Task-aware Virtual Machine Scheduling for I/O Performance

Hwanju Kim Hyeontaek Lim Jinkyu Jeong Heeseung Jo Joowon Lee[1]

Computer Science Department, Korea Advanced Institute of Science and Technology (KAIST)

hjukim@calab.kaist.ac.kr, limh@kaist.ac.kr, {jinkyu,heesn}@calab.kaist.ac.kr, joonwon@skku.edu

Abstract

The use of virtualization is progressively accommodating diverse and unpredictable workloads as being adopted in virtual desktop and cloud computing environments. Since a virtual machine monitor lacks knowledge of each virtual machine, the unpredictableness of workloads makes resource allocation difficult. Particularly, virtual machine scheduling has a critical impact on I/O performance in cases where the virtual machine monitor is agnostic about the internal workloads of virtual machines. This paper presents a task-aware virtual machine scheduling mechanism based on inference techniques using gray-box knowledge. The proposed mechanism infers the I/O-boundness of guest-level tasks and correlates incoming events with I/O-bound tasks. With this information, we introduce *partial boosting*, which is a priority boosting mechanism with task-level granularity, so that an I/O-bound task is selectively scheduled to handle its incoming events promptly. Our technique focuses on improving the performance of I/O-bound tasks within heterogeneous workloads by lightweight mechanisms with complete CPU fairness among virtual machines. All implementation is confined to the virtualization layer based on the Xen virtual machine monitor and the credit scheduler. We evaluate our prototype in terms of I/O performance and CPU fairness over synthetic mixed workloads and realistic applications.

Categories and Subject Descriptors D.4.1 [*OPERATING SYSTEMS*]: Process Management—Scheduling

General Terms Experimentation, Performance

Keywords Virtualization, Scheduling, Inference, Virtual Machine, Xen

1. Introduction

As system virtualization has matured rapidly in terms of performance, reliability, and administration, the application of virtualization is expanding into diverse parts of the computing environment. System virtualization allows a large number of machines to be consolidated in limited physical hardware so that resource utilization and management are efficient. Due to the high performance hardware and paravirtualization techniques, the degree of machine consolidation has grown considerably. Recently, Sun and VMware introduce virtual desktop infrastructure [1, 2], which consolidates virtual desktop images in a server farm. Desktop users can access their virtual desktop through terminal devices such as a thin client or a mobile device. The extension of virtualization from server to desktop environment makes the virtual machine monitor (VMM) accommodate various and unpredictable workloads.

Since diverse workloads are feasible on consolidated virtual machines (VMs), the VMM has difficulty in resource allocation due to a semantic gap with guest operating systems. The semantic gap [9] is inevitable because different operating systems have their own sophisticated mechanisms and policies, whereas the VMM has lightweight components and narrow interfaces. This problem impedes the efficient allocation of hardware resources in terms of how much and when the resources are required by each VM. To bridge the semantic gap, VMM-level inference techniques [13, 15, 14, 16], which estimate the internals of a guest operating system by monitoring system events, have been proposed with respect to buffer cache, task tracking, hidden process detection, and so forth.

The unpredictableness of workloads makes CPU allocation hard, especially with regard to timeliness. Since the VMM lacks knowledge about existing tasks in each guest operating system, it has difficulty in considering mixed workloads on VMs. In addition, the VMM does not track which I/O event is destined for which blocked tasks. Such a semantic gap can degrade responsiveness, especially when many VMs are consolidated and their workloads are diverse. For example, when a lot of VMs contain both I/O-bound and CPU-bound workloads, they could have poor I/O responsiveness because the VMM cannot timely relate a received event with an I/O-bound task. If the VMM schedules the destined VM without considering its internal workloads, unrelated CPU-bound tasks in the VM could exhaust its allocated CPU; hence, such scheduling induces poor fairness among guest VMs. Therefore, the VMM requires guest-level task awareness and the correlation between an incoming event and an I/O-bound task.

This paper presents a task-aware VM scheduling mechanism for improving the performance of I/O-bound tasks within a VM. Our main goal is to improve the responsiveness of I/O-bound tasks selectively from CPU-bound workloads with complete CPU fairness. To identify I/O-bound tasks in mixed workloads, we use gray-box knowledge based on general characteristics for I/O-bound tasks. By observing scheduling order and CPU consumption for each task, the VMM regards a task that is preemptively scheduled in response to an I/O event and consumes short CPU time as an I/O-bound task. Furthermore, the VMM considers multiple observations to statistically reinforce the inference. Our inference technique through event monitoring and time measurement keeps the VMM lightweight.

To enhance I/O performance, we introduce partial boosting and correlation mechanisms on the basis of inferred information for I/O-bound tasks. First, partial boosting is a priority boosting mechanism with task-level granularity. Partial boosting enables the VMM to boost the priority of a virtual CPU (VCPU) that has an inferred I/O-bound task in response to an incoming event. The pri-

[1] Currently at School of ICE, Sungkyunkwan University, Korea

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE'09, March 11–13, 2009, Washington, DC, USA.
Copyright © 2009 ACM 978-1-60558-375-4/09/03. . . $5.00

ority boosting is only kept up while the inferred I/O-bound task handles the event. Therefore, partial boosting efficiently improves I/O responsiveness without compromising CPU fairness among VMs. Second, our correlation mechanisms help the VMM associate an incoming event with an I/O-bound task. Based on the correlation information, the VMM initiates partial boosting only when an incoming event is highly likely to be received by an inferred I/O-bound task. This mechanism filters ineffective partial boosting that could be conducted in response to an event for non-I/O-bound tasks. Our correlation mechanisms for block and network I/O events are best-effort with lightweight inference techniques.

We implement our task-aware VM scheduling mechanism on the credit scheduler, which is the latest scheduler of the Xen VMM. Although the scheduling-related features such as partial boosting are implemented in the credit scheduler, we present common interfaces for our mechanism to be ported to other potential schedulers. The inference mechanism for estimating I/O-boundness and the correlation mechanisms are implemented in the scheduler-independent part of Xen. Since our mechanism is confined to the virtualization layer without any modification to guest kernel, various operating systems can take advantage of the mechanism. In the evaluation section, we show the I/O performance improvement in terms of responsiveness and throughput in the worst case scenario. Furthermore, the correlation mechanisms for block and network I/O are evaluated by using synthetic workloads. Finally, we demonstrate I/O performance gain for our system on realistic workloads.

The remainder of this paper is organized as follows: Section 2 describes the Xen VMM and the credit scheduler as our implementation background. Section 3 introduces the design and operation of the proposed task-aware VM scheduler. Section 4 explains implementation details of our Xen-based prototype. Section 5 demonstrates and discusses experimental results for various workloads. Section 6 compares our mechanism with related work. Finally, Section 7 concludes our work and presents a future direction.

2. Background

This section explains the terminology, the I/O model, and the credit scheduler of the Xen VMM.

2.1 Xen Overview

Xen [4] is an open-source VMM based on a paravirtualization technique, which achieves higher performance than full virtualization approaches. The paravirtualization endeavors to minimize virtualization overheads through an interface, named *hypercall*, between a guest operating system and the VMM by modifying the guest kernel. Xen makes the privileged VM, called *domain0*, in charge of managing other guest VMs, called *domainU*.

Xen introduces the *isolated driver domain* (IDD), which conducts real I/O operations to a bare device on behalf of domainUs, for reliable I/O architecture [8]. This I/O model enhances the reliability of an entire system by isolating the faults induced by device drivers in an IDD. Moreover, an IDD can operate existing device drivers and multiplexing software such as a network bridge. This I/O model requires guest domains to use virtual device drivers for transparent I/O access. A virtual frontend driver in a domainU communicates with a corresponding virtual backend driver, which resides in an IDD and forwards delivered I/O requests to a native device driver.

A frontend driver and a backend driver notify each other of an I/O event through an *event channel*. The event channel mechanism virtualizes a hardware interrupt. A virtual interrupt is pending in the corresponding event channel and then is delivered into the target domain when the domain is scheduled. The latency between pending and delivered events obviously depends on the underlying VM scheduling mechanism.

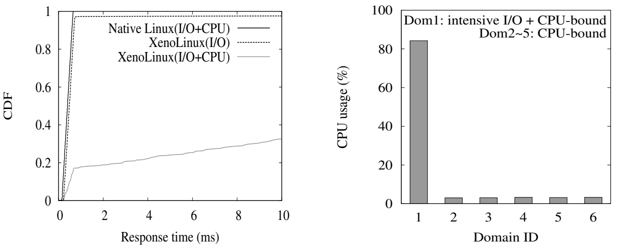
Xen allocates one or more VCPUs to a domain when the domain is created. A VCPU contains general information related to scheduling and event channels because the VCPU is a scheduling entity. The event channel information in a VCPU is shared by its owner domain and the VMM through a shared page for efficient interrupt handling. When a VCPU is scheduled, the paravirtualized event driver checks whether its event channel has a pending event. If so, the driver invokes the corresponding interrupt handler routine. In this manner, a physical interrupt that is delivered in the VMM is pending in the event channel of an IDD, and the IDD processes I/O and sends a virtual interrupt to the event channel of the target domain by using hypercall.

Xen currently enables the domain0 to choose a scheduler among two schedulers: the simple earliest deadline first (SEDF) scheduler and the credit scheduler. The SEDF scheduler makes each domainU specify a required time slice in a certain period; a (slice, period) pair represents how much CPU time a domain is guaranteed in a period. The SEDF scheduler preferentially schedules a domain with the earliest deadline. This scheduler achieves satisfactory quality on the environment where a domain includes decidable or predictable workloads such as a streaming server with a constant bit rate. The credit scheduler is a proportional share scheduler with a load balancing feature for SMP systems. The virtue of the credit scheduler is the simplicity of the operation with reasonable fairness guarantee and performance, whereas the SEDF scheduler requires a well-tuned parameter configuration.

2.2 Credit Scheduler

The credit scheduler regards a time quantum as *credit*, which is determined by the defined *weight* for each domain. The credit of a running VCPU is debited by 100 every tick period (10 ms); all active VCPUs are given the credit calculated by the weight of their domain every credit period (30 ms). The credit of a VCPU is used to determine its priority once a credit period. If a VCPU has remaining credit (credit > 0), its priority is `UNDER` (-1). Otherwise a VCPU is given `OVER` (-2) priority, which means the VCPU has consumed more than its allocated credit. All VCPUs with the priority of `UNDER` are scheduled before those with the priority of `OVER`; a run queue maintains VCPUs with `UNDER` followed by those with `OVER`, and the scheduler picks a VCPU at the head of the run queue as a next VCPU. Once a VCPU is scheduled, it receives the time slice of 30 ms and runs consuming its credit. When the time slice of a running VCPU expires, the VCPU is descheduled and is put into the tail of a list that contains VCPUs with the same priority as the descheduled VCPU. If a running VCPU does not have any runnable task in spite of time remaining in its time slice, it is blocked and leaves the run queue.

The credit scheduler allows a VCPU to preempt a running one to improve the performance of I/O-bound domains by using a boosting mechanism. If a VCPU has only I/O-bound tasks, it is usually blocked with slight credit consumption. When an event is pending to the blocked VCPU, the VMM wakes up the VCPU and inserts it into the run queue. Since the VCPU waits until the preceding VCPUs are descheduled, the event delivery can be delayed. To achieve low latency, the credit scheduler boosts the priority of a woken VCPU if its priority is `UNDER`—the VCPU has been blocked with remaining credit. The boosting mechanism assigns the priority of `BOOST` (0), the highest priority, to the woken VCPU and allows it to preempt a running VCPU. The VCPU of an I/O-bound domain usually retains `UNDER` priority because such a VCPU typically consumes much less than a tick period. Therefore, I/O-bound domains frequently preempt a running domain and thus achieve the improved responsiveness and throughput [20].



(a) Response time for three cases

(b) CPU usage for aggressive boosting

Figure 1. Necessity of task-awareness

3. Task-aware VM Scheduling

Although a VCPU-level scheduling mechanism is quite simple and supports fairness well, it has limitations due to the semantic gap. Once the VMM allocates a physical CPU to a VCPU, the VMM fully relies on the guest kernel scheduler on the VCPU during the time slice. Hence, the VCPU-level scheduler does not track the internal tasks of a VCPU. In spite of the simplicity, the lack of knowledge about the guest-level workloads could lead to I/O performance degradation, especially for timeliness. An I/O-bound task that is mixed with heterogeneous workloads cannot reveal its I/O-boundness to the VMM because the characteristic of each individual task is unrecognized at the VMM. For example, when an event is pending to an idle VCPU, which has no runnable task in its previous time slice, the credit scheduler preferentially schedules the VCPU by the boosting mechanism. However, if the VCPU is not idle, the credit scheduler does not boost the VCPU even though a corresponding event is pending. An I/O-bound task running in the non-idle VCPU does not take advantage of the boosting mechanism and consequently has low responsiveness.

Figure 1(a) shows the effect on the response time of an interactive workload over the credit scheduler in accordance with the workloads of a domain. We simply measure the response time while a client in a separate machine repeatedly requests a small network packet to a server in a domain that is consolidated with five CPU-bound domains. As shown in the graph, the server with a CPU-bound task never benefits from the boosting mechanism and consequently suffers from low responsiveness. The shape of the CDF graph in this case is totally different from that of a native Linux for the same workload. The response time is presumed to be degraded as the physical machine consolidates more domains, most of which consume CPU time. The server without a CPU-bound task, on the other hand, almost preempts a running domain without waiting for the other domains with the aid of the boosting mechanism; the improved response time is close to that of the native Linux.

In the VCPU-level scheduler, there is a critical trade-off between responsiveness and fairness. As a naive approach for better responsiveness, we consider aggressive boosting. If the VMM aggressively boosts a VCPU without considering internal workloads whenever a corresponding event arrives, the CPU fairness could be compromised, whereas the responsiveness is improved. In Figure 1(b), we show the worst case result of the aggressive boosting. Domain1 runs a network-intensive workload with a CPU-bound task, and the other five domains have CPU-bound workloads. Whenever an incoming packet is pending to domain1, the aggressive boosting mechanism preemptively schedules the VCPU of domain1 regardless of its priority and state. Since the VMM guarantees a time slice to the scheduled VCPU as long as the VCPU has runnable tasks, the CPU-bound task in domain1 exhausts the given time slice after the incoming packet is handled. Therefore, the intensive I/O of domain1 makes the other domains starve while significantly compromising fairness.

To achieve both low I/O latency and fairness of CPU allocation, the VMM needs to elaborate the boosting mechanism with knowledge about the characteristics of guest-level tasks. Our main goal is to boost a VCPU when a pending event is destined for an I/O-bound task in the VCPU while guaranteeing overall CPU fairness. This section describes tracking I/O-bound tasks, partial boosting, and correlation mechanisms.

3.1 Tracking I/O-bound Tasks

To distinguish I/O-bound tasks within the mixed workloads, the VMM should track tasks in domains at the virtualization layer. As an alternative approach, a guest kernel scheduler can cooperatively inform the VMM of the information about I/O-bound tasks. This approach, however, requires the modification of the guest kernel and assumes that all domains are trusted. To pursue a non-intrusive approach, we use the previously proposed method to track tasks at the virtualization layer by monitoring the access to the MMU hardware [14]. In MMU-enabled operating systems, a task has a private virtual address space provided by the paging facility of the MMU in the protected mode. A guest operating system should access the MMU when switching tasks by its scheduler. The VMM can capture the task switching event because the VMM virtualizes the MMU hardware.

The VMM uses gray-box knowledge to infer the I/O-boundness of guest-level tasks by observing low-level interactions between the guest kernel and hardware. The VMM controls I/O operations through event channels and monitors how tasks are scheduled by a kernel scheduler. Based on the information acquired by monitoring such events, the following general gray-box criteria can be assumed.

1. **The kernel policy for I/O-bound tasks**: A priority-based preemptive scheduler, which is prevalent in commodity operating systems, preferentially schedules an I/O-bound task when a corresponding I/O event occurs for low latency [5, 21, 19].
2. **The characteristic of I/O-bound tasks**: An I/O-bound task typically consumes little CPU time, since its execution time is dominated by the wait time for an I/O event [18].

The first inference relies on the kernel policy by regarding a task that is preemptively scheduled in response to an event as an I/O-bound task. In order to firmly characterize its I/O-boundness, the VMM also considers the CPU consumption of the inferred I/O-bound task based on the second criterion. The short CPU consumption of an I/O-bound task is a crucial characteristic to overcome the trade-off between responsiveness and fairness. A task with the two characteristics can selectively achieve high responsiveness in its VCPU without compromising overall CPU fairness among VCPUs by partial boosting, which is detailed in next section.

By checking the two criteria, we classify observations of scheduling events into three disjoint classes: *positive evidence*, *negative evidence*, and *ambiguity*. The observation of a task is positive evidence if the observation supports the task being I/O-bound. If the observation indicates that the task is not I/O-bound, it belongs to the negative evidence class. Ambiguity means that the observation cannot help the VMM infer I/O-boundness. Figure 2 shows an example of task scheduling during the time slice of a VCPU after an event is pending. We define *IOthreshold* to determine the short CPU consumption; 0.5 ms is used in this example. After the VCPU with a pending event is scheduled, `T2` immediately preempts `T1` and runs for less CPU time than *IOthreshold*. Since multiple tasks could wait for the event, we also consider `T3`, which is consecutively scheduled after `T2` with short CPU consumption. Hence, the observations of `T2` and `T3` are positive evidence. On the other hand,

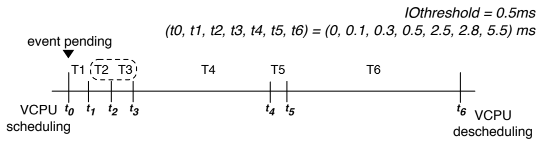


Figure 2. Inference for I/O-bound tasks

T4 and T6 have negative evidence because they satisfy neither of our two criteria. We regard the observation of T1 as ambiguity in spite of short CPU consumption, since the CPU time is likely from the immediate preemption of T2. T5 has short CPU consumption, but is scheduled after a task with long CPU time. Due to the case where an I/O-bound task has temporarily low priority in this time slice, we also regard the case of T5 as ambiguity.

Then, the VMM considers multiple observations for reliable inference. Although the above gray-box knowledge explains most activities of the task scheduler of a guest kernel, there are anomalies that violate the gray-box knowledge. For example, an I/O-bound task may show an exceptionally long CPU time when the operating system interrupts the execution of the task and processes internal data without switching the virtual address space; in the case of Linux, a *kernel thread* uses the address space of the previously descheduled task to avoid address space switching. On the other hand, a CPU-bound task may have a short CPU time when the task is preempted by the scheduling policy of its kernel. To our observation, these anomalies are rare but not negligible. We therefore relieve this uncertainty by adopting a statistical approach.

The VMM maintains the *degree of belief* on the I/O-boundness for each task. The degree of belief for a task is a variable that represents how certain the VMM is that the task is I/O-bound. The degree of belief for a task is initially zero, which means the VMM has no bias for the I/O-boundness of the task. Every time the VMM observes positive evidence, the VMM adds *PositiveEv* to the degree of belief of the descheduled task. For the negative evidence, the VMM subtracts *NegativeEv* from the degree of belief. We simply ignore ambiguity because it does not help determining I/O-boundness. The VMM assumes that a task is I/O-bound only if its degree of belief is larger than *BelThreshold*. Finally, we restrict the degree of belief for each task to be in a certain range in order to allow the VMM to quickly adapt the degree of belief to the current I/O characteristic of the task.

The degree of belief and the evidence are concepts of statistical inference techniques such as a Bayesian inference. Additive evidence can be represented as log odds, which is the weight of evidence [10]. For more intelligent inference, the VMM can dynamically change PositiveEv and NegativeEv by learning workloads on a VM. The learning technique, however, is somewhat expensive for the VMM, which is a performance-critical system. More importantly, the VMM cannot exactly identify the operation of a guest kernel, and gray-box knowledge has limitations for intelligent learning. Therefore, we use static parameters in our current implementation and evaluation. The use of dynamic parameters by learning efficiently and static parameters by deciding empirically are challenging problems to enhance the task-awareness of the VMM.

3.2 Partial Boosting

Based on inferred information for I/O-bound tasks, we devise a partial boosting mechanism to improve I/O responsiveness while keeping CPU fairness. As described in Section 3, an aggressive boosting could compromise the fairness. To improve I/O responsiveness with fair CPU allocation, we want only I/O-bound tasks to preempt a running VCPU in response to an incoming event for immediate I/O processing and yield CPU to another VCPU. When an event is pending for a VCPU, the VMM initiates partial boosting for the VCPU regardless of its priority if the VCPU has at least one inferred I/O-bound task. A partially boosted VCPU can preempt a running VCPU and handle the pending event. The VMM revokes CPU from the partially boosted VCPU when the guest operating system schedules a task that is not inferred as I/O-bound. The priority of the descheduled VCPU is reassigned by the original policy of the scheduler, and then the VCPU is inserted into the run queue based on the returned priority.

When an I/O-bound task that is mixed with CPU-bound tasks intensively conducts I/O operations, its VCPU is partially boosted very frequently. Since partial boosting is conducted regardless of the state and priority of a VCPU, unrestricted partial boosting causes excessive preemptions while repeatedly interrupting other VCPUs. Moreover, a frequently boosted VCPU can greedily use shared I/O resources. To relieve this problem, we define PBratio to restrict uncontrolled partial boosting. PBratio is partial boosting allowance, which is maintained for each VCPU and is defined as:

$$\texttt{PBratio} = \frac{Allowed\,CPU\ usage\ for\ partial\ boosting}{Total\,CPU\ usage}$$

PBratio means the CPU fraction that is allowed for partial boosting in the total CPU usage of a VCPU. Both total CPU usage and the usage by partial boosting are periodically reset to zero in order to consider a recent tendency. With low PBratio, only an interactive application, which is not I/O-intensive, has the high quality of responsiveness by partial boosting. On the other hand, high PBratio makes an I/O-intensive task achieve high throughput although its domain uses more I/O resources. If PBratio is zero, our scheduler runs in the same as the original scheduling mechanism. The evaluation of PBratio is presented in Section 5.1.

Although ideal partial boosting lasts a short time near IOthreshold, there are some cases where the duration of partial boosting is prolonged. First, partial boosting can occur in response to an I/O event that is handled in kernel only and is not delivered to any task. For example, an ARP request packet is handled in the kernel and does not wake up any task. In this case, partial boosting is prolonged until the boosted VCPU schedules a non-I/O-bound task or exhausts its time slice. For the worst case, a CPU-bound task uses up the entire time slice of the boosted VCPU. Second, an inferred I/O-bound task may start consuming CPU right after partial boosting. The effect of such varying workloads can be relieved by assigning a relatively larger value to NegativeEv than PositiveEv. Furthermore, the VMM can forcibly revoke CPU from a VCPU that keeps partial boosting for some time. This mechanism, however, can incur overheads for managing an individual timer for each partial boosting. Without the need of maintaining additional timers, we make the VMM restrict the duration of one partial boosting at a tick granularity. Moreover, the prolonged partial boosting can be significantly alleviated by our correlation mechanism described in the next subsection.

3.3 Correlation Mechanisms

Partial boosting based on only the I/O-boundness of tasks has the limitations due to the lack of correlation between an event and a task. Partial boosting could be initiated in response to an event that is destined for a non-I/O-bound task without the correlation information. Since the partial boosting mechanism revokes the CPU from a boosted VCPU as soon as the non-I/O-bound task is scheduled, such partial boosting is meaningless while incurring unnecessary preemption. Similarly, an event to be handled by the kernel only may cause useless prolonged partial boosting. The correlation mechanism therefore is essential for effective partial boosting in that the VMM partially boosts a VCPU only if an I/O-bound task in the VCPU is likely to receive an incoming event. We devise correlation mechanisms for two representative I/O devices: a block device and a network device. We consider only block read and network reception events, to which users are latency-sensitive. The main objective of our correlation mechanisms is to determine whether a pending event is destined for an I/O-bound task. The correlation mechanism addresses event identification, correlation, and accuracy issues.

3.3.1 Block I/O

The correlation for block I/O is relatively simple in that the event of block read completion is paired with its request event.

Event identification. In the case of block read I/O, a guest kernel explicitly sends a block read request to a block device driver. The device driver then requests a DMA operation to a block device. When the requested block is transferred to the memory via DMA, the block device generates an interrupt that notifies the kernel of an I/O completion. Due to the request-response procedure of block read I/O, a read I/O completion event can be identified by a requested block number.

Correlation. As a simple method, the VMM correlates a requested block I/O with the task running at the request time. Accurate correlation, however, is challenging because an actual block request can be delayed from a user request by the status of a request queue and the policy of a kernel I/O scheduler. Jones [13] proposes a more accurate correlation than the simple method by exploiting that operating systems typically copy contents in the buffer cache into a user buffer. In spite of better correlation, this technique incurs overheads for maintaining inverse memory mapping and handling intentional page faults.

In our mechanism, we are interested in whether a block read I/O is requested from an I/O-bound task. In order to consider a delayed block request, the VMM inspects not only a current task, but also previously scheduled tasks at a request time. The VMM regards a block request as sent by an I/O-bound task if at least one inferred I/O-bound task is inside *inspection window* at the request time.

For example, an I/O-bound task `T1` and a non-I/O-bound task `T2` request the 100th block and the 200th block, respectively, and the inspection window size is two. The two requests are inserted in the request queue of a block device driver. If the block device driver handles these requests when `T2` is running, the VMM inspects `T1` and `T2` within the inspection window. Since `T1` is an I/O-bound task, the requests for the 100th and the 200th blocks are considered as sent by an I/O-bound task. When a read completion event for the 100th block is pending, the VMM partially boosts the corresponding VCPU so that `T1` promptly handles the event.

Accuracy issues. This window-based correlation is a best-effort approach because it could remain some false positive partial boosting. When a read completion event for the 200th block is pending, the VMM also boosts this VCPU even though `T2`, which is supposed to receive the pending event, is not I/O-bound. Such false partial boosting, however, rarely occurs, since a task that is inferred as non-I/O-bound is unlikely to conduct I/O requests frequently. In the case of the Xen I/O model, furthermore, a batch of I/O requests from a guest domain alleviates the false positive partial boosting because an IDD also batches some responses for simultaneously requested I/O to improve throughput.

3.3.2 Network I/O

The correlation for network I/O is more complicated than that for block I/O because a network packet arrives asynchronously, whereas a block operation is only conducted in response to an explicit request from the kernel. Due to this characteristic, the VMM correlates the event of an incoming packet with a task through a posterior correlating method.

Event identification. The VMM identifies an incoming packet for correlation as it identifies a block read completion with the requested block number. Operating systems commonly use *socket* abstraction to map a network packet to a task for TCP/IP networking. A socket is identified by four-tuple (source IP address, source port number, destination IP address, and destination port number) for connection-oriented protocols such as TCP, or by two-tuple (destination IP address and destination port number) for connectionless protocols such as UDP. To identify an incoming packet exactly, the VMM should also maintain the tuples to correlate an incoming packet with a recipient task. The VMM, however, may have high overheads of memory space and processing time to maintain socket-like information, especially when a number of network connections are established. For a lightweight correlation mechanism, we consider only a destination port number as an identification clue of an incoming packet because it is the most specific information related to a recipient task.

Correlation. For the posterior correlation, we use a prediction mechanism by monitoring which task is woken up after the delivery of an incoming packet. As stated in Section 3.1, we anticipate that an incoming packet is delivered to the first woken task if this task is I/O-bound. By this anticipation, if the first woken task is an inferred I/O-bound task, the VMM regards the incoming packet is for I/O-bound. To elaborate the prediction, we use history-based approach as with the branch prediction scheme [22]. The VMM uses a *portmap*, each entry of which maintains the correlation history for each destination port number; each entry is an N-bit saturating counter, named *portmap counter*. If an incoming packet for a destination port number makes the kernel wake up an inferred I/O-bound task, the corresponding portmap counter is incremented. Otherwise, the counter is decremented. When a packet is pending to a VCPU, the VMM partially boosts the VCPU if the most significant bit of the corresponding portmap counter is set.

Accuracy issues. Since correlation accuracy depends on the amount of history, a suitable bit-width should be chosen with the consideration of space overheads; in the case of an N-bit counter, the VMM stores 2^N prediction history for each port number. Although a 1-bit counter takes up minimal space, it is vulnerable to miss correlation. In Section 5.2, we show a 2-bit counter is reasonable for both accuracy and space requirement.

A multiple bit counter has another effect to alleviate miss correlation in case where multiple tasks use one port number, for example, a multitasking TCP server. As described above, only a destination port number is regarded as a correlation unit, the VMM cannot distinguish each connection for multiple tasks using one port number. For the 1-bit counter, the newly created task using the server port can invalidate the previously established correlation because the new task is not regarded as I/O-bound. A multiple bit, on the other hand, retains the established correlation as long as request packets for the same port number reach I/O-bound tasks.

When a domain receives multiple packets for different port numbers at once, the VMM confuses which port number is related with the first woken task. To cope with this uncertainty, the VMM updates portmap only if all incoming packets are destined for one port number before the time slice of the target VCPU. Although this approach could defer partial boosting when many packets for different port numbers simultaneously reach, more precise correlation is achieved.

4. Implementation

This section describes the implementation of our scheduling and inference mechanisms in the Xen VMM. Xen uses a common interface for schedulers so that different schedulers are easily developed and adopted. We implement our scheduling-related mechanisms over the credit scheduler of Xen-3.2.1 through a *task-aware operation* interface. Our current implementation is based on a single physical core while assuming a guest domain has a VCPU. We therefore do not consider task migration between VCPUs and synchronization issues.

4.1 Task Information Management

For the management of task information, each VCPU maintains a *task hash*, which contains `task_info` structure. This structure stores a task ID, a timestamp, and the degree of belief. The task ID is the CR3 of a task; in x86, a CR3 indicates the page directory of a virtual address space. The timestamp is the last time when the task is inferred as an I/O-bound task. The timestamp is used for Xen to reclaim `task_info` of a task that has not accessed I/O during a certain period. We do not accurately track the termination of a task, which is addressed in Antfarm [14], and therefore rely on the periodic reclaiming for tasks that conduct no I/O for some time. For efficiency, `task_info` is preallocated and is managed in a pool with a bitmap-based allocator.

4.2 Partial Boosting

Partial boosting is conducted by a task-aware operation interface to the credit scheduler. When an event is pending to a VCPU, the VMM checks whether the VCPU is placed in the run queue and has at least one inferred I/O-bound task. If so, the VMM determines on partial boosting based on the correlation information. Partial boosting is implemented by assigning `BOOST` priority to the VCPU regardless of its current priority and by reinserting the VCPU in the run queue. Since VCPUs with the same priority are scheduled in a round-robin manner, a boosted VCPU should wait for already boosted ones. The credit of a partially boosted VCPU is maintained in the same way that of an ordinary VCPU is debited.

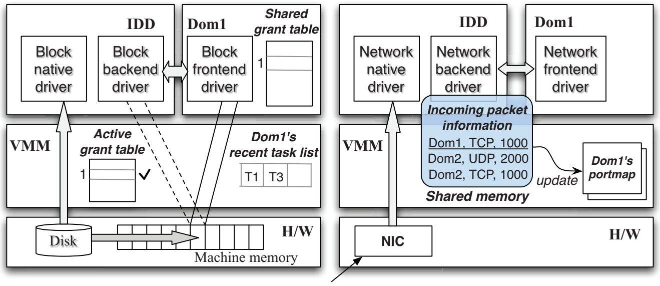
Since the credit scheduler debits the credit from a running VCPU at a tick time, the CPU consumption of a VCPU cannot be reflected to its credit if the consumed time is less than a tick granularity. Partial boosting is expected to last short-term less than a tick and thus can steal the credit of another VCPU. In the original coarse-grain accounting of the credit scheduler, therefore, excessive partial boosting affects fine-grain CPU fairness. To complement such a problem, we make the VMM account actually consumed CPU time to each VCPU. The actual CPU time is acquired by *time stamp counter* (TSC) of IA-32; the TSC has the high resolution that is same as CPU clock frequency. For example, if a partially boosted VCPU consumes 1 ms and then is descheduled, its credit is debited by 10, which is calculated as $1ms \times (100/10ms)$.

4.3 Correlation Mechanism

As described in Section 2.1, Xen shifts the management of hardware devices to an IDD instead of direct management of the VMM. The implementation of correlation therefore relies on the operation of an IDD.

4.3.1 Block I/O

Xen enables an IDD to temporarily map the foreign memory of domainU in order to avoid a memory copying overhead for block I/O. Since an IDD conducts real DMA requests on behalf of a domainU, the IDD requires the access privilege for the memory of the domainU. A domainU should permit its buffer cache memory to be mapped by an IDD before requesting block I/O. This mapping privilege is managed by using *shared grant table* whose entry contains a permitted domain, the address of buffer cache, and a status flag. The shared grant table is governed by the frontend driver of a domainU. The VMM tracks the current status of the mapping by using *active grant table*, which shadows the shared grant table and is only accessible in the VMM. When a block I/O is finished, an IDD releases the foreign mapping, and then the VMM frees the corresponding entry in the active grant table. The correlation mechanism for block I/O uses the active grant table without additional data structure.



(a) Block I/O

(b) Network I/O

Figure 3. Correlation mechanism in Xen I/O architecture

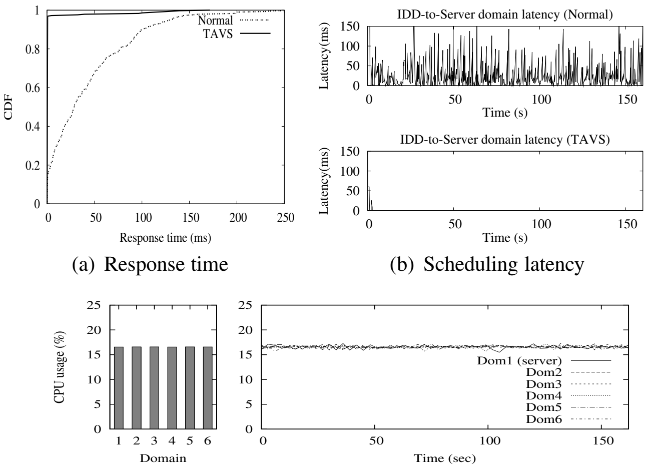
Figure 3(a) shows the operation of block I/O and our correlation mechanism. Domain1 requests a block read operation to an IDD. Prior to the request, domain1 specifies that its buffer cache page is allowed to be mapped by an IDD in an entry of its shared grant table; the entry is identified by a *grant reference*, which is an index of the table, and entry 1 is used in this example. The IDD first maps the permitted memory to its address space through hypercall. The VMM creates the requested mapping and updates the active grant table after checking permission. Since the entry of the grant table represents a block I/O request, the VMM correlates the entry with a task group, which consists of tasks scheduled in the inspection window before requesting block I/O. This figure shows that entry 1 is related with `T1` and `T3`. At grant mapping time, if either `T1` or `T3` is an I/O-bound task, entry 1 is marked as I/O-bound. When the IDD unmaps the mapping in response to the I/O completion, partial boosting is initiated if the unmapped entry is for an I/O-bound task.

4.3.2 Network I/O

The correlation for network I/O is assisted by an IDD because packet information can be readily extracted by a network backend driver, called *netback*. An incoming packet from outside is delivered to the netback driver through the native driver and multiplexing software such as a bridge. When the netback driver forwards the packet, it checks whether the packet is for TCP/IP. If so, the driver records the protocol and the port number of the packet and the destination domain ID in the memory shared by the VMM. The packet header inspection incurs negligible overheads because only a few memory accesses are required with some offset calculations (15 lines of source code). Furthermore, the memory access does not affect a hardware cache because the VMM copies the received packets to recipient domains right after the inspection. Each VCPU requires N × 8 KB memory for a portmap with N-bit counter; a TCP/IP packet has a 16-bit destination port number.

Figure 3(b) briefly shows the correlation for network I/O. When the VMM deschedules an IDD, it inspects incoming packets that are arrived during the previous time slice of the IDD. For low overheads, the limited amount of the port information is recorded; an IDD records up to 16 entries for one time slice in our current implementation. In this example, the portmap of domain1 is updated based on the first woken task of domain1 because a packet for one destination port number reaches during the time slice of the IDD.

The IDD-based I/O model of Xen may impede our inference that an incoming packet is likely delivered to the first woken task. Since an IDD is designated to perform I/O only, the IDD often preempts a requesting domain. The preemption of IDD could make a requesting I/O-bound task remain as a current task, which becomes the first task in the next time slice of its VCPU. In this case, if an in-



(a) Response time

(b) Scheduling latency

(c) CPU usage

Figure 4. Performance and fairness guarantee for simple interactive workload

coming packet for the I/O-bound task is pending, the packet will be delivered to the first task. Then, the portmap counter is incorrectly decremented if the first woken task is an unrelated non-I/O-bound task. To address the incorrect updating of a portmap, the portmap counter is decremented if neither the first task nor the first woken task is I/O-bound.

4.4 Fairness Enhancement

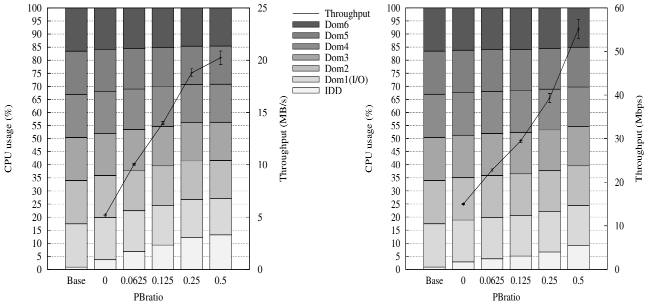
We modify the credit scheduler to show better fairness. As explained in Section 2.2, the credit scheduler puts a descheduled VCPU into the tail of its priority list. Since the I/O-intensive activity of a domain makes a boosted IDD frequently preempt the domain, such a domain is often inserted to the tail of its priority list with much less CPU consumption than other CPU-bound domains. This problem could compromise overall fairness and significantly degrade CPU-bound tasks mixed with an I/O-intensive task. For better fairness, in case where a VCPU is preempted by a boosted IDD, we make the credit scheduler locate the descheduled VCPU on the head of its priority list.

5. Evaluation

Our prototype is installed on a 3.00 GHz Intel Pentium D CPU, equipped with 2 GB RAM. We make our system run on a single physical core. A network client runs on a separate physical machine, an Intel Pentium 4 processor 2.60 GHz with 1.5 GB RAM; this machine is connected to the evaluated machine through an 100 Mbps Ethernet switch. In our evaluation, we assign 5, 20, and 20 to PositiveEv, NegativeEv, and BelThreshold, respectively. The negative evidence is regarded as a penalty and thus has higher weight than the positive evidence. In addition, we limit the degree of belief by the minimum of -100 and by the maximum of 300. IOthreshold is empirically determined as 0.5 ms.

5.1 I/O Performance

We demonstrate the improvement of I/O performance on a consolidated machine with our mechanism in terms of responsiveness and throughput. To show the improvement for the worst case consolidation scenario, we concurrently run five CPU-bound domains with one domain to be evaluated. The evaluated domain contains both I/O-bound and CPU-bound workloads so that the original scheduler does not identify the I/O-bound task. We use the domain0 as an IDD. Our mechanism is referred to as TAVS (task-aware VM scheduling) in all figures.



(a) Block

(b) Network

Figure 5. I/O throughput for different PBratios

Avg. response time (ms)	CPU + I/O			I/O		
	Dom1	Dom2	Dom3	Dom4	Dom5	Dom6
Normal	69.44	74.75	74.13	3.75	4.56	5.07
TAVS	5.09	5.69	5.67	4.95	4.95	3.76

Table 1. Average response time with different workloads

Figure 4(a) shows the response time of a simple interactive workload. One domain runs a TCP echo server with a CPU-bound task, and a remote client repeatedly requests a small network packet (40 byte) to this server with random think time, which is between 100 ms and 1000 ms in this experiment. As shown in the CDF graph, our mechanism significantly improves the response time by partial boosting compared to the normal case. Figure 4(b) shows the scheduling latency from an IDD to the server domain for delivering an incoming packet. The result of the normal case shows that the server domain has up to about 150 ms as maximum latency; this latency is resulted from the number of CPU-bound domains (5) × a maximum time slice (30 ms). In our mechanism, the latency is close to zero by partial boosting except for the initial inferring period. Figure 4(c) shows CPU usage for each domain during the experiment. This result demonstrates that our mechanism guarantees the CPU fairness for an interactive workload.

We demonstrate the throughput of block read and network as well as CPU usage for each domain according to different PBratios in Figure 5. The base case shows the reference data, which is acquired by letting all domains be CPU-bound. We use *SysBench* and *Iperf* to measure the throughput of disk and network, respectively; we measure the disk throughput by sequentially reading 8192 files, each of which has the size of 128 KB and the network throughput by having a remote client transmit the data of 512 MB over TCP connection. All results are averaged over five runs.

Figure 5 shows the throughput of block read and network is improved as more partial boosting is allowed. In addition, CPU fairness among guest domains is guaranteed for all cases. Instead, the CPU usage of the IDD increases as the I/O throughput is improved because an actual I/O operation is processed in the IDD. The credit scheduler allows an I/O-intensive domain to use its corresponding IDD in a work-conserving manner and accounts the CPU usage of the IDD by processing I/O to the IDD itself, not a requesting domain. In fact, the CPU usage on behalf of guest domains should be distributed into each requesting domain to enhance performance isolation. If an accurate accounting method such as of *SEDF-DC* in [12] is implemented in our prototype, PBratio will be a useful parameter to control the use of the IDD. The accurate accounting method is beyond the scope of this paper and is remained as future work.

We evaluate our system in case where multiple domains have different workloads, which consist of three mixed domains (CPU- and I/O-bound), three I/O-bound domains, and three CPU-bound domains. Six clients conduct requests and responses with the think time between 10 ms and 1000 ms. Table 1 shows that the domains including CPU- and I/O-bound tasks have much lower responsiveness than I/O-bound domains in the normal case. Our mechanism substantially improves the poor responsiveness of the mixed domains nearly as good as that of I/O-bound domains.

5.2 Correlation

This section presents the evaluation of our correlation mechanisms for block and network I/O. We evaluate correlation and I/O performance as changing the inspection window size and the bit-width of a portmap counter. As the metric of the correlation, *a partial boosting hit ratio* (PBHR) is measured by using TSC. PBHR is defined as:

$$\texttt{PBHR (\%)} = \frac{\sum h}{The\ number\ of\ partial\ boostings} \times 100$$

where

$$h = \begin{cases} 1 & \text{, if an I/O-bound task awakes during partial boosting.} \\ 0 & \text{, otherwise.} \end{cases}$$

We instrument our benchmarks to record a timestamp in memory whenever an I/O-bound task awakes from blocking I/O; in this experiment, a disk read program records a timestamp right after *open* and *read* system calls, and a UDP server records a timestamp right after *recvfrom* system call. Since TCP requires kernel-level instrumentation due to control packets such as *ack*, we use UDP to simply measure PBHR. Xen also records a timestamp at the start and end of partial boosting. We run five CPU-bound domains with an evaluated one.

One domain generates synthetic workloads, which are running multiple tasks with different CPU consumption between I/O operations. An I/O-bound task intensively performs I/O without CPU consumption. The others conduct I/O with CPU consumption greater than IOthreshold. In this experiment, one domain runs eight tasks with different CPU consumptions: 0 ms, 1 ms, 2 ms, 5 ms, 10 ms, 30 ms, 100 ms, and 300 ms; a task with 0 ms is an I/O-bound task. We measure PBHR and the performance of the I/O-bound task with the PBratio of 0.125. All averaged results are the 10% trimmed mean of ten runs. In addition, the figures provide PBHR and performance in the case of no correlation, named NC; no correlation means the VMM partially boosts a guest domain that includes at least one I/O-bound task whenever an event is pending to this domain.

Figure 6 shows PBHR and the throughput of the block I/O-bound task for different inspection window sizes. As stated in Section 3.3.1, the inspection window enables our scheduler to consider the I/O-bound tasks of which block requests are delayed by the guest kernel. As the window size increases, therefore, false negative partial boosting is reduced; that is, an I/O-bound task benefits from more partial boosting and achieves higher performance. On the other side, the larger window size is, the higher false positive ratio. In Figure 6, PBHR of the I/O-bound task decreases as the window size increases; false positive ratio is equal to (100 - PBHR) %. Instead, the larger window size achieves the better throughput of the I/O-bound task, since its delayed requests are compensated for partial boosting. Because partial boosting is restrictively allowed by PBratio, high false positive ratio rather reduces the partial boosting chance of the I/O-bound task (See the decline of throughput for window sizes between five and eight).

To evaluate network I/O correlation, we use the simple interactive workload, which is used in Subsection 5.1; however, random think time is between 10 ms and 1000 ms to increase intensity. The eight UDP echo servers with the same configuration for the CPU consumption of the block I/O evaluation individually serve the eight clients. Figure 7 shows the response time and PBHR for the different bit-widths of a portmap counter. In the case of 1-bit counter, PBHR of 64% shows its weakness from miss correlation and relatively low responsiveness. On the other side, 2-bit and 4-bit counters achieve PBHR of about 90% with the aid of the correlation history. Even though PBHR of 4-bit counter is a slightly higher than that of 2-bit counter, their response times are almost same. This result demonstrates 2-bit counter is the best choice for reasonable performance and memory overheads. Although no correlation shows reasonable responsiveness, its PBHR is very low resulted from exhaustive partial boosting, which is inefficient due to unproductive domain switches.

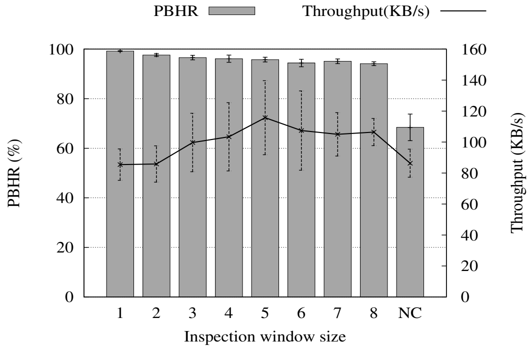


Figure 6. PBHR and throughput of a block read I/O-bound task for different inspection window sizes

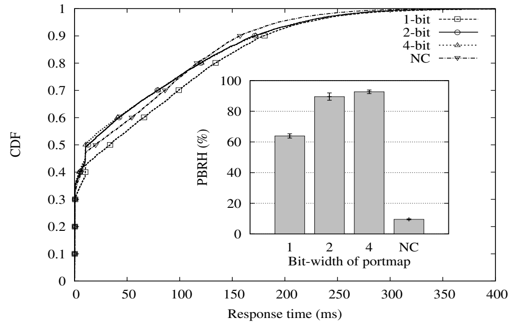
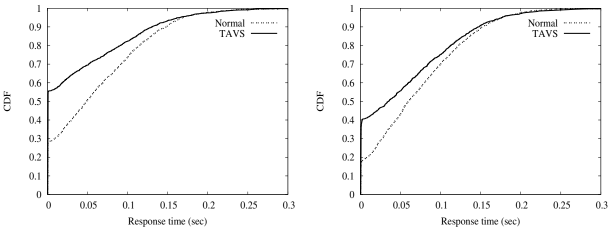


Figure 7. PBHR and response time of a network I/O-bound task for the different bit-widths of a portmap counter

5.3 Realistic Workload

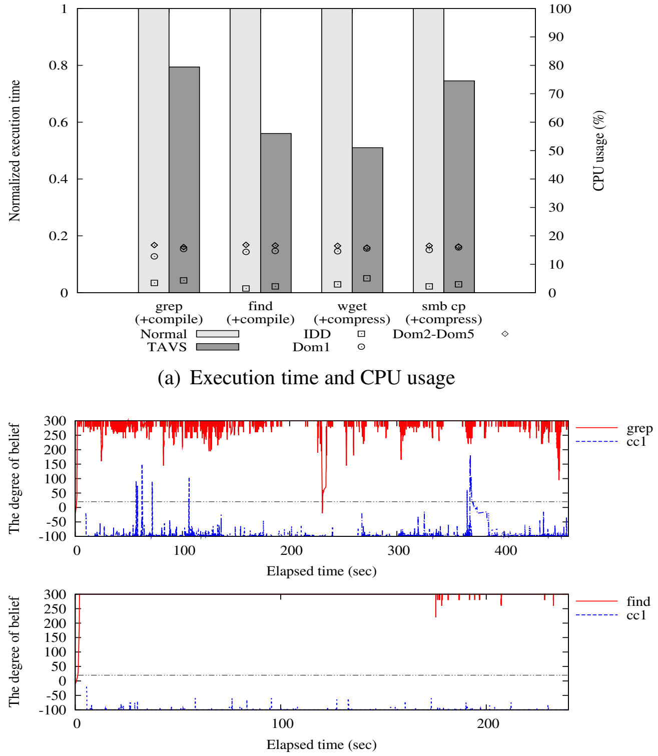
We evaluate our mechanism over realistic workloads for a virtual desktop farm and consolidated development machines. Virtualization is convenient for developing in that developers can work on their target environment anywhere with the developing tools installed in virtual machine images. As with other experiments, we concurrently run five CPU-bound domains with the PBratio of 0.125, the inspection window size of three, and the bit-width of portmap counter of two. Figure 8(a) and Figure 8(b) show the response time of a text editing task with running compilation and web browsing, respectively. The web browsing workload is made by running web browser with three sites containing several Flash animations, which is CPU-intensive. The text editing is carried out through *ssh* connection. Our mechanism improves the response time of text editing with the CPU-bound workloads.

Figure 9(a) shows the execution time and CPU usage of four different I/O-bound tasks (`grep`, `find`, `wget`, and `cp`) mixed with CPU-bound workloads (Xen compilation and file compression);



(a) Developer (b) Desktop user

Figure 8. The response time for text editing



(a) Execution time and CPU usage

(b) The degree of belief

Figure 9. Performance and CPU usage for realistic workloads

cp copies a large number of files from a remote *samba* server to the local disk. From the result, we show the performance of I/O-bound tasks is improved without compromising CPU fairness. In addition, PBHR is more than 99% for all cases. Figure 9(b) shows the degree of belief of tasks for each workload pair; the horizontal line represents BelThreshold (20). The result shows that the degree of belief well reflects the I/O-boundness of guest-level tasks. The results of wget and samba copy are omitted because they show similar cut to that of find.

5.4 Overhead

This section describes overheads for our task-aware scheduling. To evaluate the overhead for tracking I/O-bound tasks, we run 400 tasks in a domain and make them communicate each other by using *hackbench*. The average slowdown for 100 runs is 0.06% and thus shows a negligible tracking overhead. In addition, we have an IDD send network requests intensively to a domain with full CPU utilization to show the overhead for recording and checking port information. As a result, there is no degradation of network throughput for our mechanism, since port information is kept in the default shared page with the limited number; the default shared page contains frequently referenced data such as an event channel, and therefore is likely in a hardware cache.

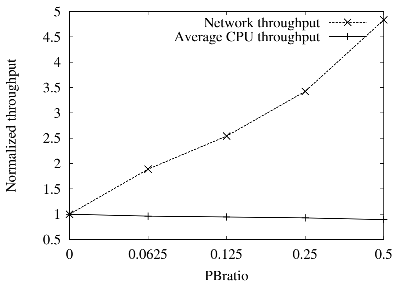


Figure 10. The overall system performance

We evaluate the overall system performance affected by partial boosting. Figure 10 shows the network throughput of a domain and the average CPU throughput of CPU-bound domains for the same experimental configuration with Figure 5(b). In this figure, the average CPU throughput of CPU-bound domains decreases as more partial boosting is allowed to one domain, since the increased I/O makes an IDD consume more CPU and results in more context switching. However, the degradation of CPU throughput is small in comparison with the increased I/O throughput. The ratio of the increased network throughput to the decreased CPU throughput is about $48 : 1$ in this evaluation.

6. Related Work

This section compares our work with previous research on VM scheduling and inference techniques using gray-box knowledge.

6.1 VM Scheduling

Performance analysis for VM schedulers has been well conducted by Cherkasova and Gupta *et al.* on the Xen VMM. They focused on the I/O performance over the I/O model of Xen using IDD. They analyzed the I/O performance of three schedulers: BVT, SEDF, and the credit scheduler [7, 6]. This work shows the I/O performance of the schedulers according to different parameters and workloads. Furthermore, they demonstrated that the I/O model of Xen makes CPU allocation and accounting complicated because an IDD processes I/O on behalf of guest domains. To enhance the accounting mechanism, they proposed SEDF-DC [12], which distributes the CPU usage of an IDD into corresponding guest domains that trigger I/O operations to the IDD.

Govindan *et al.* proposed a communication-aware VM scheduling mechanism on consolidated hosting environment [11]. Their mechanism uses *network intensity* as a scheduling metric for high throughput of network intensive workloads. In addition, they devised anticipatory scheduling for a network sender that transmits a packet periodically. Their scheduling mechanism achieves high performance over specific workloads such as a network intensive server or a streaming server. Their heuristic method, however, does not tackle the responsiveness of non-intensive interactive workloads and the performance of block I/O tasks.

Ongaro *et al.* explored the impact of a VM scheduler for various combinations of scheduling features over multiple guest domains running different types of applications [20]. They mainly focused on the operation of the credit scheduler and its enhancement. Their enhancement includes fair event channel notification, preemption minimization, and VCPU ordering based on remaining credit. In the evaluation, they experimented on the credit and SEDF schedulers according to their enhancement and original features such as boosting. They concluded that a latency-sensitive workload has poor responsiveness if the workload is mixed with CPU-bound ones in the same domain.

To cope with a semantic gap in VM scheduling, we proposed a guest-level priority-based scheduling mechanism in previous research [17]. This work is based on an intrusive approach in that a guest kernel explicitly informs the VMM of guest-level priorities of runnable and blocked tasks. In the credit scheduler-based implementation, the VMM preferentially schedules a guest domain with the highest guest-level priority if the VCPU of the domain has remaining credit. In contrast to this work, our task-aware scheduling mechanism is non-intrusive by using inference techniques and presents the enhanced correlation mechanisms.

6.2 VMM-level Inference Techniques

Many novel inference techniques monitor guest-level behaviors and achieve better resource allocation. While the use of explicit information from a guest kernel has the limitations of untrustworthiness and kernel modification, sophisticated VMM-level inference is very useful to enhance resource management transparently. Several inference techniques use gray-box knowledge, which is information acquired by monitoring output or exploiting algorithmic knowledge for operating systems [3].

Jones *et al*. presented various inference techniques for monitoring the buffer cache [15], tracking guest-level tasks [14], and detecting hidden malicious tasks [16] at the VMM-level. Antfarm is a task tracking technique that monitors virtual address space switches. In Antfarm, the VMM tracks the creation, switching, and termination of tasks while it matches an address space identifier with a task. By using this tracking technique, they proposed task-aware anticipatory scheduling, which is a disk I/O scheduling mechanism relying on task-specific information. Furthermore, they developed a hidden task detection mechanism, called Lycosid, by using Antfarm. Lycosid detects the existence of hidden malicious tasks on the basis of the task view of the user and that of the VMM. The task tracking is a crucial technique, since a task is a very important abstraction of general operating systems.

7. Conclusions

As system virtualization is more prevalent in various parts of the computing environment, a semantic gap disturbs the efficient resource management of the VMM. The inference technique using gray-box knowledge from empirical studies of operating systems can bridge the semantic gap in that the technique is transparent and can be easily deployed. This paper introduces a novel VM scheduling mechanism based on the I/O-boundness of guest-level tasks by using lightweight inference mechanisms. By adopting task-awareness in VM scheduling, we give intelligence to the VMM in favor of I/O performance while guaranteeing CPU fairness. Our inference technique for tracking I/O-boundness and the correlation mechanisms are lightweight and best-effort for preserving the economy of the VMM. Our task-aware scheduling is effective for unpredictable and varying workloads such as virtual desktop or cloud computing environments.

Our current prototype only considers the case where a guest domain has one VCPU on a single physical CPU. Thus, we do not address migration issues for guest-level tasks and VCPUs. The current credit scheduler tends to relocate a VCPU between physical CPUs mainly focusing on balancing the load of CPU. As future work, we plan to extend our prototype to support multi-core systems for improving I/O performance based on task-awareness.

Acknowledgments

This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD, Basic Research Promotion Fund) (KRF-2008-314-D00345).

References

[1] Sun virtual desktop infrastructure software. `http://www.sun.com/software/vdi/`.

[2] Virtual desktop infrastructure (VDI). White paper of VMware.

[3] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proc. SOSP*, 2001.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. SOSP*, 2003.

[5] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 3rd edition, 2005.

[6] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007.

[7] L. Cherkasova, D. Gupta, and A. Vahdat. When virtual is harder than real: Resource allocation challenges in virtual machine based it environments. Technical Report HPL-2007-25, February 2007.

[8] K. Fraser, S. H, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. Workshop on OASIS*, 2004.

[9] T. Garfinkel and M. Rosenblum. When virtual is harder than real: security challenges in virtual machine based computing environments. In *Proc. HOTOS*, 2005.

[10] I. J. Good. Weight of evidence: A brief survey. In *Proc. Second Valencia Int'l Meeting on Bayesian Statistics*, 1983.

[11] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In *Proc. VEE*, 2007.

[12] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Proc. ACM/IFIP/USENIX Middleware Conference*, November 2006.

[13] S. T. Jones. *Implicit operating system awareness in a virtual machine monitor*. PhD thesis, Madison, WI, USA, 2007. Adviser-Remzi H. Arpaci-Dusseau.

[14] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proc. USENIX Annual Technical Conference*, 2006.

[15] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proc. ASPLOS-XII*, 2006.

[16] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *Proc. VEE*, 2008.

[17] D. Kim, H. Kim, M. Jeon, E. Seo, and J. Lee. Guest-aware priority-based virtual machine scheduling for highly consolidated server. In *Proc. Euro-Par*, 2008.

[18] R. Love. *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press, 2nd edition, 2005.

[19] M. K. McKusick and G. V. Neville-Neil. Thread scheduling in FreeBSD 5.2. *Queue*, 2(7):58–64, 2004.

[20] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in virtual machine monitors. In *Proc. VEE*, 2008.

[21] M. E. Russinovich, M. E. Russinovich, D. A. Solomon, and D. A. Solomon. *Microsoft Windows Internals, Fourth Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[22] J. E. Smith. A study of branch prediction strategies. In *Proc. ISCA*, 1998.