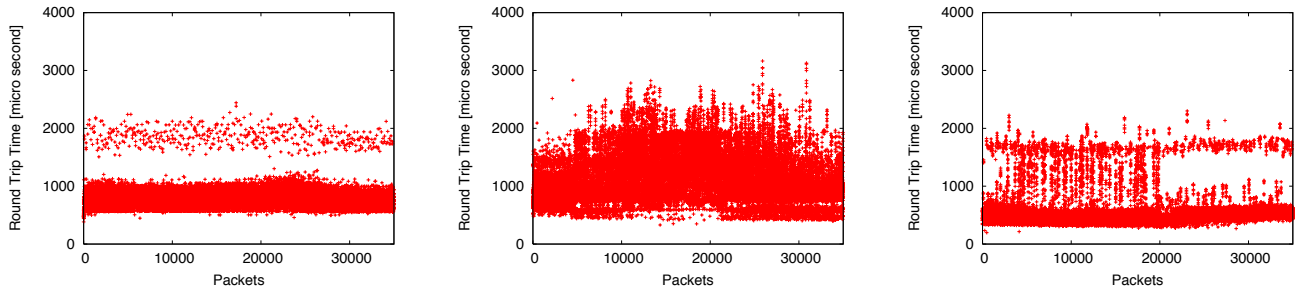


Figure 4. Comparison of RTT fluctuations. *These graphs show the distributions for RTTs observed by native Vegas, guest Vegas, and FoxyVegas. The distribution for FoxyVegas is similar to that for native Vegas. The RTTs for guest Vegas are larger than those for native Vegas and FoxyVegas.*

Tricked Algorithm	bic	cubic	newreno	htcp	highspeed
w/o FoxyVegas [KB/s]	789	921	830	1063	1030
w/- FoxyVegas [KB/s]	1281	1270	1275	1367	1291
Ratio	62.3%	37.8%	53.6%	28.6%	25.3%

hybla	westwood	scalable	vegas
1058	902	732	1466
1317	1326	1328	1433
24.5%	46.9%	81.5%	−2.3%

Table 2. Comparison of throughput in network congestion. *The table lists the improvement rates for throughput when we used FoxyVegas. In almost all cases, FoxyVegas achieves higher throughput for guest Linux under network congestion (bottleneck of 2000 KB/s, delay of 10ms and queue size of 6000 byte)*

Benchmark	Throughput [MB/s]	Decreasing rate
Native Vegas	42.442	—
Guest Vegas	23.831	43.85%
FoxyVegas	33.692	20.62%

Table 1. Comparison of throughput for various TCP Vegas configurations *The table lists the aggregated throughput for various configurations of TCP Vegas. Three Vegas are used in the experiments. The first is Linux TCP Vegas running on the physical machine (Native Vegas). The second is Linux TCP Vegas running on the Xen guest domain (Guest Vegas). The third is FoxyVegas, which tricks Linux BIC. BIC is the Linux default congestion control algorithm.*

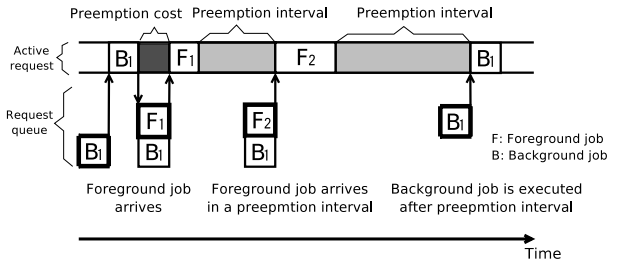


Figure 5. Idletime scheduling. *The figure shows the behavior of the idletime scheduling. A preemption interval starts whenever a foreground request finishes. Background requests are not executed while the preemption interval is active. The background jobs are started only when a preemption interval expires.*

5. Case Study: FoxyIdle

FoxyIdle is a Foxy-based module for the idletime scheduling [11] that schedules disk I/O requests. The idletime scheduler regulates disk I/O requests from background jobs that should run only when computer resources are idle. Examples of background jobs include virus checkers, disk defragmenters, and SETI@home. If not managed properly, the background jobs impede foreground, PC user's normal jobs. To avoid the interference between foreground and background jobs, the idletime scheduling controls the disk I/O request from background jobs.

5.1 Idletime Scheduling

To prevent background jobs from causing excessive preemption costs, the idletime scheduler introduces a time delay, called *preemption interval*. A preemption interval is a time period following each serviced foreground request during which no background request starts — the resource remains idle even when background requests are queued (Fig. 5). The idletime scheduler begins a preemption interval whenever an active foreground request finishes. While the preemption interval is active, the scheduler does not start servicing any background requests. The idletime scheduler starts background requests only when a preemption interval expires. Even if background requests are executed, the idletime scheduler starts servicing foreground requests as soon as possible when a foreground request arrives.

The length of the preemption interval is a parameter which controls the tradeoff between aggressive use of idle resources and impact on the performance of foreground jobs. With a longer preemption interval, foreground performance increases because the opportunity for background jobs to be executed decreases; however, resource utilization decreases. With a shorter preemption interval, utilization is higher but foreground performance decreases.

5.2 Implementation

To forge the behavior of the idletime scheduling, FoxyIdle controls virtual disk drives *differently* from the real ones. To stall background requests, FoxyIdle pretends that the disk drive has been seeking the requested data for a long time. Although the real disk drive has not been seeking at all during this time, the guest OS believes that the requested data have been sought by the disk drive. Therefore, the guest OS keeps blocking the background process. Here, we make use of graybox knowledge that a process is kept *waiting* until the disk drive raises an interrupt. By tuning this waiting time, FoxyIdle forces the guest OS to stall the background process for a preemption interval.

To pretend that a virtual disk drive is seeking data, the virtual disk drive delays sending an I/O request to the real disk drive. When the virtual disk drive detects a background request, it delays the request until the preemption interval expires. During the preemption interval, the guest OS believes that the requested data have been sought by the virtual disk drive. Thus, it blocks the background process that issued the request. When the preemption interval expires, the virtual disk drive actually sends an I/O request to the real disk drive. When the real disk drive raises an interrupt, the virtual disk drive captures and delivers it to the guest OS. Capturing this interrupt, the guest OS naturally resumes the background process. If a foreground request arrives at a virtual disk drive, the request is immediately sent to the real disk drive because we do not have to delay foreground requests. Figure 6 illustrates this behavior of FoxyIdle.

When a virtual disk drive captures a disk read request, FoxyIdle must be able to discern which process issued it and determine whether it is foreground or background. Here, we encounter a

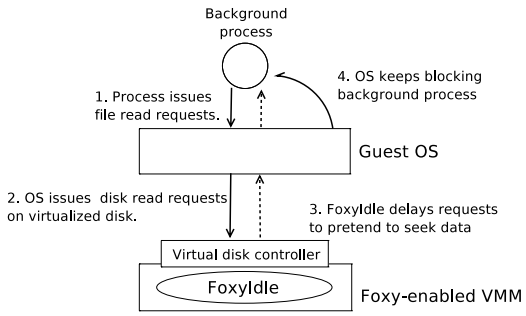


Figure 6. FoxyIdle. *FoxyIdle controls background disk read requests. When a background process issues disk read requests via the guest OS, FoxyIdle delays sending the requests to the real disk drive. The guest OS considers the data are sought by the virtual disk drive, and keeps blocking the background process.*

semantic gap between the OS and the VMM because the VMM lacks the knowledge of the 'process' and 'foreground/background.'

FoxyIdle uses three techniques to bridge this semantic-gap. First, it uses Antfarm that makes the VMM aware of the 'process'. To identify processes, Antfarm tracks virtual address spaces because they correspond to processes. Antfarm tracks address spaces on Intel x86 by observing the value of CR3 (a processor control register). CR3 stores the physical address in which the current page table is placed. Because the instruction for changing the CR3 value is privileged, control is transferred to the VMM whenever a guest OS performs context-switch. Hence, Antfarm regards the value of CR3 as a process ID; by checking the value of CR3, the VMM can know which process is currently running.

Second, to associate disk read requests with processes, we employ a strategy called *context association* [15] that associates a read request with whatever process is currently running. This strategy does not take potential asynchrony within the OS 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 already switched off the context of the processor. Although this leads to association error, it is accurate enough for our purpose.

Finally, to distinguish background requests from foreground requests, FoxyIdle assigns a 'foreground' or 'background' attribute to each CR3 value (recall that FoxyIdle regards the CR3 value as process ID). When a background process starts running, it notifies the FoxyIdle module in the VMM that a background process is running, by sending a UDP message to the FoxyIdle module. When the FoxyIdle module receives this message, it associates the current value of CR3 with the 'background' attribute. Again, we use the context association strategy; when the VMM captures this message, the sending process may already have switched off the processor. To send this notification message, a background process is linked with a library that overrides `libc_main_start()` in `libc`. We used the library preload to override `libc_main_start()`.

5.3 Experiments

To demonstrate FoxyIdle works well, we conducted the experiments on a 3.0 GHz Pentium D PC with 1 GB of RAM and a SATA hard disk drive. We used a Linux kernel version 2.6.16 in both the Xen control and guest domains. The Xen control domain is configured with 512 MB of memory, and the guest domain is assigned 128 MB of memory.

To find out whether FoxyIdle can trick various policies in guest OSes, the guest Linux is configured to use all of Linux's disk I/O schedulers: 1) no operation (`noop`), 2) anticipatory (`ac`), 3) deadline (`dl`), and 4) complete fairness queuing (`cfq`). Similarly

Preemption intervals	`Grep` (sec)	`updatedb` (sec)
5 ms	23.80	95.00
10 ms	14.65	87.30
15 ms	13.44	81.72
20 ms	13.51	88.69
Concurrent execution	78.95	110.96
Standalone	12.86	78.11

Table 3. Comparison of execution times for `grep` and `updatedb`. *The table lists execution times of `grep` (foreground) and `updatedb` (background) with FoxyIdle. Both jobs access a total of about 260 MB of files and directories.*

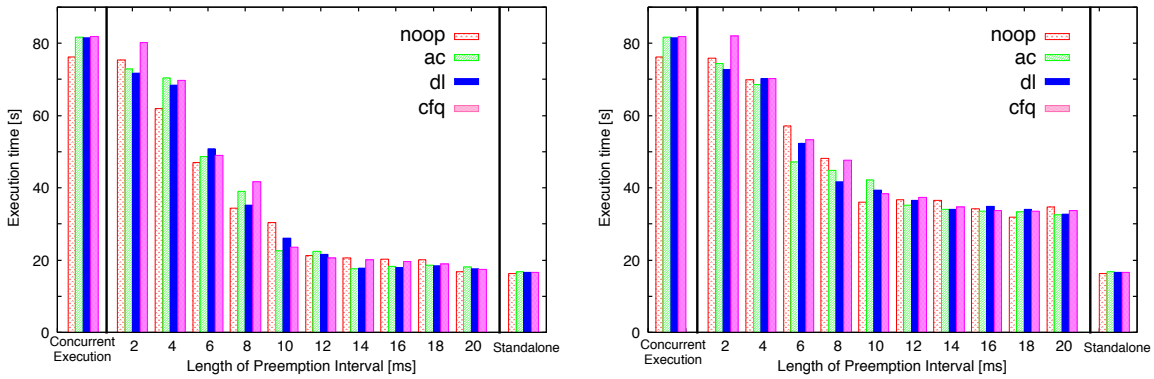
to the case of FoxyVegas, by showing a single FoxyIdle module can trick various guest policies, Foxy-based modules are expected to be portable across different OSes. We used two benchmarks for the experiments: `sequential` and `random`. `Sequential` reads a 1 GB file sequentially, and `random` reads a 1 GB file randomly 10 times. We measured the execution times for foreground and background jobs, varying the length of the preemption interval.

The experimental results are shown in Figures 7 and 8. The x-axis represents the length of a preemption interval, and the y-axis is the execution time (msec). The leftmost bars (labeled *concurrent execution*) plot the execution times when FoxyIdle is disabled. The rightmost bars (labeled *standalone*) plot the execution times when benchmarks were executed alone. In Fig. 7, both the foreground and background jobs are `sequential`. In Fig. 8, the foreground is `random` and the background is `sequential`.

Figures 7 and 8 suggest two things. The first is that FoxyIdle can produce a behavior similar to the idletime scheduling. The second is that FoxyIdle works well regardless of guest OS policies. The execution time for the foreground shortens when the preemption interval is extended. The execution time for the foreground increases with shorter preemption intervals because the background job causes disk contention with the foreground job. With longer preemption intervals, the execution time for the foreground approaches that for "standalone." In concurrent execution, the increasing ratio of the execution time for the foreground is 496% at most in the case that the foreground is `sequential`. When the preemption interval is 20 ms in the same case, the increasing ratio is 7.8% at most. On the other hand, the increasing ratio in the case that the foreground is `random` is 387% at most. When a preemption interval is 20 ms in the same case, the increasing ratio is about 22.3%. Longer preemption intervals also shorten the execution time for the background because the idletime scheduling can avoid disk contention for both jobs.

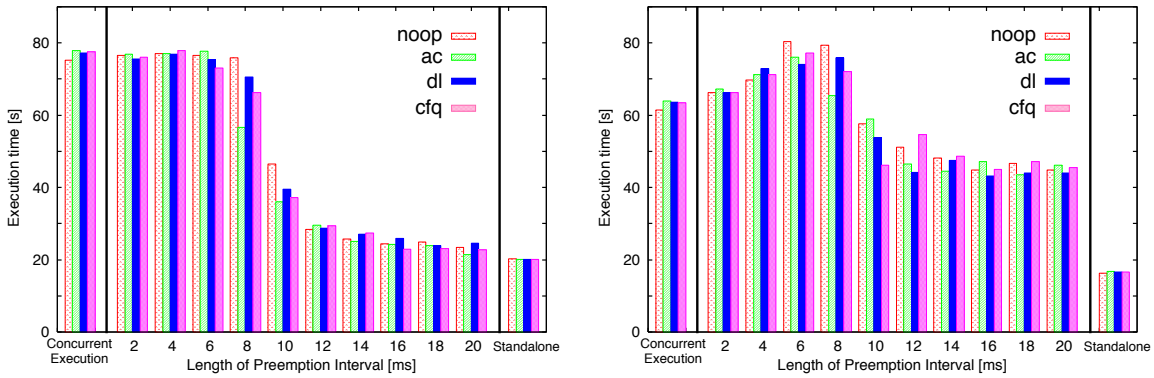
To confirm FoxyIdle is effective in a more realistic situation, we prepared other benchmarks: `grep` and `updatedb`. `Grep` searches for lines containing 'submit_bio' in the source code and documents of Linux kernel 2.6.16. `Updatedb` indexes the source and document files of the same Linux kernel. The total file size is about 260 MB. In this experiment, the foreground was `grep` and the background was `updatedb`. We measured the execution times for both benchmarks, varying the preemption interval. The guest policy used here is anticipatory scheduling, Linux's default disk scheduling policy. To discern the effect of the idletime scheduling, these benchmarks are configured to access separate copies of the Linux files. By doing so, these benchmarks do not share the file cache, and thus generates as many disk read requests as possible.

Table 3 lists the results. In all cases with FoxyIdle, the execution times for both benchmarks are shorter than that for concurrent execution. This is why FoxyIdle enables both jobs to run with less disk contention than concurrent execution (Fig. 9). Disk contention



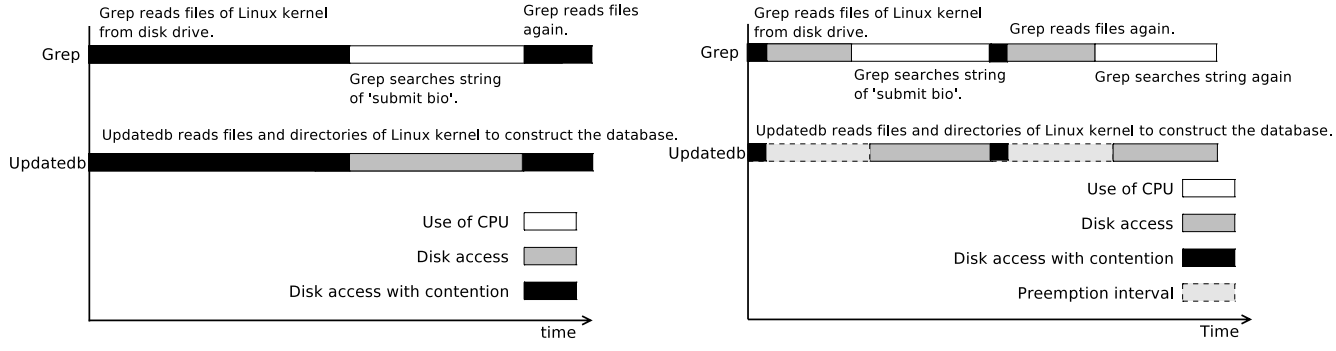
(a) Execution time for foreground `sequential` (b) Execution time for background `sequential`

Figure 7. Difference in execution time between foreground (`sequential`) and background (`sequential`). *In both graphs, the leftmost bars (labeled concurrent execution) plot the execution times when FoxyIdle is disabled. The rightmost bars (labeled standalone) plot the execution times when benchmarks were executed alone.*



(a) Execution time for foreground `random` (b) Execution time for background `sequential`

Figure 8. Difference in execution time between foreground (`random`) and background (`sequential`).



(a) Concurrent Execution (b) Execution with FoxyIdle

Figure 9. Difference between concurrent execution and FoxyIdle execution *The figure shows `grep` and `updatedb` behavior with and without FoxyIdle. In concurrent execution, both jobs make progress with disk contention. In contrast, FoxyIdle allows both benchmarks to run with less disk contention.*

especially affects the performance of the application. To make matters worse, the Xen guest domain incurs a virtualization overhead for applications. Hence, FoxyIdle can produce shorter execution times by avoiding disk contention. When the preemption interval is 15 ms, the execution time for `updatedb` is much shorter. We believe this is because `updatedb` conveniently issues disk I/O while `grep` uses CPU time to search the string of 'submit_bio' from read data. The experimental results suggest that FoxyIdle can forge the idletime scheduling behavior sufficiently well to obtain benefits in real situations.

6. Related Work

Antfarm [15] and Geiger [16] bridge OS-VMM semantic gaps. Antfarm and Geiger enable a VMM to infer the 'process' and the 'buffer cache' state of the guest OS, and allow some resource management policies to be incorporated into the VMM layer. FoxyTechnique addresses the interference between VMM-level and guest OS policies, and tricks guest OS policies by making use of this interference. For example, FoxyVegas makes use of the common features of TCP/IP protocol stacks to implement TCP Vegas. Antfarm and Geiger do not explicitly make use of the interference between the VMM layer and the guest OS.

The balloon driver [26] regulates the amount of memory assigned to guest OSes. A balloon driver is loaded into a guest OS and tricks the guest OS into believing a region of the memory is being used. The balloon process [24] controls CPU schedulers on multiprocessor VMs when each virtualized processor has different processing power. Although these approaches trick guest OS policies for resource management, they do not address the problem of introducing a new policy into guest OSes. In addition, portability is slightly reduced because a new driver must be developed.

Introvirt [17] enables us to change the OS behavior without changing the source of its kernel. The goal of Introvirt is to apply OS patches *without* directly changing the kernel source, and it uses breakpoints to gain control from the guest OS. Although Introvirt is a powerful tool for changing the OS behavior, it requires the source code of guest OSes. Hence, it is almost impossible to apply Introvirt to proprietary OSes.

To improve the performance of virtual machines, some VMMs manage physical resources in a way transparently to guest OSes. Disco [8] and Potemkin [25] apply the copy-on-write technique to enhance memory usage. They do not address the problem of incorporating a new policy into a guest OS, and the interference between a VMM layer and a guest OS.

The technologies for inserting a new resource management policy into an OS kernel have been investigated for the past decades. To enable a new policy to be incorporated into an existing OS, OS researchers have proposed the concept of extensible OSes [5, 12, 21]. Newhouse and Pasquale [18] developed user-level schedulers for CPU intensive jobs. Graybox techniques [2] enable us to infer the internal states of an OS at the user-level. Based on this inference, we can build various resource management policies at the user-level. FoxyTechnique borrows the concept of the graybox technique to deduce the internal states of guest OSes. Infokernel [3] exposes the internal states of and algorithms employed by an OS to facilitate the development of user-level resource managers. OS profilers such as Debox [20] and Dtrace [9] can be used to obtain the OS internal states.

7. Conclusion

It is difficult to integrate a new resource management policy into an existing OS despite many proposals for sophisticated, innovative policies. This is because modern OSes are too complex and large to modify their kernel code. This paper presented FoxyTechnique, a technique of enabling us to incorporate new policies into existing OSes without making any changes to its kernels. FoxyTechnique virtualizes physical devices differently from real ones, and tricks the guest OS policy into producing a behavior similar to a desired policy. To embody the concept of FoxyTechnique, we conducted two case studies, FoxyVegas and FoxyIdle, on the Xen VMM. The targets of these case studies are different; FoxyVegas targets TCP/IP congestion control and FoxyIdle targets disk I/O scheduling. Through these case studies, we demonstrated that FoxyTechnique can trick various guest OS policies into producing a behavior similar to a desired policy.

Acknowledgments

We wish to thank the anonymous reviewers who gave us their helpful comments and suggestions. This work was partially supported by a Grant-in-Aid for Core Research for Evolutional Science and Technology (CREST), from the Japan Science and Technology Agency (JST).

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX Development. In *Proceedings of Summer USENIX Conference*, pages 93–112, June 1986.

[2] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, October 2001.

[3] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, October 2003.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and Art of Virtualization. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, October 2003.

[5] Braian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267–283, December 1995.

[6] Daniel P. Bovet and Marco Cesati. Understanding the LINUX KERNEL (3rd Edition). O'Reilly Media,Inc., 2006.

[7] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of ACM SIGCOMM '94*, pages 24–35, August 1994.

[8] Edouard Bugnion, Scott Devine, and Mendel 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, December 1997.

[9] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the USENIX 2004 Annual Technical Conference (USENIX '04)*, pages 15–28, June 2004.

[10] Peter M. Chen and Brian D. Noble. When Virtual is Better than Real. In *Proceedings of Workshop on Hot Topics in Operating Systems (HotOS '01)*, pages 133–138, June 2001.

[11] Lars Eggert and Joseph D. Touch. Idletime Scheduling with Preemption Intervals. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 249–262, October 2005.

[12] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium*

on Operating Systems Principles (SOSP'95), pages 251–266, December 1995.

[13] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[14] Hermann Hőrtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schőnberg, and Jean Wolter. The Performance of μ-Kernel-Based Systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP '97)*, pages 66–77, Octorber 1997.

[15] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of USENIX Annual Technical Conference (USENIX '06)*, June 2006.

[16] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, October 2006.

[17] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter 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, October 2005.

[18] Travis Newhouse and Joseph Pasquale. ALPS: An Application-Level Proportional-Share Scheduler. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC '06)*, pages 279–290, June 2006.

[19] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 64–75, December 1981.

[20] Yaoping Ruan and Vivek Pai. Making the "Box" Transparent: System Call Performance as a First-class Result. In *Proceedings of the USENIX 2004 Annual Technical Conference (USENIX '04)*, pages 1–14, June 2004.

[21] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–227, October 1996.

[22] S. Shenker and J. Wroclawski. RFC2216: Network Element Service Specification Template, 1997. http://rfc.net/rfc2216.html.

[23] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the USENIX 2001 Annual Technical Conference (USENIX '01)*, pages 1–14, June 2001.

[24] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of Virtual Machine Research and Technology Symposium (VM '04)*, pages 43–56, May 2004.

[25] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex Snoeren, Geoffrey Voelker, and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 148–162, December 2005.

[26] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of Symposium on Operating System Design and Implementation (OSDI '02)*, pages 181–194, December 2002.

[27] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of Symposium on Operating System Design and Implementation (OSDI '02)*, pages 195–209, December 2002.