Transparently Bridging Semantic Gap in CPU Management for Virtualized Environments

Hwanju Kim[*] KAIST
Hyeontaek Lim[†] KAIST
Jinkyu Jeong[‡] KAIST
Heeseung Jo[§] KAIST

Joonwon Lee[**]
Sungkyunkwan University

Seungryoul Maeng[††]
KAIST

CS/TR-2009-311

June 29, 2009

K A I S T
Department of Computer Science

[*] hjukim@camars.kaist.ac.kr
[†] limh@ kaist.ac.kr
[‡] jinkyu@camars.kaist.ac.kr
[§] heesn@camars.kaist.ac.kr
[**] joonwon@skku.edu
[††] maeng@camars.kaist.ac.kr

Transparently Bridging Semantic Gap in CPU Management for Virtualized Environments

Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, Joonwon Lee, and Seungryoul Maeng

Abstract—Consolidated environments are progressively accommodating diverse and unpredictable workloads in conjunction with virtual desktop infrastructure and cloud computing. Unpredictable workloads, however, aggravates the semantic gap between the virtual machine monitor and guest operating systems, leading to inefficient resource management. In particular, CPU management for virtual machines has a critical impact on I/O performance in cases where the virtual machine monitor is agnostic about the internal workloads of each virtual machine. This paper presents virtual machine scheduling techniques for transparently bridging the semantic gap that is a result of consolidated workloads. To enable us to achieve this goal, we ensure that the virtual machine monitor is aware of task-level I/O-boundness inside a virtual machine using inference techniques, thereby improving I/O performance without compromising CPU fairness. In addition, we address performance anomalies arising from indirect use of I/O devices via a driver virtual machine at the scheduling level. The proposed techniques are implemented on the Xen virtual machine monitor and evaluated with micro-benchmarks and real workloads on Linux and Windows guest operating systems.

1 Introduction

As machine virtualization has matured rapidly in terms of performance, reliability, and administration, the application of virtualization has expanded into diverse parts of the computing environment. Machine virtualization allows a large number of machines to be consolidated in limited physical hardware, ensuring efficient resource utilization and management. Furthermore, the degree of machine consolidation has grown considerably under the influence of high performance hardware and sophisticated software techniques such as paravirtualization. Along with the emergence of cloud computing [1] and virtual desktop infrastructure [2], [3], an individual computing environment is encapsulated in a virtual machine (VM) that is stored and managed in a server farm. Such virtualized environments accommodate unpredictable workloads of diverse domains ranging from desktop computing to scientific computation.

The consolidated environment should present the illusion that each VM exhibits proper behavior as its user works in the native environment. The virtual machine monitor (VMM), however, has difficulty in resource management for seamless services, since a semantic gap exists between the VMM and guest operating systems (OSes). The semantic gap [4] is inevitable because different OSes have their own sophisticated mechanisms and policies, whereas the VMM consists of lightweight components and narrow interfaces. This problem impedes the efficient allocation of hardware resources in terms of the amount and time that the resources are required by each VM. Much previous work has explored various methods to bridge the semantic gap based on VM-aware optimization [5], [6] or VMM-level inference techniques [7], [8], [9], [10].

Diverse and unpredictable workloads worsen the semantic gap in CPU allocation, especially in a highly consolidated environment. Because multiple VMs use available physical CPUs in a time-sharing manner, a guest VM can run on a virtualized CPU that is fairly given by the VMM. VM scheduling without considering VM-internal workloads, however, could provide existing VMs with unexpected performance that does not fit the native environment. In particular, the VMM degrades the performance of I/O-bound workloads mixed with various others in a VM, since the VMM lacks knowledge about guest-level tasks and their relation with I/O events. This semantic gap degrades the I/O performance in terms of responsiveness and throughput, as more VMs are consolidated in a physical machine.

In addition to the task-unawareness, using a separate VM for I/O access could incur performance anomalies due to VM scheduling. Several VMMs permit only a privileged VM, called a driver VM, to conduct and multiplex physical device accesses for the sake of system reliability and software reusability [11], [12], [13]. Since the driver VM is realized as a schedulable object by the VMM, it shares physical CPUs in the same manner as other VMs. Therefore, the VMM provides each VM with virtualized I/O devices that are only available when the driver VM is given a virtualized CPU. Such indirect use of I/O devices could present false views to a guest VM that involves I/O operations due to the underlying scheduling policies.

This paper presents virtual CPU (VCPU) management techniques to transparently bridge the semantic gap, mainly focusing on I/O performance. First, we introduce a task-aware VM scheduling mechanism, by which the VMM manages VCPUs based on the characteristics of internal workloads. It statistically infers I/O-bound tasks from mixed workloads and associates I/O events with I/O-bound tasks on the basis of gray-box knowledge about general OSes. The inferred information enables the VMM to improve the performance of I/O-bound workloads while guaranteeing CPU fairness. To this end, we devise partial boosting, which is a priority boosting mechanism applied while an I/O-bound task handles incoming events.

Second, we address the semantic gap that stems from driver VM scheduling by specifically handling a driver VM at the scheduling level. In our approach, we regard a driver VM as a shared resource, rather than a general VM, since it serves guest VMs as a proxy for I/O access. From this perspective, a driver VM is adaptively scheduled depending on I/O requesting VMs. The driver VM-specific scheduling closes the gap by hiding the indirect use of I/O devices from guest VMs.

The implementation of the proposed techniques was based on the *credit scheduler*, which is the latest scheduler of the Xen VMM. To enable lightweight management, our inference technique uses event monitoring and time measurement to distinguish I/O-bound tasks. Since the implementation is confined to the virtualization layer without any modification to the guest kernel, a variety of OSes can exploit our mechanism. In the evaluation section, we demonstrate that our mechanism improves I/O performance by bridging the semantic gap without compromising CPU fairness among guest VMs. In addition, we evaluated our schemes for desktop computing and development environments using real workloads on Linux and Windows OSes.

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 and Section 4 introduce the design and operation of our task-aware VM scheduling and driver VM-specific scheduling. 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 [14] is an open-source VMM based on a paravirtualization technique, which achieves higher performance than full virtualization approaches. Paravirtualization endeavors to minimize virtualization overheads via an interface, named *hypercall*, between a guest OS and the VMM by modifying the guest kernel. Xen puts the privileged VM, called *domain0*, in charge of managing other guest VMs, called *domainU*. Xen also supports full virtualization based on hardware-assisted virtualization (Intel-VT [15] and AMD-V [16]), ensuring that it is possible to run unmodified OSes such as Windows. Such a full virtualized domain is called a hardware virtual machine (HVM) by Xen.

Xen introduces the *isolated driver domain* (IDD), which conducts real I/O operations for a bare device on behalf of domainUs, to enable a reliable I/O architecture [11]. This I/O model enhances the reliability of an entire system by isolating the faults arising from 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.

Frontend and backend drivers notify each other of an I/O event via an *event channel*, by which a hardware interrupt is virtualized. A virtual interrupt is pending in the corresponding event channel and is then 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 it is created. A VCPU contains general information related to the execution context and event channels, because the VCPU is a scheduling entity. When a VCPU is scheduled, the guest kernel checks whether its event channel has a pending event. If so, the kernel invokes the corresponding interrupt handler routine. In this manner, a physical interrupt, which is received by the VMM, is pending in the event channel of an IDD, which then sends a virtual interrupt to the event channel of the target domain after I/O processing.

For efficient CPU utilization in virtualized environments, the VMM must identify an idle VCPU, which has no runnable task for its own time slice. When a running VCPU no longer has a runnable task, it yields its CPU and is blocked by the VMM. To this end, a paravirtualized kernel interposes a hypercall that yields the CPU to the VMM in its idle thread routine, which is invoked when no runnable task exists. In the case of full virtualization with an HVM, an instruction that makes a CPU idle (e.g. `hlt` in the x86 architecture) results in a transition to the privilege level, whereby the VMM identifies an idle VCPU.

2.2 Credit Scheduler

The credit scheduler is the default Xen scheduler that provides proportional CPU sharing and load balancing for SMP systems. The credit scheduler regards a time quantum as a *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 credit based on the weight of their domain every credit period (30 ms). The credit of a VCPU is used to determine its priority once per credit period. If a VCPU has remaining credit (i.e., 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. VCPUs with UNDER priority are always scheduled before those with OVER priority; a run queue maintains UNDER priority VCPUs followed by those with OVER priority, and the scheduler picks the VCPU at the head of the run queue as the next one. Once a VCPU is scheduled, it receives a time slice of 30 ms and then consumes its credit as it runs. When the time slice of a running VCPU expires, it is descheduled and inserted at the tail of a list that contains VCPUs with the same priority. 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.

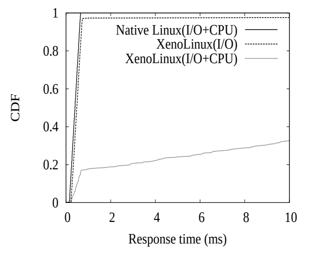
The credit scheduler allows one VCPU to preempt another running one to improve the performance of I/O-bound domains via a boosting mechanism. If a VCPU has only I/O-bound tasks, it is usually blocked and there is low credit consumption. When an event is pending to the blocked VCPU, the VMM wakes it and 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 VCPU woken, if it is UNDER—the VCPU has been blocked and there is remaining credit. The boosting mechanism assigns the highest priority, BOOST (0), to the VCPU woken, 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 time than a tick period. Therefore, I/O-bound domains frequently preempt a running domain and thus achieve improved responsiveness and throughput [17]. BOOST priority is demoted to UNDER priority at a tick time.

3 Task-aware VM Scheduling

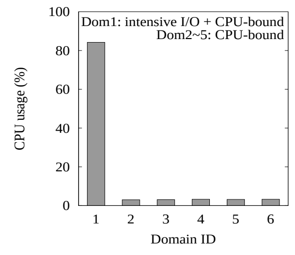
This section describes how to make the VMM aware of guest-level tasks and presents the proposed mechanisms to improve I/O performance. We first illustrate why the VMM needs task-awareness for providing seamless services.

3.1 Necessity of task-awareness

Although a VCPU-level scheduling mechanism is quite simple and effectively supports fairness, the semantic gap imposes limitations. Once the VMM allocates a physical CPU to a VCPU, it relies entirely 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 in terms of timeliness. I/O-boundness of an I/O-bound task that is mixed with heterogeneous workloads is not visible to the VMM, because it cannot recognize the characteristic of each individual task. 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 via the boosting mechanism. However, the credit scheduler does not boost the VCPU if it is not idle, even though a corresponding event is pending. An I/O-bound task running on the non-idle VCPU does not exploit the boosting mechanism and consequently has low responsiveness.



(a) Response time for three cases



(b) CPU usage for aggressive boosting

Fig. 1: Necessity of task-awareness

Figure 1(a) shows the effect of an interactive workload on the response time over the credit scheduler for various 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 are CPU-bound. The server without a CPU-bound task, on the other hand, almost preempts a running domain via the boosting mechanism, without waiting for the other domains; 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. Aggressive boosting is a naive approach for better responsiveness. 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. Figure 1(b) shows the extreme results of 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 it has runnable tasks, the CPU-bound task in domain1 has exhausted the given time slice after the incoming packet is handled. Therefore, the intensive I/O of domain1 starves the other domains while significantly compromising fairness.

To achieve both low I/O latency and fairness of CPU allocation, the VMM needs to supplement 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.

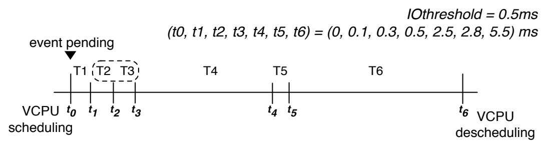


Fig. 2: Inference for I/O-bound tasks

3.2 Tracking I/O-bound Tasks

To distinguish I/O-bound tasks within mixed workloads, the VMM must track tasks in each domain 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. We favor a non-intrusive approach and use the previously proposed method to track tasks at the virtualization layer by monitoring the access to the MMU hardware [9]. In MMU-enabled OSes, a task has a private virtual address space that is provided by the paging facility of the MMU in the protected mode. A guest OS must access the MMU when switching tasks by its scheduler. The VMM can capture the task switching event because it virtualizes the MMU hardware.

The VMM uses gray-box knowledge to infer the I/O-boundness of guest-level tasks by observing the low-level interactions between the guest kernel and the hardware. The VMM controls I/O operations via 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, prevalent in commodity OSes, preferentially schedules an I/O-bound task when a corresponding I/O event occurs for low latency [18], [19], [20].
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 [21].

The first inference relies on the kernel policy. It regards a task that is preemptively scheduled in response to an event as I/O-bound. 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 characteristic that is crucial in order to overcome the trade-off between responsiveness and fairness. A task with the two characteristics can selectively achieve high responsiveness in its VCPU via partial boosting without compromising overall CPU fairness among VCPUs. This is detailed in next section.

We use the two criteria to classify observations of scheduling events into three disjoint classes: *positive evidence*, *negative evidence*, and *ambiguity*. The observation of a task is positive evidence if it provides support for 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* in order to determine what constitutes short CPU consumption; 0.5 ms is used in this example. After the VCPU is scheduled with a pending event, `T2` immediately preempts `T1` and runs for less CPU time than IOthreshold. Since multiple tasks could be waiting for the event, we also consider `T3`, which is consecutively scheduled after `T2` with a short CPU consumption. Hence, the observations of `T2` and `T3` are positive evidence. On the other hand, `T4` and `T6` provide negative evidence, because they satisfy neither of our two criteria. We regard the observation of `T1` as ambiguity in spite of a short CPU consumption, since the CPU time is likely a result of the immediate preemption of `T2`. `T5` has a short CPU consumption, but is scheduled after a task with a long CPU time. Considering the case where an I/O-bound task temporarily has low priority in this time slice, we also regard the case of `T5` as ambiguity.

Then, the VMM considers multiple observations to enable more reliable inference. Although the above gray-box knowledge explains most activities of the task scheduler of a guest kernel, certain cases may be exceptions. For example, an I/O-bound task may show an exceptionally long CPU time when the OS 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. These cases are rare but not negligible. We therefore alleviate this uncertainty by adopting a statistical approach.

The VMM maintains a *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, it adds *PositiveEv* to the degree of belief of the descheduled task. For negative evidence, the VMM subtracts *NegativeEv* from the degree of belief. We simply ignore ambiguity, because it provides no help in 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 Bayesian inference. Additive evidence can be represented as log odds, which is the weight of evidence [22]. For more intelligent inference, the VMM can dynamically change PositiveEv and NegativeEv by learning workloads on a VM. In addition to the evidence values, IOthreshold can be tuned depending on the dominant I/O-bound workloads. For example, an I/O-bound task with a text user interface can be identified by a smaller IOthreshold than that with a graphical user interface (GUI). Our current prototype uses static values, which are empirically decided, for lightweight inference. Adjusting dynamic parameters by efficiently learning workloads is a challenging issue in enhancing the task-awareness of the VMM.

3.3 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 3.1, aggressive boosting could compromise 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 to yield its CPU to another VCPU. When an event is pending to a VCPU, the VMM initiates partial boosting if it has at least one inferred I/O-bound task. A partially boosted VCPU can preempt a running one and handle the pending event. The VMM revokes the CPU from the partially boosted VCPU when the guest OS 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 it is then inserted into the run queue based on the returned priority.

When an I/O-bound task that is running with CPU-bound ones intensively conducts I/O operations, its VCPU is partially boosted very frequently. Unrestricted partial boosting, however, allows I/O-bound tasks to consume more CPU time given during a certain period than CPU-bound ones. To adjust how strongly I/O-bound tasks are favored with a given CPU, a partial boosting allowance for each VCPU, `PBratio`, is defined as follows:

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

PBratio means the fraction of a VCPU's total CPU usage that is allowed to be used for partial boosting. Both the total CPU usage and the fraction used for partial boosting are periodically reset to zero in order to consider the recent tendency. With a low PBratio, only an interactive application, which is not I/O-intensive, achieves a high responsiveness via partial boosting. On the other hand, a high PBratio ensures that an I/O-intensive task achieves a high throughput, although its colocated CPU-bound tasks have a relatively lower CPU usage. If PBratio is zero, our scheduler runs in the same manner as the original scheduling mechanism.

Although the duration of partial boosting is expected to be short due to I/O-boundness, there are some cases where it is prolonged. First, partial boosting can occur in response to an I/O event that is handled only in the kernel and is not delivered to any task. For example, an ARP request packet is handled in the kernel and no tasks are woken. In this case, partial boosting is prolonged until the boosted VCPU schedules a non-I/O-bound task or exhausts its time slice. In the worst case, a CPU-bound task exhausts the entire time slice of the partially boosted VCPU. Second, an inferred I/O-bound task may start consuming CPU time immediately after partial boosting. The effect of such varying workloads can be relieved by ensuring that NegativeEv is relatively larger than PositiveEv. Furthermore, the VMM can forcibly revoke the CPU from a VCPU that maintains prolonged partial boosting. This mechanism, however, can incur overheads for managing an individual timer for each partial boosting. Without the need for maintaining additional timers, we can ensure that the VMM restricts the duration of one partial boosting at the tick granularity. Moreover, prolonged partial boosting can be significantly alleviated by our proposed correlation mechanism that is described in the next subsection.

3.4 Correlation Mechanisms

The lack of correlation between an event and a task imposes limitations on partial boosting based only on the I/O-boundness of tasks. Without the correlation information, partial boosting could be initiated in response to an event that is destined for a non-I/O-bound task. Since the partial boosting mechanism revokes the CPU from a boosted VCPU as soon as the non-I/O-bound task is scheduled, partial boosting is meaningless when it involves unnecessary preemption. Similarly, an event that is only handled by the kernel may cause useless prolonged partial boosting. A correlation mechanism is therefore essential for effective partial boosting, in that the VMM partially boosts a VCPU only if one of its I/O-bound tasks 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.

3.4.1 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 that is running at the request time. Accurate correlation is challenging, however, 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 [7] proposes a more accurate correlation than the simple method by exploiting the fact that OSes 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 to have been sent from an I/O-bound task if at least one inferred I/O-bound task is inside the *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 and 200th blocks, 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 200th blocks are considered to have been sent from 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 some false positive partial boosting could remain. When a read completion event for the 200th block is pending, the VMM also partially 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 in order to improve throughput.

3.4.2 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 via 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 a four-tuple (source IP address, source port number, destination IP address, and destination port number) for connection-oriented protocols such as TCP, or by a 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. However, it 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, only a destination port number is considered as an identification clue of an incoming packet, because this is the most specific information related to a recipient task.

Correlation. For the posterior correlation, we use a prediction mechanism that monitors which task is woken after the delivery of an incoming packet. As stated in Section 3.2, we anticipate that an incoming packet is delivered to the first task woken, if this task is I/O-bound. Thus, if the first task woken is an inferred I/O-bound task, the VMM regards the incoming packet as for the I/O-bound task. To elaborate, we use a history-based approach, as with the branch prediction scheme [23]. The VMM uses a *portmap*; each entry maintains the correlation history for each destination port number by using an N-bit saturating counter, named *portmap counter*. If an incoming packet for a destination port number makes the kernel wake an inferred I/O-bound task, the corresponding portmap counter is incremented, otherwise it 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 the correlation accuracy depends on the amount of history, a suitable bit-width must be chosen based on the space overheads; in the case of an N-bit counter, the VMM stores a 2^N prediction history for each port number. Although a 1-bit counter uses minimal space, it is vulnerable to miss correlation. In Section 5.2, we show that a 2-bit counter is reasonable for both accuracy and space requirements.

A multiple bit counter has another means to alleviate miss correlation in case where multiple tasks use a single port number, for example, a multitasking TCP server.

As described above, since only a destination port number is regarded as a correlation unit, the VMM cannot distinguish each connection for multiple tasks using a single 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 counter, 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 at once for different port numbers, the VMM cannot distinguish which port number is related with the first task woken. To cope with this uncertainty, the VMM updates the portmap only if all incoming packets are destined for a single port number before the time slice of the target VCPU. Although this approach could defer partial boosting in case of multiple packets arriving at once for different port numbers, more precise correlation is achieved.

4 Driver VM-specific Scheduling

This section addresses the semantic gap due to scheduling in the driver VM architecture. To illustrate the importance of specifically handling an IDD, we show the performance gap from experiments and propose IDD-specific scheduling to solve the problems.

4.1 Driver VM-induced Performance Gap

As mentioned in Section 2, Xen adopts driver VM-based I/O virtualization, which permits an IDD to handle I/O operations on behalf of guest domains. An IDD is scheduled in the same manner as other guest domains because it is a schedulable object from the perspective of the VMM. Many researchers have addressed the fact that the scheduling of an IDD directly affects I/O performance [24], [25]. Since an IDD includes native I/O drivers and multiplexing software, it typically has I/O-bound characteristics, which cause it to be frequently blocked with small CPU consumption. By this characteristic, the credit scheduler mostly boosts the priority of an IDD when the VMM wakes it in response to an incoming event, thereby reducing its scheduling latency.

Scheduling an IDD as a general domain, however, incurs anomalies in a highly consolidated system. Note that the credit scheduler uses priority-based round robin scheduling with preemption support. Although this scheduling policy is widely used by general-purpose OSes, the preemption mechanism with small-ranged priority shows anomalies in a driver VM-based virtualized environment. When a domain is consolidated with several CPU-bound domains, it could suffer from driver VM-induced problems for the following cases: 1) When the priority of an IDD is lower than that of an I/O requesting domain and 2) When an I/O requesting domain is preempted by an IDD due to its I/O request.

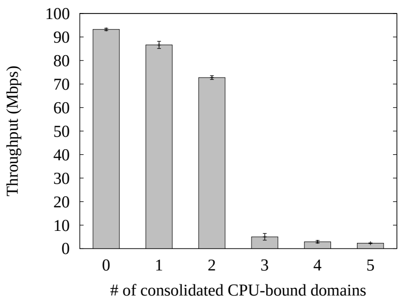
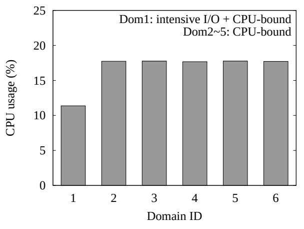


Fig. 3: Throughput degradation of a network-intensive task as the number of consolidated CPU-bound domains increases

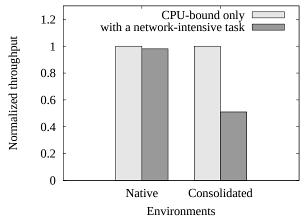
Since an I/O-intensive domain is dominated by I/O operations, its performance directly relies on the scheduling of its IDD. As mentioned, I/O-bound domains including an IDD are mostly boosted for incoming events. Ideally, an I/O requesting domain and its IDD engage in mutual preemption with successive boosted priority. The boosted priority of each domain, however, can either be reallocated every credit period or demoted to normal priority (`UNDER` or `OVER` in the credit scheduler) every tick period. The problem occurs when the priority of an IDD is lower than that of an I/O requesting domain. If an I/O requesting domain has boosted priority, an IDD with normal priority is preempted whenever delivering received I/O events to the requesting domain. Then, the preempted IDD is inserted at the tail of its priority list and must wait on the run queue. Although the domain requests I/O operations with boosted priority, these must wait until the IDD with normal priority is scheduled. Moreover, the intensive I/O requests of the boosted domain prevent the IDD from acquiring boosted priority by continuous preemption. For this problem, an I/O-intensive domain suffers from throughput degradation when consolidated with CPU-bound domains.

Figure 3 shows the I/O throughput degradation of a TCP network-intensive domain as the number of CPU-bound domains increases. Although one domain relies solely on a network device, its I/O throughput is significantly degraded as more CPU-bound domains are consolidated. In particular, when running with more than two CPU-bound domains, the I/O-intensive domain almost starves in terms of its I/O resource usage. This result demonstrates that the I/O-intensive domain severely underutilizes the I/O resource even when consolidated with a few CPU-bound domains.

Another problem arises when I/O-intensive workloads are mixed with CPU-bound workloads in the same domain. As discussed in Section 3.2, I/O-intensive tasks typically run prior to CPU-bound tasks. When an I/O-intensive task requests an I/O operation, the requested event invokes the IDD, which is mostly boosted and preempts the requesting domain. Based on the priority-based round robin algorithm, the preempted domain is



(a) Unfair CPU usage



(b) CPU throughput

Fig. 4: Unfairness problem in case where I/O-intensive workloads are mixed with CPU-bound ones in the same domain

inserted at the tail of its priority list. At the next scheduled time, the response for the previously requested I/O wakes the I/O-intensive task. If the task woken repeatedly requests I/O, the boosted IDD continuously preempts the requesting domain. Such successive preemption does not guarantee CPU fairness among guest domains, because it frequently deprives accompanied CPU-bound tasks of chances to run. Figure 4(a) shows that the requesting domain is granted unfair CPU usage even though it runs CPU-bound workloads, compared to other CPU-bound domains. As shown in Figure 4(b), this inadequate CPU allocation leads to considerable throughput degradation (up to 50 %) of CPU-bound workloads running with network-intensive workloads.

In addition, unfair preemption might delay a network response that consists of multiple network packets. If a network response is larger than the maximum segment size, it is segmented into multiple packets. In the case of connection-oriented protocols such as TCP, a recipient sends an acknowledgement packet in response to received packets for reliable data transfer. This acknowledgement packet invokes an IDD, which preempts the recipient domain that has lower priority than it does. Such preemption reinserts the recipient domain at the tail of its priority list while receiving a network response, whereby the scheduling latency prolongs the response time.

4.2 IDD-specific Scheduling

We present IDD-specific scheduling to address the aforementioned problems. The performance gap stems from the fact that I/O operations are not handled inside its domain, but via an IDD indirectly. As with monolithic VMMs such as the VMware ESX server, which include native drivers in the VMM, we consider an IDD as a shared resource that is used by guest domains for I/O processing. Although the VMM realizes an IDD as a normal domain, it can differentiate the IDD and manipulate its scheduling.

As described in Section 4.1, an I/O-intensive domain that has higher priority than its IDD suffers considerable I/O throughput degradation. While a boosted domain, for example, waits for an IDD with `OVER` priority to handle its I/O requests, other domains with `UNDER` priority can run before it does. This problem is similar to a traditional priority inversion problem. Unlike general domains, the IDD is not an independent component because it processes only I/O operations involved with other guest domains. In addition, it is a trusted component, to which the VMM grants privilege capabilities such as DMA. In light of these characteristics, the priority of the IDD should be decided more reasonably by considering each I/O requesting domain.

We propose *IDD priority inheritance*, which elevates the priority of an IDD to that of an I/O requesting domain. When a guest domain sends an I/O request to its IDD via an event channel, the IDD inherits the priority of the requesting domain if its own priority is lower. Then, it is reinserted into the run queue with the inherited priority. The original priority of the IDD is restored after it finishes handling the requested I/O operation. This mechanism guarantees that the requesting domain uses an IDD with at least equal priority for its I/O processing. When multiple domains request I/O operations to the IDD, its priority is set to the highest priority of the requesting domains. This mechanism ensures that a boosted I/O-intensive domain uses a boosted IDD for I/O processing, thus preventing I/O-intensive domain starvation. Algorithm 1 shows the procedure of IDD priority inheritance.

Algorithm 1 IDD Priority Inheritance

```
 1: procedure SENDEVENT($src, dst$)                ▷ Inter-domain
 2:     if $src = DomU$ and $dst = IDD$ then
 3:         if $src.priority > dst.priority$ then
 4:             $dst.orig\_priority \leftarrow dst.priority$
 5:             $dst.priority \leftarrow src.priority$
 6:             Reinsert $dst$
 7:         end if
 8:     end if
 9: end procedure
10: procedure SCHEDULE($prev, next$)
11:     if $prev = IDD$ then
12:         $prev.priority \leftarrow prev.orig\_priority$
13:     end if
14: end procedure
```

IDD priority inheritance does not compel the VMM to fairly allot CPU usage to an IDD, since priority is inherited regardless of the original priority system, which is based on CPU usage. Because an IDD is used in a work-conserving manner, its CPU usage should be properly distributed to each domain that involves I/O operations in order to enforce performance isolation. To distribute the CPU usage of an IDD, accurate accounting mechanisms have been proposed based on the I/O ratio for each domain [26], [27]. With accurate accounting, the credit an IDD consumes can be debited by each corresponding domain, thereby achieving enhanced performance isolation. Currently, we have not yet implemented accurate accounting in the credit scheduler and it remains as a future work.

To address the unfairness problem and inefficient reception of a network response, we ensure that the VMM retains the current domain in the run queue to prevent the preemption of an IDD (this preservation is *KeepOnRunQ*). Since our IDD-specific scheduling regards an IDD as a component for I/O processing, an I/O requesting domain should not be ousted by preemption of an IDD due to its own I/O requests. By this mechanism, the requesting domain can resume the execution for its time slice after the IDD handles its I/O requests. As a result, the VMM guarantees that CPU-bound workloads and network recipient tasks run during the given CPU of its domain.

5 Evaluation

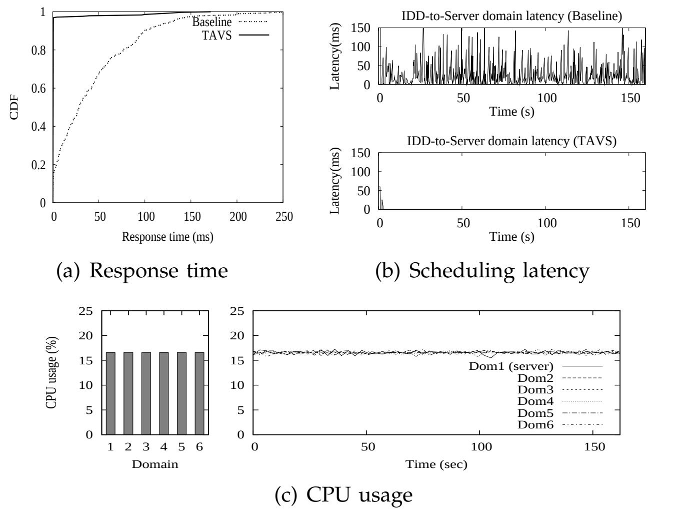
The implementation of our scheduling mechanism is based on the credit scheduler of Xen-3.2.1. To enhance the fairness of the credit scheduler, we modified the tick-based accounting method to a fine-grain one by using an Intel *time stamp counter*. Our Xen-based implementation is detailed in [28]. The proposed prototype is installed on a 3.00 GHz Intel Pentium D CPU equipped with 2 GB RAM. By default, we use domain0 as an IDD, and each domain has the paravirtualized Linux 2.6.18.8 kernel with a VCPU on a single physical core. For network workloads, a separate physical machine is connected to the consolidated machine via a 100 Mbps Ethernet switch.

We also evaluate our mechanism on Microsoft Windows XP installed on an HVM. A fully virtualized HVM should use emulated I/O operations via a QEMU device manger in an IDD. Our correlation mechanism, however, collaborates with paravirtualized features such as a grant table and a network backend driver. Our evaluation uses open-source paravirtualized drivers for Windows OS, including block and network frontend drivers. The evaluation for HVM is carried out on Intel Core 2 Quad with VT enabled.

In our evaluation, PositiveEv, NegativeEv, and BelThreshold were 5, 20, and 20, respectively. Negative evidence is regarded as a penalty and thus has higher weight than positive evidence. In addition, we constrained the degree of belief to a minimum and maximum of 100 and 300, respectively. For a guest domain with Linux, in which evaluated applications have a text user interface, the VMM tracks I/O-bound tasks with an IOthreshold of 0.5 ms. A guest domain with Windows XP, on the other hand, is tracked with an IOthreshold of 2 ms.

5.1 Partial Boosting

We demonstrate the improvement of I/O performance in terms of responsiveness and throughput on a consolidated machine with partial boosting. To show the improvement for the high consolidation scenario, we concurrently run five CPU-bound domains along with the one being 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. Our task-aware VM scheduling is always referred to as *TAVS*. Since a domain that runs I/O-bound and CPU-bound workloads is given unfair CPU usage, we applied KeepOnRunQ to both the baseline and proposed mechanisms in order to show the CPU fairness guarantee (this method is evaluated in Section 5.3).



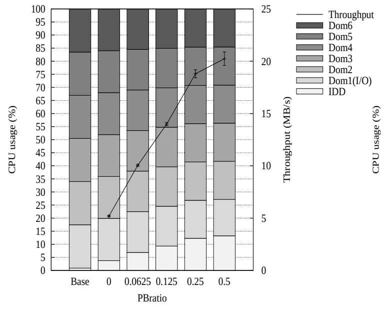
(a) Response time (b) Scheduling latency

(c) CPU usage

Fig. 5: Performance and fairness guarantee for simple interactive workload

Figure 5(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 bytes) to this server with a random think time (100 ms $\sim$ 1000 ms). As shown in the CDF graph, our mechanism significantly improves the response time by partial boosting compared to the baseline. Figure 5(b) shows the scheduling latency for delivering an incoming packet from an IDD to the server domain. The result of the baseline shows that the server domain has a maximum latency of up to about 150 ms; this latency is result of the number of CPU-bound domains (5) $\times$ a maximum time slice (30 ms). In our mechanism, the latency is close to zero as a result of partial boosting, except for the initial inferring period. Figure 5(c) shows the CPU usage for each domain. This result demonstrates that our mechanism guarantees CPU fairness for an interactive workload.

We demonstrate the throughput of the block read and network, as well as the CPU usage of each domain for different PBratios in Figure 6. The base case shows the reference data, which is acquired by permitting all domains to be CPU-bound. We use *SysBench* [29] and *Iperf* [30] to measure the throughput of the disk and network, respectively; we measure the disk throughput by sequentially reading 8192 files (the file size is 128 KB), and the network throughput by having a remote client


(a) Block

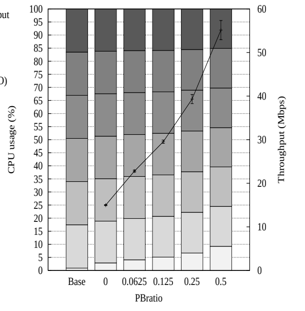

(b) Network

Fig. 6: I/O throughput for different `PBratios`

Avg. response time (ms)	CPU + I/O			I/O		
	Dom1	Dom2	Dom3	Dom4	Dom5	Dom6
Baseline	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

transmit the 512 MB of data over a TCP connection. All results are averaged over five runs. Figure 6 shows that the throughput of the 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 by the IDD.

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 a 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 the I/O-bound domains in the case of baseline. Our mechanism substantially improves the poor responsiveness of the mixed domains nearly as effectively as that of the 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 by changing the inspection window size and the bit-width of a portmap counter. As a metric of correlation, *a partial boosting hit ratio* (PBHR) is measured via 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

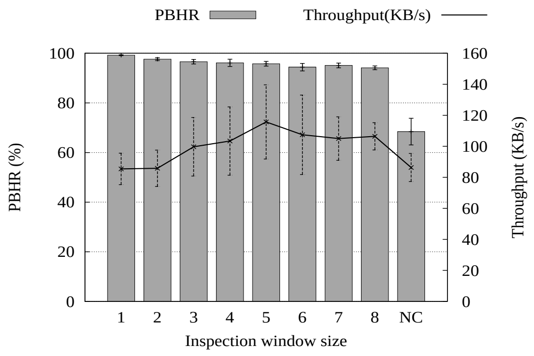


Fig. 7: PBHR and throughput of a block read I/O-bound task for different inspection window sizes

records a timestamp after the *open* and *read* system calls, and a UDP server records a timestamp after the *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 along with the one being evaluated.

A single 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 the IOthreshold. In this experiment, a single domain runs eight tasks with different CPU consumptions (ms) of 0, 1, 2, 5, 10, 30, 100, and 300; a task with 0 ms is an inferred I/O-bound task. We measure the PBHR and the performance of the I/O-bound task with a PBratio of 0.125. All averaged results are a 10% trimmed mean of ten runs. In addition, the figures provide the PBHR and the performance in the case of no correlation, named NC; no correlation means that 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 7 shows the PBHR and the throughput of the block I/O-bound task for different inspection window sizes. As stated in Section 3.4.1, the inspection window enables our scheduler to consider the I/O-bound tasks for which the block requests are delayed by the guest kernel. As stated earlier, a window-based mechanism could incur false positive partial boosting. In Figure 7, the PBHR of the I/O-bound task decreases as the window size increases; the false positive ratio is equal to (100 - PBHR) %. Instead, a larger window size achieves a better throughput of an I/O-bound task, since its delayed requests are compensated for partial boosting. Because partial boosting is restrictively allowed due to the PBratio, a high false positive ratio reduces the partial boosting chance of the I/O-bound task (Note the decline of throughput for window sizes between five and eight).

To evaluate network I/O correlation, we use the simple interactive workload used in Section 5.1; however, the random think time is between 10 ms and 1000 ms

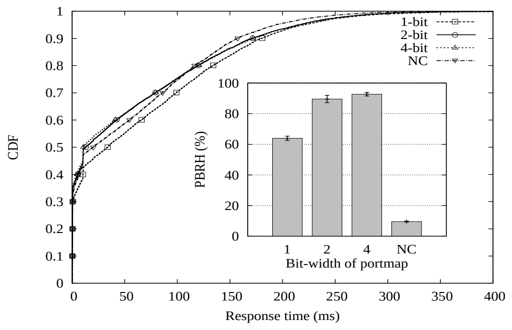


Fig. 8: PBHR and response time of a network I/O-bound task for different bit-widths of a portmap counter

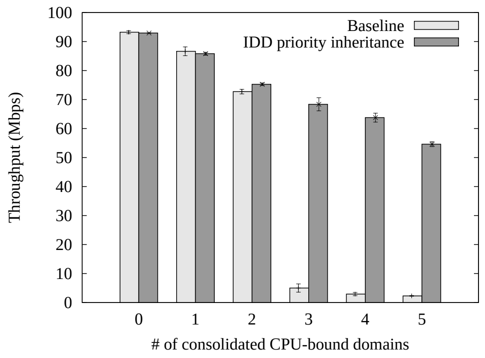
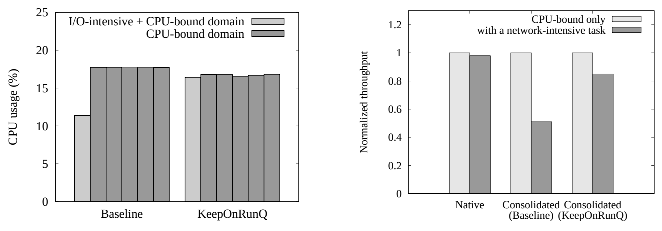


Fig. 9: Throughput improvement of a network-intensive domain for IDD priority inheritance



(a) CPU usage (b) CPU throughput

Fig. 10: CPU fairness is guaranteed by keeping an I/O requesting domain in response to an IDD's preemption (KeepOnRunQ)

in order to increase the intensity. The eight UDP echo servers with the same configuration used for the CPU consumption of the block I/O evaluation individually serve the eight clients. Figure 8 shows the response time and the PBHR for the different portmap counter bit-widths. In the case of the 1-bit counter, the PBHR of 64% shows its weakness as a result of the miss correlation and relatively low responsiveness. On the other side, the 2-bit and 4-bit counters achieve a PBHR of about 90% with the aid of the correlation history. Even though the PBHR of the 4-bit counter is slightly higher than that of the 2-bit counter, their response times are almost the same. This result demonstrates that the 2-bit counter is the best choice for reasonable performance and memory overheads. Although no correlation shows reasonable responsiveness, its PBHR is very low because of exhaustive partial boosting. This is inefficient due to unproductive domain switches.

5.3 IDD-specific Scheduling

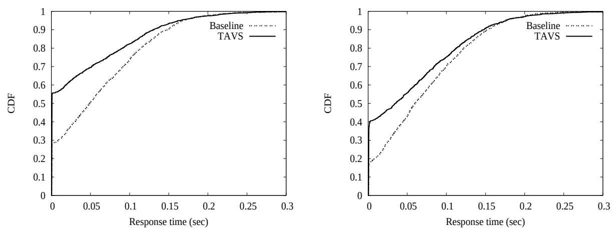
To show the impact of IDD priority inheritance, we run an Iperf server as a network-intensive task in a domain, increasing the number of CPU-bound domains. A remote client transmits 256 MB data over a TCP connection and the throughput is averaged over five runs. Figure 9 implies that the I/O-intensive domain avoids starvation when running with more than two CPU-bound domains, by preventing the IDD from having a lower priority than the I/O requesting domain. The gradual drop in throughput is a result of the lower CPU share for network processing as more CPU-bound domains are consolidated.

We evaluate that KeepOnRunQ solves the unfairness problem in case where an I/O-intensive task is running with CPU-bound workloads. Figure 10 shows the CPU usage of CPU-bound domains, one of which concurrently runs a network-intensive task. Compared to the baseline, KeepOnRunQ enables the VMM to guarantee that the CPU-bound task runs in its time slice. This mechanism maintains CPU fairness among guest domains, thus alleviating degradation of CPU throughput. As a result, we close the gap between native and consolidated environments.

5.4 Realistic Workload

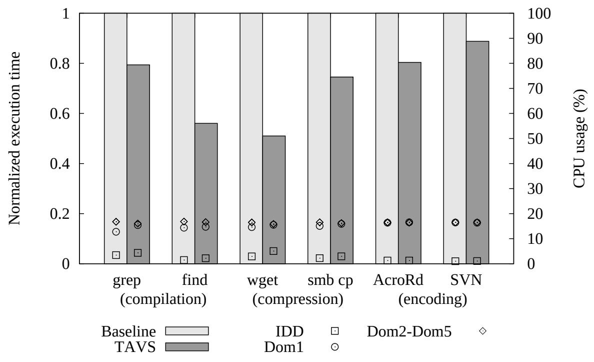
We evaluate our mechanisms over realistic workloads for a virtual desktop farm and consolidated development machines. Virtualization is convenient for developing software in that developers can work on their target environment anywhere with the developing tools installed in virtual machine images. As with the other experiments, we concurrently run five CPU-bound domains with a PBratio of 0.125, an inspection window size of three, and a portmap counter bit-width of two. Figure 11(a) and Figure 11(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 a web browser with three sites containing several Flash animations, which is CPU-intensive. The text editing is carried out via an *ssh* connection. Partial boosting improves the response time of text editing with the CPU-bound workloads.

First, we evaluate task-aware VM scheduling for mixed workloads; six I/O-bound workloads (*grep*, *find*, *wget*, *smb cp*, *Adobe Acrobat Reader*, and *Subversion* (SVN) client) are mixed with CPU-bound workloads (Xen compilation, file compression, and movie encoding). *Smb cp* copies a large number of files from a remote *samba* server to the local disk. The last two I/O-bound workloads are running on Windows XP; Acrobat Reader is initiated by loading a PDF file with cold buffer cache, and a GUI-based SVN client conducts a *checkout* operation, which

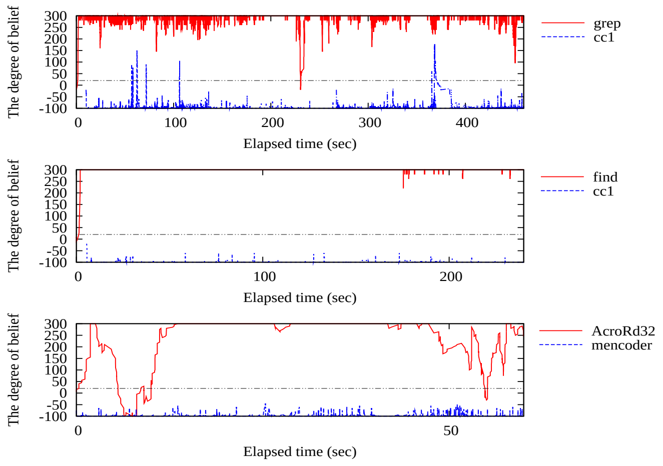


(a) Developer (b) Desktop user

Fig. 11: Response time for text editing



(a) Execution time and CPU usage



(b) The degree of belief

Fig. 12: Performance and CPU usage for realistic workloads

pulls source codes from a repository server. Figure 12(a) shows that the execution time of each I/O-bound workload is reduced without compromising CPU fairness. Figure 12(b) shows the degree of belief of tasks for each workload pair; the horizontal line represents the BelThreshold (20). The result shows that the degree of belief effectively reflects the I/O-boundness of guest-level tasks. The results of *wget*, *samba* copy, and *SVN* client are omitted because they show similar cuts.

Figure 13 shows the impact of IDD priority inheritance on the same I/O-bound workloads, except for the CPU-bound ones. Performance enhancement is in proportion to I/O intensity, since intensive I/O increases the CPU usage of an IDD, which frequently leads to lower priority of the IDD than that of a requesting domain. Since Windows applications typically consume more CPU time for GUI operations than its IDD, priority inversion rarely occurs; loading Acrobat Reader requires more CPU time for initialization. Although the checkout operation of the SVN client is I/O-intensive, it also consumes CPU time for updating graphical components. Moreover, in the baseline case, network-intensive workloads suffer from severe throughput degradation, since the IDD is continuously preempted after priority inversion occurs, due to burst and asynchronous packet arrival.

Finally, we evaluate the impact of KeepOnRunQ on the network response time. A Windows XP domain browses text and image contents with Microsoft Internet Explorer; this domain is consolidated with five CPU-bound domains, and the browsed contents are stored on a separate web server machine. We measure the response time by using *AutoIt* [31], which is a Windows automation language. Table 2 shows the web browsing response time averaged by 30 different files for each text and image; the average sizes of the text and image files (KB) are 8.6 and 300, respectively. The result shows that the improvement in the response time of image is greater than that of text, since the larger responses are divided into multiple packets, which prolong the response time without KeepOnRunQ.

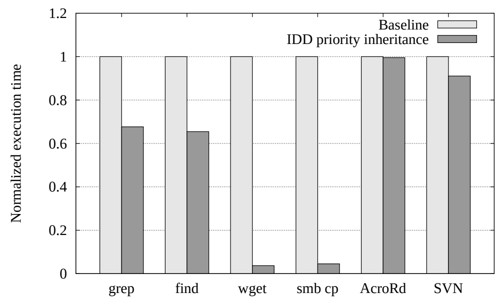


Fig. 13: Performance improvement by IDD priority inheritance for realistic workloads

Avg. response time (sec)	Text	Image
Baseline	2.12	2.78
KeepOnRunQ	2.05 (3%)	1.97 (29%)

TABLE 2: Average response time of web browsing (the value in the parenthesis denotes the performance improvement)

5.5 Overheads

This section describes the overheads for our task-aware VM scheduling. To evaluate the overhead for tracking I/O-bound tasks, we use *hackbench* [32] to run 400 tasks that communicate one another in a domain. The average slowdown for 100 runs is 0.06%, which indicates a negligible tracking overhead. In addition, we have an IDD send network requests intensively to a domain with full CPU utilization in order to show the overhead for

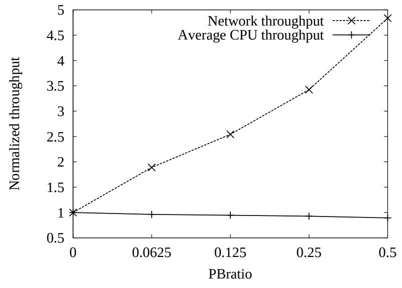


Fig. 14: Overall system performance

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 a limited number; the default shared page contains frequently referenced data such as an event channel, and therefore is likely in a hardware cache.

We evaluate how the overall system performance is affected by partial boosting. Figure 14 shows the network throughput of a domain and the average CPU throughput of CPU-bound domains for the same experimental configuration with Figure 6(b). In this figure, the average CPU throughput of CPU-bound domains decreases as more partial boosting is allowed for a domain, since the increased I/O increases the IDD's CPU consumption 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 $48:1$.

6 Related Work

This paper extends our prior work [28] with a broader point of view in terms of the semantic gap in virtual CPU management. This section compares our work with previous research on VM scheduling and inference techniques using gray-box knowledge.

6.1 VM Scheduling

Cherkasova and Gupta *et al*. have conducted an intensive performance analysis for VM schedulers 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 [24], [25]. 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 complicates CPU allocation and accounting, because an IDD processes I/O on behalf of guest domains. To enhance the accounting mechanism, they proposed SEDF-DC [26], which distributes the CPU usage of an IDD to corresponding guest domains that trigger I/O operations for it.

Govindan *et al*. proposed a communication-aware VM scheduling mechanism on the consolidated hosting environment [33], [27]. Their mechanism uses *network intensity* as a scheduling metric for the 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 or streaming server.

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 [17]. 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 previously proposed a guest-level priority-based scheduling mechanism [34]. 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, since it uses inference techniques and 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 OSes [35].

Jones *et al*. presented various inference techniques for monitoring the buffer cache [8], tracking guest-level tasks [9], and detecting hidden malicious tasks [10] 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 means of this tracking technique, they proposed task-aware anticipatory scheduling, which is a disk I/O scheduling mechanism relying on task-specific information. Furthermore, they used Anfarm to develop a hidden task detection mechanism, called Lycosid. Lycosid detects the existence of hidden malicious tasks on the basis of the task view of the user and the VMM. Task tracking is a crucial technique, since a task is a very important abstraction of general OSes.

7 Conclusions

This paper addresses the semantic gap that interferes with efficient CPU management in terms of I/O performance. Task-aware VM scheduling allows the VMM to schedule existing VMs based on the I/O characteristics of their internal workloads. We use gray-box knowledge from empirical studies of OSes to enable the VMM to transparently infer the characteristics of guest-level tasks. The inferred information for workloads assists the VMM to schedule VCPUs 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, preserving the economy of the VMM. Furthermore, this paper argues that the VMM should specifically schedule a driver VM based on guest domains that access I/O devices. We regard the driver VM as a shared resource instead of a guest domain for high I/O resource utilization and CPU fairness. As a consequence, our proposed schemes can increase the degree of consolidation while providing end-users with good quality of service.

We plan to explore the semantic gap when consolidated VMs share a multi-core CPU with multiple VCPUs. Although co-scheduling ensures that the VMM transparently serves the underlying multi-core CPUs, it can result in inefficient CPU utilization when overall VCPUs are overcommitted [36]. As a future work, we are investigating scheduling techniques to enable the VMM to provide seamless service of multi-core CPUs in an overcommitted environment.

References

[1] "Above the clouds: A berkeley view of cloud computing," white paper of UC Berkeley Reliable Adaptive Distributed Systems Laboratory.
[2] "Virtual desktop infrastructure (VDI)," white paper of VMware.
[3] "Sun virtual desktop infrastructure software," http://www.sun.com/software/vdi/.
[4] T. Garfinkel and M. Rosenblum, "When virtual is harder than real: security challenges in virtual machine based computing environments," in *Proc. HOTOS*, 2005.
[5] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in xen," in *Proc. USENIX Annual Technical Conference*, 2006.
[6] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for i/o virtualization," in *Proc. USENIX Annual Technical Conference*, 2008.
[7] S. T. Jones, "Implicit operating system awareness in a virtual machine monitor," Ph.D. dissertation, Madison, WI, USA, 2007, adviser-Remzi H. Arpaci-Dusseau.
[8] 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.
[9] ——, "Antfarm: Tracking processes in a virtual machine environment," in *Proc. USENIX Annual Technical Conference*, 2006.
[10] ——, "VMM-based hidden process detection and identification using Lycosid," in *Proc. VEE*, 2008.
[11] 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.
[12] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines," in *Proc. OSDI*, 2004.
[13] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor," in *Proc. USENIX Annual Technical Conference*, 2001.
[14] 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.
[15] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.
[16] AMD, "AMD64 virtualization codenamed "pacifica" technology: Secure virtual machine architecture reference manual," May 2005.
[17] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling I/O in virtual machine monitors," in *Proc. VEE*, 2008.
[18] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly, 2005.
[19] M. E. Russinovich, M. E. Russinovich, D. A. Solomon, and D. A. Solomon, *Microsoft Windows Internals, Fourth Edition*. Redmond, WA, USA: Microsoft Press, 2004.
[20] M. K. McKusick and G. V. Neville-Neil, "Thread scheduling in FreeBSD 5.2," *Queue*, vol. 2, no. 7, pp. 58–64, 2004.
[21] R. Love, *Linux Kernel Development (2nd Edition) (Novell Press)*, 2nd ed. Novell Press, 2005.
[22] I. J. Good, "Weight of evidence: A brief survey," in *Proc. Second Valencia Int'l Meeting on Bayesian Statistics*, 1983.
[23] J. E. Smith, "A study of branch prediction strategies," in *Proc. ISCA*, 1998.
[24] L. Cherkasova, D. Gupta, and A. Vahdat, "When virtual is harder than real: Resource allocation challenges in virtual machine based it environments," Tech. Rep. HPL-2007-25, February 2007.
[25] ——, "Comparison of the three CPU schedulers in Xen," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 2, pp. 42–51, 2007.
[26] 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.
[27] S. Govindan, J. Choi, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and co.: Communication-aware cpu management in consolidated xen-based hosting platforms," Jan 2009.
[28] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware virtual machine scheduling for i/o performance." in *Proc. VEE*, 2009.
[29] "Sysbench: a system performance benchmark," http://sysbench.sourceforge.net/.
[30] "Iperf - The TCP/UDP bandwidth measurement tool," http://sourceforge.net/projects/iperf.
[31] "Autoit - automate and script windows tasks," http://www.autoitscript.com/autoit3/.
[32] "Hackbench: New multiqueue scheduler benchmark," http://lkml.org/lkml/2001/12/11/19.
[33] 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.
[34] 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.
[35] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "Information and control in gray-box systems," in *Proc. SOSP*, 2001.
[36] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Hardware support for spin management in overcommitted virtual machines," in *Proc. PACT*, 2006.