Accelerating Critical OS Services in Virtualized Systems with Flexible Micro-sliced Cores

Jeongseob Ahn
Ajou University
jsahn@ajou.ac.kr

Chang Hyun Park
KAIST
changhyunpark@calab.kaist.ac.kr

Taekyung Heo
KAIST
tkheo@calab.kaist.ac.kr

Jaehyuk Huh
KAIST
jhhuh@kaist.ac.kr

ABSTRACT

Consolidating multiple virtual machines into a single server has been widely adopted in cloud computing to improve system utilization. However, the sharing of physical CPUs among virtual machines in consolidated systems poses a new challenge in providing an illusion of continuous CPU execution to the guest operating systems (OS). Due to the time-sharing of physical CPUs, the execution of a guest OS is periodically interrupted, while the guest OS may not be aware of the discontinuity of virtual time against the real time. The virtual time discontinuity problem causes the delayed processing of critical OS operations, such as interrupt handling and lock processing. Although there have been several prior studies to mitigate the problem, they address only a subset of symptoms, require the modification of guest OSes, or change the processor architecture. This paper proposes a novel way to comprehensively reduce the inefficiency of guest OS execution in consolidated systems. It migrates the short-lived critical OS tasks to dedicated micro-sliced cores, minimizing the delays caused by time sharing. The hypervisor identifies the critical OS tasks without any OS intervention, and schedules the critical code sections onto the dynamically partitioned cores at runtime. The dedicated micro-sliced cores employ a short sub-millisecond quantum to minimize the response latencies for consolidated virtual machines. By readily servicing the critical tasks, the proposed scheme can minimize the adverse artifact of virtual machine consolidation without any modification of guest OSes.

CCS CONCEPTS

• **Software and its engineering** → *Virtual machines*; *Scheduling*;
• **Computer systems organization** → Cloud computing;

KEYWORDS

Virtualization, Virtual Time Discontinuity, Scheduling

ACM Reference Format:
Jeongseob Ahn, Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. 2018. Accelerating Critical OS Services in Virtualized Systems with Flexible Micro-sliced Cores. In *Proceedings of Thirteenth EuroSys Conference 2018 (EuroSys '18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3190508.3190521

1 INTRODUCTION

In cloud computing, the consolidation of multiple virtual machines (VMs) with fluctuating loads reduces the computing cost by improving the overall system utilization. However, in such virtualized systems, guest operating systems often suffer from the scheduling artifact of virtualization. Operating systems have been designed to run directly on the hardware system, fully controlling the underlying CPU resources. However, in virtualized systems, physical CPUs (pCPUs) are time-shared by multiple virtual CPUs, and thus the execution of guest operating systems can be interrupted by the CPU scheduling decision of the hypervisor.

This problem, known as the virtual time discontinuity problem, has been reported to cause significant inefficiency in highly consolidated systems [2, 7, 10, 14, 26, 28, 33]. It causes the delays of critical OS services in several aspects. First, it can delay critical kernel synchronizations implemented with spin locks. Second, inter-processor interrupts (IPI) are not immediately transferred to the destination cores, delaying interactions among kernel threads. Third, external interrupts for I/O processing are not served promptly as the virtual CPUs (vCPUs) to process them are not readily scheduled. Delaying such critical OS services significantly reduces the overall throughput of consolidated systems.

To mitigate the effect of the virtual time discontinuity problem, there have been several prior approaches. Scheduling-oriented approaches attempt to assign pCPUs as quickly as possible to the vCPUs processing the critical guest OS services, if the hypervisor can identify the events involved with kernel synchronization [10, 14, 26]. The scheduling quantum between context switches is shortened to more frequently schedule vCPUs onto the pCPUs, although it can potentially degrade the performance due to direct context switch overheads and cache pollution [2, 28, 34]. However, the prior approaches address only part of the virtual time discontinuity problem [10, 14, 26, 33], require changes in guest OSes [33], need HW support [2], or cannot handle the case when a VM has mixed behaviors that require both short and long time slices [7, 28].

This paper proposes a new holistic approach to address the virtual time discontinuity problem, comprehensively addressing

	vTurbo [33]	Demand-sched [14]	Fixed-μsliced [2]	vTRS [28]	vScale [7]	Our scheme
Inter-processor interrupt		✓	✓	✓	✓	✓
I/O interrupt	✓		✓	✓	✓	✓
I/O + CPU mixed	✓		✓			✓
Lock holder preemption		✓	✓	✓	✓	✓
Precise selection	✓	✓			✓	✓
Guest OS transparency		✓	✓	✓		✓
No additional hardware	✓	✓		✓	✓	✓

Table 1: Comparing `Flexible Micro-sliced Cores` **with prior approaches**

different aspects of the problem. The hypervisor transparently identifies the critical guest OS services without any modification of guest OSes, and accelerates the critical OS services on a subset of cores with sub-milliseconds time slices. Dedicating a subset of cores to serve critical guest OS services allows rapid processing of the critical code section. Furthermore, the dedicated *micro-sliced cores* employ an extremely short time slice of 0.1ms to minimize the response time for OS services. The kernel services are commonly short-lived, and processed within the short time slices. Even if many consolidated virtual machines demand to process their OS services, the micro-sliced kernel cores process them quickly without being hampered by the rest of user-level application activities.

When dedicating some cores for such critical OS services, an important challenge is to prevent wasting CPU resources. If the critical OS services are not invoked, the reserved cores will be idle, lowering the overall CPU utilization, delaying user applications. To accommodate such potentially fluctuating demands for the micro-sliced cores, this paper proposes a dynamic mechanism to adjust the number of micro-sliced cores based on the usage of the critical OS services.

The proposed micro-sliced cores differ from the prior approaches in several aspects. First, it can address the virtual time discontinuity problem in all three aspects, including synchronization, IPI, and external I/O handling. Second, it does not require modifications of the guest operating system and the CPU hardware. Furthermore, as the critical OS services are precisely chosen to be run on the micro-sliced cores, the rest of user-level application execution does not suffer from the negative impact of short time slices. Finally, even if a VM runs heterogeneous applications such as I/O intensive workloads along with cache-intensive applications, only the I/O handling code in the guest kernel is offloaded to the micro-sliced cores, avoiding the negative impact on cache-sensitive threads with short time slices. Table 1 summarizes the differences.

The proposed dynamic micro-sliced approach has been implemented in the Xen 4.7 hypervisor. The experimental results of various consolidated scenarios show that the proposed scheme can effectively mitigate the consolidation artifact of the virtualized system. For the `exim` mail server benchmark, a single micro-sliced core increases the throughput by 4.56x while only degrading 8% of the co-runner application. For the TLB synchronization intensive cases in `dedup` and `vips`, our scheme improves the aggregate throughput by 49% and 21%, respectively.

The new contributions of the paper are as follows.

- This paper identifies the critical OS services in virtualized systems. It shows that the execution of such critical OS services can be identified during runtime without any guest OS modifications.
- The paper employs micro-sliced cores dedicated to the critical OS execution. The guest-transparent migration of kernel execution to the micro-sliced cores effectively mitigates all three aspects of virtual time discontinuity problem.
- The paper proposes a dynamic mechanism to adjust the number of micro-sliced kernel cores, to prevent underutilizing CPU resources.

The rest of the paper is organized as follows. Section 2 introduces the virtual time discontinuity problem and prior approaches, comparing them to our work. Section 3 presents the critical OS services, and their performance implication. Section 4 describes the proposed design and Section 5 discusses the implementation details on the Xen hypervisor. Section 6 presents the experimental results, and Section 7 concludes the paper.

2 BACKGROUND AND DESIGN APPROACHES

2.1 Virtual Time Discontinuity Problem

In consolidated environments, virtual CPUs (vCPUs) share limited physical CPUs (pCPUs) to maximize the resource utilization. Although virtualization provides an illusion of continuous execution for guest operating systems, in reality their executions are interrupted because vCPUs are periodically scheduled out. Since the underlying hypervisor is not aware of the semantic of guest OS execution, vCPUs running critical OS services can be suspended. When the vCPUs of a guest VM are scheduled out by the hypervisor, the executing critical OS services such as interrupt handling can be suspended before completion. This violates the OS design assumption that urgent contexts, such as interrupt handlers or critical sections, will not be preempted; leading to significant degradation of system performance.

The discontinuity of virtual time causes many problems. First, spinlock mechanisms are handled inefficiently. If a vCPU holding a lock is scheduled out by the hypervisor, another vCPU waiting for the lock cannot make forward progress until the lock-holder vCPU is rescheduled and releases the lock (the lock holder preemption problem). This phenomenon is also observed in the RCU (Read-copy Update) synchronization [24]. In addition, the lock waiter preemption problem occurs with the FIFO ticket lock, where only the waiter with the subsequent ticket can acquire the lock. Even though the lock is not being held by any vCPUs, the vCPU with the next ticket can be preempted. Second, the discontinuity of virtual time slows down the handling of external and inter-processor interrupts. If the vCPU receiving an interrupt request is not running on a pCPU, the interrupt cannot be handled until the next scheduling turn of the vCPU. These problems share the same root cause, the *virtual time discontinuity (VTD)* problem [2].

2.2 Prior Work

There have been significant efforts throughout software and even hardware community to mitigate the performance degradation imposed by the virtual time discontinuity (VTD) problem.

Scheduler-based approaches: Co-scheduling addresses the virtual time discontinuity problem by scheduling all the sibling vCPUs of a VM at the same time [29]. However, the scalability of co-scheduling is limited by the increasing numbers of vCPUs per VM, as it requires synchronous scheduling of multiple vCPUs. Balanced scheduling was proposed to minimize the negative effect of co-scheduling [26]. Both of approaches can be used without guest-level information. Ding et al. proposed scheduler modifications of both the hypervisor and the guest OS to address the blocked waiter wakeup problem [10]. Prioritizing vCPUs which receive I/O interrupts has been widely adopted for the current hypervisor designs [21]. Advanced techniques to infer the semantics of guest OSes by examining SW/HW events were proposed [7, 14–16, 23].

Pool-based approaches: Pool-based approaches tackle the problem by arranging physical cores into different classes of cores. This leads to the dedication of physical cores for the vCPUs which experience the virtual time discontinuity problem. vTurbo [33] dedicated cores for I/O requests and applies a short time quantum for the cores. However, it requires modifying the guest OS, which separates critical I/O handling codes from the rest of the guest OS. The hypervisor must run the I/O handling codes on the dedicated cores. vTRS [28] categorized vCPUs into different classes based on their time slice preference at runtime and schedules each classified vCPU group to a CPU pool with an appropriate time slice.

Introducing new locking mechanisms: The Linux community introduced a queue-based spinlock to replace the ticket-based spinlock for virtualized environments [13, 17]. Recently, Teabe et al. introduced i-spinlock which allows only the thread which has sufficient remaining time slice to acquire the lock [27]. Therefore, the critical section can be completed before being interrupted. Another approach redesigned the spinlock used in OSes to mitigate the lock holder preemption (LHP) and lock waiter preemption (LWP) problems [22]. Meanwhile, there have been efforts delegating critical sections into different cores (or servers) by leveraging remote procedure calls to improve performance of acquiring highly contended locks [18, 25].

Architectural solutions: Ahn et al. proposed to shorten the time quantum for all CPUs to address both lock and interrupt problems [2]. To mitigate the overhead caused by frequent context switches, they introduced architectural techniques. However, this approach required hardware modifications which are not currently available. In commercial processors, an architectural technique was introduced to reduce wasting CPU cycles in spin locks [30]. Intel and AMD processors detect cores excessively executing spinlocks [4, 9]. If a processor executes too many `PAUSE` instructions in a short period of time, the processor generates an exception (`VMEXIT`). Upon receiving the exception, the hypervisor de-schedules the vCPU and picks another vCPU to increase the utilization [30].

2.3 Design Approaches

Our proposed mechanism identifies the preempted critical OS services, and allows them to run on the *micro-sliced cores* to quickly complete and exit the critical region. The micro-sliced cores are a pool of cores scheduled with a short 0.1ms time slice, used only for the critical OS services. To adapt to diverse virtual machine loads, a dynamic mechanism is supported to adjust the number of micro-sliced cores. Finally, our scheme aims to provide the aforementioned functionality without requiring any guest OS modifications. In Table 1, we compare our scheme, flexible micro-sliced cores, with the prior approaches. We consider the following three design approaches, which comprehensively address all the aspects of the virtual time discontinuity.

Precise selection of urgent tasks: The limitation of the prior work is that the granularity of detecting urgency is virtual CPUs, not the critical OS services [28, 33]. In this study, we pinpoint critical OS services and offload the regions to a designated pool of micro-sliced cores. The proposed mechanism executes the critical kernel routines in cores with a small time quantum, resulting in a very short turn-around time. Our design decision has the benefit of allowing the main work to run under normal time slices with the minimized context switching overheads, while the critical kernel tasks are automatically offloaded to the micro-sliced cores.

Dynamic adjustment of the micro-sliced cores: Our work proposes dynamic adjustments of the number of micro-sliced cores to prevent underutilizing CPUs. When requests for handling critical tasks are infrequent, the number of micro-sliced cores is decreased to allow more cores to be used for regular the guest application execution. When the system encounters critical OS tasks which are insufficient with a single micro-sliced core, the number of micro-sliced cores is increased to meet the demand.

Guest OS transparency: Requiring guest OS modification to address the virtual time discontinuity problem is not desirable in cloud systems, which support diverse guest operating systems. Our approach is transparent to the guest operating system without any OS modification. The hypervisor accesses the kernel symbol table of the guest OS, and identifies the location of critical services. The guest OS neither needs to be aware of our mechanism, nor requires any modifications. Although the guest OS needs to explicitly provide the hypervisor with the kernel symbol table, the symbol table is readily available in common OSes. For example, most Linux distributions including RedHat and Debian place the symbol table file under the `/boot/` directory. Even in cases where the guest kernel is compiled by the user, the symbol table is commonly generated automatically and placed alongside the kernel binary. We will discuss this issue in more details in Section 4.4.

2.4 Comparison with Prior Work

In this section, we discuss main differences between our work and the recent prior studies. The first difference is that our scheme is able to precisely pinpoint critical OS services while not modifying guest operating systems. By leveraging the existing kernel symbol table, the proposed technique avoids any guest OS changes and precisely reacts to the delayed processing of critical services. As only the critical kernel codes run on the micro-sliced cores, the user-level execution is not affected by the short time slice in the micro-sliced cores.

Unlike the prior approaches, with the selective critical section acceleration, the proposed technique efficiently mitigates all the aspects of the VTD problem including inter-processor interrupt, I/O interrupt, I/O and CPU mixed environment, and synchronization as shown in Table 1. On the other hand, vTurbo [33] focuses only on the I/O performance. It improves the I/O throughput and latency by dedicating a core in a static manner. It modifies the guest OS to separate the I/O handling codes and runs the I/O handler on the dedicated core. vScale [7] dynamically adjusts the number of virtual CPUs to mitigate the IPIs and synchronization problems, but does not address the mixed VM cases, where I/O and CPU-intensive behaviors are mixed on a single VM. vScale also requires the modifications of guest OSes as well as the hypervisor.

To avoid the guest modification, vTRS [28] estimates the behaviors of virtual machines through runtime profiling. vTRS identifies the types of virtual CPUs as lock intensive, I/O intensive, cache thrashing, or cache sensitive. Based on the time slice preferences of vCPUs, it clusters virtual CPUs to assign the best time quantum for each type of vCPUs. However, due to the coarse-grained classification on vCPUs, it may not address certain use cases. If conflicting time slice preferences are mixed in a single VM, for example, exhibiting I/O and cache-sensitive behaviors simultaneously, the optimal time slice for the VM cannot be selected. The scheme proposed in this paper only migrates the critical OS services to the micro-sliced cores, instead of classifying the entire vCPUs which run user-level and non-critical kernel codes as well as the critical ones. In addition, the profiling based on HW performance counters and other stats may not be responsive enough for changing VM behaviors.

3 ANALYSIS OF CRITICAL OS SERVICES

This section investigates which parts of the operating system become critical bottlenecks due to virtual time discontinuity. To identify the reason for the significant guest OS execution delay, we first examine the sources of `yield` events, which occur when the guest OS execution cannot make forward progress. Second, we analyze the I/O handling stacks to investigate the causes of delay in I/O interrupt processing in consolidated systems. The analysis will be used to determine which virtual CPUs are executing critical OS services and need to be migrated to the micro-sliced cores. Finally, we measure the performance impact caused by the delayed critical OS tasks due to the virtual time discontinuity problem in the baseline system.

3.1 Critical OS Components

To identify which kernel tasks are delayed, we analyze the Linux kernel functions which are frequently preempted due to the virtual time discontinuity problem. Whenever a vCPU yields its pCPU, the profiler reads the instruction pointer of the vCPU and interprets the semantic of the address by referencing the kernel symbol table to figure out what the vCPU is processing in the kernel execution.

workloads	# of yields	
	solo	co-run
exim	157,023	24,102,495
gmake	79,440	295,262,662
dedup	290,406	164,578,839
vips	644,643	57,650,538

Table 2: The number of yields of workloads run in `solo` and `co-run` (w/ *swaptions*):

For the experiments in this paper, we used Xen 4.7 for the hypervisor and the guest virtual machines were running Linux kernel 4.4 optimized for virtualization by default. We used Intel Xeon E5645, which has 12 hardware threads. In `solo` configuration, one virtual machine with 12 vCPUs runs on the system. In `co-run` configuration, two virtual machines with 12 vCPUs for each VM run on the system, with 2:1 overcommit ratio. One virtual machine runs `PARSEC` or `MOSBENCH`, while the other virtual machine runs `swaptions` which exhibits the highest CPU utilization among `PARSEC` applications.

We first assess the severity of yield increase caused by consolidation in the `co-run` configuration. Table 2 shows the number of yields that occur in `solo` and `co-run` settings. We observe that the number of yields increases significantly when the pCPUs are time-shared between two virtual machines. Yielding pCPUs results in poor performance because the critical OS tasks are delayed in the yielding vCPUs. Section 3.3 will present the performance impact.

Table 3 summarizes the critical components frequently preempted in the consolidated system for the Linux operating system. The operations are identified based on the kernel address at the yielding event. The table presents the commonly preempted operations and their semantics. The operations commonly involve spinlocks, TLB flushes, IPIs for scheduling, and interrupt handling. We further analyze the behaviors of the operations with Linux Perf and Xentrace [32] logs.

First, pause-loop exiting (PLE) incurs yields to avoid wasting CPU cycles while trying to acquire spinlocks. PLE is an architectural extension in the x86 processors which allows the hypervisor to detect whether a CPU is excessively executing the `pause` instruction. It is used as a sign that the CPU is likely wasting CPU cycles by spinning on the lock variable. Whenever a PLE exception occurs, the common hypervisors including Xen and KVM de-schedule the vCPU and select another runnable vCPU. In our experimental setup, `exim` and `gmake` frequently cause PLE exceptions due to excessive lock spinning when running with `swaptions` in a different virtual machine.

Second, waiting for acknowledgments after sending inter-processor interrupts (IPIs) causes frequent yields. The IPI-based messaging mechanism is widely used in the Linux systems. It is used for TLB synchronization and SMP load balancing. A guest VM running `dedup` or `vips` frequently yields pCPUs when co-running with another VM hosting `swaptions`. `dedup` and `vips` are applications

Module	File	Operation	Semantic
irq	softirq.c	irq_enter()	increase the preemption count
	softirq.c	irq_exit()	decrease the preemption count
	chip.c	handle_percpu_irq()	wakeup the irq handler
kernel	smp.c	smp_call_function_single()	send an inter-processor interrupt(IPI) to another core
	smp.c	smp_call_function_many()	send an inter-processor interrupt(IPI) to other cores
mm	tlb.c	do_flush_tlb_all()	TLB flush received from remote
	tlb.c	flush_tlb_all()	flush all processes TLBs
	tlb.c	native_flush_tlb_others()	send TLB shootdown IPI to others
	tlb.c	flush_tlb_func()	invoked by the TLB shootdown IPI
	tlb.c	flush_tlb_current_task()	flush the current mm struct TLBs
	tlb.c	flush_tlb_mm_range()	flush a range of pages
	tlb.c	flush_tlb_page()	flush one page
	tlb.c	leave_mm()	invoked in the lazy tlb mode
	page_alloc.c	get_page_from_freelist()	try to allocate a page
	page_alloc.c	free_one_page()	free a page in a memory zone
	swap.c	release_pages()	release page cache
sched	core.c	scheduler_ipi()	invoked by reschedule IPI
	core.c	resched_curr()	trigger the scheduler on the target CPU
	core.c	kick_process()	kick a running thread to enter/exit the kernel
	core.c	sched_ttwu_pending()	try to wake-up a pending thread
	core.c	ttwu_do_activate()	enqueue a selected thread
	core.c	ttwu_do_wakeup()	mark the task runnable and perform wakeup-preemption
spinlock	spinlock_api_smp.h	__raw_spin_unlock()	release a spinlock
	spinlock_api_smp.h	__raw_spin_unlock_irq()	release a spinlock & enable irq
	spinlock_api_smp.h	_raw_spin_unlock_irqrestore()	release a spinlock & restore irq
	spinlock_api_smp.h	_raw_spin_unlock_bh()	release a spinlock & enable bottom half IP
rwsem	rwsem-spinlock.c	__rwsem_do_wake()	wake up a waiter on the semaphore
	rwsem-xadd.c	rwsem_wake()	

Table 3: A summary of critical components in Linux 4.4

which intensively invoke *mmap* and *munmap* system calls to manage the shared address space among threads [8]. *mmap* and *munmap* involve expensive TLB synchronization process which makes sure all CPUs have the up-to-date address mapping information in their TLBs. The vCPU which initiates TLB synchronization calls `flush_tlb_*()` which uses IPIs. If one or more sibling vCPUs are preempted, the vCPU which initiated the TLB synchronization, cannot receive the IPI acknowledgments until the recipients are scheduled again. In the co-run configuration, dedup spends 89% of its CPU cycles in the `smp_call_function_many()` function, waiting for IPI acknowledgments. In Table 3, the *tlb.c* file in the `mm` module corresponds to the TLB synchronization.

The SMP load balancer also takes advantage of the IPI mechanism to send a signal to another CPU, called a reschedule IPI. The vCPU initiating an IPI in `smp_send_reschedule()` waits for the IPI acknowledgment from the recipient thread, and cannot proceed if the recipient is preempted. In addition, we discover that a few kernel functions, such as `kick_process()` and `resched_curr()`, suffer from the effects of multiple inefficiencies. They initiate reschedule IPIs while holding a spinlock or disabling guest level preemption. As a result, the delay of reschedule IPI processing makes the lock holder preemption problem more severe in the functions.

3.2 Critical I/O Path

System virtualization complicates the I/O path as it involves the handling of physical interrupts by the hypervisor, and that of virtual interrupts by the guest operating system. Once a network packet arrives at the virtualized systems, the hypervisor is responsible for handling the physical interrupt and determining the destination virtual machine of the interrupt. After that, the hypervisor sends a virtual interrupt to the target virtual CPU. In the guest operating systems, the NIC interrupt handler (e.g., `e100_intr()`) is invoked, but traversing the TCP/IP stack is deferred to the softIRQ context.

The I/O handling of the guest OS consists of a chain of operations involving potentially multiple vCPUs. The core handling the hardware interrupt can differ from the core receiving softIRQ. When the packet has been processed, the softIRQ handler (e.g., `net_rx_action()`) wakes up the registered user process waiting for the data, using the CPU scheduler (e.g., `ttwu_do_activate()`). The function in turn, invokes the reschedule IPI mechanism that we discussed in the previous section. To provide the optimal I/O performance, the hypervisor needs to quickly pass the IPI signal from one vCPU (the IRQ handler) to another vCPU (the waiting user process).

Kernel component	Wait time (avg.) solo	co-run
Page reclaim	1.03	420.13
Page allocator	3.42	1,053.26
Dentry	2.93	1,298.87
Runqueue	1.22	256.07

(a) Spinlock waiting time (μsec) in `gmake`

		Avg.	Min.	Max.
dedup	solo	28	5	1927
	co-run	6354	7	74915
vips	solo	55	5	2052
	co-run	14928	17	121548

(b) TLB synchronization latency (μsec)

	Jitters (ms)	Throughput (Mbits/sec)
solo	0.0043	936.3
mixed co-run	9.2507	435.6

(c) Latency and throughput in `iPerf`

Table 4: Performance loss: `solo` **vs.** `co-run` **(w/** ***swaptions*****)**

Furthermore, unlike the spinlock, TLB synchronization, and IPIs, the kernel-level I/O processing is coupled with user-level processing because the final destination of an I/O operation is the application. Eventually, the user-level applications are expected to consume the incoming data as soon as possible. Our design must consider the scheduling of the target application as well as the acceleration of the interrupt service routine of operating systems for maximum I/O throughput and latency improvement.

3.3 Performance Implication

This section presents the performance degradation due to the lock holder preemption, TLB synchronization, and I/O in the same `solo` and `co-run` environment as used in Section 3.1. The `co-run` scenarios host two virtual machines on a system with 2:1 consolidation ratio. One virtual machine executes a target application while the other virtual machine runs `swaptions`. In the `solo` scenarios, a single virtual machine can consume all pCPUs without any contention.

First, we compare the latencies for acquiring spinlocks between `solo` and `con-run` environments. Table 4a shows the average waiting time to acquire spinlocks in `gmake` for four kernel components reported by Lockstat [11]. Lock contention is severe in `gmake` co-running with the VM hosting `swaption`. The latencies of acquiring the spinlock are increased by orders of magnitude due to the system consolidation. Note that the recent Linux kernel does not use ticket spinlocks[1] in virtualized environments to eliminate the latency increased due to the lock waiter preemption problem. However, even with the latest locking scheme the lock holder preemption problem still exists.

The second results present the latencies for synchronizing TLBs in the two scenarios. To measure the latencies, we observe a kernel function, `native_flush_tlb_others()`, through Systemtap [1]. Since all the sibling vCPUs have to acknowledge their completion for the TLB synchronization, the completion time can be increased significantly because of preempted stragglers. Table 4b shows the performance implications of TLB shootdown IPIs. Since the virtual machine in the `solo` environment can occupy all the pCPUs without any contention, the latencies for TLB synchronization are tolerable with 28 and 55μsec on average. On the other hand, the latencies are significantly increased in `co-run`. According to this analysis, the critical OS services such as spinlock and TLB synchronization usually take less then 100μsec without contention. Therefore, we can exploit this observation by scheduling critical OS services with a time slice of sub-milliseconds, as such critical services can be completed even within the short time slice. The short time slice reduces the waiting time by shortening the scheduling turn-around latency.

[1]Since the Linux 4.2 kernel, qspinlock has replaced the ticket lock [17].

The last evaluation shows the I/O latency and throughput in `solo` and `mixed co-run`. In the `mixed co-run` scenarios, the target virtual machine runs `iPerf` along with `swaptions` on the same VM. The `mixed co-run` scenario presents a virtual machine running both CPU intensive application and I/O intensive application at the same time. It represents the case where the Xen hypervisor cannot boost the virtual CPU which exhibits the mixed behavior. In this evaluation, we used a 1Gbit network environment between the server and client. The results are shown in Table 4c. Jitters and throughputs of `iPerf` benchmark in the `mixed co-run` are significantly affected, while the `solo` configuration exhibits the upper bound performance. Jitters, representing the latency variance of the I/O operations, is significantly increased due to consolidation, from 0.0043ms in `solo` to 9.2507ms in `co-run`.

4 DESIGN

This section presents the design and architecture of our proposed scheme. The goal of our system is to minimize the delay of processing critical OS services while not sacrificing the performance of other applications. The proposed system determines whether a critical OS task is preempted by glancing at the instruction pointer (instruction address) of the preempted vCPU and matching it with the guest kernel symbol table at runtime.

To quickly resume and complete suspended critical tasks, the proposed system takes advantage of one or more **micro-sliced cores**. The micro-sliced cores are a pool of physical CPUs (pCPUs) that have a very short time slice of 100μsecs. They are used to quickly but briefly schedule vCPUs that need urgent attention to run the critical OS services. We chose the short time slice based on our analysis in Section 3.3, which shows that 100μsec is sufficient to complete the critical services. The micro-sliced cores minimize the queuing delay of the critical tasks: each vCPU in the runqueue is given a short time slice of 100μsec, and the vCPUs waiting for their turns in the runqueue will be scheduled with much shorter scheduling latencies compared to conventional longer time slices.

The remainder of this section describes how to detect critical OS services based on the guest kernel symbol table, how to handle the

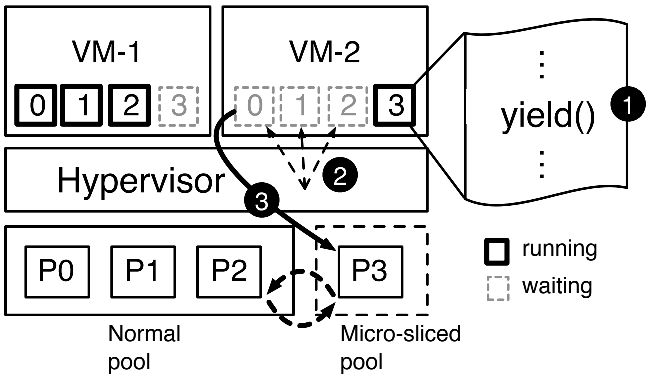


Figure 1: Identifying and offloading critical OS services based on `yields`

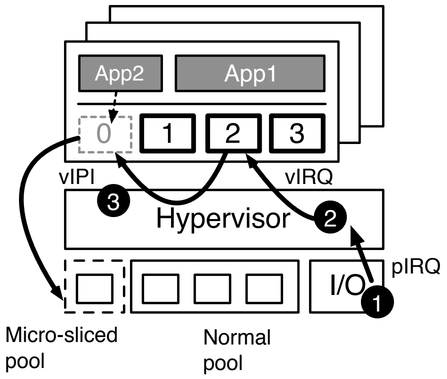


Figure 2: Identifying vCPUs running critical OS services with I/O events

critical tasks, and how to adjust the number of micro-sliced cores. Lastly, we discuss the availability of the symbol table.

4.1 Detecting Critical OS Services

Our mechanism uses two types of hints to determine whether a critical OS service is required. The first one is a *yield* event that occurs either involuntarily (hardware triggered), or voluntarily (software triggered). The second one is an incoming I/O IRQ into the system. Both events can hint that the guest system may require attention to immediately process critical services. However, the two types of events are handled in different ways.

Detecting from yield events: Figure 1 shows how a critical service signaled by a yield event is accelerated. The physical CPUs (pCPUs) are grouped into the normal pool and the micro-sliced pool, denoted as P0~3. ❶ When a vCPU (vCPU 3 of VM-2) yields its physical CPU (pCPU), the yield signal triggers the detection mechanism. To precisely locate what the vCPU is executing, the detection mechanism consults the instruction pointer of the yielding vCPU and looks up the guest kernel symbol table. ❷ Depending on the reason of yield, the instruction pointers of the already preempted sibling vCPUs (vCPUs 0~2 of the same VM) may need to be inspected as well. ❸ If the preempted vCPU was conducting a critical OS task, the hypervisor migrates the vCPU which was running in the `normal pool` of physical cores onto the `micro-sliced pool`. If necessary, the hypervisor migrates the preempted sibling vCPUs into the micro-sliced pool for a certain type of critical OS services, such as TLB synchronization.

Depending on the types of the kernel tasks which caused yield events, each type is handled differently. The identification of the kernel task starts from the instruction address at the preemption point. With the value of the instruction pointer register, the hypervisor looks up the kernel symbol table of the guest operating system, identifying whether the preempted operation is critical or not based on the whitelist derived from Table 3. The details are explained in Section 4.2. Since the proposed system accelerates kernel services instead of applications, the kernel symbol table is sufficient to obtain the semantics of yield events.

Detecting from IRQ events: Figure 2 shows the flow of an incoming network packet. ❶ When a physical interrupt is triggered by I/O devices, the interrupt generates a VMEXIT to handle the I/O related interrupt on the hypervisor. ❷ Then the hypervisor forwards the interrupt to a designated vCPU (vCPU 2) of the recipient VM. The recipient vCPU invokes the IRQ handler of the guest operating system. In the case of network I/Os, the IRQ handler offloads the remaining parts (e.g., TCP/IP stack traversing) to softIRQ. Finally, when the softIRQ finishes handling the packet, and the process is blocked, the process that the packet directed to is rescheduled. If the process is on another vCPU (vCPU 0), a reschedule IPI is sent to the vCPU ❸.

There are two critical services that must be handled timely to improve I/O performance. First, the virtual IRQ sent by the hypervisor to the guest VM needs to be handled quickly, to prevent the delayed processing of the packet. Second, rescheduling the process that eventually receives the incoming packet is also critical to the performance of I/O. The hypervisor can identify both of the critical events by the following two clues: first, an incoming physical IRQ that is addressed to a VM, and second, an IPI sent to another sibling vCPU. These two signals can be used to detect two types of critical IRQ events.

Limited support for accelerating such IRQ handling is already available in the current hypervisor, called `BOOSTING`. The Xen hypervisor boosts a vCPU that was not previously in a runqueue. The boosted vCPU is immediately scheduled in anticipation that they will need to handle urgent tasks such as I/O interrupts. However, the I/O prioritization mechanism in the current hypervisor is not always effective. As previously identified by Xu et al. [33], vCPUs that host both I/O intensive and CPU intensive workloads may show poor I/O performance. This mixed behavior within a vCPU prevents `BOOSTING` from being triggered, severely degrading the I/O performance. When another CPU intensive task is running on the vCPU in such mixed workloads, the vCPU is already on the runqueue (if it has been preempted), and this will prevent the boosting of the preempted vCPU. This leads to the degradation of I/O performance on a mixed behavior vCPU. The proposed system improves the performance of such mixed workloads by migrating critical IRQ handling tasks to the micro-sliced cores.

4.2 Handling Critical Services

Depending on the types of critical services, acceleration to the micro-sliced cores are handled differently. First, if the vCPU generating the yield exception is in the middle of TLB synchronization, the hypervisor wakes up and migrates all the preempted sibling vCPUs of the virtual machine onto the micro-sliced cores, as those sibling vCPUs need to participate in the TLB synchronization. The TLB synchronization uses a one-to-many IPI, where all the preempted vCPUs must be re-scheduled immediately to process the incoming IPI. If the system does not have enough micro-sliced cores to accelerate all the vCPUs, more physical cores can be added onto the pool of micro-sliced cores, as will be discussed in Section 4.3.

Second, if the yield exception occurs while spinning on a lock (e.g., *_raw_spin_lock()*), where the vCPU is waiting for a lock held by another already preempted vCPU. As shown in Table 3, the hypervisor checks whether the preempted vCPU siblings are executing in the critical section. If so, it wakes up and migrates the vCPUs to the micro-sliced pool to complete the suspended critical section. After running on the micro-sliced cores, the vCPUs are migrated back to the normal cores, to prevent wasting the micro-sliced cores on non-critical tasks.

I/Os are handled in a similar manner, where a selected vCPU is migrated into the micro-sliced pool. When the hypervisor receives a physical IRQ and sends a corresponding virtual IRQ to the guest VM, the hypervisor migrates the recipient vCPU onto the micro-sliced cores, to allow the IRQ to be timely handled. When the handling of the IRQ is finished, if the user process waiting for the packet is on a different vCPU, an IPI will be sent to the vCPU (❸ of Figure 2). The hypervisor will relay the IPI for the guest VM, and before the relay of interrupt, it moves the recipient vCPU onto the micro-sliced pool.

4.3 Adaptive Adjusting of Micro-sliced Cores

In consolidated environments, a single micro-sliced core may not be able to handle all the critical OS services immediately when multiple vCPUs need to be woken up. In such cases, the hypervisor needs to increase the number of micro-sliced cores. However, if the demands for critical tasks decrease, idle micro-sliced cores need to be reassigned to the normal core pool to run normal vCPUs. To adapt to the changing demands, the hypervisor needs to dynamically adjust the number of micro-sliced cores in a system. Figure 3 depicts the migration of a normal core into a micro-sliced core. P3 (pCPU 3) is removed from the pool of normal cores, and reassigned to the pool of micro-sliced cores.

In the figure, ❶ the hypervisor constantly profiles how many times each event occurs. ❷ Once a normal core needs to migrate to the micro-sliced pool, the hypervisor picks a physical core from the normal pool and the vCPUs waiting on the runqueue of the selected core are reassigned to other normal cores. ❸ The physical core is assigned to the micro-sliced pool with a short time slice. In the next profiling turn, the hypervisor decides whether to keep the micro-sliced core or return it to the normal pool based on the profiled statistics of the next round.

Algorithm 1 describes the selection algorithm for the number of micro-sliced cores. Initially, the hypervisor does not reserve any micro-sliced cores. If the virtual machines do not contend for

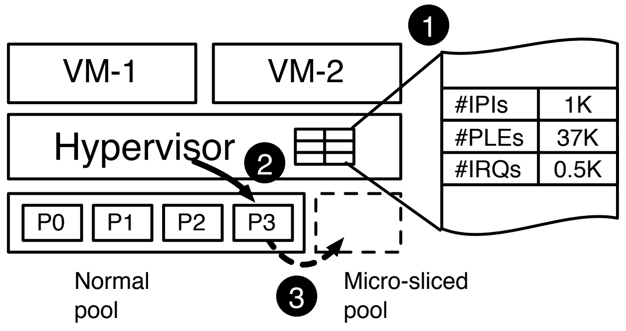


Figure 3: Migrating a normal core to the micro-sliced pool

pCPUs, the hypervisor simply skips this profiling turn. The number of micro-sliced cores becomes zero till the next profiling turn.

Adjusting the micro-sliced core is done by inspecting the execution statistics of the virtual machines. The hypervisor profiles the number of inter-processor interrupts (IPIs), pause-loop exits (PLEs), and virtual IRQs for I/O devices (vIRQ). Periodically, the hypervisor determines which event is dominant in terms of the number of events. If the IPI type is dominant, the hypervisor needs to determine how many micro-sliced cores are enough to serve the IPI tasks, as the IPI handling may involve multiple vCPUs. If the PLE or IRQ type is dominant, the hypervisor simply reserves a micro-sliced core, as one micro-sliced core can cover the necessary loads for the two types of critical tasks in the target 12-core machine.

For IPI dominant cases, the algorithm finds the best number of micro-sliced cores iteratively in a trial-and-error manner. Since the hypervisor migrates all the preempted sibling vCPUs onto the micro-sliced pool for the IPI cases as discussed in Section 4.2, the required number of micro-sliced cores may vary by the states of vCPUs. The algorithm iteratively increases the number of micro-sliced cores and measures the number of IPI events until the number of cores reaches the limit of maximum micro-sliced cores. Since the overall performance of applications is highly affected by the number of normal cores, the algorithm maintains a limit on the maximum number of micro-sliced cores.

On the other hand, if IPI and PLE events do not occur, the micro-sliced core pool is decreased. The hypervisor can allow the system administrator to manually tune the number of micro-sliced cores. The manual mode delegates the process of identifying the optimal number of micro-sliced cores to the administrator. The downside is that the administrator needs to have prior knowledge about their workloads. Such static policy is suitable for private clouds where the characteristics of applications are already known.

4.4 Discussion

Although the proposed mechanism of detecting critical OS services with the instruction pointer requires the kernel symbol table for virtual machines, obtaining the kernel symbol information is straightforward in common systems. In variants of Linux systems, the kernel symbol table is compiled into the `System.map` file along

Algorithm 1 Adjusting the number of micro-sliced cores

```
procedure AdaptiveMicroSlicedCores
    if $profileMode == False$ then
        // Initialize the profiling phases
        $num\mu Cores \leftarrow 0$
        $profileMode \leftarrow True$
        $interval \leftarrow ProfileInterval(e.g., 10ms)$
        goto out
    end if

    // Gather the statistics of urgent events for the numμCores
    $curr \leftarrow urEvents[num\mu Cores] \leftarrow systemWideStats$
    $interval \leftarrow ProfileInterval$

    if $num\mu Cores == 0$ then
        // Any urgent events did not occur
        if $CheckUrgentEvents(urEvents) == False$ then
            $profileMode \leftarrow False$
            $interval \leftarrow EpochInterval(e.g., 1000ms)$
            goto out
        end if

        $num\mu Cores \leftarrow num\mu Cores + 1$
        if $curr.numIPIs > curr.numPLEs$ OR
            $curr.numIPIs > curr.numIRQs$ then
            // IPI dominant case
            goto out
        else
            // Early termination for IRQ or PLE cases
            $profileMode \leftarrow False$
            $interval \leftarrow EpochInterval$
        end if
    else if $num\mu Cores < NUM\_LIMIT\_\mu CORES$ then
        $num\mu Cores \leftarrow num\mu Cores + 1$
    else if $num\mu Cores == NUM\_LIMIT\_\mu CORES$ then
        $num\mu Cores \leftarrow FindBest\mu CoreCnt(urEvents)$
        $profileMode \leftarrow False$
        $interval \leftarrow EpochInterval$
    end if

out:
    $SetTimer(AdaptiveMicroSlicedCores, interval)$
end procedure
```

with the kernel binary to help diagnose kernel panics. For Windows, Microsoft maintains a symbol server to help the debugging process [19]. In most of the public cloud environments, virtual machine images must be imported to cloud service components (e.g., virtual machine image server) before using them in compute nodes. A possible systematic methodology for acquiring kernel symbols is to expand the image server to provide the kernel symbols corresponding to each virtual machine image.

In practice, customers commonly leverage the virtual machine images served by cloud providers. For example, Amazon provides Linux and Windows images called Amazon Machine Images (AMI) [3]. Similarly, Microsoft Azure and Google Cloud Platform have also their own marketplace to provide popular OS images to customers [12, 20]. Considering the popularity of cloud-provided VM images, we anticipate that acquiring kernel symbol tables of virtual machines will not be challenging, as the cloud providers can guarantee that the symbol table is included for each VM image.

Limitations: The proposed technique is applied only to the critical section of guest operating systems, but not to guest applications. As the major performance impact of the virtual time discontinuity problem is within the kernel execution, the proposed technique effectively mitigates the majority of performance degradation caused by the problem. However, it opens a new possibility to accelerate certain critical codes in applications. A new user-level interface can be added to describe the user-level critical sections, and make them accessible from the hypervisor. The hypervisor will be able to register the critical regions in its separate per-process symbol table, and accelerate those regions on the micro-sliced CPU pool, if necessary.

5 IMPLEMENTATION

We implemented our prototype in Xen 4.7, confining all the necessary modifications in the hypervisor. Our mechanism to read the instruction pointer register and to look up the kernel symbol table does not require any changes to the guest operating systems and guest applications. For I/O operations, we leverage the Xen I/O handling mechanism, which relays IRQs and IPIs to guest operating systems. We made modifications at these relay paths in the hypervisor. Our modifications comprise of 1454 lines of code changes. The source code is available at `http://github.com/microslicedcore`.

Handling urgent signals: There are two sources of yields: PLE and guest OS-initiated yields. First, when a PLE occurs while a virtual CPU (vCPU) is being executed, the VMEXIT handler, `vmx_vmexit_handler()`, is invoked with the `EXIT_REASON_PAUSE_INSTRUCTION` reason in Xen. Then, the vCPU involuntarily yields the physical CPU (pCPU) to prevent wasting CPU cycles. Second, the operating system may voluntarily invoke the yield hypercall (e.g., in the `xen_smp_send_call_function_ipi()` function). We take advantage of the yield signals as the trigger point for detecting preemptions of critical services. The `vcpu_yield()` function is eventually triggered by the two sources. The handling of the yields is dealt at this function. For I/O operations, the Xen hypervisor has been designed to boost events such as virtual IRQs and virtual IPIs. However, the vanilla Xen cannot boost a vCPU when it is running a mix of different workloads and exhausted all the credits. We allow the vCPU which is on the runqueue to run on the micro-sliced cores if it receives the interrupt.

CPU pool management: Our mechanism implements pools of normal cores and micro-sliced cores by extending the *cpupool* mechanism of Xen [31]. We fork a *micro-sliced* pool which becomes the child cpupool of the normal pool. The micro-sliced pool inherits the scheduling policy and parameters from the parent cpupool except for the time quantum. In the micro-sliced pool, we do not maintain a master CPU which is responsible for managing credits in the credit scheduler. Instead, the micro-sliced pool relies on the master CPU in the parent cpupool to manage credits of the pool. Therefore, the micro-sliced pool does not need to consider the accounting mechanism separately.

We relax the restriction of Xen where all the vCPUs of a virtual machine must be in the same cpupool. Some vCPUs can be in the normal pool while others are in the micro-sliced pool. Once we decide to accelerate a suspended vCPU which is identified as urgent, the `vcpu_migrate()` function in Xen is extended to migrate the vCPU to the micro-sliced pool. We limit the length of the per pCPU runqueue in the micro-sliced pool to one vCPU to avoid the case stacking vCPUs. So, we check the length of runqueues before initiating the acceleration.

Tracking critical events: As mentioned in Section 4.3, the best number of micro-sliced cores depends on the type of urgent tasks such as inter-processor interrupts (IPIs), pause-loop exits (PLEs), and virtual IRQs (vIRQs). Our dynamic mechanism first figures out which critical service is currently dominant in the system without having micro-sliced cores for 10ms (profile phase). The Xen hypervisor already keeps track of the events for the PLE and vIRQ. We extend the existing tracking mechanism and add an IPI counter to record the number of yields caused by IPIs. During the profile interval, if events are not counted, the system is considered as not contended. In such cases, the micro-sliced cores are not allocated for one second (run phase). Otherwise, we start exploring for an adequate number of micro-sliced cores. Depending on the type of critical services, we take two different approaches. If the IPI is dominant source of yields, we need to know how many micro-sliced cores are sufficient. To find the answer, we profile the number of IPI events by increasing the number of cores in the micro-sliced pool one by one until a predefined limit. Without the limit, the profiling cost could be expensive because it gradually reduces the number of cores in the normal pool. Based on the profiling results, we can simply select a configuration which generates the least number of yields. For IRQ or PLE dominant cases, a single micro-sliced core is allocated because the two types of critical services can be easily covered by a micro-sliced core. The decision from the above profiling phase is enforced for one second (run phase). The run phase is long enough to avoid unnecessary fluctuations on the number of micro-sliced cores, which can be affected by short-lived behavior changes in VMs.

Other considerations: While running on the micro-sliced cores, we prevent the vCPUs from being preempted by the boosting and load balance mechanism to rapidly complete the urgent tasks without any interruption. Also, we do not allow the load balancer to migrate vCPUs from the normal to micro-sliced pool because the vCPUs to run on the micro-sliced pool must be selected by our mechanism. After consuming the allocated time slice (`0.1ms`) on the micro-sliced cores, we always move the vCPUs back to the normal pool. Otherwise, the vCPUs that have been migrated to the micro-sliced cores can monopolize the micro-sliced pool. For example, upon preemption at the micro-sliced core if we insert the preempted vCPU back to the micro-sliced core runqueue again, the preempted vCPU prevents other urgent vCPUs from being migrated into the micro-sliced pCPU due to our restriction on the length of each runqueue.

6 EVALUATION

6.1 Experimental Setup

We evaluate our proposed scheme with a server system equipped with two Intel Xeon E5645 processors which have 12 physical CPUs. Each physical CPU has two hardware threads enabled by hyperthreading, and thus, the total number of hardware context is 24. However, to avoid the effect of NUMA and Dom-0, only a socket (12 threads) is used to evaluate the performance. A total of 32GB memory is installed in the socket. We use Xen 4.7 as our hypervisor, and the time slice for the baseline is set to the default `30ms`. The micro-sliced cpupool is configured to have a time slice of `0.1ms`.

Environment: To mimic cloud-like consolidated environments, we deploy two virtual machines which are configured with 12 virtual CPUs (vCPUs) and 8GB memory each. The guest virtual machines run Ubuntu 14.04 and Linux kernel 4.4. Since the Linux kernel has been optimized for virtualization, we did not change the default kernel configuration options such as `CONFIG_PARAVIRT_SPINLOCKS=y`. In addition, Pause-Loop Exiting (PLE) is turned on by default.

Benchmarks: A subset of the applications from PARSEC [5] and MOSBENCH [6] are used to evaluate this work. From PARSEC, `dedup` and `vips` are selected as they are known to pressure the TLB synchronization component in the kernel. Both applications use the native input. From MOSBENCH, `exim`, `gmake`, and `psearchy` are chosen for application-level evaluation. These workloads are designed to stress various kernel components. `exim` frequently creates processes and small files. `gmake` is known to intensively exercises the kernel to incur LHP problems. We also use `memclone` as a microbenchmark. We use the default input parameters for the MOSBENCH workloads. In the latter part of this section, we analyze and discuss the overhead of our technique for CPU intensive applications from SPECCPU2006.

6.2 Experimental Results

Performance improvements: Figure 4 shows the performance improvements by varying the number of micro-sliced cores from 1 to 6, compared to the baseline. Since the number of normal cores decreases as the number of micro-sliced cores increases, we limit the number of micro-sliced cores to 6 out of the 12 cores in this evaluation. We execute two virtual machines, with the first virtual machine running one of six applications and the second co-runner virtual machine running `swaptions`. The overall performance trend shows that the execution time is reduced for the target virtual machine (the first VM) whereas the performance of co-runner virtual machine (the second VM) is slightly degraded.

For the consolidation of `gmake` and `swaptions`, the execution time of `gmake` is significantly reduced by the micro-sliced cores, but `swaptions` is slowed down compared to the baseline. The main reason is that in the baseline configuration, `gmake` was not able to fully exercise the allocated CPUs in consolidated environments because of the lock holder preemption problem. As a result, the guest operating system hosting `gmake` is frequently scheduled out by the PLE hardware exception, while spinning on the lock. Once a yield event occurs, the hypervisor selects a runnable vCPU mostly from the VM with `swaptions`, to avoid wasting CPU cycles in the system. Thus, `swaptions` was able to use more CPU time than `gmake` in the baseline. Note that this behavior is due to the work-conserving policy in the hypervisor scheduler. However, with a micro-sliced core, the number of yields for `gmake` is significantly reduced because the hypervisor detects the preempted lock holder vCPU and wakes up the vCPU on the micro-sliced core. With micro-sliced cores, `gmake` is able to utilize CPUs much better than the baseline. The combined throughput improvement from the VM

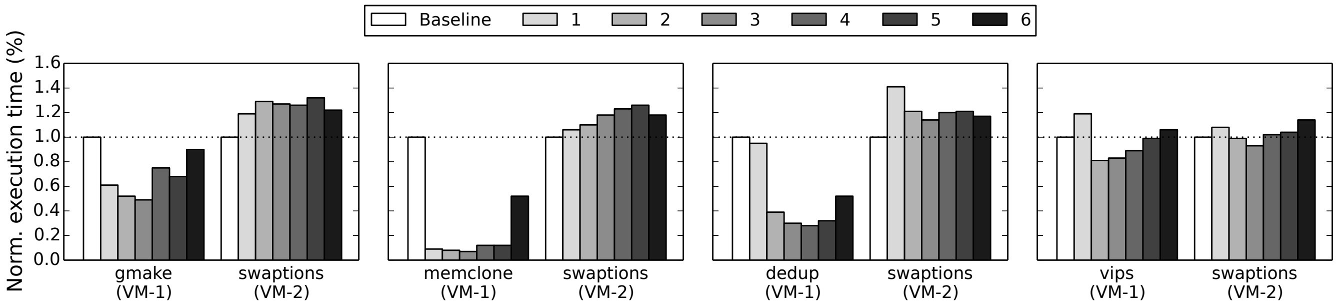


Figure 4: Performance results by varying the number of micro-sliced cores: `gmake`**,** `memclone`**,** `dedup`**, and** `vips`

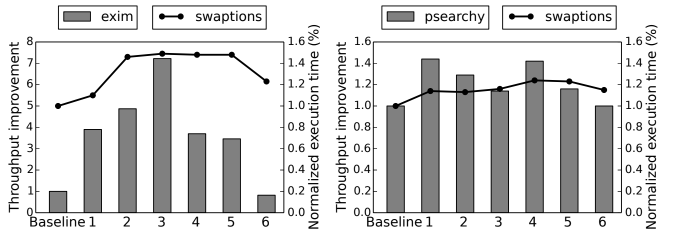


Figure 5: Performance results by varying the number of micro-sliced cores: `exim` **and** `psearchy`

pair is 20% when running with a single micro-sliced core. Since the workload combination is sensitive mostly only to the lock holder preemption problem, one single micro-sliced core was sufficient to serve the urgent services.

`memclone` creates threads and invokes the `mmap` system call to allocate memory on each thread. It also suffers from the lock holder preemption problem. When `memclone` runs with `swaptions`, a micro-sliced core significantly improves the performance of `memcl-one` by 91%, while increasing the execution time of `swaptions` only by 6%. Although `memclone` shows the best performance with three micro-sliced cores, the performance difference is minor, compared to a single micro-sliced core.

On the other hand, for the executions of `dedup` and `vips`, we found that a single micro-sliced core incurs negative effects. The reason is that these applications are frequently involved in the TLB synchronizations. Unlike spinlocks, TLB synchronizations require all the vCPUs belonging to the same virtual machine to participate in the process. Although the number of preempted vCPUs in the same virtual machine depends on the system load status, in our evaluation, a single micro-sliced core was insufficient. Increasing the micro-sliced cores to two or three effectively reduces the delay in the TLB synchronizations. From four micro-sliced cores, however, the performance was degraded because the gain in the kernel performance from the four micro-sliced cores is lower than the performance degradation in non-critical parts by the reduction of normal cores. With three micro-sliced cores, the yielding time of `dedup` is significantly reduced by 70% while the execution time of `swaptions` is only increased by 14%. The combined throughput improvement is 56% for the two applications. For the case of `vips` and `swaptions`, both applications are improved by 17% and 7%, respectively.

Figure 5 shows two more virtual machine pairs with with a different performance metric, as the performance metric of `exim` and `psearchy` is throughput. On the left y-axis, the graph shows the throughput improvements and the right y-axis indicates the execution time of `swaptions` running on the co-runner VM. For both cases, the introduction of micro-sliced cores can improve the overall system performance. For `exim`, the throughput is improved by 3.9 times over the baseline by employing a single micro-sliced core, at the cost of 10% `swaptions` performance. In the baseline configuration, `swaptions` has benefited from the unused CPU cycles yielded from the virtual machine serving `exim`. For the `psearchy` and `swaptions` case, a micro-sliced core exhibits 1.4 times throughput improvement. The throughput of `psearchy` was improved by reducing the number of halts and spinlock-induced yields.

Throughout the experiments, the best number of micro-sliced cores varies, depending on the type of workloads because each workload has a different dominant kernel component. We observe that TLB synchronization-sensitive applications, `dedup` and `vips`, work effectively with three micro-sliced cores in the system configuration with 12 HW threads. A single micro-sliced core is sufficient for the spinlock sensitive applications, because the lock cannot be held by multiple vCPUs.

Dynamic adjusting of micro-sliced sores: Since each workload combination has a different performance benefit with the different number of micro-sliced cores, the number of micro-sliced cores needs to be adjusted dynamically. We evaluate our proposed dynamic technique described in Section 4.3, which finds the best number of micro-sliced cores at runtime.

Figure 6 shows the performance comparison of the static best case (`static`) with the dynamic micro-sliced cores (`dynamic`). For the static best configuration, for each run, we pick the number of micro-sliced cores which exhibits the best performance in the previous experiments. In general, our simple scheme of estimating the required number of micro-sliced cores shows comparable performance to the static ideal cases.

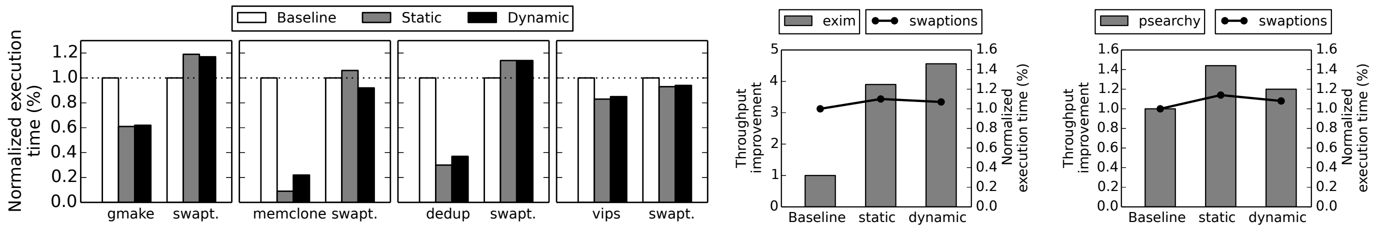


Figure 6: Performance comparison: `static` vs. `dynamic` micro-sliced cores

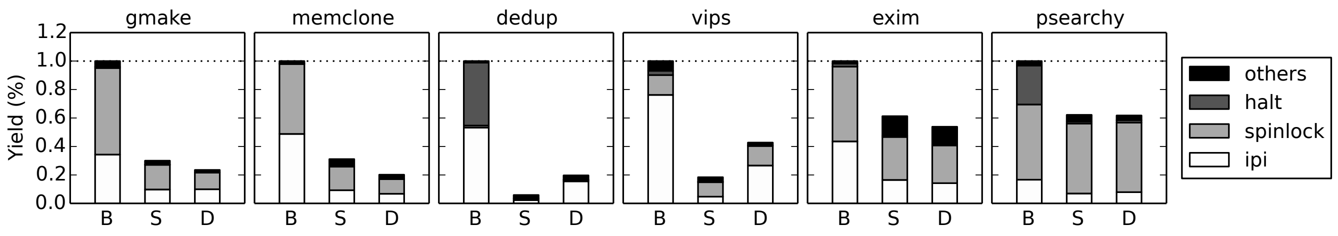


Figure 7: Reduction of the number of `yield` events (`B`: Baseline, `S`: Static, and `D`: Dynamic)

The `memclone` and `dedup` cases show a small performance loss of around 5%, compared to the static ideal case. In `exim`, however, the throughput is slightly improved with the dynamic technique. For `psearchy`, its throughput decreases compared to the static ideal case, but still shows much better than the baseline by 20%.

Effectiveness of dynamic micro-sliced cores: Figure 7 presents the number of yield events that can be reduced by our schemes (static and dynamic), compared to the baseline. We decompose the sources of yields and plot their stacked bar graphs. For the multi-threaded applications, the inter-processor interrupts are the dominant source of yields. Interestingly, `dedup` generates many `halt` exceptions in addition to the IPIs in the baseline, as the CPU utilization of `dedup` is lowered significantly by the delays in the TLB shootdown operations.

However, our proposed scheme is able to reduce the number of IPI-induced yields by accelerating the suspended vCPUs on the micro-sliced cores. It results in a decreased number of `halt`-induced yields. For `gmake`, `memclone`, and `exim`, the numbers of PLEs, which account for most of the yields, are effectively reduced by our schemes. With the micro-sliced cores, the yielding of CPUs is significantly reduced. As the yielding event is reduced, the applications exhibit higher CPU usages than the baseline.

Overhead analysis: We also evaluate the proposed technique with workloads which do not intensively exercise OS services. The goal of this evaluation is to show that our scheme provides better performance with the applications which are sensitive to OS services, while not sacrificing the performance of unaffected applications. We selected two multi-threaded applications from PARSEC and several single threaded applications from SPECCPU2006. These applications are known to spend most of their runtime in the user-level execution, and do not spend much time in the kernel execution. Figure 8 presents the execution times of those applications with our scheme enabled. On average, the performance overhead is around 2-3%. The benchmark applications for Figure 8 do not suffer from the virtual time discontinuity problem. The proposed dynamic scheme with its periodic profiling, does not incur any noticeable performance degradation for the applications, showing that the extra performance overheads of the proposed scheme is negligible.

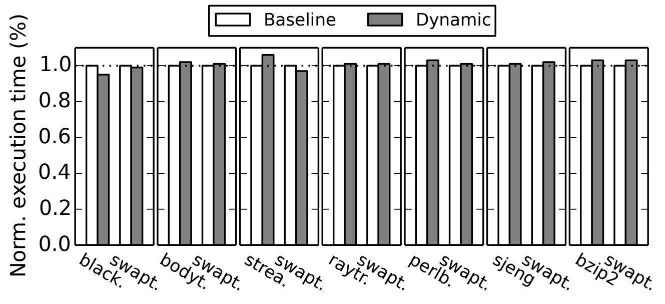
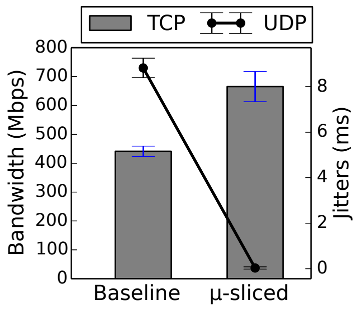


Figure 8: Non-affected workloads

I/O performance: In this experiment, we evaluated the I/O performance for virtual machines with mixed behaviors. It uses a scenario where I/O and compute intensive applications are hosted on the same virtual machine, and the I/O intensive process is placed on the same vCPU as a compute intensive application. Due to the mixed behavior of the vCPU, the vCPU will not benefit from the boosting mechanism of the Xen hypervisor because the CPU intensive application can always occupy the vCPU. This causes delays in the processing of the network packets in the baseline hypervisor. In addition, timely acknowledgments and subsequent requests are delayed, reducing the performance of I/O.



(a) Latency and throughput

	Workloads
VM-1	iPerf lookbusy
VM-2	lookbusy

(b) Workload combinations

Figure 9: Performance of mixed co-run cases

To evaluate the I/O performance of the mixed VM, we use two virtual machines with one vCPU each, pinned onto the same physical CPU (pCPU). The applications running on each VM are shown in Figure 9b. Lookbusy is used to represent a compute bound application, constantly utilizing the CPU.

Figure 9a shows the throughputs and jitters of the network benchmark `iPerf`. First, the TCP throughput of the baseline, which suffers from the mixed behavior vCPU interfering with the hypervisor boost mechanism, is improved in the micro-sliced scheme. Jitters are also significantly improved, from over 8ms to near 0ms. The reason for the significant jitter reduction is due to the prompt scheduling of the vCPU handling the IRQ signals. In the baseline, the IRQ signal is sent to the guest by the hypervisor, but because the guest vCPU was not boosted, the handling of the IRQ signal was delayed. The same principle applies to the TCP bandwidth. Our work migrates the suspended vCPUs onto the micro-sliced cores to allow rapid processing of the urgent vCPUs, leading to the improved jitters and throughput.

We show that our approach of offloading critical OS services to the micro-sliced cores can significantly improve the I/O performance in the presence of mixed behavior CPUs. This is because our scheme pinpoints short critical I/O handling codes, and offloads only these regions of codes onto the micro-sliced cores, without affecting the rest of execution.

7 CONCLUSIONS

In this work, we proposed a new approach to mitigate the virtual time discontinuity problem in consolidated environments. To precisely detect virtual CPUs preempted while executing critical OS services, we took advantage of the instruction pointer of vCPUs and consulted with kernel symbols of the guest operating systems. Once vCPUs executing critical urgent services were identified, we offloaded the vCPU to the micro-sliced cores: a pool of CPUs that schedule vCPUs in very short time slices. Our proposed system did not require any guest OS modifications, the hypervisor transparently identified the critical guest OS services and accelerated them. The evaluation showed that our proposed scheme improves the performance of problematic consolidation scenarios without degrading the performance of other workloads.

ACKNOWLEDGMENT

This research was supported by the National Research Foundation of Korea (NRF-2017R1C1B5075437, NRF-2017M3C4A7081088) and by the Institute for Information & communications Technology Promotion (IITP-2017-0-00466). Both grants were funded by the Ministry of Science, ICT and future Planning (MSIP), Korea.

REFERENCES

[1] 2015. A scripting language and tool for dynamically instrumenting Linux kernel. (2015). Retrieved March 6, 2018 from https://sourceware.org/systemtap/

[2] Jeongseob Ahn, Chang Hyun Park, and Jaehyuk Huh. 2014. Micro-Sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*.

[3] Amazon. 2018. Amazon Machine Images (AMI). (2018). Retrieved March 6, 2018 from https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

[4] AMD. 2010. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Programmer's Manual. (2010).

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*.

[6] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*.

[7] Luwei Cheng, Jia Rao, and Francis C. M. Lau. 2016. vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*.

[8] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*.

[9] Intel Corporation. 2017. Intel 64 and IA-32 Architectures Software Developer's Manual. (2017).

[10] Xiaoning Ding, Phillip B. Gibbons, Michael A. Kozuch, and Jianchen Shan. 2014. Gleaner: Mitigating the Blocked-waiter Wakeup Problem for Virtualized Multicore Applications. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC '14)*.

[11] Linux Kernel Documentation. 2015. Lock Statistics. (18 September 2015). Retrieved March 6, 2018 from https://www.kernel.org/doc/Documentation/locking/lockstat.txt

[12] Google. 2018. Google Compute Engine. (2018). Retrieved March 6, 2018 from https://cloud.google.com/compute/docs/images

[13] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2015. Scalability in the Clouds!: A Myth or Reality?. In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys '15)*.

[14] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. 2013. Demand-based Coordinated Scheduling for SMP VMs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*.

[15] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. 2009. Task-aware Virtual Machine Scheduling for I/O Performance.. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*.

[16] Min Lee, A. S. Krishnakumar, P. Krishnan, Navjot Singh, and Shalini Yajnik. 2010. Supporting Soft Real-time Tasks in the Xen Hypervisor. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '10)*.

[17] Waiman Long. 2014. qspinlock: a 4-byte queue spinlock with PV support. (7 May 2014). Retrieved March 6, 2018 from https://lwn.net/Articles/597672/

[18] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2012. Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC '12)*.

[19] Microsoft. 2017. Symbols for Windows debugging. (23 May 2017). Retrieved March 6, 2018 from https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/symbols

[20] Microsoft. 2018. Microsoft Azure Marketplace. (2018). Retrieved March 6, 2018 from https://azuremarketplace.microsoft.com/en-us/marketplace

[21] Diego Ongaro, Alan L. Cox, and Scott Rixner. 2008. Scheduling I/O in Virtual Machine Monitors. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*.

[22] Jiannan Ouyang and John R. Lange. 2013. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '13)*.

[23] Jiannan Ouyang, John R. Lange, and Haoqiang Zheng. 2016. Shoot4U: Using VMM Assists to Optimize TLB Operations on Preempted vCPUs. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '16)*.

[24] Aravinda Prasad, K Gopinath, and Paul E. McKenney. 2017. The RCU-Reader Preemption Problem in VMs. In *Proceedings of the 2017 USENIX Conference on Annual Technical Conference (USENIX ATC '17)*.

[25] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. Ffwd: Delegation is (Much) Faster Than You Think. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[26] Orathai Sukwong and Hyong S. Kim. 2011. Is Co-scheduling Too Expensive for SMP VMs?. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*.

[27] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagimont. 2017. The Lock Holder and the Lock Waiter Pre-emption Problems: Nip Them in the Bud Using Informed Spinlocks (I-Spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*.

[28] Boris Teabe, Alain Tchana, and Daniel Hagimont. 2016. Application-specific Quantum for Multi-core Platform Scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*.

[29] VMware. 2010. VMware vSphere 4: The CPU scheduler in VMware ESX 4.1. Technical Whitepaper. (12 July 2010).

[30] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. 2006. Hardware Support for Spin Management in Overcommitted Virtual Machines. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06)*.

[31] Xen Project wiki. 2016. Cpupools Howto. (9 May 2016). Retrieved March 6, 2018 from https://wiki.xen.org/wiki/Cpupools_Howto

[32] Mark A. Williamson. 2015. Xen Trace. (28 August 2015). Retrieved March 6, 2018 from https://xenbits.xen.org/docs/4.7-testing/man/xentrace.8.html

[33] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. 2013. vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-sliced Core. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC '13)*.

[34] Cong Xu, Sahan Gamage, Pawan N. Rao, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. 2012. vSlicer: Latency-aware Virtual Machine Scheduling via Differentiated-frequency CPU Slicing. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '12)*.