Micro-sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems

Jeongseob Ahn, Chang Hyun Park, and Jaehyuk Huh
Computer Science Department, KAIST
{*jeongseob, changhyunpark, and jhuh*}*@calab.kaist.ac.kr*

***Abstract*—Although time-sharing CPUs has been an essential technique to virtualize CPUs for threads and virtual machines, most of the commercial operating systems and hypervisors maintain relatively coarse-grained time slices to mitigate the costs of context switching. However, the proliferation of system virtualization poses a new challenge for the coarse-grained time sharing techniques, since operating systems are running on virtual CPUs. The current system stack was designed under the assumption that operating systems can seize CPU resources at any moment. However, for the guest operating system on a virtual machine (VM), such assumption cannot be guaranteed, since virtual CPUs of VMs share limited physical cores. Due to the time-sharing of physical cores, the execution of a virtual CPU is not contiguous, with a gap between the virtual and real time spaces. Such a virtual time discontinuity problem leads to significant inefficiency for lock and interrupt handling, which rely on the immediate availability of CPUs whenever the operating system requires computation.**

This paper investigates the impact of virtual time discontinuity problem for lock and interrupt handling in guest operating systems. To reduce the gap between virtual and physical time spaces, the paper proposes to shorten time slices for CPU virtualization to reduce scheduling latencies of virtual CPUs. However, shortening time slices may lead to the increased overhead of context switching costs across virtual machines. We explore the design space of architectural solutions to reduce context switching overheads with low-cost context-aware cache insertion policies combined with a state-of-the-art context prefetcher.

***Keywords*-virtual time discontinuity; virtualization; context prefetch; context preservation**

I. Introduction

Although time-sharing CPUs has been an essential technique to virtualize CPUs to threads and virtual machines (VMs) for many decades, most of the commercial operating systems and hypervisors maintain relatively coarse-grained time slices to mitigate the costs of frequent context switching. However, with the proliferation of system virtualization, the performance impact of time slices needs to be re-evaluated under these new environments. One of the changes by virtualization is that operating systems are running on virtual CPUs (vCPUs) which time-share physical cores. From the perspective of operating systems, one critical difference between virtualized and native systems is the gap between the virtual and real time spaces. Guest operating systems schedule user and kernel threads, and are often interrupted by exception handling, under the assumption that the virtualized CPUs are continuously running. However, the vCPU time space for a VM is in fact a sequence of discrete time slices in real time space. We call this the virtual time discontinuity problem.

Traditionally, the system software stack has been designed with the assumption that the operating system can seize the control of CPUs at any moment. Due to the assumption, the virtual time discontinuity problem can potentially incur significant inefficiency in two critical mechanisms which the current operating systems rely on for synchronization and I/O handling. The first mechanism is a spin lock for synchronization among kernel threads. Such a low-overhead simple spin lock is used in kernels since the critical sections of kernel threads are very short. However, with the virtual time discontinuity problem, the vCPU running the lock holder thread can be preempted by the hypervisor, making lock waiters spin, wasting CPU resources. The problem also incurs the reduction of I/O throughput and the slowdown of inter-processor interrupts (IPI). The fundamental assumption of the interrupt mechanism is that the CPU is readily available to serve incoming interrupts.

There have been several advancements in hypervisor schedulers and architectural supports to address the two problems individually. Each solution mitigates the inefficiency of lock processing or interrupt handling in virtualized systems in an ad-hoc manner. For example, the popular Xen hypervisor boosts the priority of a vCPU if a virtual interrupt is pending for the vCPU, preempting a running vCPU. Furthermore, the PLE (Pause-Loop Exiting) support in the x86 architecture allows the hypervisor to detect spin lock execution on a core, providing the physical core to another ready vCPU. However, such scheduling or architectural solutions address only a small portion of the virtual time discontinuity problem.

This paper explores the solution space of the virtual time discontinuity problem, which encompasses both lock processing and interrupt handling. This paper proposes to shorten the time slice for vCPU scheduling to sub-millisecond ranges. By reducing the time slice to a more fine-grained time quantum of less than 1ms, the scheduling artifact of the current time multiplexing of vCPUs in the range of tens of milliseconds can be hidden. By multiplexing

vCPUs with such a short time slice, the scheduling turn of each vCPU arrives quickly without a significant delay. Computation latencies for the spin lock holder or interrupt handler are relatively short, commonly within 1ms, and the critical section or interrupt handling can be completed within the short time period. The proposed time slice solution differs from prior scheduler-based solutions, since it provides a general solution to the virtual time discontinuity problem. Instead of modifying the scheduler case by case, which can often lead to a patch work with unpredictable behaviors in certain cases, the time slice method can potentially eliminate the artifact of coarse-grained scheduling of vCPUs.

However, one of the most significant downsides of the time slice method is the overhead of frequent context switches. Reducing the overheads of context switching, especially in virtualized environments, has been studied in prior studies [5], [25]. The proposed context prefetchers aim to restore cache states at the beginning of a time slice, by prefetching a burst of data for the saved context. This paper complements the context prefetching technique with an alternative *context preservation* technique. Some contexts do not exploit the large last-level cache capacity effectively, and for the other co-running, cache-efficient applications, the context preservation technique attempts to keep as much data of these cache-efficient contexts as possible. The proposed technique uses a time-sampling technique to identify the caching efficiency of each context, and apply a different cache insertion policy to prevent cache pollution by cache-inefficient contexts.

Although the prior studies addressed the costs of context switching with context prefetchers [5], [25], they are based on the assumption that highly consolidated systems will have frequent context switches due to the bounded scheduling window which certain hypervisors are employing. However, in such systems, time slices will decrease only when many active contexts are in the runqueue. This paper focuses on addressing the virtual time discontinuity problem, and argues that regardless of hypervisor schedulers or the ratio of overcommitments, shortening time slices to a much shorter quantum than previously used, mitigates the discontinuity problem in consolidated systems.

The new contributions of this paper are as follows. First, this paper is one of the first studies to address the virtual time discontinuity problem in a holistic manner. Prior studies approach spin locks, external interrupts, or IPIs in VMs as an individual problem. This paper generalizes them into the virtual time discontinuity problem. Second, this paper emphasizes the importance of reducing time slice to resolve virtual time discontinuity. The paper also investigates the downside of sub-millisecond time slicing, and the architectural implication for future support for VM consolidation. Finally, this paper proposes a context preservation technique based on time sampling. This technique requires minimal changes in current cache architectures unlike context prefetchers. The context preservation technique can be combined with context prefetchers to improve their efficiency further.

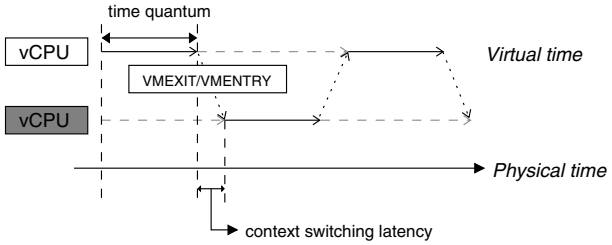


Figure 1: Virtual and physical time window

Our real machine evaluation shows that decreasing time slices to less than 1ms can mitigate the virtual time discontinuity problem effectively both for lock and interrupt-intensive VMs. To overcome the performance degradation of certain overcommitted workloads in SPEC applications, our context preservation technique can reduce the degradation significantly, when cache-inefficient contexts are running with cache-efficient contexts by time-sharing.

The rest of the paper is organized as follows. Section II presents the background on the virtual time discontinuity problem with case studies on spin lock and interrupt handling. Section III presents the study of short time slices and their effect on spin lock and interrupt handling. Section IV introduces context preservation to reduce the overheads of frequent context switching. Section V presents our experimental results. Section VI discusses prior work and section VII concludes this paper.

II. Motivation

In this section, we introduce the virtual time discontinuity problem in virtualized systems, and present the background and current solutions for two affected kernel mechanisms, spin lock processing and interrupt handling by guest operating systems.

A. Virtual Time Discontinuity Problem

In virtualized systems, many VMs share a physical system. Virtual CPUs (vCPUs) of the VMs are multiplexed to physical CPUs by the hypervisor. When the number of active vCPUs exceeds the available physical cores, vCPUs time-share physical cores, receiving their share of time slice periodically. The quantum of time slice for each scheduling round is coarse-grained, commonly several to tens of milliseconds, to amortize costs for context switching. The guest operating system assumes virtually continuous execution of its vCPUs, but in real time space, each vCPU is periodically scheduled in and out. In current virtualized systems, virtual machines are commonly overcommitted to share physical resources more efficiently, and thus, virtual time discontinuity occurs frequently. Figure 1 describes the gap between virtual and real time in a single-core system,

	Avg. waiting time		Avg. holding time	
Workloads	solo	co-run	solo	co-run
bodytrack	6.73	13,641.45	7.88	111.35
canneal	3.42	5,765.86	15.97	1,165.09
facesim	8.86	37,514.69	8.43	212.37
fluidanimate	0.59	13,615.83	3.44	632.20
raytrace	2.93	3,952.83	5.52	433.40
streamcluster	1.22	3,197.97	1.20	120.98

Table I: Spinlock waiting and holding times (μusec) with co-running `swaptions`: 1 VM (solo) vs. 2 VMs (co-run)

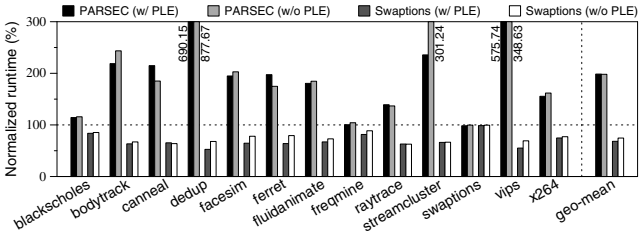


Figure 2: Performance: execution time of PARSEC co-running with `swaptions` normalized to capped solo-runs

where two VMs with a single vCPU share the system. At the end of each time slice, a `vmexit` event occurs to transfer the control to the hypervisor, and the hypervisor schedules the next VM with a `vmentry` event.

One critical problem of virtual time discontinuity is that the operating system is not aware of the discrete CPU execution slices. An important design assumption of current operating systems is that the operating system can seize the control of CPUs immediately to process kernel operations or serve interrupt requests quickly. Two most important mechanisms that rely on this assumption are spinlock-based critical sections in kernel and handlers for external and inter-processor interrupts. The next two subsections describe the two problems in details.

B. Spin Lock in Kernel

The kernel uses spinlocks since they have low acquire latencies when the lock contention is low. Since the critical sections are commonly short in the kernel, kernel threads need to spin for only a short period of time without wasting CPU resources significantly in native systems. However, with virtual time discontinuity, the vCPU running a lock-holding thread can be scheduled out, while lock waiting threads are spinning on the lock. The lock waiting threads can potentially waste CPU resources, until the vCPU running the lock holder thread is scheduled in again.

In the current Linux kernel implementation, a variant of spin lock called *ticket lock* is commonly used to protect a critical section accessing kernel data structures. With the ticket lock, every thread requesting a lock receives a ticket number, which is incremented for each request. If the ticket number is less than or equal to the current lock counter, the thread acquires the lock. The current lock counter is incremented when the lock is released. Compared to an alternative test-and-set spin lock, the ticket lock can provide ordering among lock requesters, and thus, prevent starvation of lock acquisition.

With the ticket lock used in the guest operating system kernel, two potential problems can arise. The first problem is the lock holder preemption (LHP) problem. The vCPU running a lock-holding thread can be preempted by the hypervisor, and lock-waiting threads have to waste CPU cycles until the lock is released. The other problem called the lock waiter preemption (LWP) problem is due to the ordering of lock grants in ticket locks. A vCPU running the next lock waiter can be preempted by the hypervisor, and the other lock waiters must wait for the preempted vCPU to be scheduled in.

Table I shows the average lock waiting and holding times used by kernel locks. The `solo` column represents a configuration with a single VM with four vCPUs on a system without any contention. The `co-run` column represents a contended case with two quad-core VMs on the same system. In the `co-run` configuration, each PARSEC application on a VM share the system with another VM running `swaptions` as a co-runner on a quad-core Xeon CPU system. In the `solo` configuration, the waiting and holding times are both negligible. However, with a co-running application, both latencies are increased by several orders of magnitude. As discussed by Kim et al [9]., the lock waiting problem is exacerbated in the current Linux system, since the kernel locks commonly protect critical sections involving TLB shootdowns or reschedule IPIs.

Architectural support for reducing LHP and LWP problem is available. Current x86 processors support the Pause-Loop Exiting (PLE) mechanism. Spin lock implementation in the guest kernel executes a pause instruction for each spin, and if a core detects that too many pause instructions are executed, exceeding a configurable threshold within a short period, a `vmexit` event is generated for the hypervisor. The hypervisor preempts the spinning vCPU, and schedule another ready vCPU, expecting the new vCPU to run useful instructions instead of spinning. Although PLE improves the throughput of the physical systems by reducing spins, it can impact the fairness across different VMs.

Figure 2 presents execution time changes by consolidating two VMs running the PARSEC benchmarks. The figure shows two configurations with and without PLE support, and for each application, two execution times for the application, and two execution times for the co-running `swaptions` are shown. The execution times are normalized to *equal-capped solo* runs. In the equal-capped solo runs, although only a VM is running, the CPU share of the VM is capped to a half of the available CPU share. In the co-run configuration, each VM is entitled to receive at least the same amount of CPU share as the capped solo run, unless the VM releases CPUs

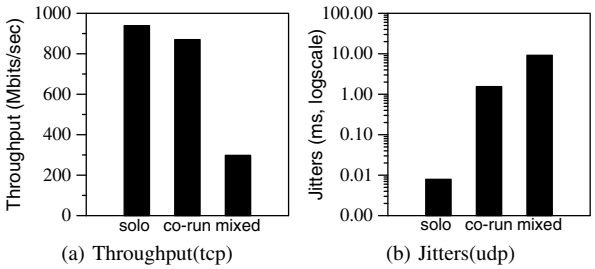
voluntarily. Ideally, without any extra overhead due to time-sharing, a VM in the co-run configuration should perform equally to the same VM in the capped solo configuration, since both configurations provide the same CPU share to the VM. However, when consolidated, the execution time increases significantly for workloads which utilize kernel locks frequently. For `dedup, streamcluster, vips`, which have been reported as heavy busy waiters (either lock spinning, IPI or both)[9], the execution times increase substantially. Enabling PLE can mitigate the performance degradation for certain workloads. For example, the execution time of `dedup` is reduced significantly with PLE while the co-running swaptions also benefits from PLE. However, for some applications, PLE has mixed effects, improving one application while degrading the other co-running one as shown in `vips` with `swaptions`. PLE does not significantly mitigate the performance problem by lock handling across different workloads. Another negative aspects of the lock problem is fairness between two co-running VMs. While a VM is suffering from LHP and LWP, the co-running `swaptions` can benefit compared to the capped solo-runs, since it can take some CPU resources from the other applications.

C. Interrupt Handling

Interrupt handling can also suffer from virtual time discontinuity. In non-virtualized systems, the interrupt is a mechanism for a HW device to send a request to the kernel to receive immediate services. An occurrence of interrupt immediately invokes the corresponding handler, with the assumption of the availability of CPUs by preemption. I/O throughput is affected by such interrupt handling efficiency. Furthermore, IPIs (inter-processor interrupts) for communication among cores also rely on the interrupt mechanism. For example, during a page table update and TLB shootdown process, IPIs are sent to invoke TLB flushes in the other processors.

The processing of an interrupt is relatively short-lived, and the user process is resumed quickly after interrupt handling in native systems. However, in virtualized systems, the hypervisor sends a virtual interrupt to a VM, and if the vCPU is not scheduled, the virtual interrupt cannot be served immediately. The vCPU will be able to process the interrupt only when it receives its time slice again from the hypervisor scheduler, postponing the processing of urgent interrupts. Although the actual CPU processing time for interrupt handling is short, the scheduling time slice will force a VM to delay interrupt handling by tens of milliseconds or longer, depending on how many vCPUs share a physical core. Such a delay can significantly reduce the I/O throughput, and increase the latencies of TLB shootdown processes.

To mitigate the problem, the current Xen hypervisor optimizes the hypervisor scheduler for interrupts. The credit scheduler has three states for each vCPU, `under`, `over`, and `boost`. Depending on the availability of remaining credits to execute on a core, a vCPU can be either in under state, which means it still has credits left, or over state, meaning the vCPU is out of credits. When a vCPU in the under state receives a virtual interrupt, its state is elevated to boost, preempting the running vCPU to seize the physical core immediately. Such a scheduling optimization mimics the behavior of interrupt handling in native systems, by scheduling a vCPU to a physical core immediately.



(a) Throughput(tcp) (b) Jitters(udp)

Figure 3: `iperf` throughput and jitters for `solo`, `co-run`, and `mixed` configurations

To evaluate the effect of interrupt handling optimization in Xen, Figure 3 shows the I/O throughput and jitters for three configurations, `solo`, `co-run` and `mixed` environments. Jitter for iperf is the mean of differences between consecutive transit times. In the `solo` configuration, only one VM runs an `iperf` benchmark to measure raw I/O throughput and jitters. In the `co-run` configuration, one VM runs iperf, and the other VM runs `x264` of PARSEC, to show the CPU contention between I/O-intensive and CPU-intensive VMs. The `mixed` configuration represents a more complicated consolidation scenario. Both I/O-intensive and CPU-intensive applications are running on a VM, and another VM is sharing the system. Such a mixed VM, which has both I/O and CPU intensive processes in one VM, appears in many server applications with both intensive data I/Os and a certain amount of processing of data. Performance optimizations for such mixed VMs have been discussed in several recent studies [23], [24].

As shown in the figure, the `co-run` case almost matches the I/O throughput of the `solo` configuration, since the interrupt handling optimization works effectively in Xen. However, in the `mixed` environment, I/O throughputs drop and jitters increase significantly even with the scheduler optimization. One of the reason for the throughput reduction is because the CPU-intensive application in the mixed VM exhausts its credit of CPU resources during a scheduling round. Since the CPU-intensive process has already exhausted the credit of the vCPU, the scheduler cannot boost the vCPU with pending interrupts. Such a `mixed` case represents a consolidation scenario which a simple scheduler optimization cannot address effectively.

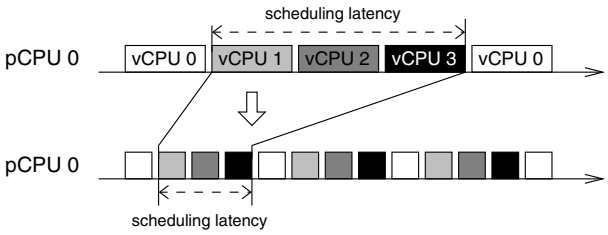


Figure 4: Shorter time slices lead to shorter scheduling latencies

KVM	Dynamically calculated by CFS depending on the amount of runnable processes. As number of processes increases time slice decreases until it reaches a lower bound value.
PowerVM	For each dispatch window, assign a timeslice proportional to the processing units owned by the VM [6].
Xen	Fixed time slice for each vCPU

Table II: Time slice policies of different hypervisors

III. Micro-sliced Context Switch

A. Reducing Scheduling Latencies

Current hypervisors have several different ways to determine time slices for virtual CPUs. Table II shows the policies used to decide the time slices in three hypervisors. There are broadly two different categories. In the first approach, *bounded window*, the scheduling window is fixed, and the number of active vCPUs during the window determines time share for each vCPU. KVM and PowerVM adopt such a scheduling window approach. KVM, based on the Linux CFS scheduler, dynamically increases the scheduling window size if the number of active vCPUs (threads) increases, and the time slice becomes shorter than a configurable threshold (4ms by default). PowerVM has a fixed 10ms window size, which can be shared by up-to 10 vCPUs per window [3]. The minimum time slice per vCPU is also lower bounded to 1ms. For the bounded window approach, until the number of active vCPUs reaches a certain limit, the scheduling latency for each vCPU is bounded by the window size. The second approach, *unbounded window*, uses a fixed time slice for each vCPU, as used in Xen. The Xen hypervisor uses a default 30ms time slice for each vCPU. In such a policy, the scheduling latency for a VM may increase as the number of active vCPUs increases.

In this paper, to reduce the gap between virtual and real time spaces, we propose to decrease time slices for each VM to reduce the scheduling latency. Although the approach can be applied for both of the aforementioned approaches, we apply it to the latter, unbounded window case, as our evaluation uses the Xen hypervisor. In the former approach with the fixed scheduling window, as the number of active vCPUs per core increases, the time slice is decreased automatically. However, as shown in the previous section, even if two vCPUs are competing for a physical core, the virtual time discontinuity problem occurs. Therefore, even if the scheduling window is bounded, it is necessary to reduce time slices explicitly by reducing the window and allowing a very short time slice for each context.

	Nehalem	Westmere	Haswell
Cycles	4,986	1,988	1,484

Table III: The number of cycles for a null hypercall for each Intel architecture generation. 95th percentile values shown

In the latter approach without a fixed scheduling window, the length of the time slice can directly affect the scheduling latency as shown in the following equation.

$$scheduling\_latency = time\_slice \times (\#vCPUs - 1) \quad (1)$$

The #vCPUs of equation 1 denotes the number of vCPUs queued in the given physical CPU's runqueue.

To minimize the scheduling latency, our approach shortens the time slice on the Xen hypervisor, visualized in Figure 4. The figure illustrates the shortened scheduling latency due to more frequent scheduling of each vCPUs. Reducing the scheduling latency alleviates the previously mentioned LHP, LWP, and I/O problems. The preempted lock holder will be scheduled earlier, and thus will release the lock sooner. Any busy waiting lock waiters will be able to acquire the lock sooner, or will be scheduled out after busy waiting for a shorter period of time. As for the interrupt handling, whenever an interrupt is issued to a VM, the vCPUs of the VM will be scheduled with a shorter delay. This will allow a more prompt handling of the interrupt, resulting in improved I/O performance.

On the other hand, the short time slice will be accompanied by overheads which are classifiable into two types. The first type is the direct cost incurred by context switches, and virtual mode extensions (VMX) mode switches such as VM-exits and VM-enters. However, the direct cost has been decreasing to 1000-2000 cycles. Table III shows the direct cost of `vmexit` + `vmentry`. A null hypercall was executed 1000 times, once per second on an idle system. The actual latencies vary, and the table shows 95th percentile values. Therefore, for 95% of null hypercalls, the latencies are shorter then the ones in the table. The cycles for `vmexit` and `vmentry` have decreased from 5000 cycles to about 1500 cycles, as the processor designs improve for better support of virtualization. The second type is the indirect cost, which is the pollution of architectural structures such as caches, TLBs, branch predictors, etc. To minimize the negative impacts, architectural support in mitigating the impacts is necessary, which will be discussed in depth in Section IV.

B. Experimental Results

In this section, we evaluate the effects of shortening time slices with a real system equipped with two Intel

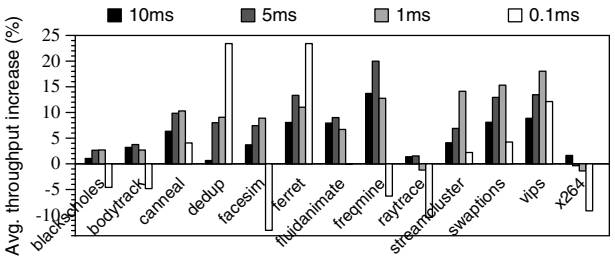


Figure 5: Average of aggregate throughputs for all combinations of PARSEC: 10ms, 5ms, 1ms, and 0.1ms

Xeon E5620 2.40GHz processors and 48GB of RAM. Dynamic voltage and frequency scaling, turbo boost, and hyper-threading were disabled for consistent performance evaluation. We use Xen 4.2.3 as a hypervisor, with the dom0 VM running Ubuntu 12.04 (kernel 3.8.0). To minimize the interference by dom0, one CPU socket is dedicated to dom0 and the other socket is used to run guest VMs. We employ hardware assisted virtual machines, running Ubuntu 12.04 and 3.5.7 Linux kernel.

1) **Single-application VM Scenarios:** We evaluate the overall performance by running each of the PARSEC benchmark [4] in two different VMs. For completeness, we run every combination of two PARSEC applications and executed them together, excluding cases with two same applications. Figure 5 presents the average aggregated throughput improvements for each tenant benchmark, compared to the default time slice of 30ms. For example, the values for `dedup` are averages of aggregate throughput of `dedup` running with 12 other PARSEC benchmarks. For most cases, a short time slice of 1ms improved the average throughput effectively. As for the case of 0.1ms, we observe a large variation of results, with some workloads improving, and others decreasing. The performance increase of `dedup` with 0.1ms is because the lock preemption problem is severe in the application with frequent TLB shootdowns. The reason for the performance drop with the 0.1ms slice is due to the cost incurred, both direct and indirect, by frequent context switching.

2) **Multi-application VM Scenarios:** In this section, we evaluate two consolidated scenarios with multiple applications in a VM. Table IV describes the workload combinations for the two scenarios. All workloads in the same cell run on the same VM, and the value inside the parenthesis indicates the number of threads for the application. In the `non-mixed` set, the workloads consist of only CPU-oriented workloads with VM-4 running multiple applications. In the `mixed` set, VM-1 has both CPU-oriented application (ferret) and I/O-oriented application (iperf). The default number of threads is four for the unspecified workloads, and each VM has 4 vCPUs. We vary the time slices to 30, 1, and 0.1ms.

	VM-1	VM-2	VM-3	VM-4
non-mixed	ferret	vips	dedup	3x swaptions(1), streamcluster(1)
mixed	ferret+**iperf(1)**	vips	dedup	3x swaptions(1), streamcluster(1)

Table IV: Non-mixed, and mixed workload combinations

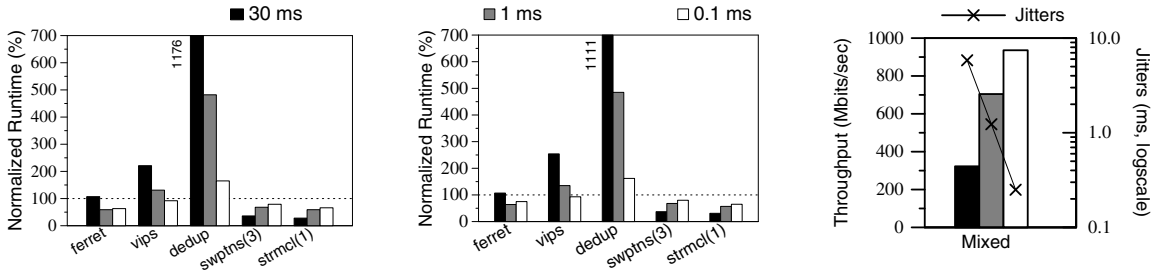
The execution times are normalized to those with `equal-capped solo` runs as used in Section II-B. In the equal-capped solo runs, one VM runs on a physical system, but the CPU share is equal to the consolidated configration in Table IV. The solo-runs provide the same amount of CPU to a VM as the consolidated runs when each VM receives the same share of CPUs, but they do not have the negative effect of time-sharing. In the setup, four vCPUs share a physical core, and in the equal-capped solo runs, a quarter of the total CPU share is allocated to the VM.

Non-mixed set: Figure 6a shows the execution times with the non-mixed workloads, normalized to those with the equal-capped solo runs. Note that the parenthesis in figures indicate the number of executed benchmarks. `Swaptions(3)` indicates the average of three concurrent executions of single-threaded `swaptions`.

With the 30ms time slice, the execution times increase by more than 10 times in `dedup` compared to the capped solo-runs. Even if the VM with `dedup` in both consolidated and capped solo runs can use the same CPU share, the negative effect on lock handling degrades the performance of `dedup` significantly. `Ferret` and `vips` also suffer significantly by the artifact of time-sharing. However, `swaptions` and `streamcluster` improve the performance by time-sharing, since they receive more CPU share taken from the other applications. By decreasing the time slice to 1ms or 0.1ms, the execution times of `ferret`, `vips`, and `dedup` are reduced significantly. However, the performance of the other two applications degrade since they can no longer take more CPU share from the other three applications.

From the perspective of fairness, reducing the time slice makes the performance of all VMs closer to the capped solo-runs. Such improvement of fairness is another benefit of short time slices in addition to throughput improvement. Furthermore, with 0.1ms time slice, all VMs exhibit lower execution times compared to the capped solo-runs except for `dedup`. `Dedup` shows considerable performance improvement as well; not to the point where total fairness is achieved, but improving the overall system throughput nonetheless.

Mixed set: Figure 6b presents the normalized execution times for CPU-oriented applications with three different time slices. The results are in general similar to those in the non-mixed results. The additional application in this case is `iperf`, and Figure 6c shows the I/O throughput and jitters with `iperf` as the time slice is reduced. With the



(a) Non-mixed workload runtime over capped solo-runs

(b) Mixed workload runtime over capped solo-runs

(c) `iperf` Throughput and Jitters

Figure 6: Normalized execution time and I/O performance of mult-application VM scenarios

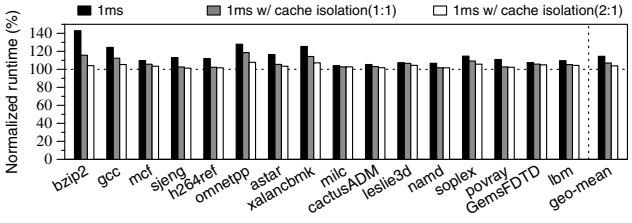


Figure 7: Runtime with cache isolation based on coloring on a real machine (co-running with `libquantum`): normalized to 30ms time slice

30ms time slice, `iperf` suffers from significant throughput degradation, because it executes on a VM with the CPU-intensive `ferret` application. In addition, the jitter exhibits a high variance of 5.83ms with the 30ms slice. As the time slice is reduced, the VM running `iperf` and `ferret` is more frequently scheduled, resulting in `iperf` being scheduled more frequently. When the time slice is 0.1ms, the throughput recovers to that of a solo-running `iperf`, as shown in Figure 3a. Furthermore, jitters decrease to 1.23 with 1ms, and 0.25 with 0.1ms.

Based on our experiments, using 1ms and 0.1ms time slices almost eliminates the effect of virtual time discontinuity problem for lock and I/O-intensive workloads. However, using 0.1ms time slice can often increase the context switching costs significantly. We conclude that reducing the time slice of the scheduler can achieve both improved fairness and throughput on consolidated systems. If such context switching cost can be lessened, more performance gain would be expected.

IV. Context Preservation and Prefetching

Although decreasing time slices to sub-millisecond ranges reduces the negative artifact of virtual time discontinuity, frequent context switches can potentially reduce system performance due to overheads. In this section, we identify the last-level cache as the most important source of performance degradation which accompanies short time slices. To mitigate the overheads, we investigate context-aware cache insertion policies, which attempt to preserve the data of previous contexts, as long as the current context is not negatively affected. This simple solution does not require significant changes in processors. In addition, we investigate how the context prefetcher from prior studies can be combined with the context-aware insertion policy.

A. Overheads of Short Time Slices

One of the main penalties for sub-millisecond context switch is the cache pollution effect. When a vCPU is scheduled to a core, the private cache and parts of the LLC are filled with data fetched for the vCPU. The new vCPU evicts the data used by the prior vCPU, eventually flushing the cached data of the prior vCPU entirely. In current virtualized systems with 10-30ms coarse-grained context switches, the flushing effect has a minor impact for the virtual machine performance, since the time slice is long enough to amortize the effect of cache flushing, and new data fill the caches quickly at the beginning of the time slice. However, when the time slice is reduced, the phases in which cache contents are replaced occur more frequently, resulting in more cache misses.

First, we use our real machine setup to measure the overhead of context switching in a 1ms time slice system. To isolate the effect of cache pollution due to fine-grained context switches, we evaluate the effect of decreasing the time slice without cache interference among VMs. To emulate a system without cache interference, we use page coloring when a VM memory is allocated by the hypervisor. Page coloring is a technique to control the location of a page in the cache by exploiting parts of set numbers in machine address. With page coloring, each VM receives different sets of colors, so that they never interfere in the cache. However, note that the effective cache capacity for each VM is reduced to an Nth of the physical cache capacity, when N VMs share the system.

Figure 7 presents the effect of isolating cache effect for 1ms time slice. The experimental setup evaluates overcommitted VMs running a SPECCPU 2006 application and `libquatum` with cache thrashing behaviors. The `1:1`

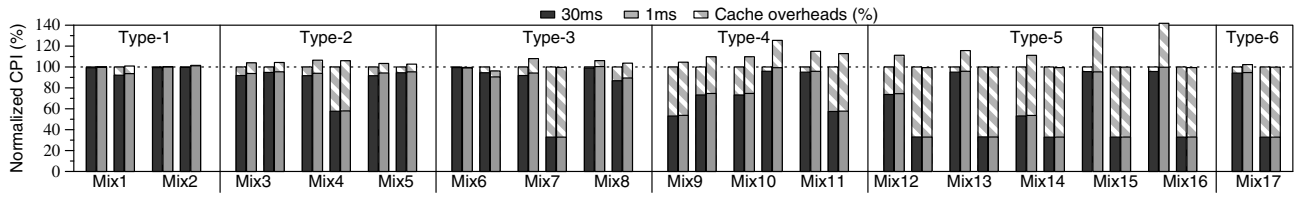


Figure 8: Performance effects of cache pollution: simulated system results

configuration means the cache capacity is equally partitioned between two virtual machines and the `2:1` configuration indicates each SPECCPU application has twice more cache space than `libquantum` to isolate the negative effects. For example, bzip2 suffers from the reduced time slice (1ms) with 42% increase of execution time compared to 30ms time slice. However, after isolating the cache effect, the performance degradation is reduced significantly for most of the applications. For the PARSEC applications we mainly used in the previous sections of this paper, the cache effect was minor, even for short time slice runs. There are two main reasons for the small cache effect. First, the working sets of the PARSEC applications are relatively small, compared to the large LLC. Second, part of multiple vCPUs in a VM can be running while the rest of vCPUs of the same VM are preempted. The running vCPUs can hold on to a subset of the shared data in the LLC.

To clearly identify the main source for performance degradation, we used the simulation setup described in Section 5.1. Figure 8 presents the CPI (cycles per instruction) results for 17 mixes of applications with the simulated setup. For each mix, two sets of bars are shown for each application, as the mixes consist of two applications. For each set, CPIs of 30ms and 1ms context switching time are compared. The portion of bars with striped patterns represent the CPI increases due to cache pollution. As shown in the figure, when CPI increases with 1ms time slice, the CPI caused by cache pollution increases, while the rest of the solid bar remains unchanged. From the results, we conclude that for any performance overhead incurred by the 1ms time slice, almost all the overheads result from the cache pollution by multiple contexts.

B. Context-aware Cache Insertion

To reduce the impact of caches for fast context switches, this paper investigates two possible approaches. The first approach, *context prefetching*, fetches essential data of an incoming context, as soon as the incoming context is scheduled. The cache addresses are recorded to a specified memory region when the blocks are evicted by another competing contexts from the previous time slice. Daly and Cain first proposed such a context prefetcher to prepare the LLC for the next context in a virtualized configuration similar to PowerVM [5]. RECAP later optimized the bandwidth consumption of the context prefetcher further [25]. Our implementation of the context prefetcher follows the details of the optimized RECAP.

An alternative approach newly explored in this paper is *context preservation*. As indicated by our previous analysis with SPEC and PARSEC benchmarks in the previous section, only a small subset of SPEC applications are negatively affected by 1ms time slice, and almost none of PARSEC applications suffered from the short time slice. Instead of adding new HW context prefetchers which consume significant memory bandwidth at the beginning of each time slice, the LLC can be used more effectively to retain the data of previous contexts.

The context preservation technique exploits how each context benefits from the LLC. There are three types of contexts based on their caching behaviors. The first type, *CPU-intensive*, does not use the LLC much, as their working set fits in the modest capacity of the second-level cache. Due to the small working set size, bringing in the data at the beginning of each time slice does not require significant delays. The second type, *cache-friendly*, uses the LLC very effectively, as their working sets exceed the second-level cache, but fit in the last-level cache. For the cache-friendly type, the LLC has a very important role for their performance. The third type, *thrashing*, does not use the LLC effectively, as the working set exceeds the LLC.

The caching behavior of a context not only affects the performance of its own context, but also can be exploited for preserving other contexts sharing the LLC. As CPU and thrashing type contexts do not use the LLC efficiently, the data of those contexts should be forced not to evict the data of previous cache friendly contexts. Only cache-friendly contexts should be allowed to use the entire capacity of the LLC.

To support such context preservation, two mechanisms are necessary. First, the cache control mechanism must prevent the data of CPU or thrashing type, from evicting the data of important cache-friendly contexts. Second, another mechanism must detect the behavior of each context to identify its type.

For the first control mechanism, we use a simple dual insertion mechanism. For cache-friendly contexts, a new cache block is inserted to the MRU position of the corresponding set (MRU). As the context is executed for a time slice, the

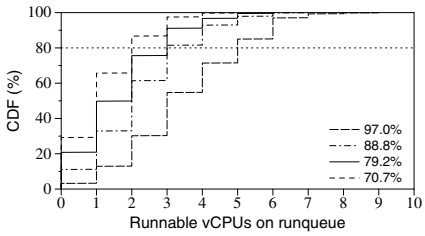


Figure 9: Simulated runqueue length for a fair scheduler with four different aggregate utilizations shown in CDF

Parameter	Value
Processor	Out-of-order x86 ISA, 3.4GHz 128-entry ROB, 80-entry LSQ 5-issue width, 4-commit width 36-issue queue, 6-ALU, 6FPU
Branch Predictor	4K entry BTB, 1K entry RAS Two-level branch predictor
L1 I/D Cache L2 Cache L3 Cache	2/4-cycle, 32KB, 4-way, 64B block 6-cycle, 256KB, 8-way, 64B block 27-cycle, 2MB, 16-way, 64B block
Memory	DDR3-1600, 800MHz, 1 memory controller

Table V: Simulated system configurations

cache blocks of the context will evict all previous context data. However, for CPU-intensive or thrashing contexts, new cache blocks of the contexts are inserted to the LRU position. If the data is reused soon, the cache block will be moved to the MRU position. If the data are not reused in the near future, they will be evicted from the LLC without evicting the data of previous contexts. The dual insertion policies were originally proposed by Qureshi et. al for a single core cache [16], and later extended for multi-core shared caches [8]. We employ a similar dual insertion policy between time-sharing VMs, although our scheme does not use the set-sampling mechanism to select the best policy for the current VM, and our dual policy uses MRU and LRU, instead of MRU and BIP (Bimodal Insertion Policy) in the original study.

For the second identification mechanism, we use a simple time-sampling mechanism which does not require any extra hardware. For every 10 quanta of 1ms time slice, MRU and LRU insertions are tried for two sampling quanta. The remaining 8 quanta use whichever policy that causes less cache misses. If the current context uses the cache effectively with lower misses with MRU insertion compared to that of LRU insertion, the context uses the MRU insertion policy for the rest of the 8 quanta. If both insertion policies yield similar misses, LRU insertion is used to prevent evicting the data of previous contexts.

C. Active Contexts in Overcommitted Systems

One of the important factors for context preservation is how many contexts are sharing the LLC. If there are too many contexts, the effectiveness of context preservation will be reduced, as more contexts compete for the limited LLC. This section investigates how many active contexts are ready and competing for the LLC. In highly consolidated virtualized systems, it is possible that 10s of VMs share a physical system. However, such a high consolidation ratio does not always lead to a large number of active vCPUs competing physical cores and cache at the same time. One of the constraints in such consolidated systems is that the total CPU utilization does not exceed the CPU capacity of the physical system. The recommended ratio of consolidation of vCPUs to pCPUs is dependent upon the aggregate utilization of vCPUs [11]. The target aggregate utilization is commonly up to 80%, and if the utilization exceeds 90%, one of the VMs should be migrated to another physical system [20].

To analyze the actual ready vCPUs under the constrained aggregate utilization, Figure 9 shows the cumulative distribution function (CDF) of the length of the runqueue for different target utilizations of the overcommitted virtual machines. We analytically simulate an environment with 10 vCPUs (from 10 VMs with a single vCPU) on a 1 pCPU machine. The utilization of each VM is set to be equal, and the sum of all utilizations from VMs is the target utilization. The job arrival from each VM follows an exponential distribution, although the jobs would have some dependent factors in reality. The emulated scheduling is based on fair scheduling across different VMs with work conserving not to waste CPU resources.

Figure 9 shows that if the utilization is 88.8%, then 80% of the time the runqueue length is three or less. When the utilization is 79.2%, 90% of the time the runqueue is three or less. We used the target utilizations for this study as it has been recommended to overcommit VMs to the point where the CPU is upper bounded to 80%, and 90% indicates an overloaded state[20]. From the analysis, we observe that even if a system is highly consolidated, if the target utilization is set to a certain threshold to achieve performance targets, the number of actual active contexts at a given time slice is relatively small. With such small active contexts, preserving the contexts in the LLC can be efficient.

V. Evaluation

A. Methodology

To evaluate our proposed schemes, we use the Marss86 full-system simulator [15], with a detailed DRAM model [17]. The simulated system runs Linux 3.11 on the simulator. Although we did not run a real hypervisor on our infrastructure, we modified the completely fair scheduler of Linux to mimic the fixed time slice scheduler. In this study, we evaluate the time slices for 30ms (Xen default) and 1ms as the baseline. The detailed system configurations are shown in Table V. On a 3.4GHz processor context switches

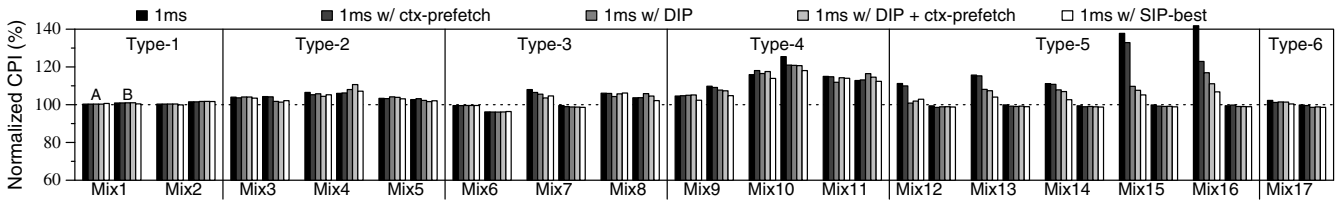


Figure 10: Performance 2-to-1: mixed type applications

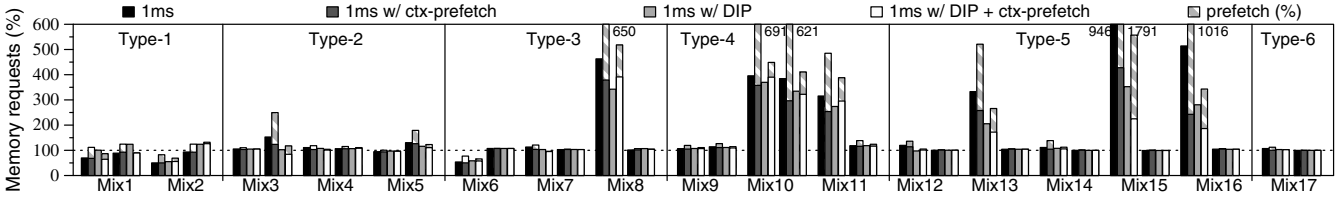


Figure 11: Memory requests 2-to-1: mixed type applications

Type-1 (CPU - CPU)	Mix-1	namd, sjeng
	Mix-2	namd, h264
Type-2 (CPU - CFR)	Mix-3	sjeng, astar
	Mix-4	sjeng, soplex
	Mix-5	sjeng, bzip2
Type-3 (CPU - THR)	Mix-6	namd, milc
	Mix-7	sjeng, libquantum
	Mix-8	h264, mcf
Type-4 (CFR - CFR)	Mix-9	omnetpp, gcc
	Mix-10	astar, xalancbmk
	Mix-11	bzip2, soplex
Type-5 (CFR - THR)	Mix-12	gcc, libquantum
	Mix-13	bzip2, libquantum
	Mix-14	omnetpp, libquantum
	Mix-15	astar, libquantum
	Mix-16	xalancbmk, libquantum
Type-6 (THR - THR)	Mix-17	milc, libquantum

Table VI: Workloads used in experiments

occur at an interval of 102,000,000 and 3,400,000 cycles for 30ms and 1ms respectively.

To focus on the performance impact of cache interference, we use SPEC2006 applications. Table VI describes the details of benchmark applications and scenarios. We evaluate two overcommitted scenarios for 2 and 4 applications running on a single core processor.

To create mixes of workloads with various mix scenarios, we group applications into three classes. *CPU* indicates that the working set of the application fits in the L2 cache. The performance is not highly affected by the cache efficiency as shown in Figure 8. *CFR* indicates a cache friendly application which utilizes the LLC effectively. *THR* is an application which incurs cache thrashing. Based on this classification, we create 6 mix scenarios as shown in Table VI.

B. Experimental Results

First, we evaluate our proposed schemes on a single core with 2:1 and 4:1 overcommitted setups. In the 2:1 overcommit ratio, two VMs share a core, and in the 4:1 overcommit ratio, four VMs share a core. 4 VM mix workloads are composed by doubling each application in the 2 VM mixes.

Figure 10 shows the performance impacts by shortening the time slice on the simulator for all 17 mixes. A and B stand for the two applications shown in Table VI. All results are CPIs normalized to those of 30ms time slice. The goal is to reduce CPIs of 1ms time slice to those of 30ms time slice with little context switching cost. The first bar, `1ms`, represents the CPI of 1ms time slice with neither context prefetching nor preservation. The second bar, `1ms w/ ctx-prefetch`, shows the CPI with a context prefetcher. The third bar, `1ms w/ DIP`, shows the dynamic insertion policy (DIP) with time sampling. The fourth bar, `1ms w/ DIP + ctx-prefetch`, shows the combination of DIP and context prefetcher. The final bar, `1ms w/ SIP-best`, is an ideal implementation of the proposed dynamic insertion policy. The applications were run with all possible insertion policies, and we selected the best insertion policy for each context.

In the figure, type-1, 2 and 3 workloads show only minor performance degradation even with 1ms time slice, and both of the context prefetching and preservation do not affect the performance significantly with little room to improve. Type-6 workloads also show negligible performance degradation with 1ms, since both applications of the mixes are thrashing the LLC and do not use caches efficiently.

The best scenario for context preservation is the mix scenario of type-5 where a cache-friendly application runs with a thrashing application. The data of the thrashing context are inserted to the LRU position, not polluting the data of the cache-friendly context. In such cases, the simple dynamic insertion policy outperforms the context prefetcher. Although the context prefetcher with the reuse-bit optimization does not prefetch data without locality for the thrashing workloads, the thrashing context eventually evicts

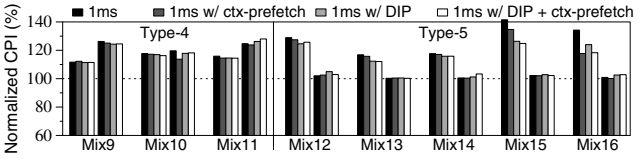


Figure 12: Performance 4-to-1: mixed type applications

the data of the co-running cache friendly context. Type-3 with CPU-intensive and thrashing workloads also shows the benefit of the context-aware insertion policy, although the performance benefit is much smaller than that of type-5.

However, for type-4 which are both cache-friendly contexts, context preservation does not improve the performance significantly. In this mix type, the context prefetcher effectively reduces the performance degradation of the 1ms time slice. However, the performance improvement even with the context prefetcher is relatively limited in our three-level cache hierarchy with a DDR3-1600 memory model.

Figure 11 presents the memory transactions normalized to `30ms` for each configuration. Bars with the context prefetching are decomposed into the normal lower portion and the upper portion by the context prefetcher. For type-4 and 5, memory requests with `1ms` increase significantly, and the context prefetch further increases the memory bandwidth usage. On the other hand, the context preservation schemes can effectively reduce the number of prefetches.

Figure 12 presents the performance with the 4:1 overcommit ratio. Our context preservation scheme shows similar performance improvements of type-5 with the 2:1 runs. However the improvements are reduced, since multiple cache-friendly contexts must share the LLC, competing the limited cache capacity.

In summary, context preservation can effectively prevent the cache pollution caused by thrashing contexts, and protect the cache data of cache friendly contexts. However, its effectiveness can be sensitive to the number of simultaneous active contexts. On the other hand, the context prefetching is effective when both cache friendly applications share a physical core. Combining both approaches will be able to cover both cases effectively, with the additional benefit of reducing prefetching traffic. One aspect not explored in this paper and prior context prefetching studies is the sharing of LLCs among vCPUs of multiple VMs. Such time and spatial sharing of LLCs will pose an interesting challenge for context preservation and prefetching.

VI. Related Work

There have been many prior efforts to mitigate the artifact of virtualization including synchronization and interrupt handling by improving the hypervisor scheduler. Traditionally, co-scheduling provides a simple way to minimize the negative effects of CPU virtualization, running virtual CPU siblings at the same time [19]. It can give an illusion to the guest operating system that their CPUs are dedicated to the VM. Although it can resolve the lock holder and waiter preemption problem, it causes CPU fragmentation problems, which lower the utilization of CPU resources to allow synchronized scheduling of sibling vCPUs. To minimize the negative effect, balanced scheduling was introduced [18]. It attempts to co-schedule the virtual CPU siblings, but it may occasionally break the strict requirement to avoid wasting CPU resources. However, the scalability of co-scheduling for many-core VMs is not fully investigated. Recently, for fast responses to IPI (inter-processor interrupts) demand-based scheduling boost the virtual CPU priority by capturing IPI signals [9]. Ouyang and Lange proposed the preemptable ticket spinlock to mitigate the lock holder preemption problem at a guest OS level [14].

In addition, the interrupt processing delay was widely discussed by several prior studies [10], [13], [22], [23], [24]. Although the Xen hypervisor introduced a boosting mechanism which preempts the current running vCPU to handle I/O requests quickly, VMs with both I/O and CPU oriented applications can suffer a low I/O throughput. vSlicer uses different scheduling quantums for latency sensitive VMs and non-latency senitive VMs for fast interrupt handling of latency sensitive VMs [24]. In vSlicer, VM users or the admin must assign such latency sensitive VMs manually. vTurbo addresses such mixed VM scenarios by using a dedicated core for interrupt handling [23]. As mentioned before, these works have focused on fixing each individual problem in an ad-hoc manner. Our approach is to look at the lock, IPI, and I/O related problems as a whole, and we proposed solving the virtual time discontinuity problem with the use of shorter time slices.

The architecture community also have discussed the spinlock problems in virtualization. Spin Detection Buffer [21] is an architectural support mechanism to detect CPUs being wasted due to the lock holder preemption problems. The goal of Spin Detection Buffer is similar to the newly added Intel's PLE [7] and AMD's Pause Filter [1]. The AMD virtual interrupt controller (AVIC) [2] allows interrupt handling for APIC register r/w, IPIs, and I/O interrupts to bypass the hypervisor.

To prevent thrashing workloads from polluting the cached data of co-running applications, adaptive insertion policy was proposed [16]. We adopt the adaptive insertion policy and use time sampling method along with the context prefetcher in effort to achieve the best of the two worlds. Recently, to minimize the negative effects of cache sharing in overcommitted systems, cache restoration techniques were introduced [5], [25]. The cache contexts of the virtual machine being scheduled are restored by prefetching. In traditional operating systems, Mogul and Antia studied the effects of context switches incurring cache pollution in a single processor [12].

VII. Conclusions

This papers explored the solution space of the synchronization and interrupt handling problems caused by the virtual time discontinuity in virtual machines. Although problem-specific scheduler optimizations can mitigate each problem case individually, the general time slice method proposed in this paper can potentially eliminate this negative artifact of CPU virtualization. However, the downside of supporting sub-millisecond time slice can be significant for some combinations of workloads in consolidated environments. This paper proposed a low cost solution of preserving as many contexts in the LLC as possible. Based on our analysis of active contexts in highly consolidated systems, exploiting the large LLC with context preservation will provide a promising alternative or complement a more aggressive context prefetcher.

Acknowledgment

This research was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. NRF-2012R1A1A1014586 and No. NRF-2013R1A2A2A01015514).

References

[1] "AMD64 architecture programmer's manual volume 2: System programming," Programmer's Manual, AMD, 2010.

[2] "Introduction of AMD virtual interrupt controller," Presentation Slides, XenSummit, AMD, 2012. Available: http://www-archive.xenproject.org/xensummit/xs12na_talks/M6b.html

[3] N. Bhatia, "Capacity planning and performance management on IBM PowerVM virtualized environment," 2011.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[5] D. Daly and H. Cain, "Cache restoration for highly partitioned virtualized systems," in *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[6] "Power7 virtualization best practice guide," IBM, 2012.

[7] "Intel 64 and IA-32 Architectures Software Developer's Manual," Software Developer's Manual, Intel, 2010.

[8] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 208–219.

[9] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, "Demand-based coordinated scheduling for SMP VMs," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[10] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik, "Supporting soft real-time tasks in the xen hypervisor," in *Proceedings of the 6th International Conference on Virtual Execution Environments (VEE)*, 2010.

[11] S. D. Lowe, "Best practices for oversubscription of CPU, memory and storage in vSphere virtual environments," Technical Whitepaper, Dell, 2013.

[12] J. C. Mogul and A. Borg, "The effect of context switches on cache performance," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[13] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling I/O in virtual machine monitors," in *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE)*, 2008.

[14] J. Ouyang and J. R. Lange, "Preemptable ticket spinlocks: improving consolidated performance in the cloud," in *Proceedings of the 9th International Conference on Virtual Execution Environments (VEE)*, 2013.

[15] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: A full system simulator for multicore x86 CPUs," in *Proceedings of the 48th Design Automation Conference (DAC)*, 2011.

[16] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007.

[17] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, Jan 2011.

[18] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for SMP VMs?" in *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, 2011.

[19] "VMware vSphere 4: The CPU scheduler in VMware ESX 4.1," Technical Whitepaper, VMware, 2010.

[20] "Performance best practices for VMware vSphere 5.0," Technical Whitepaper, VMware, 2011.

[21] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Hardware support for spin management in overcommitted virtual machines," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.

[22] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," in *Proceedings of the International Conference on Embedded Software (EMSOFT),*, 2011.

[23] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu, "vTurbo: Accelerating virtual machine I/O processing using designated turbo-sliced core," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC)*, 2013.

[24] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu, "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)*, 2012.

[25] J. Zebchuk, H. Cain, X. Tong, V. Srinivasan, and A. Moshovos, "RECAP: A region-based cure for the common cold (cache)," in *Proceedings of the 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.