IPSJ Transactions on Advanced Computing Systems Vol. 1 No. 1 144–159 (June 2008)

Regular Paper

Introducing New Resource Management Policies Using a Virtual Machine Monitor

Hiroshi Yamada[†1] and Kenji Kono[†1]

Integrating new resource management policies into operating systems (OSes) is an ongoing process. Despite innovative policy proposals being developed, it is unrealistic to widely deploy a new one because it is a difficult, costly and often an impractical endeavor to modify an existing operating system 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 behavior similar to a desired policy. FoxyTechnique has three 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.

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.

†1 Keio University

Despite many sophisticated, innovative policy proposals being presented, it is quite difficult to widely deploy an 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 an 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 deploying 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 get used widely since cross-platform applications would only be able to use the features 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)–3)], extensible OSes [4)–6)], and infokernels [7)] 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 an OS, many techniques have been developed that enable us to build resource management policies on "as is" operating systems. Newhouse, et al. [8)] have shown that a user-level CPU scheduler for CPU intensive jobs can be implemented on unmodified FreeBSD. Graybox techniques [9)] 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 [10)] 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) has 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 brings us 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 [11),12)] that facilitate the development of innovative functionalities

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 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 the graybox knowledge that time slices are measured by counting timer interrupts in the guest OS [⋆1].

FoxyTechnique has the following advantages:

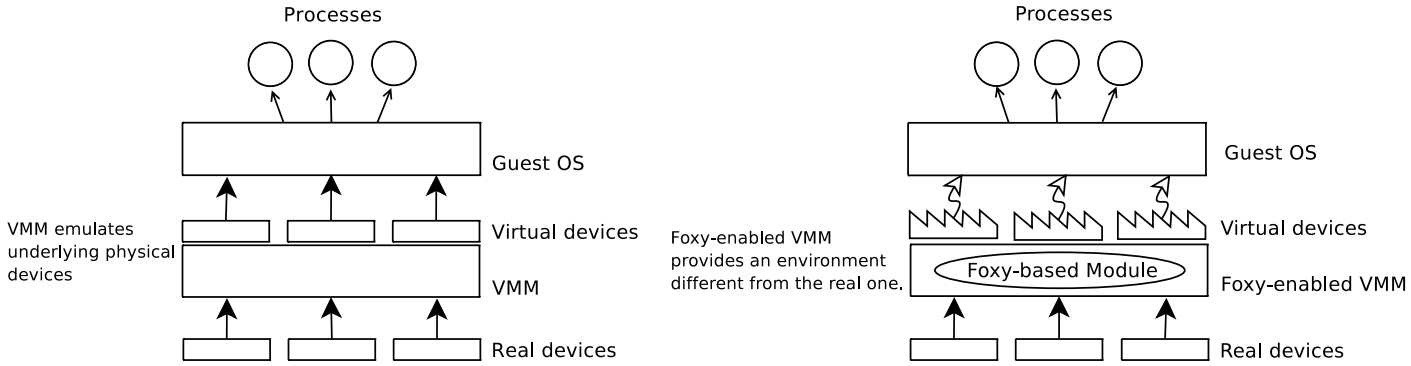
- **No changes to OS kernels required.**
  FoxyTechnique enables us to incorporate new resource management policies into an existing OS without the need to make any changes to it. Foxy-based resource management policies are implemented within a VMM and thus are isolated from guest OSes by stable virtual 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, one of our Foxy-based modules can trick binary increase congestion (BIC) control, NewReno, CUBIC TCP, TCP-Hybla, TCP Westwood+ and Scalable TCP into TCP Vegas. On the other hand, FoxyTechnique does not work well in some situations. The limitation of FoxyTechnique is discussed later in this paper.
- **Better performance estimation.**
  Resource management policies are often required to measure performance indicators to estimate the cost of the next possible operations. The accuracy of performance indicators affects the performance of resource management. For example, TCP Vegas measures round trip times (RTTs) of network packets to detect network congestion. If the RTT contains a large error, 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 an overall better performance than guest OS policies. In fact, our Foxy-based module for TCP Vegas shows better performance than TCP Vegas running in the guest Linux.

Note that FoxyTechnique does *not* facilitate the implementation of resource management policies. Sometimes, it is not easy to implement Foxy-based modules because FoxyTechnique indirectly introduces a new policy. At the expense of this difficulty, FoxyTechnique brings the three advantages described above.

To embody the concept of FoxyTechnique, we conducted two case studies, *FoxyVegas* and *FoxyIdle*, on the Xen virtual machine monitor [13]. 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 an illusion that a receiver is overloaded to force a guest OS to make its window size smaller. 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 Idletime scheduling [14]. This scheduling prevents background processes from degrading the performance of foreground processes. The idletime schedul-

⋆1 This is an example to help you understand the FoxyTechnique concept. If you actually want to change a CPU scheduling policy of commodity OSes, you might not only change the rate of timer interrupts but need to employ another technique.



(a) Conventional virtual machine monitor (b) Foxy-enabled virtual machine monitor

Fig. 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.

ing stalls disk read requests from background processes if they interfere with the 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 show 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 the implementation of FoxyTechnique. Sections 4 and 5 report the design, implementation and experiments of case studies, FoxyVegas and FoxyIdle, respectively. 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 virtual machine monitors (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 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 transferred to the VMM. At this point, the VMM can catch 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. To achieve an expected behavior of a guest OS, FoxyTechnique takes the guest policy into consideration. In other words, we make use of *graybox knowledge* [9] 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 change the interrupt rate of the virtual timer device to alter the policy of 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 a timer. For example, as explained earlier, if the Foxy-based module raises more timer interrupts than a real one, the guest OS clock advances more quickly. If the Foxy-based module raises less 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 at the guest OS, it considers that network congestion has occurred and reduces the congestion window size. As a result, the TCP window size is reduced and the effective bandwidth is throttled.
- **Rewriting contents 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 sets data into registers, the Foxy-based module rewrites values of the registers, and then sends interrupts to the guest OS. Rewriting the contents of data registers enables us to control the window size of packets in TCP/IP. By rewriting the server receivable data size included in packets, we can make the guest OS believe 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* [15].

To bridge OS-VMM semantic gaps, FoxyTechnique uses techniques already proposed. Antfarm [11] infers 'process' abstraction, and Geiger [12] infers 'buffer cache' abstraction from the VMM layer. Unfortunately, the techniques for inferring other OS-level abstractions have not been established yet. To obtain the information about these abstractions, FoxyTechnique assumes that a user-level process (running on the guest) informs OS the Foxy-enabled VMM of such information. There is one thing to be noted. Currently, we assume the information coming 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, a single virtual machine would monopolize all the 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 [16] if the guest OS employs the idletime scheduling [14]. 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 the background processes. Even if the deadline for an I/O request from a background process has expired, 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, an interesting tradeoff exists between accuracy and portability. If a Foxy-based module uses detailed knowledge of a single guest OS, the policy forged by this Foxy-based module would behave very similarly to the expected one, but the portability would be lost. If a Foxy-based module only uses common features of ordinary OSes, the 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 that FoxyTechnique can incorporate a new policy without losing the portability. In fact, the tricked guest OS shows behavior similar to the expected policy, even though the guest policies employed in the guest OS are completely different.

2.3 Discussion

We need to discuss the scope of resource management policies that FoxyTechnique can successfully introduce. It is quite difficult to generally determine the scope, which depends largely on how to make use of graybox knowledge about target OSes. Instead, we are trying to show the applicability of FoxyTechnique by applying it to various kinds of resource management. As a first step, we implemented a couple of resource management policies based on FoxyTechnique in this paper.

FoxyTechnique may complicate the process of the guest OS maintenance since it forges a device different from the real one. We assume that the developers should debug and tune a guest OS without Foxy-enabled VMM to eliminate side effects of Foxy-based modules. Thus, they can maintain the OS on a regular basis.

We also discuss the difficulty in designing the tricking actions of Foxy-based modules. First, when we introduce a new policy with FoxyTechnique, we need to study the policy in detail and judge whether it can be introduced by altering device behavior or not. Therefore, we need to be familiar with the targeted OS, and know how the OS reacts to devices whose behavior has been transformed. Second, there is a trade-off between portability and efficiency. If we use the knowledge specific to the targeted OS, the portability of the Foxy-based module is lowered, as we described in Section 2.2.

3. Implementation

To demonstrate the usefulness of FoxyTechnique, we have built two Foxy-based modules: FoxyVegas and FoxyIdle. The target resources of 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 [13] virtual machine monitor version 3.0.2-2. Xen is an open source VMM for the intel x86 architecture. Xen provides a paravirtualized [17] 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 virtual machine monitor such as VMware [18),19].

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 1,500 lines of code.

4. Case Study: FoxyVegas

FoxyVegas is a Foxy-based module for TCP Vegas [20], a TCP congestion control algorithm. TCP congestion control algorithms adjust the size of the congestion window to reduce packet loss under network congestion. A lot of congestion control algorithms have been proposed each of which is superior under different circumstances and different workloads. TCP Vegas detects network congestion more sensitively and keeps high throughput in network congestion.

4.1 TCP Vegas

TCP Vegas utilizes RTT values of network packets to decide the size of the congestion window (`cwnd`). To detect network congestion, TCP Vegas detects

fluctuation of RTTs. If RTT values are becoming larger, TCP Vegas considers network congestion has occurred and makes `cwnd` smaller. If RTT values are becoming smaller, `cwnd` is made larger. If RTT values are similar to previous ones, TCP Vegas does not change `cwnd`.

4.2 Implementation

To mimic TCP Vegas, FoxyVegas controls a virtual network interface card (NIC). Since `cwnd` is a kernel data and not visible to a VMM, FoxyVegas adjusts the window size of a guest OS. To control the window size, FoxyVegas gives an illusion that a receiver is loaded heavily. Tricked by this illusion, the guest OS starts reducing the window size. If this behavior is similar enough 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: 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 explained above, FoxyVegas controls `rwnd` since `cwnd` is not visible to the VMM, whereas the value of `rwnd` is included in ACKs and thus visible to the VMM.

To create an illusion that a receiver is under heavy load, a virtual NIC rewrites the `rwnd` included in an ACK packet (**Fig. 2**). When a virtual NIC receives an 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 that FoxyVegas works successfully, we conducted experiments on three machines. Each machine is 2.4 GHz Pentium 4 PC with 512 MB of RAM and an IDE hard disk drive. These machines run as a sender, a router and a receiver, respectively. The sender and the receiver are connected with the router via the 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

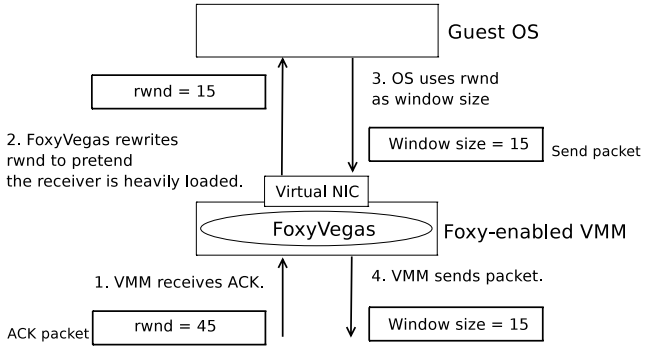
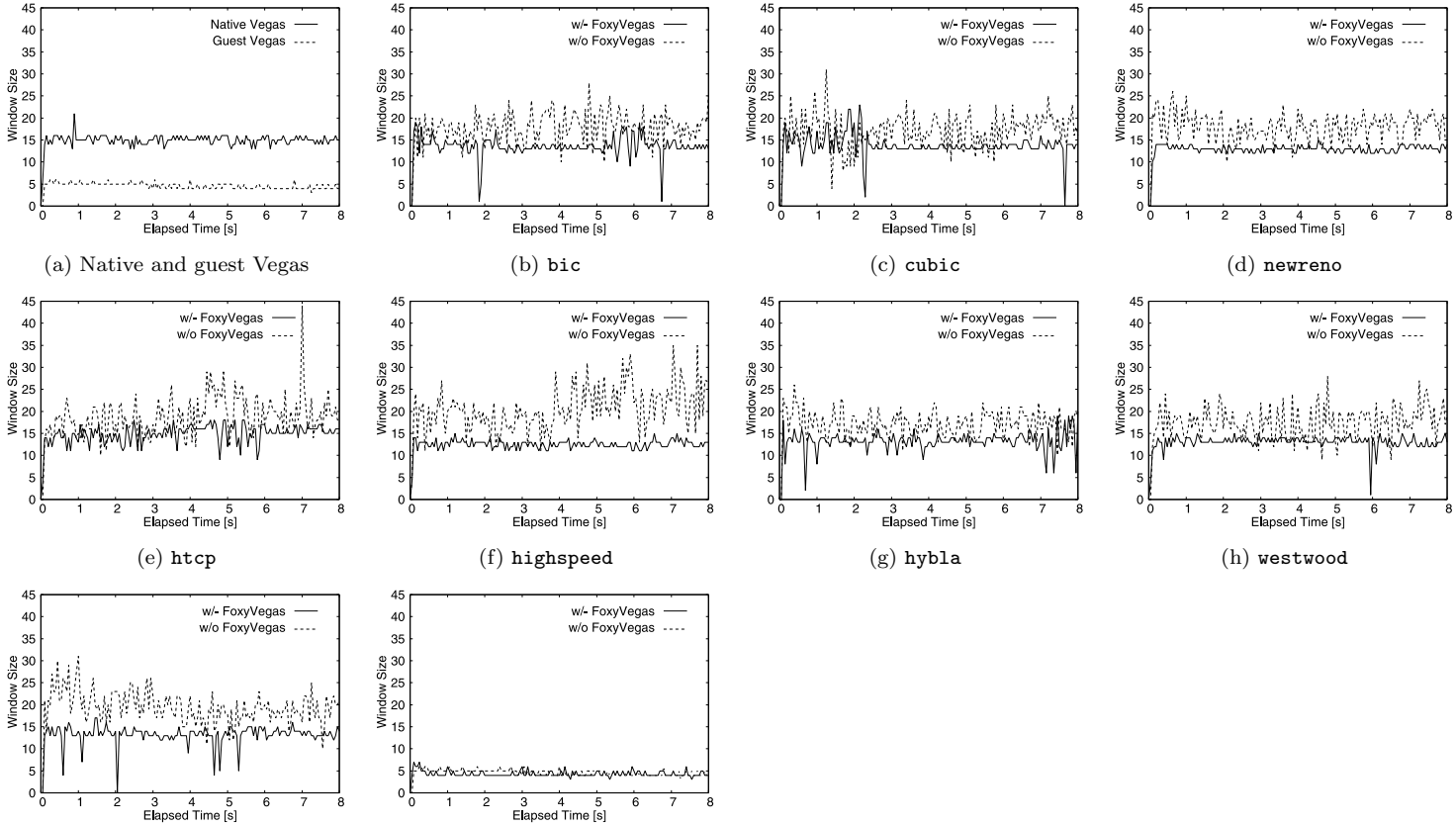


Fig. 2 FoxyVegas. When a receiver's acknowledgment packet (ACK) arrives, FoxyVegas rewrites the advertised window size (rwnd) included in the ACK. The guest OS considers the receiver is loaded heavily, and changes the window size conforming with the rewritten rwnd.

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`), 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 the 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 10,500 KB. To emulate the bottleneck, the router used the token bucket filter [21].

The experimental results are shown in **Fig. 3**. The x-axis is the elapsed time, and the y-axis is the window size. For comparison, we also showed the window sizes of Linux TCP Vegas running on physical hardware (*Native Vegas*) and of Linux TCP Vegas running in the Xen guest domain (*Guest Vegas*) in Fig. 3 (a). We can see that FoxyVegas tricks all the guest OS policies, except for `vegas`, into behaving like TCP Vegas. FoxyVegas does not work well for the guest OS running TCP Vegas. Since the guest OS calculates `cwnd` smaller than FoxyVegas,



(a) Native and guest Vegas (b) `bic` (c) `cubic` (d) `newreno`

(e) `htcp` (f) `highspeed` (g) `hybla` (h) `westwood`

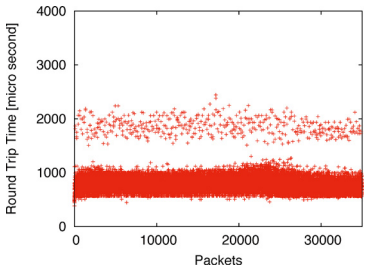
(i) `scalable` (j) `vegas`

Fig. 3 Comparison of the window size. The figure shows the aggregate window size for various TCP congestion control algorithms whether FoxyVegas is used or not. The experiment uses nine congestion control algorithms.

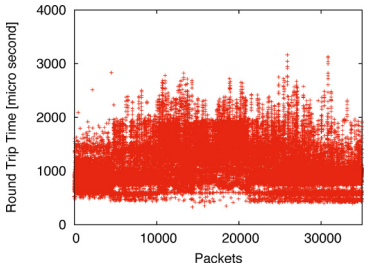
the window size is set to the smaller `cwnd`.

Note that FoxyVegas behaves more similarly to native Vegas than guest Vegas. Since FoxyVegas is running on physical hardware, FoxyVegas can measure RTTs as accurately as native Vegas. **Figure 4** shows the distribution of RTTs observed by native Vegas, guest Vegas, and FoxyVegas. The distribution of 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 keeps a larger window size than guest Vegas. Therefore, an experiment in which we measured throughput between the sender and the receiver shows FoxyVegas outperforms guest Vegas (**Table 1**).

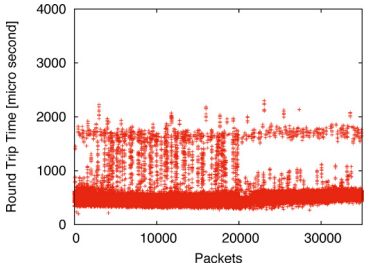
To demonstrate that FoxyVegas provides benefits of TCP Vegas, we measured



(a) Native Vegas



(b) Guest Vegas



(c) FoxyVegas

Fig. 4 Comparison of RTT fluctuations. These graphs show the distribution of RTTs observed by native Vegas, guest Vegas and FoxyVegas. The tendency of the distribution of FoxyVegas is similar to one of native Vegas. The RTTs of guest Vegas is larger than native Vegas and FoxyVegas.

throughput under network congestion. The sender transmits 400 MB data to the receiver. The router emulates a bottleneck of 2,000 KB/s and a delay of 10 ms, and a maximum queue size of 6,000 KB.

Table 1 Comparison of throughput for various configurations of TCP Vegas. The table reports the aggregated throughput for various configurations of TCP Vegas. The experiments use three Vegas. 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 last is FoxyVegas which triked Linux BIC. BIC is the Linux default congestion control algorithm.

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

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

5. Case Study: FoxyIdle

FoxyIdle is a Foxy-based module for the idletime scheduling [14] that schedules disk I/O requests. The idletime scheduler regulates disk I/O requests from background jobs which 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. To the authors' knowledge, there is no commodity OS which supports the idletime scheduling.

5.1 Idletime Scheduling

To prevent background jobs from causing excessive preemption cost, 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

Table 2 Comparison of throughput in network congestion. The table reports the improvement rate of throughput when we used FoxyVegas. In almost cases, FoxyVegas makes guest Linux achieve higher throughput under network congestion (a bottleneck of 2,000 KB/s, a delay of 10 ms and queue size of 6 MB).

Tricked Algorithm	bic	cubic	newreno	htcp	highspeed	hybla	westwood	scalable	vegas
w/o FoxyVegas [KB/s]	789	921	830	1,063	1,030	1,058	902	732	1,369
w/- FoxyVegas [KB/s]	1,281	1,270	1,275	1,367	1,291	1,317	1,326	1,328	1,235
Ratio	62.3%	37.8%	53.6%	28.6%	25.3%	24.5%	46.9%	81.5%	−9.8%

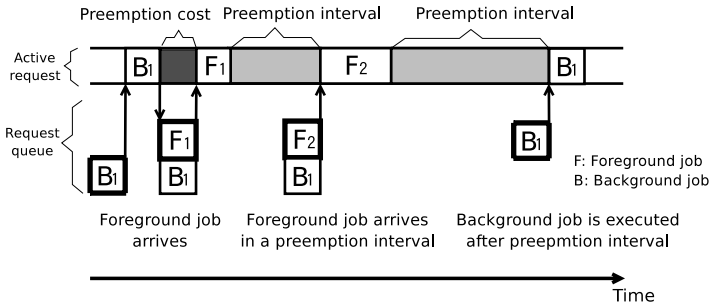


Fig. 5 Idletime scheduling. The figure shows the behavior of the idletime scheduling. A preemption interval starts whenever a foreground request finishes. While the preemption interval is active, the background requests are not executed. The background jobs are started only when a preemption interval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

idletime scheduler starts background requests only when a preemption interval expires. Even if background requests are executed, the idletime scheduler starts serving 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 performance of foreground jobs. With a longer preemption interval, foreground performance increases because the opportunity for background jobs to be executed decreases, but the resource utilization becomes lower. With a shorter preemption interval, the 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 is seeking the requested data for a long time. During this time, although the real disk drive is not seeking at all, 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 the 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 a virtual disk drive detects a background request, it delays the request until a 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 the I/O request to the real disk drive. When the real disk drive raises an interrupt, the virtual disk drive catches and delivers it to the guest OS. Catching 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 catches a disk read request, FoxyIdle must be able to discern which process issues it and determine whether it is foreground or background. Here, we encounter a semantic gap between the OS and the VMM because the VMM lacks the knowledge of 'process' and 'foreground/background.'

FoxyIdle uses three techniques to bridge this semantic-gap. First, it uses Antfarm [11] that makes a VMM aware of 'process'. To identify processes, Antfarm can track virtual address spaces because they correspond to processes. Antfarm tracks address spaces on Intel x86 by observing the value of CR3 (a processor

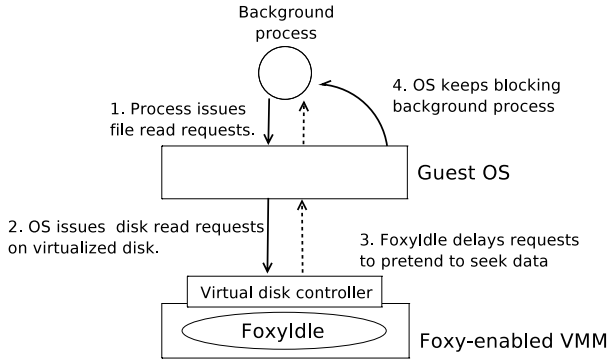


Fig. 6 FoxyIdle. FoxyIdle controls background disk read requests. When a background process issues disk read requests via the guest OS, FoxyIdle delays sending the requests to the real disk drive. The guest OS considers the data is sought by the virtual disk drive, and keeps blocking the background process.

control register). CR3 stores the physical address at which the current page table is placed. Because the instruction to change CR3 value is privileged, the control is transferred to the VMM whenever a guest OS performs context-switch. Hence, Antfarm regards the value of CR3 as process ID; by checking the value of CR3, the VMM can know which process is currently running.

Second, to associate disk read requests to processes, we employ the strategy called *context association* [11] that associates a read request with whatever process is currently running. This 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 already context switched off the processor. This leads to association error, but is accurate enough for our purpose.

Finally, to distinguish background requests from foreground requests, FoxyIdle assigns the 'foreground' or 'background' attribute to each CR3 value (recall that FoxyIdle regards 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 catches this message, the sending process may have already 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()`.

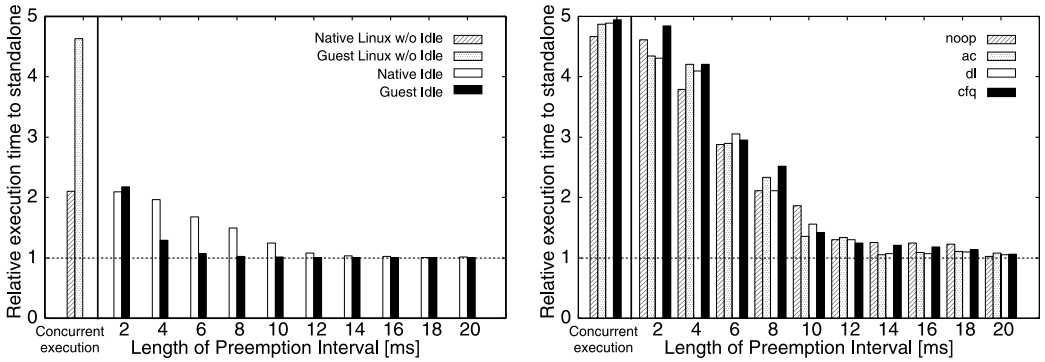
5.3 Experiments

To demonstrate that 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 Linux kernel version 2.6.16 in both the Xen control domain 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 disk I/O schedulers of Linux: 1) no operation (`noop`), 2) anticipatory (`ac`), 3) deadline (`dl`), and 4) complete fairness queuing (`cfq`). Similarly 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 also implemented the idletime disk scheduler into Linux 2.6.16 to compare its behavior to one forged by FoxyIdle. In the following experiment, we used two benchmarks: `sequential` and `random`. `Sequential` reads a 1 GB file sequentially, and `random` reads a 1 GB file randomly for 10 times. We measured the execution time of foreground and background jobs, varying the length of a preemption interval.

The experimental results are shown in **Figs. 7** and **8**. The x-axis is the length of a preemption interval, and the y-axis is the relative execution time to the "standalone" execution time when each benchmark ran alone. In each figure, (a) and (c) are the results when the idletime disk scheduler was executed on physical hardware (*Native Idle*) and the Xen guest domain (*Guest Idle*). On the other hand, (b) and (d) are the ones when FoxyIdle ran. The leftmost bars (labeled *concurrent execution*) plot the execution times when the idletime scheduler and FoxyIdle were disabled. In Fig. 7, both foreground and background jobs are `sequential`. In Fig. 8, the foreground is `random` and the background is `sequential`.

These figures indicate two things. One is that FoxyIdle can produce behavior similar to the idletime scheduling; FoxyIdle preserves the performance of the fore-



(a) Execution time of foreground `sequential` on native Idle and guest Idle

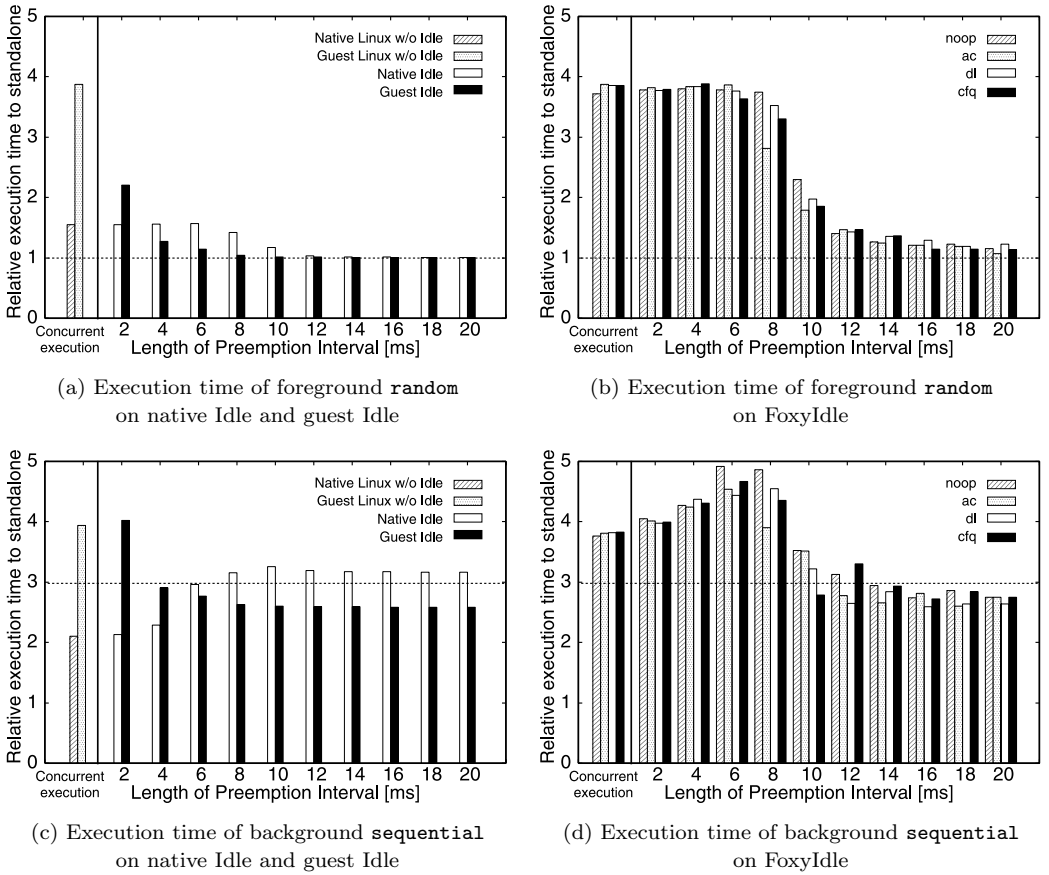
(b) Execution time of foreground `sequential` on FoxyIdle

(c) Execution time of background `sequential` on native Idle and guest Idle

(d) Execution time of background `sequential` on FoxyIdle

Fig. 7 Difference of execution time between the foreground (`sequential`) and the background (`sequential`). In each graph, the leftmost bars (labeled concurrent execution) plot the execution times when the idletime scheduler is disabled.

ground jobs gradually by lengthening a preemption interval. The other is that FoxyIdle works well regardless of guest OS policies. In the idletime scheduling, the execution time of the foreground gets smaller and closer to that of "standalone" when the preemption interval is set larger. In fact, Fig. 7 (a) and Fig. 8 (a) reveal that native Idle and guest Idle make the relative execution times of the foregrounds close to 1 when a preemption interval is larger than 14 ms. On FoxyIdle, the relative execution times similarly become about 1 with larger preemption intervals, as shown in Fig. 7 (b) and Fig. 8 (b). Figure 7 (c) exhibits that native Idle and guest Idle make the relative execution times of the backgrounds close to 2 when the preemption intervals become larger. In addition, the relative times get close to 2 on FoxyIdle, as shown in Fig. 7 (d). When the foreground is `random`, the relative execution times of the backgrounds get close to 3 on native Idle with



(a) Execution time of foreground `random` on native Idle and guest Idle

(b) Execution time of foreground `random` on FoxyIdle

(c) Execution time of background `sequential` on native Idle and guest Idle

(d) Execution time of background `sequential` on FoxyIdle

Fig. 8 Difference of execution time between the foreground (`random`) and the background (`sequential`).

the preemption intervals larger. On the other hand, FoxyIdle gradually makes the relative times less than 3 when the preemption intervals are larger, similarly to guest Idle (see Fig. 8 (c) and (d)).

These experimental results also reveal that guest Idle preserves the performance of the foregrounds with smaller preemption intervals than native Idle. In Fig. 7 (a), for example, the relative execution time is 1.29 on guest Idle while it is 1.96 on native Idle when the preemption interval is set to 4 ms. This is because the timer which guest Idle uses to stall background requests is not accurate due to the virtualization overhead. **Figure 9** shows the absolute values of the errors between the calculated interval and the actual interval with the box and whiskers plot, when the preemption interval is 4 ms. The calculated interval is an interval which the idletime scheduler or FoxyIdle calculated for stopping background

Table 3 Comparison of execution times of `grep` and `updatedb`. The table reports execution times of `grep` (foreground) and `updatedb` (background) with FoxyIdle. Both jobs access totally about 260 MB of files and directories.

	Native Idle		Guest Idle		FoxyIdle	
Preemption intervals	`Grep` (sec)	`updatedb` (sec)	`Grep` (sec)	`updatedb` (sec)	`Grep` (sec)	`updatedb` (sec)
5 ms	9.72	101.13	24.34	113.44	24.92	113.74
10 ms	9.86	101.91	17.50	109.07	16.40	107.94
15 ms	9.25	101.17	15.40	107.59	15.50	107.60
20 ms	8.72	103.46	14.63	107.30	15.41	107.46
Concurrent execution	10.14	102.02	82.94	123.31	82.94	123.31
Standalone	8.67	92.04	14.34	87.62	14.34	87.62

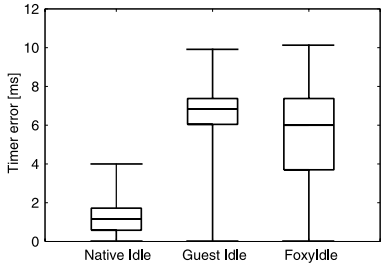


Fig. 9 Timer Errors of the idletime schedulers. The figure shows the absolute values of the errors between the calculated interval and the actual interval. The calculated interval is an interval which the idletime scheduler or FoxyIdle calculated for stopping background requests. The actual interval is an interval for which background requests were actually stopped.

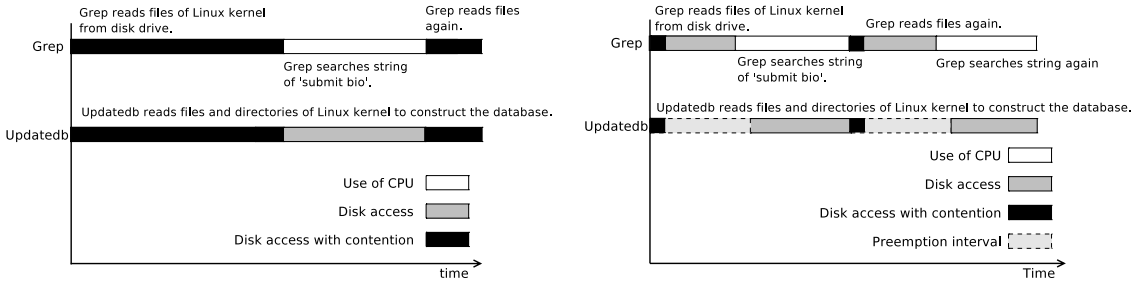
requests. The actual interval is an interval for which background requests were actually stopped. Since the timer functionality in Linux is basically unsophisticated, the median timer error is 1.16 ms even on native Idle. On guest Idle, the median is higher than native Idle and FoxyIdle, 6.83 ms. As a result, guest Idle excessively stalls background requests and thus the foregrounds are more easily able to make progress. Although the timer errors on FoxyIdle are also higher than native Idle, FoxyIdle does not restrict the execution of the backgrounds as excessively as guest Idle because the timer on FoxyIdle is more accurate than that on guest Idle (see Fig. 9).

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 time of both benchmarks, varying a preemption interval. The guest policy used here is anticipatory scheduling, the default Linux 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 generate as many disk read requests as possible.

Table 3 lists the results. In all the cases with FoxyIdle, the execution times of both benchmarks are shorter than in the case of concurrent execution, similarly to native Idle and guest Idle. This is because FoxyIdle enables both jobs to run with less disk contention than concurrent execution (**Fig. 10**). The experimental results suggest that FoxyIdle can forge the idletime scheduling behavior sufficiently to obtain its benefits in a realistic situation.

6. Related Work

Antfarm [11] and Geiger [12] 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 VMM layer and the guest OS.



(a) Concurrent Execution (b) Execution with FoxyIdle

Fig. 10 Difference between the concurrent execution and the FoxyIdle execution. The figure shows `grep` and `updatedb` behavior whether FoxyIdle is used or not. In concurrent execution, both jobs make progress with disk contention. In contrast, FoxyIdle allows both benchmarks to run with less disk contention.

The balloon driver [19] regulates an 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 memory is used. The balloon process [22] 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, the portability is slightly reduced because a new driver must be developed.

Introvirt [23] 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 uses breakpoints to get the control from the guest OS. Introvirt is a powerful tool for changing the OS behavior, but 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 transparently to guest OSes. Disco [24] and Potemkin [25] apply the copy-on-write technique to enhance the memory usage. They do not address the problem of incorporating a new policy into a guest OS, and the interference between VMM layer and a guest OS.

The technologies for inserting a new resource management policy into an OS kernel have been investigated over the past decades. To enable a new policy to be incorporated into an existing OS, the OS researchers have proposed the concept of extensible OSes [4–6]. Newhouse and Pasquale [8] developed user-level schedulers for CPU intensive jobs. Graybox technique [9] enables us to infer 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 guess the internal states of guest OSes. Infokernel [7] 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 [26] and Dtrace [27] can be used to get the OS internal states.

7. Conclusion

It is difficult to integrate a new resource management policy into an existing OS despite many proposals of sophisticated, innovative policies. This is because modern operating systems are too complex and large to modify their kernel code. In this paper, we presented FoxyTechnique, a technique of enabling us to incorporate new policies into existing OSes without any changes to their kernel. FoxyTechnique virtualizes physical devices differently from real ones, and tricks the guest OS policy into producing behavior similar to a desired policy. To embody the concept of FoxyTechnique, we conducted two case studies, FoxyVegas

and FoxyIdle, on the Xen virtual machine monitor. 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 have demonstrated that FoxyTechnique can trick various guest OS policies into producing a behavior similar to a desired policy.

References

1) Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M.: Mach: A new kernel foundation for UNIX Development, *Proc. Summer USENIX Conference*, pp.93–112 (1986).
2) Hărtig, H., Hohmuth, M., Liedtke, J., Schőnberg, S. and Wolter, J.: The Performance of μ-Kernel-Based Systems, *Proc. 16th ACM Symposium on Operating Systems Principles* (*SOSP '97*), pp.66–77 (1997).
3) Rashid, R.F. and Robertson, G.G.: Accent: A communication oriented network operating system kernel, *Proc. 8th ACM Symposium on Operating Systems Principles* (*SOSP '81*), pp.64–75 (1981).
4) Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M.E., Becker, D., Chambers, C. and Eggers, S.: Extensibility, Safety and Performance in the SPIN Operating System, *Proc. 15th ACM Symposium on Operating Systems Principles* (*SOSP '95*), pp.267–283 (1995).
5) Engler, D.R., Kaashoek, M.F. and O'Toole, J.: Exokernel: An Operating System Architecture for Application-Level Resource Management, *Proc. 15th ACM Symposium on Operating Systems Principles* (*SOSP'95*), pp.251–266 (1995).
6) Seltzer, M.I., Endo, Y., Small, C. and Smith, K.A.: Dealing With Disaster: Surviving Misbehaved Kernel Extensions, *Proc. 2nd Symposium on Operating Systems Design and Implementation* (*OSDI '96*), pp.213–227 (1996).
7) Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Burnett, N.C., Denehy, T.E., Engle, T.J., Gunawi, H.S., Nugent, J. and Popovici, F.I.: Transforming Policies into Mechanisms with Infokernel, *Proc. 19th ACM Symposium on Operating Systems Principles* (*SOSP '03*), pp.90–105 (2003).
8) Newhouse, T. and Pasquale, J.: ALPS: An Application-Level Proportional-Share Scheduler, *Proc. 15th IEEE International Symposium on High Performance Distributed Computing* (*HPDC '06*), pp.279–290 (2006).
9) Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H.: Information and Control in Gray-Box Systems, *Proc. 18th ACM Symposium on Operating Systems Principles* (*SOSP '01*), pp.43–56 (2001).
10) Goldberg., R.P.: Survey of Virtual Machine Research, *IEEE Computer Magazine*, Vol.7, No.6, pp.34–45 (1974).
11) Jones, S.T., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H.: Antfarm: Tracking Processes in a Virtual Machine Environment, *Proc. USENIX Annual Technical Conference* (*USENIX '06*) (2006).
12) Jones, S.T., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H.: Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment, *Proc. 12th ACM Architectural Support for Programming Languages and Operating Systems* (*ASPLOS '06*) (2006).
13) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and Art of Virtualization, *Proc. 19th ACM Symposium on Operating Systems Principles* (*SOSP '03*), pp.164–177 (2003).
14) Eggert, L. and Touch, J.D.: Idletime Scheduling with Preemption Intervals, *Proc. 20th ACM Symposium on Operating Systems Principles* (*SOSP '05*), pp.249–262 (2005).
15) Chen, P.M. and Noble, B.D.: When Virtual is Better than Real., *Proc. Workshop on Hot Topics in Operating Systems* (*HotOS '01*), pp.133–138 (2001).
16) Bovet, D.P. and Cesati, M.: *Understanding the LINUX KERNEL* (3rd Edition), O'Reilly Media, Inc. (2006).
17) Whitaker, A., Shaw, M. and Gribble, S.D.: Scale and Performance in the Denali Isolation Kernel, *Proc. 5th USENIX Symposium on Operating System Design and Implementation* (*OSDI '02*), pp.195–209 (2002).
18) Sugerman, J., Venkitachalam, G. and Lim, B.-H.: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, *Proc. USENIX 2001 Annual Technical Conference* (*USENIX '01*), pp.1–14 (2001).
19) Waldspurger, C.A.: Memory Resource Management in VMware ESX Server, *Proc. 5th USENIX Symposium on Operating System Design and Implementation* (*OSDI '02*), pp.181–194 (2002).
20) Brakmo, L.S., O'Malley, S.W. and Peterson, L.L.: TCP Vegas: New Techniques for Congestion Detection and Avoidance, *Proc. ACM SIGCOMM '94*, pp.24–35 (1994).
21) Shenker, S. and Wroclawski, J.: RFC2216: Network Element Service Specification Template (1997). http://rfc.net/rfc2216.html
22) Uhlig, V., LeVasseur, J., Skoglund, E. and Dannowski, U.: Towards Scalable Multiprocessor Virtual Machines, *Proc. 3rd USENIX Virtual Machine Research and Technology Symposium* (*VM '04*), pp.43–56 (2004).
23) Joshi, A., King, S.T., Dunlap, G.W. and Chen, P.M.: Detecting Past and Present Intrusions through Vulnerability-Specific Predicates, *Proc. 20th ACM Symposium on Operating Systems Principles* (*SOSP '05*), pp.91–104 (2005).
24) Bugnion, E., Devine, S. and Rosenblum, M.: Disco: Running Commodity Operating Systems on Scalable Multiprocessors, *Proc. 16th ACM Symposium on Operating Systems Principles* (*SOSP '97*), pp.143–156 (1997).
25) Vrable, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A., Voelker, G. and Savage, S.: Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm, *Proc. 20th ACM Symposium on Operating Systems Principles* (*SOSP '05*), pp.148–162 (2005).

26) Ruan, Y. and Pai, V.: Making the "Box" Transparent: System Call Performance as a First-class Result, *Proc. USENIX 2004 Annual Technical Conference* (*USENIX '04*), pp.1–14 (2004).
27) Cantrill, B., Shapiro, M.W. and Leventhal, A.H.: Dynamic Instrumentation of Production Systems, *Proc. USENIX 2004 Annual Technical Conference* (*USENIX '04*), pp.15–28 (2004).

(Received October 9, 2007)
(Accepted March 15, 2008)

Hiroshi Yamada was born in 1981. He received his B.E. and M.E. degrees from the University of Electro-communications in 2004 and 2006, respectively. He is currently a Ph.D. candidate in School of Science for Open and Environmental Systems at Keio University, since 2006. His research interests include virtual machine technology, operating systems, and system software. He is a member of IPSJ, ACM, USENIX and IEEE/CS.

Kenji Kono received the B.Sc. degree in 1993, M.Sc. degree in 1995, and Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE/CS, ACM and USENIX.