Virtual Asymmetric Multiprocessor for Interactive Performance of Consolidated Desktops

Hwanju Kim Sangwook Kim Jinkyu Jeong Joonwon Lee
Sungkyunkwan University, Suwon, South Korea
hwandori@gmail.com, swkim@csl.skku.edu, jinkyu@skku.edu, joonwon@skku.edu

Abstract

This paper presents virtual asymmetric multiprocessor, a new scheme of virtual desktop scheduling on multi-core processors for user-interactive performance. The proposed scheme enables virtual CPUs to be dynamically performance-asymmetric based on their hosted workloads. To enhance user experience on consolidated desktops, our scheme provides interactive workloads with fast virtual CPUs, which have more computing power than those hosting background workloads in the same virtual machine. To this end, we devise a hypervisor extension that transparently classifies background tasks from potentially interactive workloads. In addition, we introduce a guest extension that manipulates the scheduling policy of an operating system in favor of our hypervisor-level scheme so that interactive performance can be further improved. Our evaluation shows that the proposed scheme significantly improves interactive performance of application launch, Web browsing, and video playback applications when CPU-intensive workloads highly disturb the interactive workloads.

1. Introduction

Virtual desktop infrastructure (VDI) [1, 2, 35] has drawn attention as a type of virtualized environments where individual desktops are consolidated in a server farm. The desktop consolidation is particularly appealing in organizations that manage personal desktops for a large number of their members or employees. By locating desktops as virtual machines (VMs) in a centralized data center, those organizations are capable of managing considerable desktop machines flexibly and securely. In addition, high density of desktop consolidation can lead to significant cost savings due to efficient resource utilization. Case studies about the real-world adoptions of VMware VDI show that virtual desktops are successfully deployed with the consolidation ratio of 15:1 or greater [35].

VDI hypervisors, however, provide limited support for consolidated desktops with respect to interactive performance, which is crucial for desktop user experience. For interactive performance, user-interactive workloads can be prioritized with larger CPU bandwidth and dispatch preference over background ones in a user's desktop VM. Most hypervisors shift the discretion of preferentially scheduling interactive workloads to guest operating systems (OSes) while themselves fulfilling workload-oblivious CPU provisioning. Hence, existing hypervisors provide a VM with virtual symmetric multiprocessor (vSMP) whose virtual CPUs (vCPUs) have identical computing bandwidth (i.e., CPU shares) as general workload containers. Since vSMP does not differentiate a VM's vCPUs regardless of their hosted workloads, an interactive workload running on a vCPU cannot benefit from any scheduling preference over background ones on the other vCPUs. In highly consolidated desktops, vSMP could restrict the performance of interactive workloads whose hosting vCPUs compete with those of other VMs for limited physical CPUs (pCPUs).

In this paper, we propose *virtual asymmetric multiprocessor* (vAMP), which is capable of dynamically adjusting the computing capacity of vCPUs in order to surmount the limitation of vSMP regarding interactive performance. The vAMP enables CPU shares allocated to a VM to be asymmetrically distributed to the vCPUs that belong to the VM. Since the asymmetric provisioning of CPU shares is allowed only within per-VM share budget, CPU fairness is preserved between independent VMs. By means of vAMP, we propose a VDI-friendly scheduling scheme that allows interactive workloads to run on faster vCPUs, which are given scheduling preference with larger CPU shares on contended pCPUs, than those hosting background workloads. To this end, we introduce a hypervisor extension that requests the hypervisor scheduler to allocate more CPU shares to vCPUs that host interactive workloads based on the estimation of workload characteristics.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

VEE '14, March 1–2, 2014, Salt Lake City, Utah, USA.
Copyright © 2014 ACM 978-1-4503-2764-0 /14/03. . . $15.00.
http://dx.doi.org/10.1145/2576195.2576199

The primary role of our hypervisor extension is to infer which vCPUs are involved in user-interactive workloads. Accurately identifying interactive workloads, however, is challenging in the hypervisor layer, which can access only a small set of hardware interactions (e.g., I/O operations and privileged instructions). Considering this challenge, we devise background workload identification on the basis of user I/O and per-task CPU loads rather than tracking interactive tasks. By using hypervisor-level task tracking [16], this scheme detects a set of tasks that have been generating nontrivial CPU loads before a user I/O event as background tasks. In this manner, non-background tasks are regarded as potentially interactive workloads. In addition, our hypervisor extension excludes CPU-intensive multimedia workloads such as video playback from background workloads by monitoring audio-generating tasks. To be more practical, multimedia tasks that indirectly generate audio output through server/client-based sound systems [27] can be also filtered by tracking remote wake-up recognized as an inter-processor interrupt (IPI).

Finally, we propose a guest OS extension for vAMP towards further improvement of interactive performance. Our hypervisor-level scheme has an intrinsic problem where it cannot manipulate OS scheduling policy in favor of vAMP. A vAMP-oblivious OS scheduler could multiplex interactive and background tasks on a single vCPU, which can thus be frequently changed between fast and slow modes. The frequent mode changes ineffectively increase the scheduling latency of interactive workloads. To avoid such multiplexing, our guest OS extension allows the OS scheduler to isolate interactive tasks from background ones on separate vCPUs. This extension is meant to be optionally installed in a guest OS to assist our hypervisor extension with vAMP-friendly isolation.

The proposed scheme was evaluated based on a KVM/SPICE VDI environment [1]. We chose application launch, Web browsing, and video playback as representative interactive workloads. For evaluation, we replayed a recorded interactive session to an 8-vCPU VM that ran a CPU-bound multithreaded workload (*freqmine* in the PARSEC suite) while another CPU-bound 8-vCPU VM was corunning. Our evaluation shows that the vAMP hypervisor extension alone improves application launch time by up to 41% and Web browsing time by up to 31% compared to vSMP, while achieving further improvement by up to 70% and 41% for launch and browsing time if assisted by our guest OS extension. In addition, the vAMP significantly improves video playback quality by reducing the frame drop rate from 28.1% to 1.5–2.2%.

The remainder of this paper is organized as follows: Section 2 describes background and related work on CPU scheduling schemes for interactive performance. Section 3 introduces the design and implementation of our vAMP extensions for hypervisor and guest OS. In Section 4, we present our measurement methodology and evaluation results. Finally, Section 5 concludes our work and presents future direction.

[1] http://www.spice-space.org

2. Background and Related Work

This section describes existing proportional share (PS) based scheduling mechanisms by which the hypervisor gives an illusion of an SMP to each VM. We then argue why the hypervisor scheduler should be enhanced for user-interactive performance in consolidated desktops.

2.1 PS-based Schedulers for SMP VMs

PS-based CPU scheduling [10, 14, 36, 37] is a simple and powerful scheme that provides performance isolation in CPU resources to each schedulable entity. PS-based schedulers guarantee available CPU bandwidth to be proportionally allocated to schedulable entities on the basis of their given shares. By doing so, fairness can be preserved between independent entities (e.g., processes or vCPUs), which time-share CPU resources. Since performance isolation is one of primary objectives in the hypervisor [15, 31], most commodity hypervisors have adopted PS-based CPU scheduling [9, 23, 34].

PS-based schedulers in hypervisors perform hierarchical share distribution in order to support SMP VMs. Firstly, each individual VM is given shares in proportion to its assigned weight. Inter-VM fairness is ensured in this level. Then, the VM's shares are redistributed to its own vCPUs. In order to provide a virtual SMP (vSMP), per-VM shares are equally distributed to each vCPU in the same VM. This simple scheme, however, could reduce effective shares given to active vCPUs in the case where per-VM shares are unnecessarily distributed to inactive (i.e., idle) vCPUs. To alleviate such wastage of shares, most schedulers keep track of idleness of vCPUs based on their recent CPU utilization. Once a vCPU is regarded as an idle vCPU, it is excluded from the target of share distribution, thereby increasing effective shares for active vCPUs.

2.2 Why Hypervisor Support for Interactive Performance?

In OS research communities, huge volume of work on CPU scheduling has focused on interactive performance enhancement. Most proposals improved user responsiveness by giving an interactive task higher priority or weight than throughput-oriented ones [11, 14, 24, 25, 40]. Such scheduling allows unfair CPU allotment between interactive and background tasks, for the first class support for user experience. In order to distinguish user-involved tasks from background ones, researchers have proposed various classification methods such as user-driven [24, 38], application-directed [25], middleware-assisted [11], and OS-level [40] schemes.

As a virtualization layer has been introduced and become prevalent as the most privileged layer in conventional software stack, we should rethink the support of CPU scheduling for interactive workloads hosted on consolidated systems. In a virtualized environment, a hypervisor scheduler sits on the bare-metal CPUs dictating how much and when to provide hosted VMs with shared CPU resources. Since OS schedulers work only on online vCPUs, which are scheduled by the hypervisor, OS scheduling policy for interactive performance could be invalidated if the hypervisor is oblivious to user-interactive workloads hosted on vCPUs. Making the hypervisor aware of user interaction is a challenging issue, because only a small set of hardware interactions are visible to the hypervisor.

3. Virtual Asymmetric Multiprocessor for Interactive Performance

This section presents vAMP, which dynamically adjusts the shares of vCPUs within a VM's share budget depending on workloads hosted on the vCPUs. As mentioned in the previous section, for high quality of user experience, vCPUs that are involved in user interaction need to be given more shares than others running background workloads. To differentiate vCPUs dynamically, the hypervisor should be able to determine whether a vCPU is currently hosting user-interactive workloads. In designing vAMP, we consider several challenges with regard to the different characteristics of the hypervisor from OSes.

Firstly, a vCPU, a minimum schedulable entity in the hypervisor, is a general container of various types of workloads. Since a thread can be migrated across vCPUs in a very fine-grained manner by an OS scheduler, the characteristic of a vCPU is dynamically changed at any given time. Secondly, available information for identifying workload characteristics is restrictive at the hypervisor. A small set of hardware accesses (e.g., I/O operations and privileged instructions [3]) can be unobtrusively monitored by the hypervisor, whereas upper layers in a VM can access abundant information to infer interactive workloads; this characteristic leads to semantic gap [8]. Thirdly, performance isolation between VMs limits the best-effort boosting of computing power required for interactive performance. Since VM-based consolidation environments typically have multi-tenant nature, favoring a certain VM is not allowed to compromise the computing power of other independent VMs [15, 31]. Finally, the hypervisor and guest OSes make their own scheduling decisions independently without considering the intent of each other, letting the decision of one layer less effective.

Based on the issues, we have the following design goals in mind:

1. **A simple hypervisor extension aiding the hypervisor scheduler**: Our hypervisor extension can identify user-involved vCPUs based on user I/O, per-task CPU loads, and IPIs, all of which are unobtrusively monitored by the hypervisor.
2. **Asymmetric share distribution to sibling vCPUs within per-VM share budget**: No share exchange is allowed between independent VMs for performance isolation, but the shares of a VM are flexibly distributed to its sibling vCPUs. Nonetheless, idle CPU cycles of dormant VMs, which idle all vCPUs, can be used by other active VMs in a work-conserving manner.
3. **An optional guest OS extension for further enhancement of interactive performance**: A guest OS extension can further improve interactive performance by preventing an OS scheduler from making ineffective decisions against vAMP. Such an extension can optionally be installed in a guest OS if a user needs more enhanced interactive performance.

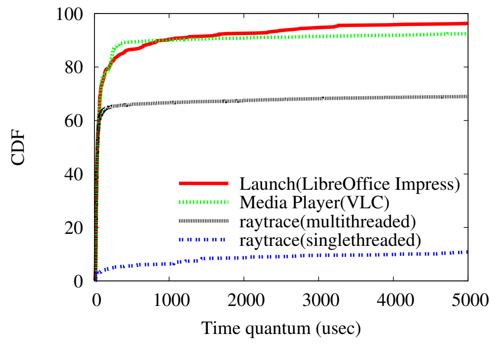


Figure 1. The cumulative distribution of time quanta observed by the hypervisor for interactive and throughput-oriented workloads.

3.1 Which vCPUs Have More Computing Power?

The challenging part of vAMP is to identify user-interactive workloads at the hypervisor. To cope with the issue, we investigated some possible ways proposed in previous work.

3.1.1 Limitations on Alternatives

We first look into the I/O-bound task tracking, which is previously proposed in [18, 19]. In this scheme, a task with short time quanta in response to I/O events is regarded as an I/O-bound task on the basis of the traditional heuristic of distinguishing I/O-bound and CPU-bound tasks. Since task execution time can be measured on address space switches [16], which are visible to the hypervisor, this technique is a viable solution for the hypervisor to identify interactive tasks. However, modern interactive applications based on GUI and multimedia are not always I/O-bound tasks with short time quanta because of their nontrivial computation. Conversely, as many throughput-oriented workloads such as RMS (recognition, mining and synthesis) applications [6] have been multithreaded, frequent synchronizations between the threads in a throughput-oriented task can lead to short time quanta.

Figure 1 shows the cumulative distribution of per-task time quanta of various workloads: application launch and video playback as interactive workloads and *raytrace* as a throughput-oriented one. Per-task time quantum was measured at the hypervisor by monitoring address space switches as with the I/O-bound task tracking [18]. We ran an 8-vCPU VM on eight pCPUs. As shown in the figure, 10% time quanta of the interactive workloads are longer than 1ms, whereas the multithreaded raytrace has its 62% time quanta shorter than 100 microseconds. Note that the single-threaded raytrace is clearly distinguished from the interactive workloads by its sufficiently long time quanta (90% is longer than 5ms). In the case of the multithreaded version, however, inter-thread synchronization significantly shortens the time quantum by frequently blocking and waking up communicating threads. Based on the result, the I/O-bound task tracking based on time quanta may not identify non-I/O-bound interactive workloads.

The second alternative is user I/O-driven detection of interactive tasks by tracking user I/O and inter-process communications (IPCs) [40]. This work exploits the general knowledge in which user-interactive tasks are scheduled being derived from user I/O such as keyboard, mouse, and audio events. More importantly, multiple tasks are involved in a single user interaction in most cases. For example, a keyboard event is firstly delivered to an X server, which then forwards it to a specific task with which a user directly interacts. Although this technique is powerful in identifying a group of tasks involved in a user interaction, it heavily relies on OS-level structures and information for tracking various types of IPCs, which are solely carried out by an OS and inaccessible by the hypervisor.

3.1.2 Background Task Identification

Considering the strength and limitation of the previous work, we propose *background task identification* based on user I/O and per-task CPU loads. As opposed to existing ways of tracking interactive tasks, the proposed scheme identifies background tasks at the moment of a user I/O event and then regards the non-background ones as potentially interactive workloads. To this end, our scheme inspects recent CPU load imposed by each task when a user I/O event occurs. If a task has generated nontrivial CPU load within a certain period prior to the user I/O event, it is tagged as a background task that likely interferes with the following user interaction.

The rationales behind the proposed scheme are as follows: Firstly, a user I/O event typically initiates a user-interactive workload, so-called an *interactive episode*, for which corresponding tasks begin consuming CPU resources [12, 13]. Secondly, background workloads literally mean the CPU loads (or noises) being generated at the moment of user interaction, which typically triggers a foreground job. Finally, identifying background tasks enables a simple hypervisor extension to unobtrusively monitor user I/O and per-task CPU loads.

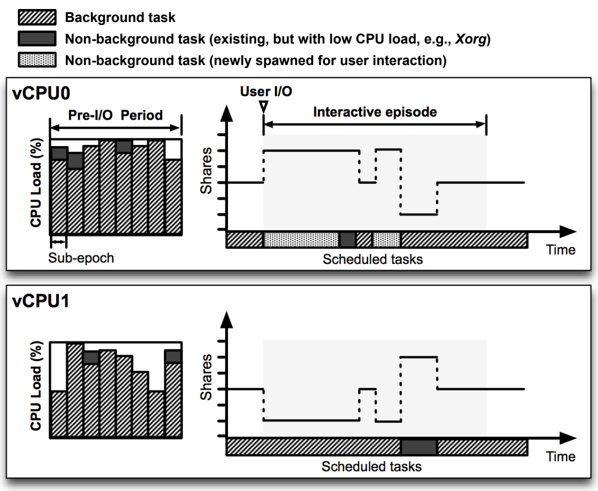


Figure 2. The background task identification based on user I/O and per-task CPU loads, and share adjustment during an interactive episode. In this example, the total amount of the VM's shares is six and the weight ratio of slow and fast vCPUs is 1:5.

In order to monitor per-task CPU loads, as with the I/O-bound task tracking [18], our hypervisor extension measures the time elapsed between two address space switches. In x86 architecture, a CR3 update with a different value represents an address space switch; an update with the same value is mainly used to invalidate translation lookaside buffer. While trapping a CR3 update by the hypervisor is essential for software-based memory virtualization, it is not mandatory and disabled by default in hardware-assisted one such as Intel EPT (Extended Page Tables) [32] and AMD RVI (Rapid Virtualization Indexing) [5]. Nonetheless, tracking address space switches is still feasible simply by enabling VMEXIT in the event of a CR3 update [19]. Although additional VMEXITs come at a cost of hypervisor intervention, improvement achieved by vAMP (hundreds of milliseconds to a few seconds shown in Section 4.3) outweighs VMEXIT overhead of hundreds of CPU cycles [4] by many orders of magnitude. Furthermore, CR3 trap can be selectively enabled when a VM is likely to take advantage of vAMP (e.g., while user interactions frequently take place with high background load). Such optimization can avoid the VMEXIT overhead while interactive workloads are solely running.

The algorithm of background task identification is illustrated in Figure 2 and performs as follows. Per-task CPU loads are maintained for each vCPU within a certain time window, named *pre-I/O period*; task execution time is accumulated by sub-epoch in a round-robin manner. Once a user I/O event occurs, per-task CPU loads are examined. If a task has generated higher CPU load than *background load threshold* (`bgload_thresh`), it is tagged as a background task. This threshold is used to consider only nontrivial loads that likely disturb a following interactive workload. More importantly, it helps filter daemon tasks (e.g., X server) that generate low CPU loads and possibly serve the invoked interactive tasks. After tagging tasks, the algorithm starts an interactive episode during which non-background tasks are assumed to be potentially interactive workloads. The episode is finished when another user input occurs or the episode lasts for *maximum interactive period* (`max_int_period`), which is long enough to generally cover a user-interactive period. After the episode ends, background tasks return to normal by removing their background tags.

When checking per-task loads, the algorithm also accounts *stolen time* as a potential load. Stolen time is the time taken to wait on a runqueue since a vCPU became runnable; the CPU time, which would otherwise be consumed without time-sharing, is stolen by another one during the wait-time. For example, a 100% CPU-bound task shows only 20% CPU load when running on a vCPU that contends with four CPU-bound vCPUs. Since stolen time is measured per vCPU, it should be accounted to existing tasks on a vCPU. In our algorithm, per-vCPU stolen time is accounted to a task in proportion to the amount of CPU time it has actually consumed while running on a vCPU.

For user-interactive I/O detection, the hypervisor extension monitors keyboard and mouse events as user input, and audio device accesses as user output by default. Most GUI-based applications in virtual desktops are interacted with a user via keyboard and mouse input devices. In addition, multimedia applications such as a media player typically involve audio output. Therefore, the default setting generally covers user interactions with virtual desktops. Other types of user I/O (e.g., user interaction via *ssh*) can also be monitored by simply interposing I/O virtualization layer; a packet with a well-known port number (e.g., 22 for ssh) can be inspected at network virtualization layer.

Inspecting per-task CPU loads on every user I/O may adversely affect the response time of a user event itself. For example, fast keyboard typing can degrade user-perceived responsiveness by repeatedly inspecting per-task loads on every keyboard input. To reduce the overheads, specific key codes are registered as a signal of when an interactive episode begins. In our current prototype, the Enter key is registered by default, since pressing it typically triggers interactive loads such as application launch and Web browsing. Regarding mouse inputs, the release event of a click is regarded as a start signal. Finally, since an audio device is intensively accessed for audio playback, it is heavyweight to start a new interactive episode by inspecting per-task loads on every audio output. To address this problem, once an audio device access initiates an interactive episode, subsequent accesses are ignored until the episode is finished.

3.1.3 Multimedia Task Identification

Although interactive CPU loads typically start to be generated in response to a user input, a user output may not be a start signal of CPU loads for output-based interactive workloads, which generate audio/video output to interact with a user. For example, a media player continuously generates CPU load required for video decoding and rendering while accessing audio/video devices. Hence, the media player task could be misidentified as a background workload by our scheme if its CPU load is higher than bgload_thresh. In order to prevent this misidentification, output-based multimedia tasks should be excluded from background ones.

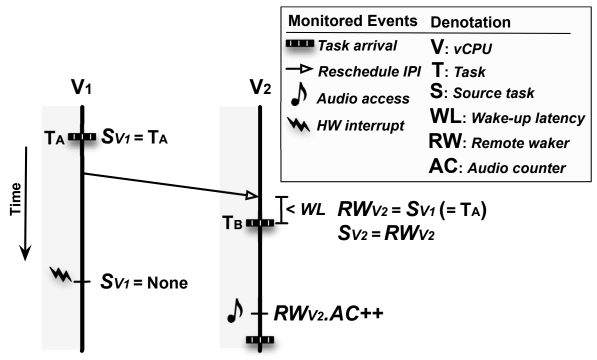


Figure 3. Remote wake-up tracking for identifying sound clients. In this example, T_A and T_B is a sound client and server, respectively.

A simple solution is to exclude audio-generating tasks from background workloads, since multimedia applications generally accompany audio output [40]. To this end, the hypervisor can consider a task that is running when audio device is accessed as an audio-generating task. Since an audio device, however, can also be accessed in audio interrupt handler, which is asynchronously invoked, such an audio device access cannot be associated with a currently running task. As in [20], our scheme does not associate a currently running task with an audio device access in the interrupt handler within which an audio interrupt vector bit is set in the interrupt service register of a corresponding vCPU.

The simple method, however, cannot identify the multimedia tasks that indirectly generate audio output through server/client-based sound systems [27] where a user-level sound server is in charge of all audio accesses requested from other tasks. In most Linux OS distributions, *pulseaudio* [2] is a user-level sound server by default for the sake of compatible and flexible sound service [27]. In this case, the current solution can identify only the sound server as a multimedia task, while missing actual audio-requesting tasks (i.e., sound clients). It is challenging to exactly identify a sound client, since server-client communication is done by IPCs without hypervisor intervention.

Our scheme is enhanced to identify sound clients as well by introducing *remote wake-up tracking*, which is a statistical method based on IPIs between vCPUs. IPI is a signaling mechanism used to send a message to a remote CPU (vCPU in our case). In particular, *reschedule IPI* is used for a vCPU to wake up a task on a remote vCPU and notify it of

[2] `www.pulseaudio.org`

the newly runnable task [21]. If a client sends a request to a server via IPC, this communication could involve a reschedule IPI if the server is woken up on a remote vCPU. Since the hypervisor is in charge of delivering IPIs between vCPUs, it can infer server-client communication by tracking such IPI-driven remote wake-up. Although this scheme cannot track local communication on the same vCPU, remote wake-up more frequently happens than local wake-up in a multiprocessor VM so that the communication can be captured statistically.

Figure 3 illustrates how the remote wake-up tracking works in the case of a server/client-based sound system. Each vCPU (V_i) maintains a source task (S_{V_i}), which holds a currently running task (e.g., $S_{V_1} = T_A$) updated at each task arrival (i.e., scheduling). If V_1 sends V_2 a reschedule IPI, the hypervisor checks whether a task (T_B) is woken up on V_2 immediately within *wake-up latency* (WL) after the IPI being injected; we set WL to 20μs by profiling the time between reschedule IPI injection and its corresponding task wake-up using a micro-benchmark. If so, V_2 sets its remote waker (RW_{V_2}) to S_{V_1} ($= T_A$) based on the assumption that T_A and T_B are the source and destination of the communication, respectively. At this point, S_{V_2} is set to RW_{V_2} in order to maintain an initial source task of the communication in the case of consecutive remote wake-ups across more than two vCPUs. Such consecutive remote wake-ups can happen if a sound server or client is multithreaded; pulseaudio is multithreaded. Finally, if T_B accesses an audio device during this time slice on V_2, the remote waker's audio counter ($RW_{V_2}.AC = T_A.AC$) is incremented. In this manner, T_A and T_B are regarded as a sound client and server, respectively.

As illustrated in Figure 3, S_{V_i} is updated as `None` when an external interrupt is injected to V_i (V_1 in the figure). A sound server can be remotely woken up by an audio interrupt handler, which is invoked asynchronously regardless of any task context. In this case, a task currently running on a vCPU where the handler is invoked may not be a sound client. Without this update, we observed that the audio counter of a background task is often falsely incremented.

To consider recent audio activity, an audio counter is moving-averaged at every interactive episode. In the case where a multimedia application is generating audio output without any user input, it is averaged at every `max_int_period`. In addition, before calculating the average, the audio counter of a task is divided by how many times the task is tagged as background in order to curtail the false increment of background tasks. To this end, *background confidence* is incremented when it is tagged as a background task and is decremented otherwise; the maximum background confidence is 10 by default.

Using this tracking, the hypervisor can statistically filter sound clients out of background tasks on the basis of their audio counters. If a task is a sound client that continuously sends requests to a sound server, it could have relatively larger audio counter than others. This property enables a threshold-based classification. In our prototype, we use a simple policy that filters a dominant sound client with the highest audio counter. This simple policy is reasonable in practice, since it is uncommon to simultaneously run multiple sound clients.

3.2 Asymmetric Share Distribution

After identifying background tasks excluding multimedia ones, a VM's shares start to be asymmetrically distributed to active vCPUs during an interactive episode. Since a vCPU can host either a background or non-background task at any given time, per-vCPU shares are dynamically adjusted during the period. To this end, as shown in Figure 2, every time a background task arrives at a vCPU during an interactive episode, the vAMP hypervisor extension gives the scheduler a command to throttle the weight of the vCPU to which relatively smaller shares are distributed. Conversely, once a non-background task is switched back, the weight is restored. By doing so, the vCPUs that host potentially interactive tasks (i.e., non-background tasks) have more computing power as fast vCPUs to improve interactive performance.

Although the asymmetric share distribution is effective for improving interactive performance, there is an intrinsic limitation arising from manipulating only a single scheduling layer (i.e., the hypervisor scheduler). With the hypervisor extension alone, guest OS schedulers are agnostic about the presence of underlying vAMP. Commodity OSes typically assume their online CPUs to have identical computing power. As real AMP has drawn attention as an energy-efficient architecture, several efforts have been done to make OS schedulers aware of performance-asymmetric computing cores [29]. They have focused on, however, which workloads are beneficially scheduled on fast cores that are statically wired in architecture. Moreover, most proposals aimed at high throughput per watt for throughput-oriented workloads without considering interactive workloads.

An OS scheduler can make an ineffective decision without the knowledge of vAMP. A vAMP-oblivious OS scheduler may schedule an interactive task on a slow vCPU, which has smaller shares. In PS-based schedulers, a vCPU with smaller shares suffers higher scheduling latency while waiting its turn until those with larger shares exhaust their time slices. Therefore, the scheduling latency of the interactive task is increased when it is multiplexed with a background task on the same vCPU. Furthermore, an interactive task is frequently scheduled on a slow vCPU when a background workload tries to occupy all available vCPUs. Such workload mix is common in a multiprocessor VM, since many throughput-oriented applications are parallelized and commonly run with the configuration to use all available cores for high throughput.

In order to prevent the ineffective multiplexing from offsetting the benefit of vAMP, the weight ratio between slow and fast vCPUs should be carefully chosen. Since the hypervisor is unaware of where a guest OS will schedule its task, it is challenging to completely avoid the adverse effect of the multiplexing. Instead, a conservative setting for the ratio can reduce the extent of adverse effect while achieving high interactive performance. From the evaluations in Section 4, vAMP shows large and stable improvement with conservative weight ratio (e.g., 1:3).

3.3 Guest OS Extension for vAMP

To eliminate the intrinsic limitation of the hypervisor-only scheme, a guest OS can be enlightened about vAMP. We propose a lightweight OS extension by which vAMP becomes more effective. The design goals of the guest OS extension are as follows. Firstly, the OS extension is not mandatory for vAMP so that it can be optionally installed in an OS for further improvement of interactive performance. With this goal, main decisions for vAMP are still made by the hypervisor while the OS extension plays a role of reducing adverse effect of OS scheduling. Since the adverse effects stem from multiplexing interactive and background workloads on a vCPU, its primary role is to isolate interactive tasks from background ones on separate vCPUs during an interactive episode. Secondly, kernel changes are kept as small as possible for low maintenance cost. To this end, the primary functionalities of the extension are performed in user space, to which the kernel exports necessary information. Most OSes commonly provide user land with a knob to place a task on a specific set of CPUs (e.g., *cpuset* [23] and *sched_setaffinity* in Linux, and *SetThreadAffinityMask* in Windows).

Keeping the design goals in mind, we implemented a Linux extension for vAMP. To isolate non-background workloads from background ones, the kernel exposes the list of background tasks identified by the hypervisor extension to user space via *procfs*. The kernel obtains the information from the structure shared with the hypervisor. This small addition requires a few lines of source codes in the kernel. A user space extension, named *vamp-daemon*, refers to the exposed list of background tasks and carries out the isolation between background and non-background tasks. The vamp-daemon performs by responding to user I/O in an event-driven fashion. To this end, it listens the Linux input interface (i.e., `/dev/input/eventN`) for user inputs, while communicating with a sound server (pulseaudio in our case) via IPC for audio output; *pacmd*, which is a helper tool for pulseaudio, was used to inspect audio output generation. Finally, the vamp-daemon performs isolation by using cpuset [23], which provides a filesystem-based interface for user space to manipulate CPU affinity of a group of tasks.

In addition to the task placement, interrupts from I/O devices that can serve interactive workloads are isolated to fast vCPUs where non-background tasks are placed. Since interactive workloads can involve disk or network I/O operations during an interactive episode, delivering an interrupt from such an I/O device to a slow vCPU could increase the latency of interrupt delivery. To solve this problem, interrupts from I/O devices that are involved in user interaction are isolated to fast vCPUs. In the Linux extension, it can be simply done by vamp-daemon via `/proc/irq/number/smp_affinity`.

The guest OS extension has an isolation policy of how vCPUs are partitioned (i.e., how many vCPUs are dedicated to interactive tasks). Interactive workloads typically require a small number of vCPUs, since they have low thread-level parallelism in general [7, 13]. Based on this characteristic, one or two vCPUs are enough to be initially dedicated to interactive tasks, while remaining vCPUs run background ones. In the case of audio output, the number of vCPUs for interactive tasks is set at least two in order for the hypervisor to track remote wake-up between a sound server and client. Interactive workloads may need more vCPUs as their given vCPUs become saturated involving in multiple tasks and threads. To address such demand, the vamp-daemon periodically monitors the CPU utilization of the vCPUs that host interactive workloads, so that it provisions an additional vCPU when they are fully utilized. This periodic monitoring also detects the end of an interactive episode at which all tasks become normal and therefore isolation is revoked. In our current vamp-daemon, an initial number of vCPUs for interactive tasks and a monitoring interval are given as configurable parameters.

One possible opportunity is that the guest OS extension performs workload identification and forwards the result of it to the hypervisor. Since a guest OS can access much more information about workloads than hypervisor, the guest-side workload identification could more precisely guide vAMP to boost interactive performance. In this case, only the substrate that adjusts vCPU shares is implemented in the hypervisor while any policy to use the substrate is implemented in guest OSes. As a simple method, task priority can be provided to hypervisor as a hint [17]. This type of separation could enhance accuracy and modularity while eliminating the overheads for hypervisor-level task tracking, but requires the guest OS extension to be mandatory, not optional. In this work, we aim to have vAMP not rely on guest OS extension, but benefit from it if any. We will explore guest-side workload identification for vAMP in the future work.

4. Evaluation

The vAMP hypervisor extension was implemented based on the KVM, a Linux module for virtualization, in Linux kernel 3.0.0. In the KVM-based virtualization, I/O requests from guest OSes are handled by a QEMU (version 1.0) running in the host Linux. Once QEMU handles a predefined user I/O event, it notifies the vAMP hypervisor extension, which then initiates an interactive episode. The extension adjusts vCPU weight interacting with Completely Fair Scheduler (CFS), which is the default PS-based scheduler in Linux. KVM uses CFS group scheduling [23], which allows multiple threads to share per-group (per-VM) shares; all threads of vCPUs and QEMU in the same VM are grouped in the VM's share budget. We set per-VM shares to the default shares (1024) multiplied by the number of pCPUs.

The prototype was installed on Dell PowerEdge R410, equipped with a quad-core Intel Xeon X5550 2.67GHz processor and 8GB RAM. In this setting, eight pCPUs are available with hyperthreading enabled. We used Ubuntu 11.04 with Linux kernel 3.0.0 as a guest OS and gave each VM eight vCPUs and 3.5GB memory. A remote desktop client ran on a separate machine (identical hardware with the server) connected through a 1Gbps Ethernet switch.

4.1 Measurement Methodology

A KVM-based desktop environment is effectively built by means of SPICE, a remote desktop solution optimized to access virtualized hardware. A SPICE server handles user I/O requests from remote SPICE clients by interacting with QEMU. Since consolidated virtual desktops are remotely accessed by end-users in general, we evaluated interactive performance at a client side.

To measure interactive performance, user-perceived response time should be properly quantified. Previous work used snapshot-based record/replay for robust replay of user-recorded interactive sessions [28, 39]. In the case where system loads (or computing power) may be different between the times of record and replay, snapshot can be used as a user-perceived (or synchronization) point to avoid unsynchronized replay. We implemented such measurement functionality in a SPICE client. As with *DeskBench* [28], at every interaction to be evaluated, a user can explicitly mark the completion of perception (by inserting a special key), at which a snapshot is recorded. During an evaluation, recorded user inputs are replayed by synchronizing the their corresponding snapshots; similar to fuzzy matching [28], small snapshot differences are allowed to continue a replay for reducing replay failure due to unexpected screen update.

We evaluated a multimedia workload using the VLC open-source media player. In a video playback workload, the performance metric is displayed frames per second (FPS). Since a media player drops frames when it cannot meet the defined rate of a video, FPS is used to evaluate how well a CPU scheduler satisfies the computing demand for video playback. We measured FPS at the server side in order to evaluate the CPU scheduling impact, because client-side performance of video playback highly depends on remote desktop protocols. Protocol supports of graphical operation offloading affects the performance of client-side video quality.

The evaluation of interactive performance was done in a consolidated environment where VMs are competing for available pCPUs. Since evaluated workloads have different completion times from each other, we fully overlapped each workload by repeated executions and considered only the overlapped runs for averaged results. Each throughput-oriented workload was repeatedly run at least five times (ten times in the case of the Web browsing evaluation), while an interactive session made progress.

Parameter	Role	Default
Pre-I/O period	Background task identification	1024msec
bgload_thresh	Background task identification	50%
max_int_period	Duration of an interactive episode	5sec

Table 1. The vAMP parameters, their roles, and default values used in the evaluations.

4.2 Parameters

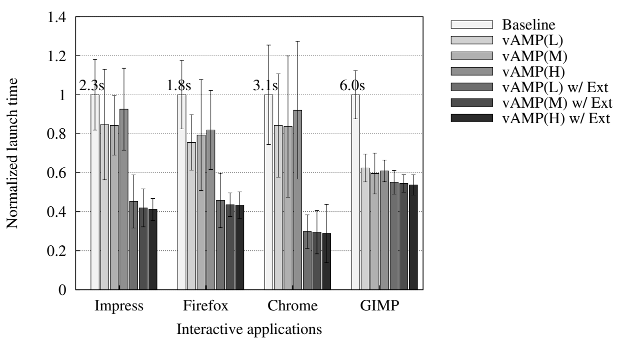
Table 1 summarizes the parameters, their roles, and default values for the vAMP hypervisor extension. The pre-I/O period and bgload_thresh are used to identify background tasks. The main role of bgload_thresh is to prevent a daemon task that is supposed to service interactive workloads from being misidentified as a background task. Since such daemons typically have low CPU utilization in the common case while handling requests in an event-driven manner, the threshold can be empirically set to a value higher than the maximum achievable CPU utilization of well-known daemon tasks. One thing to consider is a display service daemon such as an X server, since display can service both background and non-background ones [11, 40]. We determined the default value of bgload_thresh considering the highest CPU utilization of the display daemon; in our case, an X server showed less than 50% CPU utilization during the high rate of display service from video playback.

Our asymmetric share distribution is not sensitive to max_int_period as long as it sufficiently covers a general interactive episode. If an interactive workload is prematurely finished and thus background tasks solely run until the end of an interactive episode, all vCPUs have eventually the equal shares. Although previous research suggested that two seconds are reasonable as an interactive period [26, 30, 40], we used five seconds by default considering that modern interactive applications show relatively longer response time such as application launch.

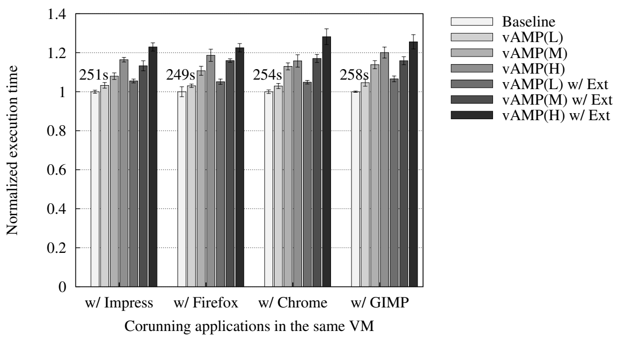
As mentioned in Section 3.2, the weight ratio of slow and fast vCPUs can affect the interactive performance. A higher ratio can improve interactive performance if interactive tasks are scheduled on fast vCPUs, whereas it may degrade the performance otherwise. The Linux CFS scheduler associates pre-set shares with each *nice* value [22], with which vAMP adjusts weight. We used three configurations for the weight ratio: low (`L`), medium (`M`), and high (`H`) ratios, which approximately match 1:3, 1:9, and 1:18, respectively. With these configurations, we show the performance impact by different weight ratios.

4.3 Application Launch

Application launch is a representative interactive workload where desktop users desire low response time. Many GUI-based applications require considerable CPU bandwidth for initialization during launch period. We used *LibreOffice Im-*



(a) The Normalized average launch time of interactive tasks



(b) The normalized average execution time of background tasks (freqmine)

Figure 4. Application launch time and the execution time of background workloads mixed with launched applications (error bars represent standard deviations and the value over a box is the elapsed time in each baseline case).

press, *Mozilla Firefox*, *Google Chrome*, and *GNU Image Manipulation Program (GIMP)* for the evaluation. Each application was repeatedly launched and closed at a user-perceived point. We inserted one-second interval between close and relaunch during replay. We used freqmine, a multi-threaded data mining application in the PARSEC benchmark suite [6], as a background workload; this type of application (e.g., data mining, data analysis, simulation, encoding) represents the workloads of knowledge workers [33]. Among PARSEC applications, freqmine is one of CPU-bound applications with a little communication while fully utilizing all vCPUs. We ran freqmine with eight threads each in two 8-vCPU VMs while repeated application launch was performed in one of the VMs. For comparison, we used the default CFS scheduler (vSMP) as the baseline.

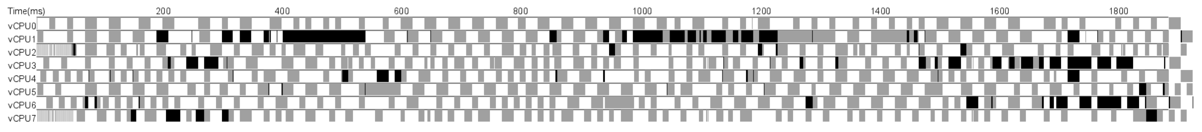
Figure 4(a) shows the average launch time of each interactive application. `Ext` stands for our guest OS extension, in which one vCPU is initially assigned for interactive tasks and, if fully utilized, is increased at every one second. As shown in the figure, the vAMP hypervisor extension alone improves the average launch time by up to 41%, while achieving further improvement by up to 70% if assisted by the guest OS extension. One important thing to note is that higher weight ratio shows less improvement without the guest OS extension. In addition, the applications that show such adverse effect by higher weight ratio (Impress, Firefox, and Chrome) show significant performance improvement in the case where the guest OS extension assists vAMP. On the other hand, GIMP shows larger improvement by about 40% even without the guest OS extension, which has a little impact in this case.

As mentioned in Section 3.2, the vAMP hypervisor extension alone may have negative effect when an OS scheduler multiplexes interactive and background tasks on a single vCPU. Since higher weight ratio leads to larger scheduling delay of a slow vCPU, the negative effect could offset the benefit of vAMP if such multiplexing frequently happens. In order to show the multiplexing behavior of a vAMP-oblivious OS scheduler, we obtained the scheduling traces of Chrome and GIMP launches. Figure 5 shows the scheduling traces where gray and black colors represent background and non-background tasks, respectively. In the figure, Chrome shows frequent multiplexing of background and non-background tasks, whereas GIMP shows that different types of tasks are mostly isolated on separate vCPUs for its launch time.

These behaviors come from the different characteristics of the applications. Chrome involves many threads in a fine-grained manner during launch so that the threads spread over multiple vCPUs with communicating with each other. By this characteristic, ineffective multiplexing with background tasks frequently happens. In GIMP, on the other side, one thread is dominantly compute-intensive conducting most of the jobs for launch, while the others perform small computation. Hence, the compute-intensive thread occupies one vCPU for most of its launch time, thereby leading to spontaneous isolation from background tasks (except for the last 500ms). Such isolation results in large performance improvement even without the guest OS extension.

We also measured how much the performance of a background workload is degraded by throttling the shares of its hosting vCPU (i.e., slow vCPU). Figure 4(b) shows the average execution time of freqmine mixed with each launched application. Note that the performance degradation of background workloads depends on how intensively an interactive application requires computation. As mentioned earlier, we used a 1-second interval between close and relaunch to simulate highly interactive workloads. As shown in the figure, the performance of freqmine is degraded by 3–20%. The higher weight ratio is used, the more degradation is observed due to the smaller shares of slow vCPUs. With the same weight ratio, the guest OS extension more degrades the performance of freqmine by preventing it from using the vCPU that is dedicated to interactive tasks during an interactive episode. The degradation of background workloads is reasonable considering the highly interactive workloads and their significant improvement by up to 70%.

Finally, we evaluated the impact of I/O interrupt isolation, which forwards an I/O interrupt to a fast vCPU during


(a) Chrome

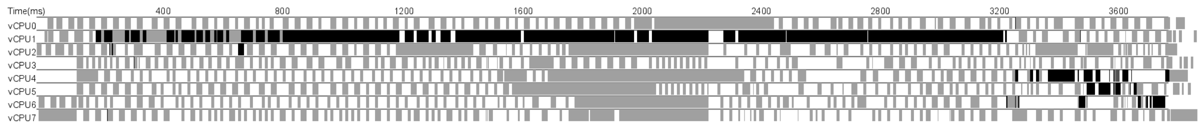

(b) GIMP

Figure 5. Scheduling traces during launch for Chrome and GIMP; gray and black colors represent the execution of a background task (freqmine) and a non-background task, respectively. No color means that a vCPU is not scheduled on a pCPU due to the time-sharing with another VM, which runs freqmine.

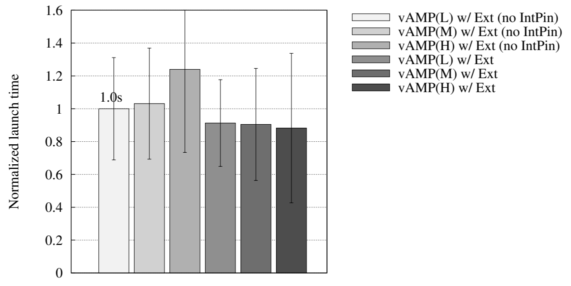


Figure 6. The normalized launch time of Chrome for vAMP with the guest OS extension. `IntPin` represents the interrupt isolation scheme in the guest OS extension.

an interactive episode. We chose Chrome for this evaluation, since it involves synchronous file writes during launch. The synchronous file write requires disk I/O even for warm launch, in which file reads are served from disk cache without I/O. Figure 6 shows the normalized launch time of Chrome for vAMP with our guest OS extension; `IntPin` means the I/O interrupt isolation (pinning). As shown in the figure, without the interrupt isolation, higher weight ratio worsens the launch performance, since the interrupt delivery of synchronous file writes can be delayed if a disk interrupt is pending to a slow vCPU. The isolation of disk interrupts resolves this problem by ensuring that the interrupts are delivered to fast vCPUs, which are scheduled with shorter latency.

4.4 Web Browsing

Web browsing has been a prevalent interactive workload as desktop users usually have their jobs done by using Web and cloud services. Web browsing is computationally intensive during an interactive period especially for rendering a Web page with complex structures and images. For evaluation, we used the contents of Web sites in *bbench* and placed them in a VM local disk in order to preclude the effects of network I/O. An interactive session of Web browsing consecutively visits ten Web sites: Amazon, BBC, CNN, Craigslist, eBay, ESPN, Google, MSN, Slashdot, and Twitter. [3] We used Firefox as a web browser and inserted three-second interval between visits.

Figure 7 shows the normalized response time of browsing each Web site in the same consolidation scenario as the evaluation of application launch. For the average, the vAMP hypervisor extension improves the browsing time by up to 31% while achieving the further improvement by up to 41% if assisted by the guest OS extension. As with the result of application launch, higher weight ratio without the guest OS extension shows less performance. In the case of Amazon and Google pages, the baseline outperforms vAMP(H) that is not assisted by the guest OS extension. This result implies that negative effect of multiplexing outweighs the benefit of vAMP. Considering the results of application launch and Web browsing, vAMP with conservative weight ratio, vAMP(L), achieves stable and noticeable performance improvement both for the hypervisor and guest OS extensions.

4.5 Media Player

As an output-based interactive workload, we evaluated a video playback application by using the VLC media player. For a video playback workload, the requirement of CPU bandwidth relies on video contents. For highly compute-intensive video playback, we chose a high definition video clip with 1920x800 resolution, *The Simpsons Movie Official Trailer HD*; its running time and FPS are 137 seconds and 23.976 FPS. Since it shows higher CPU utilization than bgload_thresh (50%), it can be misidentified as background workloads without the multimedia workload identification. Regarding audio playback, the VLC media player requests pulseaudio to generate the audio output of the video clip.

Figure 8 shows the video playback quality (FPS) in the same consolidation scenario as the evaluation of application launch. `Mult` means our multimedia identification scheme, and percentage over each bar represents a drop rate (= dropped frames × 100 / total frames). In the baseline case,

[3] YouTube was excluded for robust replaying, since it automatically plays a video when visited.

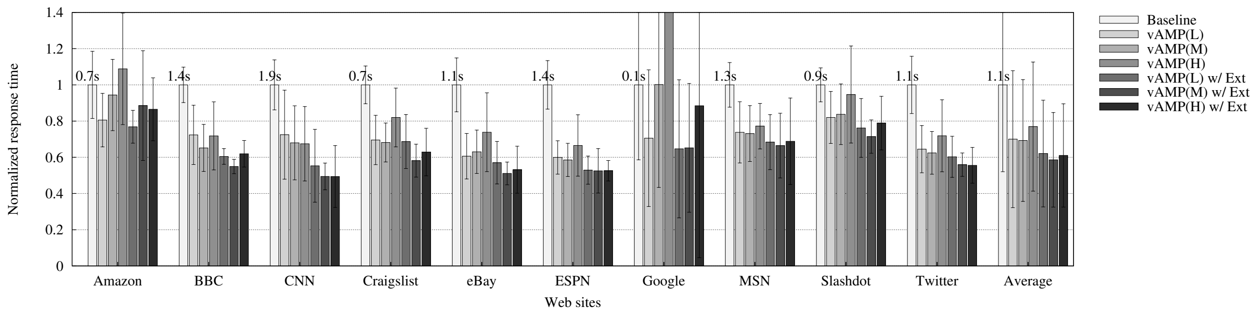


Figure 7. Normalized browsing time for each Web site in bbench. The error bars represent standard deviations and the value over a box is the elapsed time in each baseline case.

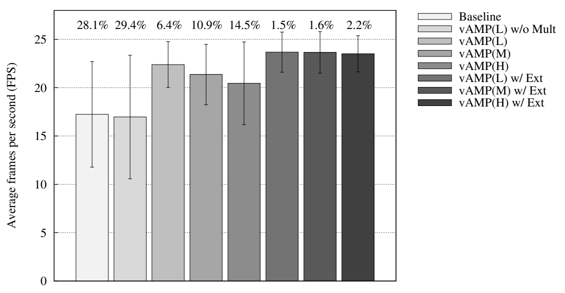


Figure 8. The displayed FPS of a 1920x800 video with 23.976 FPS. `Mult` means the multimedia identification support, and percentage over each bar is a drop rate.

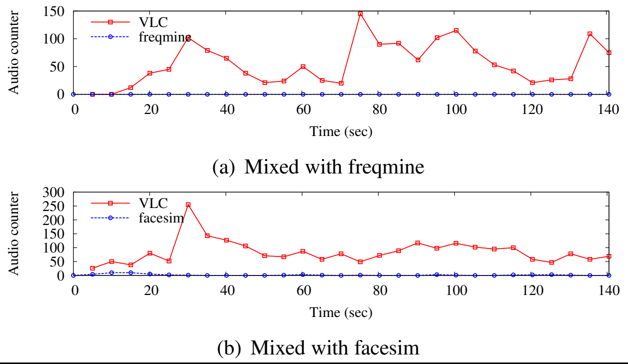


Figure 9. Audio counters during video playback

vSMP significantly degrades video quality with 28.1% drop rate. Without the multimedia identification scheme, vAMP also shows similar degradation by classifying the CPU-intensive VLC as a background task. Once the multimedia identification filters VLC out of background workloads, the video playback quality is improved. Without the guest OS extension, as with the other evaluations, higher weight ratio shows less improvement due to the negative effect of multiplexing of video playback and background tasks on a single vCPU. The guest OS extension remedies the problem by achieving much better quality with only 1.5–2.2% drop rate.

To figure out how well VLC is filtered out of background workloads, Figure 9(a) shows the moving-averaged audio counter as time progresses. In the figure, VLC has always much higher audio counter than freqmine so that it can be identified as a dominant sound client. We also ran VLC with *facesim*, which intensively generates reschedule IPIs for the repeated wake-up of worker threads, to disturb our remote wake-up tracking. In Figure 9(b), VLC also shows higher audio counter than facesim, which increases its audio counter only up to ten. As a result, our remote wake-up tracking effectively filters the multimedia workloads that indirectly generate audio output through server/client sound systems.

5. Concluding Remarks

In this paper, we present the design and implementation of vAMP for improving user-interactive performance of consolidated desktops. For differentiating the computing power of vCPUs, we devised a simple and effective scheme for identifying background workloads at the hypervisor layer. One important finding is that manipulating only the hypervisor scheduler for interactive performance limits the improvement due to the negative effect of multiplexing of interactive and background tasks on a single vCPU. To address this limitation, the lightweight guest OS extension prevents such ineffective multiplexing, thereby achieving significant improvement of interactive performance. With the hypervisor and guest OS extensions for vAMP, the consolidated desktops enable high quality of user experience while compute-intensive jobs are concurrently running with interactive workloads. We plan to investigate collaborative scheduling between the hypervisor and guest OSes for vAMP. In addition, we are exploring the applicability of vAMP for various workloads other than interactive applications.

Acknowledgments

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2013-003871), and by the IT R&D program of MKE/KEIT [10041244, Smart TV 2.0 Software Platform].

References

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

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

[3] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proc. of ASPLOS*, 2006.

[4] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proc. of USENIX Annual Technical Conference*, 2012.

[5] AMD. AMD64 virtualization codenamed "pacifica" technology: Secure virtual machine architecture reference manual, May 2005.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. of PACT*, 2008.

[7] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *Proc. of ISCA*, 2010.

[8] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proc. of HotOS*, 2001.

[9] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007. ISSN 0163-5999.

[10] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. of SOSP*, 1999.

[11] Y. Etsion, D. Tsafrir, and D. G. Feitelson. Process prioritization using output production: Scheduling for multimedia. *ACM TOMCCAP*, 2(4):318–342, 2006. ISSN 1551-6857.

[12] K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for linux. In *Proc. of OSDI*, 2002.

[13] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism and interactive performance of desktop applications. In *Proc. of ASPLOS*, 2000.

[14] P. Goyal, X. Guo, and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proc. of OSDI*, 1996.

[15] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Proc. of Middleware*, 2006.

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

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

[18] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for I/O performance. In *Proc. of VEE*, 2009.

[19] H. Kim, H. Lim, J. Jeong, H. Jo, J. Lee, and S. Maeng. Transparently bridging semantic gap in cpu management for virtualized environments. *JPDC*, 71(6):758 – 773, 2011. ISSN 0743-7315.

[20] H. Kim, J. Jeong, J. Hwang, J. Lee, and S. Maeng. Scheduler support for video-oriented multimedia on client-side virtualization. In *Proc. of MMSys*, 2012.

[21] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-based coordinated scheduling for SMP VMs. In *Proc. of ASPLOS*, 2013.

[22] R. Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010. ISBN 0672329468, 9780672329463.

[23] P. B. Menage. Adding generic process containers to the linux kernel. In *Proc. of OLS*, 2007.

[24] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: operating system support for multimedia applications. In *Proc. of ICMCS*, 1994.

[25] J. Nieh and M. S. Lam. A SMART scheduler for multimedia applications. *ACM TOCS*, 21(2):117–163, 2003. ISSN 0734-2071.

[26] J. Nielsen. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, Thousand Oaks, CA, USA, 1999. ISBN 156205810X.

[27] L. Poettering. Cleaning up the linux desktop audio mess. In *Proc. of OLS*, 2007.

[28] J. Rhee, A. Kochut, and K. Beaty. Deskbench: Flexible virtual desktop benchmarking toolkit. In *Proc. of IM*, 2009.

[29] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proc. of EuroSys*, 2010.

[30] B. Shneiderman. Response time and display rate in human performance with computers. *ACM Comput. Surv.*, 16(3): 265–285, Sept. 1984. ISSN 0360-0300.

[31] G. Somani and S. Chaudhary. Application performance isolation in virtualization. In *Proc. of CLOUD*, 2009.

[32] 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*, 38 (5):48–56, 2005. ISSN 0018-9162.

[33] VMware. VMware Infrastructure 3: VDI server sizing and scaling, May 2006.

[34] VMWare. VMware, Inc. VMware vSphere 4: The CPU scheduler in VMware ESX 4.1. Technical report, 2010.

[35] VMware Inc. Enabling your end-to-end virtualization solution. `http://www.vmware.com/solutions/partners/alliances/hp-vmware-customers.html`.

[36] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of OSDI*, 1994.

[37] C. A. Waldspurger and E. Weihl. W. Stride scheduling: Deterministic proportional- share resource management. Technical report, Cambridge, MA, USA, 1995.

[38] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Redline: first class support for interactivity in commodity operating systems. In *Proc. of OSDI*, 2008.

[39] N. Zeldovich and R. Chandra. Interactive performance measurement with vncplay. In *Proc. of USENIX Annual Technical Conference*, 2005.

[40] H. Zheng and J. Nieh. RSIO: Automatic user interaction detection and scheduling. In *Proc. of SIGMETRICS*, 2010.