Transparently bridging semantic gap in CPU management for virtualized environments

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

Abstract
Consolidated environments are progressively accommodating diverse and unpredictable workloads in conjunction with virtual desktop infrastructure and cloud computing. Unpredictable workloads, however, aggravate 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-boundedness inside a virtual machine using inference techniques, thereby improving I/O performance without compromising CPU fairness. In addition, we address performance anomalies arising from the 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 and virtual desktop infrastructure, 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 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, in which the resources are required by each VM. Much previous work has explored various methods to bridge the semantic gap based on VM-aware optimization or VMM-level inference techniques.

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 accesses 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 [9,24, 36]. Since the driver VM is realized as a schedulable entity by the VMM, it shares physical CPUs in the same manner as other VMs. Accordingly, the time at which an I/O request from a VM is processed depends on when the driver VM is scheduled on a physical CPU. Because a driver VM is a dedicated agent for guest VMs, general scheduling without considering the dependency could incur unexpected performance anomalies.

This paper presents virtual CPU (VCPU) management schemes 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 VCUs 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 partially while I/O-bound tasks handle 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, the VM scheduler regards a driver VM as an entity that hosts shared I/O resources, rather than a general VM, since it serves guest VMs as a proxy for I/O accesses. 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. Sections 3 and 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, I/O model, and credit scheduler of the Xen VMM.

2.1. Xen overview

Xen [5] 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 [39] and AMD-V [2]], 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 adopts driver VM-based I/O virtualization by introducing an 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 [9]. 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 a 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 VCUs 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 VCUs 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. VCUs with UNDER priority are always scheduled before those with OVER priority; a run queue maintains UNDER priority VCUs 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 priority boosting mechanism. If a VCPU has only I/O-bound tasks, it is usually blocked with 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 from an idle state, if it is UNDER—the VCPU has been blocked with remaining credits. 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 is usually granted BOOST 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 [30]. The BOOST priority of a running VCPU is demoted to UNDER when a tick occurs.

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-boundedness of a 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.

Fig. 1(a) shows how much mixed workloads in a domain degrades its responsiveness in a consolidated environment. 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 (Xen/Linux(I/O+CPU)) 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 (Xen/Linux(I/O)), on the other hand, mostly 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. Fig. 1(b) shows the extreme results of aggressive boosting. Domain 1 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 domain 1, the aggressive boosting mechanism preemptively schedules the VCPU of domain 1 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 domain 1 has exhausted the given time slice after the incoming packet is handled. Therefore, the intensive I/O of domain 1 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.

3.2. Tracking 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 [19]. 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 involvement at task switching, however, is not a mandatory operation in the system that supports hardware-assisted MMU virtualization: extended paging of Intel-VT [39] and nested paging of AMD-V [2]. This facility enables hardware to translate a guest-virtual address to host-physical one, so that the VMM does not need to track address space switching in guest OSes. Nevertheless, the VMM is still capable of tracking task switching by allowing the VMM to selectively intercept address space changes. For example, both extended and nested paging enable the VMM to specify a certain privileged instruction to intercept. For task tracking, the VMM enables a MOV-To-CR3 instruction to notify the VMM via a VMEXIT operation. Although a VMEXIT operation comes at a cost of VMM intervention, it achieves transparent task tracking without modifications to guest OSes; the cost of a VM–VM roundtrip varies across architectures and is expected to be improved further.

3.3. Inferring I/O-bound tasks

While tracking guest-level tasks, the VMM uses gray-box knowledge to infer their I/O-boundedness by observing the low-level interactions between the guest kernel and 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 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 [6,3,12,6].

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 [25].

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-boundedness, the VMM also considers the CPU consumption of the scheduled task based on the second criterion. A task that satisfies the two characteristics can selectively achieve high responsiveness in its VCPU via partial boosting without compromising overall CPU fairness among VCPUs. The partial boosting is detailed in the 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-boundedness. If a scheduled task satisfies both criteria, we classify the scheduling event into positive evidence. Otherwise, if a task violates the second criterion (i.e. long CPU running time), this case belongs to negative evidence. Finally, if a task satisfies the second criterion, but not the first one, we consider this case as ambiguity.

Fig. 2 shows an example of task scheduling during the time slice of a VCPU after an event is pending. We define I0threshold in order to determine that a task satisfies the second criterion; 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 I0threshold. 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 do not satisfy the second criterion. We regard the observation of T1 as ambiguity in spite of the short CPU consumption, since the CPU time is likely a result of the immediate preemption by T2. T5 has a short CPU consumption, but is scheduled after a task with a long CPU time. In this case, we cannot draw inference that the short CPU consumption of T5 is a result from the pending event, since T5 is scheduled after T4 that is inferred as a non-I/O-bound task. According to our rule, we regard this case as ambiguity.

Then, the VMM considers multiple observations to enable more reliable inference. Although the above gray-box knowledge explains most activities of a guest OS scheduler, 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-boundedness 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. Algorithm 1 describes the procedure of tracking I/O-bound tasks. The degree of belief and evidence borrow concepts from statistical inference techniques such as Bayesian inference; additive evidence can be represented as log odds, which is the weight of evidence [11].

For transparent inference, we use a static IOthreshold that is empirically decided. An IOthreshold plays an important role in distinguishing I/O-bound and CPU-bound tasks. As a white-box approach to choose an appropriate IOthreshold, each guest domain gives information of CPU demand for each I/O-bound application. Although this approach enables the VMM to dynamically adjust an IOthreshold according to applications’ demands, it requires application modifications and explicit channels between the VMM and guest domains. In order to decide an IOthreshold without guest involvement, we empirically adopt a static IOthreshold to our inference mechanism. We observed that I/O-bound tasks with text user interface can be identified by a smaller IOthreshold than those with graphical user interface (GUI); the evaluation of different IOthresholds is presented in Section 5.4.

3.4. 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 then to yield its CPU. 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 revoices 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:

\[ PBratio = \frac{\text{Allowed CPU usage for partial boosting}}{\text{Total CPU usage}}. \]

PBratio means the fraction of a VCPU’s total CPU usage that is allowed to be used for partial boosting.

PBratio is adjustable based on I/O-intensity of workloads. With a low PBratio, only an interactive application, which is not I/O-intensive, achieves a high responsiveness via partial boosting, since it consumes relatively small CPU for I/O operations. 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. In order to restrict unexpectedly prolonged partial boosting, the VMM restricts a VCPU within a partial boost state for less than 10 ms, which is the same as a tick duration. Moreover, prolonged partial boosting can be significantly alleviated by our proposed correlation mechanism, which is described in the next subsection.

3.5. Correlation mechanisms

The basic partial boosting mechanism described above has a potential problem of issuing unprofitable boosting. Note that our main goal is to selectively improve the performance of I/O-bound tasks from various others in a domain. Unfortunately, not every event is destined for I/O-bound tasks, since a non-I/O-bound task that conducts both I/O and computation can request I/O events. Without being aware of which task will receive an event, partial boosting could be initiated in response to an event destined for a non-I/O-bound task. Such partial boosting is harmful due to the unproductive preemption, since the VMM revokes the CPU from a partially boosted VCPU as soon as the non-I/O-bound task is scheduled.

As a solution to this problem, we devise a correlation mechanism, which maintains whether an event is likely received by an I/O-bound task. This mechanism keeps track of what tasks are involved in a certain event by monitoring I/O events and task scheduling. With this information, the VMM initiates partial boosting only when an I/O-bound task likely handles an incoming event. The correlation mechanism enables selective partial boosting for the performance of I/O-bound tasks.

We implement 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 correlation mechanism addresses event identification, correlation, and accuracy issues.

3.5.1. 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 [17] proposes a more accurate correlation than the simple
method by exploiting the fact that OSes typically copy contents in the buffer cache into user space. In spite of better correlation, this technique incurs overheads for maintaining inverse memory mapping and handling intentional page faults.

For better association, we use window-based correlation, which considers multiple tasks when a block request occurs. Note that we are interested in whether a block read I/O is requested from an I/O-bound task. In order to consider a deferred block request, the VMM inspects not only a current task, but also a certain number of previously scheduled ones when a request occurs. To this end, we use inspection window to decide how many tasks are involved in a deferred request. When the VMM observes a block read request, it checks whether at least one I/O-bound task is inside the inspection window. If so, the requests are considered to be associated with the I/O-bound task. In our current implementation, the inspection window considers tasks scheduled in the current VCPU time slice, since most I/O requests are eventually sent to the VMM within a VCPU time slice.

In Fig. 3, for example, an I/O-bound task T2 and a non-I/O-bound task T3 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 T3 is running, the VMM inspects T2 and T3 within the inspection window. Since T2 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 T2 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 T3, 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 catches some responses for simultaneously requested I/O in order to improve throughput.

3.5.2. 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 large 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 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 3.3, 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 [34]. 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. Fig. 4 shows the state machine of a 2-bit portmap counter. 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 (gray states are considered as I/O-bound).

Fig. 3. Inspection window for block I/O correlation.

Fig. 4. The state machine of a 2-bit portmap counter for network I/O correlation (gray states are considered as I/O-bound).
3.5.3. Implementation

We implement the correlation mechanisms by using the feature of Xen I/O virtualization. The implementation aims at low overheads for event identification and correlation.

To track pending block I/O events, we use grant table, which is maintained to share memory between an IDD and a domainU. Xen enables an IDD to temporarily map the foreign memory owned by a domainU for a DMA operation. Fig. 5(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; entry 1 is used to index the page. The IDD first maps the permitted memory to its address space through hypercall. The VMM creates the requested mapping (dashed lines) 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.

For network reception tracking, event identification is assisted by a network backend driver in an IDD. Packet identification is the primary role of the driver, since the driver forwards an incoming packet to a target domain. When the netback driving forwards the packet, it checks whether the packet is for TCP/IP. If so, the driver records the protocol, destination port number, and the target domain ID of the packet 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 $N \times 8$ KB memory for a portmap with $N$-bit counter; a TCP/IP packet has a 16-bit destination port number. Fig. 5(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.

4. Driver VM-specific scheduling

Driver VM model has been prevalent in many virtualization software such as Xen [9], L4 [24], and Hyper-V [29], as a method of I/O virtualization. This I/O model locates unmodified device drivers in a separate and privileged VM, which handles I/O operations from guest VMs. The reuse of existing drivers eliminates engineering efforts for porting enormous drivers to a specific VMM. Furthermore, shifting device drivers into a separate domain prevents a driver fault, which is a major source of system crashes, from collapsing guest VMs, thereby improving fault tolerance [9,24].

Despite the benefits, this I/O model raises dependency issues arising when a driver VM hosts a shared resource among guest VMs. An I/O request sent from a guest VM is pending until the driver VM eventually handles it. For this dependency, I/O performance is heavily affected by when the driver VM is scheduled. Although the indirect use of I/O devices is transparent to each guest VM, the I/O performance affected by scheduling can make a semantic gap. In particular, priority-based schedulers that are oblivious to such resource dependencies could incur performance anomalies such as a priority inversion problem. Performance issues of dependency-agnostic scheduling have been addressed for real-time [23,33] and general-purpose OSes [35,42].

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

4.1. Driver VM-induced performance gap

Since an IDD includes native I/O drivers and multiplexing software, it typically has I/O-bound behaviors, 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. In most cases, such low latency by priority boosting helps to hide the effect of indirect I/O processing from guest domains.

Priority boosting itself, however, may be insufficient for bridging the semantic gap arising from the indirect I/O through a separate domain. Note that the scheduler manipulates priority for each domain only based on I/O-boundedness and fair CPU allocation. Accordingly, the scheduling mechanism views an IDD as a general I/O-bound domain without considering any dependencies from guest domains. With the dependency-agnostic scheduling, we observed two performance anomalies: (1) I/O-intensive domain starvation and (2) unfair CPU allocation.
4.1. I/O-intensive domain starvation

An I/O-intensive domain could suffer from I/O resource starvation when continuously preempting an IDD, which has lower priority, by its I/O requests. For I/O performance, an I/O-intensive domain and its IDD are typically scheduled with successive boosted priorities. The boosted priority of each domain, however, can be eventually demoted to normal priority; in the credit scheduler, BOOST is demoted every tick period. The problem occurs when the priority of an IDD is demoted when a domain with boosted priority intensively conducts I/O operations. In this case, 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. This case is regarded as priority inversion, since other domains with normal priority run ahead of the boosted domain. Moreover, the intensive I/O requests of the boosted domain prevent the IDD from acquiring boosted priority by continuous preemption. Fig. 6 depicts such priority inversion during the network communication. In this case, network transmission of the boosted domain (D1) is delayed until the preempted IDD with UNDER is scheduled.

The priority inversion degrades I/O throughput as IDD scheduling latency increases. Fig. 7 shows the I/O throughput degradation of a TCP network-intensive domain as the number of CPU-bound domains increases. Although one domain solely uses 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. Fig. 7(b) shows a scheduling trace taken from this experiment when the I/O requesting domain runs with five CPU-bound domains. This trace shows that the priority of the IDD is mostly lower than that of the I/O-intensive domain when inter-domain events occur; this phase dominantly appears in the experiment. During the communication, the preemption by the boosted domain deters the IDD from being boosted.

4.1.2. Unfair CPU allocation

Unfair CPU allocation could occur when I/O-intensive workloads are running with CPU-bound ones in the same domain. As discussed in Section 3.3, 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 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 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. Fig. 8 illustrates this problem where a CPU-bound task (a dotted box) cannot run due to the preemption by the IDD.

Fig. 8 shows CPU allocation and computation throughput in case where a TCP network-intensive task runs with a CPU-bound one in a domain, which is consolidated with five CPU-bound domains. As shown in Fig. 9(a), the domain that includes an I/O-intensive task is granted unfair CPU allocation compared to other CPU-bound domains, even though it also has a runnable CPU-bound task. Fig. 9(b) demonstrates that this inadequate CPU allocation leads to considerable degradation in computation throughput (up to 50%) of the CPU-bound task running with the network-intensive task; computation throughput is measured as the number of busy-loop per second.

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. Note that 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 enforce the VMM scheduler to view an IDD as a shared resource that is used by guest domains for I/O processing. On the basis of the dependency between an IDD and I/O requesting 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. Unlike general domains, an IDD is not independent 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 2 shows the procedure of IDD priority inheritance.

**Algorithm 2 IDD Priority Inheritance**

```plaintext
1: procedure SendEvent(src_vcpu, dst_vcpu) -> Inter-domain  
2: if src_vcpu.domain = DomU and dst_vcpu.domain = IDD then  
3:     if src_vcpu.priority > dst_vcpu.priority then  
4:         dst_vcpu.orig_priority ← dst_vcpu.priority  
5:         dst_vcpu.priority ← src_vcpu.priority  
6:     Reinsert dst_vcpu  
7: end if  
8: end if  
9: end procedure  
10: procedure VCPUSchedule(prev_vcpu, next_vcpu)  
11: if prev_vcpu.domain = IDD then  
12:     prev_vcpu.priority ← prev_vcpu.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 [14,12]. 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 unfair CPU allocation, we enable the scheduler to enqueue a VCPU that is preempted by an IDD to the head of its priority list. Since an IDD is regarded as a component for I/O processing, the domain should not be ousted from the current location on its priority list by its own I/O requests. This policy ensures an I/O requesting domain to resume the execution for its time slice after the IDD handles requested I/O operations. As a result, it solves the performance anomalies arising from unfair CPU allocation by avoiding interruption to a given time slice due to the preemption by an IDD. This preservation on the run queue is referred to as KeepOnRunQ.

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 grained one by using an Intel timestamp counter. Our Xen-based implementation is detailed in [22]. 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 evaluated our mechanism on Microsoft Windows XP installed on an HVM. A fully virtualized HVM uses 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 VT-enabled Intel Core 2 Quad without an extended paging facility.

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 a
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 I/O threshold of 0.5 ms. A guest domain with Windows XP, on the other hand, is tracked with an I/O threshold of 2 ms. Our task-aware VM scheduling is referred to as TAVS.

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. We concurrently run five CPU-bound domains along with a mixed one, which contains both I/O-bound and CPU-bound workloads: the original scheduler does not identify the I/O-bound task in the mixed domain. We applied KeepOnRunQ to both the baseline and TAVS in order to show the CPU fairness guarantee (this method is evaluated in Section 5.3). Priority inheritance is excluded from both cases in the evaluation for task-aware VM scheduling.

Fig. 10(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 (100–1000 ms). We used a PBratio of 0.125 and 2-bit portmap counter. As shown in the CDF graph, our mechanism significantly improves the response time by partial boosting compared to the baseline. Fig. 10(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 the result of the number of CPU-bound domains (5) × 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. Fig. 10(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 Fig. 11. The base case shows the reference data, which is acquired by permitting all domains to be CPU-bound. We use SysBench [38] and Iperf [16] 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 transmit the 512 MB of data over a TCP connection. All results are averaged over five runs.

Fig. 11 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. Despite the improvement of I/O performance, the increased CPU usage of the IDD curtails CPU allocation of other CPU-bound domains (the impact on computation throughput is evaluated in Section 5.5). In order to mitigate aggressive use of an IDD, we conservatively chose a PBratio of 0.125 for the rest of experiments. Note that 0.125 is not the best choice for maximizing I/O performance, but we chose this conservative parameter in order to cap the IDD usage for partial boosting.

We evaluated 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 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; we exclude three CPU-bound domains, which have no I/O operation. 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 evaluated 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

\[
\text{PBHR} = \frac{\sum h}{\text{The number of partial boostings}} \times 100
\]

where

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

We use this metric to identify unprofitable partial boosting, during which only non-I/O-bound tasks run, since our correlation mechanisms aim at reducing such partial boosting; the false positive ratio equals \((100 - \text{PBHR})\%\).

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 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 I/O threshold. 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.

Fig. 12 shows the PBHR and the throughput of the block I/O-bound task for different inspection window sizes. As stated in Section 3.5.1, 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 Fig. 12, the PBHR of the I/O-bound task decreases as the window size increases. 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 ran multiple network interactive workloads in the same domain. We enabled the eight UDP echo servers with the different CPU consumptions to individually serve the eight clients; each client has a random think time between 10 and 1000 ms. Fig. 13 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 hand, the 2-bit and 4-bit counters achieve a PBHR of about 90% with the aid of the correlation history. In general, the bit-width of a portmap depends on the trade-off between accuracy and space overhead. 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. We used 2-bit portmap counter in the rest evaluations, since 4-bit counter results in little increase of hit ratio while it doubles the memory consumption. Although no correlation shows reasonable responsiveness, it is inefficient because it results in 13 times more partial boosting than the correlation, leading to a very low PBHR.

5.3. IDD-specific scheduling

We evaluated IDD-specific scheduling (IDD priority inheritance and KeepOnRunQ) with micro-benchmarks to show improvement
demonstrates that the I/O-intensive domain PBHR (%)

0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

100
80
60
40
20
0

20
40
60
80
100

0  50  100  150  200  250

(a) Response time.

(b) PBHR (the value on top of each bar is the number of partial boosting).

Fig. 13. PBHR and response time of a network I/O-bound task for different bit-widths of a portmap counter.

5.4. Realistic workloads

We evaluated 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, 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.

We evaluated that KeepOnRunQ solves the unfairness problem in case where an I/O-intensive task is running with CPU-bound workloads. Fig. 15 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.

Fig. 18 shows the degree of belief with changing an IThreshold. For each experiment, an IThreshold is increased on the fly after 20 s. As shown in the figure, text-based wget was well identified by an IThreshold of 0.5 ms, but a smaller one, 0.1 ms, did not distinguish wget and bzip2. Instead, an IThreshold of 0.5 ms was not enough to pick out GUI-based svn from mencoder. The average period of adaptation for distinguishing I/O-bound tasks is about ten seconds after the IThreshold changes. For agile adaptation, we can choose a higher PositiveEv. Across the evaluated real workloads, we observed that GUI-based applications require a relatively larger IThreshold (2 ms) than text-based ones. Since an IThreshold is a tunable parameter for each domain, we can set the value according to the characteristics of I/O-bound applications.

First, we evaluated 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 (cc1), file compression (bzip2), and movie encoding (mencoder)). 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 pulls source codes from a repository server. Fig. 17(a) shows that the execution time of each I/O-bound workload is reduced without compromising CPU fairness. Fig. 17(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-boundedness of guest-level tasks. The results of wget, samba copy, and SVN client are omitted because they show similar cuts.

5.4. Realistic workloads

We evaluated 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, 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.

First, we evaluated 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 (cc1), file compression (bzip2), and movie encoding (mencoder)). 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 pulls source codes from a repository server. Fig. 17(a) shows that the execution time of each I/O-bound workload is reduced without compromising CPU fairness. Fig. 17(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-boundedness of guest-level tasks. The results of wget, samba copy, and SVN client are omitted because they show similar cuts.

Fig. 18 shows the degree of belief with changing an IThreshold. For each experiment, an IThreshold is increased on the fly after 20 s. As shown in the figure, text-based wget was well identified by an IThreshold of 0.5 ms, but a smaller one, 0.1 ms, did not distinguish wget and bzip2. Instead, an IThreshold of 0.5 ms was not enough to pick out GUI-based svn from mencoder. The average period of adaptation for distinguishing I/O-bound tasks is about ten seconds after the IThreshold changes. For agile adaptation, we can choose a higher PositiveEv. Across the evaluated real workloads, we observed that GUI-based applications require a relatively larger IThreshold (2 ms) than text-based ones. Since an IThreshold is a tunable parameter for each domain, we can set the value according to the characteristics of I/O-bound applications.

Fig. 19 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 evaluated 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 [4], 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 are 8.6 KB and 300 KB, 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.

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 [15] 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 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 evaluated how the overall system performance is affected by partial boosting. Fig. 20 shows the network throughput and the average computation throughput of CPU-bound workloads, for the experimental configuration used in Fig. 11(b); computation throughput is measured as the number of busy-loop per second. In this figure, the average computation throughput decreases as more partial boosting is allowed, since the increased I/O raises the IDD’s CPU consumption as shown in Fig. 11. However, the degradation of computation throughput is small in comparison with the increased I/O throughput. The ratio of the increased network throughput to the decreased computation throughput is about 48:1. For such a highly I/O-intensive workload, PBratio can be favorably assigned the maximum value (i.e. 1) in order to largely improve I/O performance with a little drop in computation throughput of CPU-bound domains. In the real world, PBratio can be used as a knob for a cloud provider to adjust policy between high I/O performance and strict fairness.

6. Related work

This paper extends our prior work [22] 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 [7,8]. 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 [14], 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 [13,12]. 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 [30]. 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

Table 2

Average response time of web browsing (the value in the parenthesis denotes the performance improvement).

<table>
<thead>
<tr>
<th></th>
<th>Text</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>2.12</td>
<td>2.78</td>
</tr>
<tr>
<td>KeepOnRunQ</td>
<td>2.05 (3%)</td>
<td>1.97 (29%)</td>
</tr>
</tbody>
</table>

Fig. 20. Overall system performance.
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 [21]. 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.

Compared to our preliminary work [22], this paper addresses performance issues due to naive driver VM scheduling, by showing experiments on highly consolidated system. In addition, we have extended our prototype to support HVM in order to evaluate the proposed mechanisms for an unmodified Windows guest OS. The evaluation emphasizes our transparent mechanism without guest OS modifications.

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 [3].

Jones et al. presented various inference techniques for monitoring the buffer cache [18], tracking guest-level tasks [19], and detecting hidden malicious tasks [20] 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 Antfarm 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 the task is a very important abstraction of general OSes.

6.3. Driver VM model

The previous research on driver VM model has been mainly focused on bridging performance gap caused by the indirect use of I/O devices. Sugerman et al. presented several optimization techniques for the VMware Workstation to reduce hosted I/O overheads [36]. LeVasseur et al. proposed driver VM model based on L4 microkernel and addressed performance anomalies arising from time-shared I/O devices [24]. Menon et al. [27] and Santos et al. [32] optimized the performance of network I/O virtualization by analyzing architectural overheads caused by interactions between the Xen I/DD and domainU. Furthermore, TwinDrivers [28] operates performance-critical driver codes in the VMM, while locating others in an I/DD, for both high performance and reliability. Compared to the previous research, which primarily considers a single guest VM, our work addresses performance problems induced by scheduling multiple VMs that have heterogeneous workloads.

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-boundedness 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 [41]. In addition, the VMM can schedule driver VM’s VCPUs by considering a shared cache and dedicated core, in order to minimize performance gap incurred by the indirect use of I/O devices. 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.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0000829).

References

Hwanju Kim received his B.S. degree in Information and Computer Engineering from Ajou University, Korea, in 2006, and M.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2008. He is a Ph.D. candidate of KAIST, and his research interests include virtual machine, distributed system, and storage system.

Hyeontaek Lim received a B.S. in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), Korea, in 2009. He began his graduate study in the Computer Science Department at Carnegie Mellon University. His research interests include distributed systems and networking.

Jinkyu Jeong received the B.S. degree from the Computer Science Department, Yonsei university, and the M.S. degree in Computer Science from the Korea Institute of Science and Technology. He is currently a Ph.D. candidate in the Computer Science Department, Korea Advanced Institute of Science and Technology. He is current research interests include real-time system, operating systems, virtualization and embedded systems.

Heesung Jo received the B.S. degree in Computer Science from the Sogang University, Korea, in 2000, and the Ph.D. degree in Computer Science from the Korea Advanced Institute of Science and Technology (KAIST), Korea, in 2010. He joined the faculty of the Chonbuk National University in 2010. His research interests include computer hardware and system software.

Joowon Lee received the B.S. degree from Seoul National University in 1983 and the Ph.D. degree from the Georgia Institute of Technology in 1991. After working for IBM, he joined the faculty of the Korea Advanced Institute of Science and Technology (KAIST) in 1992. Currently, he is the faculty of the SungKyunKwan University. His research interests include operating systems, low-power computing, and virtual machine.

Seunghyoun Maeng received the B.S. degree in Electronics Engineering from Seoul National University, Korea, in 1977, and the M.S. and Ph.D. degrees in Computer Science from KAIST in 1979 and 1984, respectively. Since 1984 he has been a faculty member at KAIST. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar. His research interests include computer architecture, cluster computing, and embedded systems.