vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines

Luwei Cheng
Facebook
chengluwei@fb.com

Jia Rao
University of Colorado at Colorado Springs
jrao@uccs.edu

Francis C.M. Lau
The University of Hong Kong
fcmlau@cs.hku.hk

Abstract

SMP virtual machines (VMs) have been deployed extensively in clouds to host multithreaded applications. A widely known problem is that when CPUs are oversubscribed, the scheduling delays due to VM preemption give rise to many performance problems because of the impact of these delays on thread synchronization and I/O efficiency. Dynamically changing the number of virtual CPUs (vCPUs) by considering the available physical CPU (pCPU) cycles has been shown to be a promising approach. Unfortunately, there are currently no efficient mechanisms to support such vCPU-level elasticity.

We present `vScale`, a cross-layer design to enable SMP-VMs to adaptively scale their vCPUs, at the cost of only *microseconds*. vScale consists of two extremely light-weight mechanisms: i) a generic algorithm in the hypervisor scheduler to compute VMs' CPU extendability, based on their proportional shares and CPU consumptions, and ii) an efficient method in the guest OS to quickly reconfigure the vCPUs. vScale can be tightly integrated with existing OS/hypervisor primitives and has very little management complexity. With our prototype in Xen/Linux, we evaluate vScale's performance with several representative multithreaded applications, including NPB suite, PARSEC suite and Apache web server. The results show that vScale can significantly reduce the VM's waiting time, and thus can accelerate many applications, especially synchronization-intensive ones and I/O-intensive ones.

1. Introduction

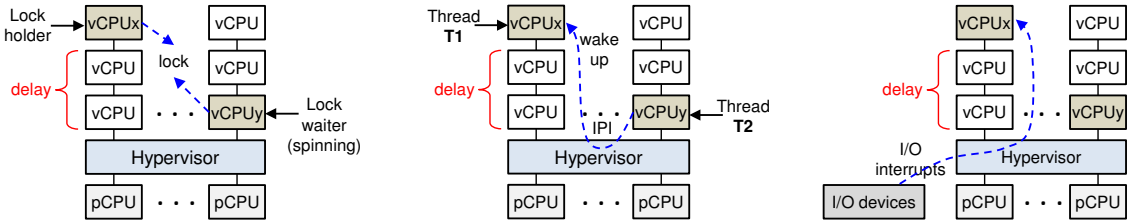
Multicore computing systems are prevalent in cloud datacenters. In order to exploit hardware parallelism, SMP-VMs are commonly adopted when hosting multithreaded applications. To boost cost-effectiveness, cloud providers allow their datacenters to be oversubscribed, i.e., by consolidating multiple independent VMs onto a small number of cores or servers. For example, in private clouds, running 40–60 VMs per server is not rare, which means 2–4 VMs per pCPU in a typical 16-core SMP machine; some deployment can run as many as 120 VMs per host [38]; in desktop virtualization, VMware even suggests putting as many as 8–10 virtual desktops on a pCPU [9]. With such a high degree of sharing a pCPU, the scheduling delays can easily be in the order of tens or even hundreds of milliseconds. The coarse-grained scheduling of modern hypervisors (Xen uses a time slice of 30ms by default [4] while VMware uses 50ms [8]) attests to this fact.

The performance of multithreaded applications critically depends on the synchronization latency between cooperative threads as well as the delay due to external I/O (if any). However, abrupt delays to vCPUs, which do not exist in dedicated environments, can seriously twist the expected behaviors, as in the following three situations. First, for applications that rely on *busy-waiting* synchronization primitives (e.g., spin lock), when a lock-holder vCPU is preempted, the contending vCPUs have to wait for a long time, wasting a tremendous amount of CPU cycles. This is illustrated in Figure 1(a): $vCPU_x$ is holding a spin lock but has been preempted by the hypervisor; meanwhile, $vCPU_y$ is vainly busy-waiting there but cannot make any progress until $vCPU_x$ gets scheduled again and releases the lock. Second, for applications that adopt *blocking* primitives (e.g., mutex and semaphore), where one thread *sleep-waits* until being woken up by the kernel scheduler via an inter-processor interrupt (IPI), the wakeup signal can also be delayed by the hypervisor scheduler (as shown in Figure 1(b)). Third, for I/O-bound applications, both throughput and responsiveness are highly sensitive to vCPU scheduling delays because preempted vCPUs are unable to respond to I/O interrupts (as shown in Figure 1(c)).

To deal with these problems, many sophisticated solutions have been proposed in previous works which mainly

EuroSys '16, April 18–21, 2016, London, United Kingdom

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901321



(a) Wasted time on preempted lock-holder. (b) Delayed virtual IPIs for kernel wakeup. (c) Delayed processing for I/O.

Figure 1: The delays from the scheduling queue (the vCPU stack) can seriously affect an SMP-VM's performance in three situations: (a) wasted CPU time in *busy-waiting* synchronization primitives such as spin lock, (b) thread's response latency in *blocking* synchronization primitives such as mutex and semaphore, (c) delayed processing for I/O interrupts.

fall into two categories: i) to modify the hypervisor scheduler to evade the virtualization reality, such as to avoid the delays by prioritizing certain types of interrupts [27, 34], to shorten the delays with soft real-time methods or smaller time slices [29, 39, 49, 50], or to make vCPUs progress at similar rates by co-scheduling them [8, 41, 46]; ii) to para-virtualize certain OS components to make them delay-aware, such as to refactor the kernel's spin lock [20, 35], the interrupt balancer [14], the transport layer [15, 21, 25], etc. Although these methods can alleviate performance degradation in their assumed scenarios, none of them offers a systematic solution to address or mitigate the problems illustrated in Figure 1 as a whole.

Actually, if an SMP-VM could let its weaker vCPUs join a few stronger ones, a large part of the scheduling delays can be avoided because each vCPU will have a higher chance of occupying a dedicated pCPU. VCPU-Bal [40] is the first work that followed and validated the effectiveness of this idea. Unfortunately, due to the lack of light-weight tuning mechanisms in the guest OS and the hypervisor, the authors are only able to "simulate" the scenario rather than doing a real run of it. To implement the idea, two key challenges must be properly addressed. The first challenge has to do with the CPU availability in the hypervisor, which often *fluctuates*. In a physical server, if one VM does not use up its CPU allocation, under work-conserving schemes, the scheduler should proportionally allocate the surplus CPU resources to the other VMs. This makes one VM's pCPU allocation highly dependent on the other VMs' consumption patterns. For example, a VM running HPC applications may tend to chew up many CPU cycles, whereas a virtual desktop running interactive applications may only need to consume CPU cycles sparingly. So when a CPU-intensive VM co-locates with an interactive VM, their pCPU availabilities should not be the same. However, current hypervisor schedulers mainly focus on fair allocation of CPU cycles, and pay minimal effort to *predict each VM's CPU extendability*, i.e., if a VM adds more vCPUs to compete for CPU cycles, what should be the VM's maximum allocation under the current machine-wide load? This semantic gap (due to the VM's lack of knowledge of pCPU utilization and co-located VMs' pCPU consumptions) precludes SMP guests from accurately determining their optimal number of active vCPUs.

The second challenge is that, as vCPU scaling may happen frequently in response to the VM's changing CPU extendability, it is important that vCPU reconfiguration be sufficiently efficient. Unfortunately, Linux's CPU hotplug (the only available mechanism in the kernel) is too heavy-weight to react promptly enough: adding or removing a vCPU can take more than 100 milliseconds. Although Chameleon [36] and Bolt [37] succeeded to significantly reduce the overhead, they are not purposely designed for *virtual* environments and thus they are suboptimal when reconfiguring vCPUs (§6 has more discussions). Since virtual SMP is significantly different from physical SMP in many respects, such as their representations and the way they interact with the underlying layer, we believe a reconfiguration design focusing specifically on *vCPUs rather than pCPUs* can be more effective.

In this paper, we propose `vScale` to address the above challenges. To make SMP-VMs aware of their CPU availability, vScale extends the hypervisor scheduler to predict each VM's maximum CPU allocation, which is derived from their entitlements and consumptions. This value is accessible to each VM via vScale's high-performance communication channel (between the hypervisor and the guest OS). In the user space, vScale's daemon closely monitors the resource changes, and then instructs vScale's kernel module to redistribute the workload onto a more reasonable number of vCPUs, i.e., pack them into fewer vCPUs or spread them over more vCPUs. Our approach does not introduce any new abstraction for the kernel to manage, but embeds the necessary actions seamlessly in Linux's current load balancing policies. The most attractive feature of vScale is its extremely low overhead: both resource monitoring and vCPU reconfiguration incur only *microsecond-level* costs. This efficiency makes possible real-time and self-scaling of vCPUs, which is infeasible in state-of-the-art virtualization platforms.

We implement a prototype of vScale with Xen 4.5.0 and Linux 3.14.15. Experiments with NPB and PARSEC show that more than half of the applications can be accelerated, especially those that involve heavy thread synchronization via either application-level busy-waiting, kernel-level busy-

waiting, or blocking. For example, in NPB's evaluations, the execution time of LU is reduced by more than 70%; in several other applications, 20% to 40% reduction of execution times can be easily observed. For the PARSEC suite, vScale can accelerate some applications from 10% to 20%. When evaluating the Apache web server which is affected by both inter-vCPU communication and I/O efficiency, vScale remarkably improves the throughput, the connection time and the response time, especially under very high load.

In the following, §2 discusses previous studies on performance problems caused by VM scheduling delays, which motivate us to work on vCPU scaling. §3 introduces the design of vScale. We describe the implementation details of our prototype with Xen/Linux in §4. Evaluation results are presented in §5, including both low-level and application-level results. §6 discusses the related work. We conclude the paper and present our plan for future work in §7.

2. Prior Solutions

Regarding performance problems caused by VM scheduling delays, previous studies followed two main directions: i) *scheduling approaches* in the hypervisor to make VMs agnostic, and ii) *para-virtualization approaches* to customize selected individual OS components

2.1 Scheduling Approaches

To tackle the Lock-Holder Preemption (LHP) problem, VMware adopts strict co-scheduling (gang scheduling) in ESX 2.x; however, this approach has the CPU fragmentation problem as well as the vCPU priority inversion problem [40], i.e., a vCPU with higher priority has to run *after* a vCPU with lower priority just because the latter is in a gang. ESX 3.x introduces relaxed co-scheduling to only track the slowest vCPU and let each vCPU make co-scheduling decisions independently [8]. Balance scheduling [41] proposes to place sibling vCPUs in different pCPUs to increase the likelihood of co-scheduling, but it affects CPU fairness because it prevents the hypervisor from migrating vCPUs for load balancing. The hypervisor can also leverage advanced hardware support to detect the guest's excessive spinning, such as Intel's Pause Loop Exiting (PLE) or AMD's Pause Filter (PF) [18, 44, 51]. For blocking primitives like mutex and semaphore, demand-based scheduling [27] identifies TLB shutdown and reschedules IPI as heuristics to preempt vCPUs. Overall, there is no known solution that can solve these problems combined that could happen to both busy-waiting and blocking synchronization primitives.

To improve VM's I/O performance, the *boost* mechanism [34] allows a blocked vCPU to preempt the current one if CPU fairness is not compromised. vSlicer [50] uses a smaller time slice for latency-sensitive VMs, but it requires the users to explicitly specify such VMs, which is difficult in practice as many VMs run a mixed workload. Soft real-time schedulers [29, 47] can also improve interrupt responsiveness, but maintaining CPU fairness remains a challenge. Side-core approaches [24, 49] partition pCPUs into *fast-tick* cores (1ms or 0.1ms) and *slow-tick* cores (30ms) in order to schedule them differentially. vAMP [26] proposes to allocate interactive vCPUs more pCPU resource. These approaches require either the hypervisor to identify the VM's I/O patterns or binding I/O tasks to specific vCPUs in the guest. Besides, a common problem is that the number of VM context switches would inevitably increase because the hypervisor keeps swapping VMs in response to I/O events. Such frequent VM switching would cause many cache flushes and thus reduce memory access efficiency (see [24]). This is probably the reason why Xen adopts 30ms as the default time slice while VMware uses 50ms.

2.2 Para-Virtualization Approaches

For the LHP problem, an OS-informed approach [42] lets the hypervisor delay the preemption of lock-holder vCPUs, but this method has security concerns as well as the CPU fairness problem. Paravirtualized spin locks [20, 35, 45] avoid wasting CPU cycles in LHP by asking the lock waiters to yield CPU control after spinning for a limited time. The essence of these approaches is to turn busy-waiting into spin-then-yield, so they can still suffer from delayed virtual IPIs for kernel synchronization. Besides, all the above approaches only target at a very specific lock: *kernel's spinlock*, but offer no way to deal with the spinning happening in *other* places, such as OpenMP's user-space spinning and the ad-hoc spinning in many applications [48].

For I/O processing, in the transport layer, vSnoop [25] and vFlood [21] leverage Xen's driver domain to continue sending and receiving packets on behalf of the VM when it is preempted; PVTCP [15] focuses on detecting spurious timeouts and abnormal RTTs. In the interrupt layer, vBalance [14] dynamically migrates interrupts from preempted vCPUs to the running ones; vPipe [22] introduces a new set of APIs to offload data processing from the VM to the hypervisor. In general, para-virtualization often requires non-trivial re-engineering effort, making kernel management more complicated. Besides, it is unclear how one solution can be integrated with another.

2.3 Motivation

All previous works assume a *fixed* number of vCPUs. On the other hand, the VM's kernel scheduler keeps distributing the workload onto *all* vCPUs. As a result, every vCPU takes part in competing for the VM's limited pCPU allocation, with each one only earning a slice per cycle and therefore suffering from a long scheduling delay. As of now, the cooperation between the OS scheduler and the hypervisor scheduler is still not fully studied. To our best knowledge, VCPU-Bal [40] is the first work that proposes the idea of *dynamic vCPUs*; but they only *simulated* the idea, primarily because there are currently no light-weight knobs for reconfiguring vCPUs: the overhead of Linux's CPU hotplug

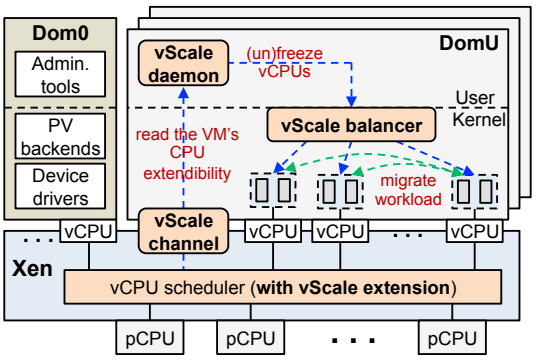


Figure 2: The architecture of `vScale` in Xen.

is over 100 milliseconds, which is too disruptive to applications with fine-grained synchronizations. Additionally, when determining the number of active vCPUs, VCPU-bal only considers the VMs' weight but not their consumption, making it not work-conserving in the hypervisor. Finally, VCPU-Bal uses a *centralized* management VM (e.g., Xen's dom0) to control the guest domains, which can potentially become a performance bottleneck as the number of VMs increases.

3. Design

This section presents the design of vScale. Using Xen [10] as a reference, Figure 2 illustrates vScale's architecture. There are three components: a user-space daemon, a kernel-space balancer and an extension in the hypervisor scheduler. vScale's user-space daemon constantly probes the VM's CPU extendability via the hypervisor and then instructs vScale's kernel balancer to swiftly redistribute the workload onto an appropriate number of vCPUs.

3.1 Design Considerations

To enable greater practicability as well as ease of use, we subscribe to the following principles when designing vScale:

- **Scalability** – Xen's dom0 actually has a full-featured toolstack which can perform many operations such as monitoring and vCPU hotpluging. Therefore, for convenience, one would be tempted to directly build a solution based on it. However, as modern hardware becomes increasingly powerful, one machine may host hundreds or even thousands of tiny VMs in the future (such as libOS VMs [28, 30]), and dom0 can easily become a performance bottleneck. We believe a scalable design should be decentralized and completely bypass dom0.
- **Flexibility** – We purposely implement vScale as an optional *service* rather than an inseparable OS component. As such, we only push new mechanisms to the low-level stacks (OS kernel and hypervisor), but export the interfaces to the user space. This flexibility is important in that if some application sets CPU affinity (so the threads should not be migrated) or assumes a fixed number of processors, vScale's service can be selectively disabled.

Algorithm 1 The calculation of VM CPU extendability

Variables: Virtual CPU v; Weight of the i_{th} VM w_i; The number of pCPUs in the shared CPU pool P; System-wide unused CPU capacity c_{slack}; Competitor VM set $\mathbb{S}$.
Output: The optimal virtual CPU number n_i for the i_{th} VM.
`/* CPU extendibility calculation period t */`
$c_{slack} = 0$
$\mathbb{S} = \emptyset$
for each VM **do**
  $s_{i,fair}(t) = \frac{w_i}{\sum w_j} \cdot t \cdot P$
  if $s_i(t) < s_{i,fair}(t)$ **then**
    $c_{slack} \mathrel{+}= s_{i,fair}(t) - s_i(t)$
    $s_{i,ext}(t) = s_{i,fair}(t)$
    $n_i = \left\lceil \frac{s_{i,ext}(t)}{t} \right\rceil$
  else
    add VM to competitor set $\mathbb{S}$
  end if
end for
for each VM in competitor set $\mathbb{S}$ **do**
  $s_{i,ext}(t) = \frac{w_i}{\sum_{\mathbb{S}} w_j} \cdot c_{slack} + s_{i,fair}(t)$
  $n_i = \left\lceil \frac{s_{i,ext}(t)}{t} \right\rceil$
end for

- **Generality** – When computing the VM's CPU capability, we do not modify the resource allocation policy but only *extend* it. The extension will also be applicable to other proportional-share schedulers.

3.2 Calculate VM's CPU Extendability

To bridge the semantic gap between the OS scheduler and the hypervisor scheduler, it is important for the VM kernel to know its maximum CPU availability, so as to maintain the workload on an appropriate number of vCPUs. vScale defines a VM's CPU *extendability* as the maximum amount of CPU it is able to receive from the hypervisor, assuming fair CPU sharing between VMs and the work-conserving CPU allocation. CPU extendability and per-pCPU capacity together determine the optimal number of vCPUs a VM should have in order to avoid scheduling delays.

In proportional CPU sharing, a VM's CPU entitlement is determined by three parameters: *weight* (relative importance), *reservation* (lower bound) and *cap* (upper bound). Under work-conserving scheduling, a VM's unused CPU time can be allocated to other VMs that have unsatisfied demands. Although a proportional-share scheduler at the hypervisor is able to distribute the unused CPU time to VMs according to their weights, the set value of the vCPU number is crucial for VM performance. While a large number of vCPUs will inevitably lead to fragmented CPU allocations that cause scheduling delays, a small number of vCPUs would likely fail to exploit hardware parallelism in SMP VMs. To this end, vScale dynamically changes VMs' vCPU numbers based on the calculation of CPU extendability.

Algorithm 1 shows the calculation of VM CPU extendability. vScale classifies VMs into two roles based on their CPU demands:

- **Competitor** – VMs that have over-consumed their fair allocation. They compete for unused CPU resources of other VMs.
- **Releaser** – VMs that have under-utilized their fair allocation. The CPU resources released by these VMs would be proportionally allocated to all competitor VMs.

vScale assumes a pool of P CPUs shared by all VMs and calculates VMs' extendability in a predefined period t. It first determines a VM's fair CPU share, i.e., $s_{i,fair}(t)$ based on the weight of the VM (line 7). The difference between a releaser VM's fair share and its actual CPU consumption, i.e., $s_i(t)$, is the VM's contribution to the overall CPU slack (line 9). The competitor VMs are added to the competitor set $\mathbb{S}$. The CPU extendability of a releaser VM is set to its fair CPU share (line 10) even it cannot fully utilize its allocation. This design ensures that a releaser VM is always able to exploit the deserved parallelism when its CPU demand ramps up. In contrast, the CPU extendability of a competitor VM is the sum of its fair share and its proportional share in the CPU slack (line 17). The optimal vCPU number of a VM is calculated as the ratio of CPU extendability and the period t (line 11 and 18). The ratio indicates how many pCPUs with full capacity a VM can possibly have given its potential maximum CPU availability (i.e., CPU extendability). Note that a VM's CPU extendability should also satisfy its reservation and cap. vScale uses a ceiling function to allow a VM to have one additional vCPU for the partial CPU allocation to the last vCPU.

The algorithm is generic and thus can be easily integrated into various proportional-share schedulers, such as the virtual-runtime based ones and their variations [13, 19, 43]. Besides, it enforces max-min fairness between VMs and prevents VMs from manipulating vCPU numbers for extra CPU allocations. CPU extendability effectively shapes the VMs for their competition for pCPUs, thereby mitigating scheduling delays.

3.3 Fast vCPU Reconfiguration

In Linux, there are mainly two types of schedulable entities that can eat up CPU cycles: *threads* and *interrupts routines*. Threads can be divided into user-level threads (uthreads) and kernel-level threads (kthreads), based on their memory address spaces. Uthreads are dynamically launched at runtime to encapsulate applications, so migrating them across CPUs will not break the kernel. As for kthreads, some have system-wide functions that serve the whole OS, and so can run on any CPU, such as filesystem related daemons (`ext4-xxx`), the kernel auditing daemon (`kauditd`) and the RCU daemon (`rcu_sched`) which is responsible for detecting RCU's grace period. Per-CPU kthreads are not migratable because they are statically created at boot time to only serve the *local* CPU. For example, `ksoftirqd` is responsible for processing local soft interrupts; `kworker` is a placeholder for kernel worker threads, which perform most of the actual processing for the kernel; the `swapper` kthread instructs the local CPU to enter idle when there are no active threads. Migrating them at random will undoubtedly cause the kernel to panic.

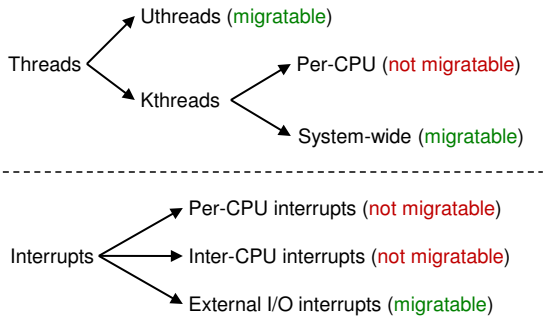


Figure 3: Linux kernel's schedulable entities.

In Linux, there are mainly two types of schedulable entities that can eat up CPU cycles: *threads* and *interrupts routines*. Threads can be divided into user-level threads (uthreads) and kernel-level threads (kthreads), based on their memory address spaces. Uthreads are dynamically launched at runtime to encapsulate applications, so migrating them across CPUs will not break the kernel. As for kthreads, some are system-wide to serve the whole OS, and so can run on any CPU, such as filesystem related daemons (`ext4-xxx`), the kernel auditing daemon (`kauditd`) and the RCU daemon (`rcu_sched`) which is responsible for detecting RCU's grace period. Per-CPU kthreads are not migratable because they are statically created at boot time to only serve the *local* CPU. For example, `ksoftirqd` is responsible for processing local soft interrupts; `kworker` is a placeholder for kernel worker threads, which perform most of the actual processing for the kernel; the `swapper` kthread instructs the local CPU to enter idle when there are no active threads. Migrating them at random will undoubtedly cause the kernel to panic.

Regarding interrupts, according to their sources and destinations, they can be divided into *local interrupts*, *inter-processor interrupts (IPIs)* and *external I/O interrupts*. Both local interrupts and IPIs are generated for *specific* CPUs. For instance, timer interrupts are generated for the local CPU to create timed events for many kernel components (e.g., the scheduling tick and resource accounting); reschedule IPIs are largely used to wake up a thread on another CPU, or to pull a thread remotely in an effort to balance the workload. Apparently, migrating these interrupts can cause correctness problems because the original CPU will lose some important events. These issues, however, are not associated with external I/O interrupts, and as long as the kernel can receive them, they can be redirected to any active CPU.

We classify the schedulable entities into different categories, as in Figure 3. Though not all of them are migrat-

Algorithm 2 The algorithm to freeze one vCPU in vScale. Unfreezing vCPU follows the similar order.

```
1: Operations performed on the master vCPU:
2: begin
3:     /* The following operations must be executed
4:     in this order */
5:     (1) Set the corresponding bit of cpu_freeze_mask, so that
6:     other vCPUs will not push tasks to the target vCPU;
7:     (2) Update the power of the scheduling domain and group
8:     that include the target vCPU;
9:     (3) Notify the hypervisor so that the target vCPU will
10:    stop earning CPU credits;
11:    (4) Send a reschedule IPI to the target vCPU, to trigger
12:    its local scheduler;
13: end
14: Operations performed on the target vCPU:
15: begin
16:    /* The following operations have no
17:    particular order */
18:    (a) In the scheduler function, move all migratable threads
19:    to other active vCPUs;
20:    (b) When the vCPU goes to idle, do not pull tasks from
21:    other active vCPUs;
22:    (c) Migrate I/O interrupts to other active vCPUs;
23: end
```

able, we find that *when uthreads, system-wide kthreads and I/O interrupts are all scheduled away, the hosting vCPU becomes completely idle*. This can be explained as follows: 1) per-CPU kthreads are actually *servants* for others, so when there are no applications and I/O interrupts to drive them, they become quiescent; 2) timer interrupt can be automatically suspended when one vCPU becomes idle (the feature of *dynamic ticks*); 3) no expensive memory context switches will happen because all kthreads share the same kernel address space; 4) as the scheduling queue is empty, other vCPUs will not issue reschedule IPIs here; 5) a vCPU that stays idle does not need to participate in RCU's grace period detection [32]. Note that in Xen, a vCPU never receives TLB shutdown IPIs due to the para-virtualization of the memory management unit (MMU) [7].

Based on the above insights, we depict vScale's steps to reconfigure vCPU in Algorithm 2. For coordination, we introduce a global CPU mask variable `cpu_freeze_mask`. Once the user-space daemon decides to *freeze* one vCPU, the master vCPU (vCPU0) immediately sets the corresponding bit of `cpu_freeze_mask`. In this way, the target vCPU will stop *pulling* tasks from other vCPUs; meanwhile, other vCPUs will not *push* tasks to the target vCPU. Another issue is that Linux CFS has the concepts of *scheduling domain* and *scheduling group* [16], so their scheduling power must also be updated accordingly. After that, the master vCPU notifies the target vCPU to migrate its local threads, by trapping into the scheduler function. vCPU deactivation follows similar operations. This *split* design ensures that the master vCPU's overhead is minimized because it does not need to block to wait for the target vCPU to finish all the migration work.

4. Implementation

We have implemented vScale in Linux 3.14.35 and Xen 4.5.0. To minimize code changes, the implementation reuses existing functions and primitives as much as possible. In the following, we describe our major modifications in detail.

4.1 Modifications to Linux Guest OS

User-level component. It is important that vScale daemon executes deterministically because we want the SMP-VM to respond timely to the underlying change of CPU extendability. To achieve this, first, we put vScale daemon in the real-time scheduling class to ensure that other fair-share threads cannot preempt it; second, we bind it to vCPU0 (the master vCPU) so that it will not be migrated. Two system calls and two hypercalls are added to enable the communication with the kernel and the hypervisor: with `sys_getvscaleinfo` and `SCHEDOP_getvscaleinfo`, the daemon can directly obtain the VM's CPU extendability from the hypervisor scheduler; with `sys_cpufreeze` and `SCHEDOP_cpufreeze`, the change of one vCPU's status can be quickly synchronized to the guest kernel as well as the hypervisor.

Kernel-level component. Linux's SMP load balancing is triggered in the following scenarios: 1) *idle balance* – when one core goes idle, 2) *fork balance* – when a new task is created, 3) *wakeup balance* – when a task wakes up, 4) *periodic balance* – the kernel regularly checks each scheduling domain to make sure that the load imbalance is within a satisfactory range. In vScale, all the above operations will consult `cpu_freeze_mask`. Specifically, after one vCPU has been frozen, *push-based* runqueue selection is forbidden in `find_idlest_cpu()` while *pull-based* load balancing is disabled in `idle_balance()`. To update the power of the scheduling domain/group that includes this vCPU, vScale calls `update_group_power()` to proactively notify them of the vCPU's absence. The hypervisor is also informed so that it can adjust the resource allocation policy. Finally, vCPU0 calls `resched_task()` to tickle the target vCPU's scheduler function to carry on with the migration work. When unfreezing a vCPU, the only difference is that vCPU0 calls `wake_up_idle_cpu()` to ask the target vCPU to migrate some threads from other active vCPUs.

In `__schedule()`, if the vCPU finds that its corresponding bit has been set in `cpu_freeze_mask`, it will first activate all migratable threads in the local `wake_list`, and then iterate the runqueue to migrate all threads away. Runqueue selection for these threads is based on the load of all active vCPUs, using `select_task_rq()`. Uthreads and kthreads can be easily distinguished with the `PF_KTHREAD` flag. Another way to tell them apart is that kthread's `mm_struct` is empty because they do not need to be accessible in the user space. For per-CPU kthreads, they are marked in `task_struct`

with an additional flag so that vScale will not migrate them (which would otherwise cause kernel panic).

Interrupts also consume considerable CPU cycles, so they must be redirected. For I/O interrupts, they are not migrated until they occur. In Xen, as all I/O interrupts are of `IRQT_EVTCHN` type, vScale can easily identify them and then redirect them by calling `rebind_irq_to_cpu()`, which in fact would make a hypercall to change the event's vCPU mapping. The current implementation only assumes *virtualized* devices. For passthrough devices (e.g., SR-IOV), we presume that it would require the hypervisor's cooperation to redirect hardware interrupts [5]. For timer interrupts, Linux kernel's *dynamic ticks* guarantee that idle vCPUs will not receive them. Otherwise, if the guest OS adopts *periodic ticks*, vScale explicitly suspends `VIRQ_TIMER` after the target vCPU becomes idle. To avoid sporadic firings, frozen vCPUs are skipped in `clocksource_watchdog`. Note that although a frozen vCPU can still respond to IPIs, it will never receive them (unless during system shutdown, via `smp_call_function`) as it does not have runnable threads for other vCPUs to tickle.

4.2 Modifications to Xen Hypervisor

The first issue is that in Xen 4.5.0, *weight* is defined as a *per-vCPU* parameter. In this model, if an SMP-VM freezes some vCPUs, it will earn less credits, which is unfair. We modify *weight* to be *per-VM* so that no matter how many vCPUs the VM uses, it receives unchanged CPU credits.

vScale is built upon the concept of *CPU-Pool*: a set of pCPUs are put together to enforce the same scheduling policy. In each pool, there is a master pCPU in charge of credit allocation for all VMs. In the master pCPU, we add another function called `vscale_ticker_fn()` to periodically calculate CPU extendability for all SMP-VMs (UP-VMs are omitted because they have no room for scaling). The default recalculation period is 10ms. Each VM's CPU consumption is accurately tracked in `burn_credits()`. The pool's idle CPU time is actually the aggregation of all *idle vCPUs'* running times (Xen runs an idle vCPU on each pCPU, similar to Linux's per-CPU idle process). To enable SMP-VMs to read their CPU extendability from vScale channel, the `struct domain` is augmented to store this value. In Xen's credit allocation function `csched_acct()`, if a vCPU has been marked as *frozen* by the guest OS, it will be removed from the domain's active list so that the other vCPUs can earn more credits (thus more chances to run).

When the master vCPU (vCPU0) freezes or unfreezes another vCPU, it is important that thread migration happens on the target vCPU in a timely fashion. Therefore, the reschedule IPIs from vCPU0 to the target vCPU (sent in `resched_task()` and `wake_up_idle_cpu()`) must be delivered as soon as possible. To this end, we ensure that the hypervisor tickles the reconfigured vCPU and prioritizes its scheduling whenever there are IPIs pending for it.

Table 1: The overhead of reading from vScale channel.

The breakdown of one operation	Overhead (μs)
(1) System call (`sys_getvscaleinfo`)	= 0.69
(2) Hypercall (`SCHEDOP_getvscaleinfo`)	+0.22 = 0.91

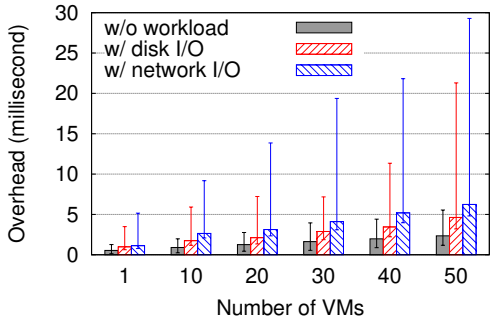


Figure 4: The min-avg-max overhead of reading VMs' CPU consumptions using `libxl`, with different background I/O workload in dom0.

5. Evaluation

We conduct the experiments on two Dell PowerEdge M1000e blade servers connected by a GbE switch. Each server has two quad-core Intel Xeon 5540 2.53GHz CPUs with hyper-threading enabled, 16 GB RAM and two 250GB SATA disks. We run Xen 4.5.0 as the hypervisor and Linux 3.14.15 as the OS for both dom0 and domUs. Because dom0 serves as the I/O proxy for all domUs, to guarantee its efficiency, we run it on 4 dedicated logical cores while domUs are put in a separate pCPU pool. We first measure vScale's mechanism-level overheads and then use several representative multi-threaded applications for a more comprehensive evaluation.

5.1 Low-level Results

vScale incurs overhead in three places: 1) computing VM's CPU extendability, 2) resource monitoring via the vScale channel, and 3) workload migration via the vScale balancer.

To compute VM's CPU extendability, vScale directly utilizes the hypervisor scheduler's runtime data, i.e., each VM's CPU consumption and allocation. The cost is similar to but intuitively smaller than Xen's credit accounting function `csched_acct()`, because vScale only needs to compute domain-level results while `csched_acct()` needs to compute both domain-level and vCPU-level results.

5.1.1 Resource Monitoring via vScale Channel

We vary the number of co-located VMs from 1 to 50. In one VM's user space, we read from the vScale channel for 1 million times to measure the execution time. Table 1 shows that the average overhead is only 0.91μs, because the operation only involves a system call and a hypercall.

To make a comparison, we evaluate the efficiency of Xen's `libxl` [2] toolstack in dom0, which is used to mon-

Table 2: The number of timer interrupts and reschedule IPIs received by each vCPU, before and after vCPU3 are frozen. The other types of IPIs such as function-call IPIs are not generated. The tick rate of the guest OS is 1000 HZ.

vTimer INTs / sec	vCPU0	vCPU1	vCPU2	*vCPU3*
all are active	1000	1000	1000	1000
vCPU3 is frozen	1000	1000	1000	**0**

vIPIs / sec	vCPU0	vCPU1	vCPU2	*vCPU3*
all are active	21.2	20.7	21.6	20.7
vCPU3 is frozen	27.1	28.7	28.3	**0**

itor each VM's CPU consumption. VCPU-Bal [40] actually adopts this centralized approach. Since dom0's major responsibility is to forward I/O for all guest domains, we evaluate `libxl` under three different scenarios: a) all VMs are idle; b) one VM conducts disk I/O using Linux `dd` command; c) one VM transmits data to another machine via the network using *netperf* [3]. Figure 4 shows the results of 10 thousand executions. First, when dom0 has no workload (the gray bar), it spends around 480μs on each VM, but the overall overhead increases linearly along with the number of VMs. Second, when dom0 gets busier with I/O, the execution time of `libxl` becomes much longer: for example, with network I/O traffic, reading 50 VMs' CPU consumption takes more than 6ms, with the maximum delay approaching 30ms. Note that this increase is only caused by *one* VM's I/O activities. If more VMs become I/O-intensive, we anticipate the overhead to be much larger because dom0 would be more congested to forward I/O data. Moreover, as future hardware will be more powerful to host more VMs per server, especially with the emergence of tiny VMs such as Mirage [30] and OSv [28], it will be more costly to monitor all VMs in dom0. In contrast, vScale is a *per-VM* approach which completely bypasses the centralized dom0.

5.1.2 vCPU Reconfiguration with vScale Balancer

We first validate the effect of freezing vCPU with the vScale balancer. In a 4-vCPU VM, we run kernel-build workload in parallel and deactivate one vCPU at runtime. We read from `/proc/interrupts` to get the number of interrupts each vCPU receives. Table 2 shows that although vScale does not disable vCPU3's interrupts, after it is frozen, it stays desirably in quiescence without being disturbed by the other vCPUs. This is because timer interrupt completely stops when the vCPU stays in idle and IPIs have been moved to the other vCPUs due to thread migration. This effect is exactly the same as that of Linux CPU hotplug, but vScale realizes it at a significantly lower cost (see below).

With the vScale balancer, reconfiguring vCPUs involves the participation of both the master vCPU and the target vCPU. To evaluate how the time is spent on the master vCPU, we instrument the `sys_freezecpu` system call to force an early return from different depths. Table 3 details the results averaged out of 1 million executions. Since vCPU0 only undertakes minimum necessary work without being blocked, its overhead is only 2.1μs. Deactivating a vCPU incurs the same overhead. To evaluate the cost on the target vCPU, we write a program to generate a desired number of threads on each vCPU, and use `ktime_get()` in `__schedule()` to record the overall migration time. It shows that on average, migrating one thread only takes 0.9–1.1μs. We speculate the time is mainly spent on runqueue selection because the vCPU needs to acquire the scheduler locks of two runqueues. For device interrupts (e.g., network and disk), vScale migrates them only when they occur. In Xen, it is realized by changing the event channel's binding vCPU via a hypercall, incurring the overhead of 0.8–1.2μs.

Next, we measure the overhead of Linux's CPU hotplug, because dom0 depends on it to add and remove vCPUs. In Xen's `libxl`, this is realized by writing the vCPU's availability to a shared database (XenStore) between dom0 and

Table 3: The overhead of freezing one vCPU in vScale balancer. Unfreezing one vCPU has a similar cost. The variable of `cpu_freeze_lock` is to prevent concurrent executions of vScale's system call.

Operations on the master vCPU (vCPU0)	Overhead (μs)
(1) System call (`sys_freezecpu`)	= 0.69
(2) Acquire and release `cpu_freeze_lock` with interrupts' status saved and restored	+0.06 = 0.75
(3) Change `cpu_freeze_mask`	+0.03 = 0.78
(4) Update the power of sched domains and groups with an RCU lock held	+0.12 = 0.90
(5) Notify the hypervisor about the change via a hypercall (`SCHEDOP_freezecpu`)	+0.22 = 1.12
(6) Send a reschedule IPI	+0.98 = 2.10

Operations on the target vCPU	Overhead (μs)
(a) Migrate N threads	= $N\times$ (0.9 $\sim$ 1.1)
(b) Migrate device interrupts	= (0.8 $\sim$ 1.2)

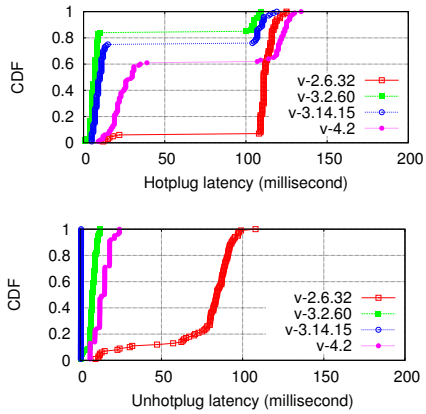


Figure 5: The overhead of Linux's CPU hotplug/unhotlpug with different kernel versions.

domU through XenBus [6]; in domU's callback function, the kernel will invoke its CPU hotplug. We evaluate 4 different kernel versions, ranging from Linux 2.6.32 to 4.2. With each version, we add and remove vCPU3 for 100 times and record the latency of each operation. Figure 5 shows the results in CDF format. It can be seen that the overhead of removing one vCPU is from a few milliseconds to over 100 milliseconds; adding one vCPU is relatively faster, being 350–500μs at best with Linux 3.14.15, but still incurs tens of milliseconds in the other 3 kernels. Compared with vScale which only incurs microsecond-level overheads, Linux CPU hotplug is slower by 100$\times$ to 100,000$\times$.

5.2 Application Results

We evaluate vScale using three popular multithreaded applications (suites). Each of them exhibits unique characteristics that can be affected by VM scheduling in different aspects, as has been illustrated in Figure 1.

- NPB-OMP 3.3 – the suite consists of 10 applications written in `OpenMP`. OpenMP allows programmers to specify the *spinning time* for thread synchronization.
- PARSEC 3.0 – the suite consists of 13 applications. Except `freqmine` (written in `OpenMP`), the others are all compiled with `Pthread` library, which adopts *sleep-then-wakeup* primitives for thread synchronization.
- Apache Web Server 2.2.15 – this application is also written in `Pthread`. Its performance is affected by the efficiency of both *inter-vCPU interrupts* and *I/O interrupts*.

5.2.1 Experimental Settings

In multi-tenant clouds, users can essentially run arbitrary workload in their VMs. As vScale is particularly useful when pCPU consumptions of co-located VMs frequently change, we set up several virtual desktops as *background VMs* to generate fluctuating workload. Desktop applications are mostly interactive-oriented and therefore generate spiked CPU consumption, such as launching an application, waiting for human's input, etc. In our settings, each background VM has 2 vCPUs and runs a "photo-slideshow" application which periodically opens a 2802$\times$1849 jpeg image file. We evaluate vScale in two types of VMs: a 4-vCPU VM and an 8-vCPU VM. During the tests, for consolidation purpose, we keep an average of 2 vCPUs per pCPU by launching a proper number of background VMs. Though with higher vCPU densities, it will be much easier for vScale to demonstrate its advantages, we find the ratio of 2 is already enough to illustrate vScale's effect. The VMs' weights are also properly configured so that all vCPUs will be treated equally by the hypervisor scheduler.

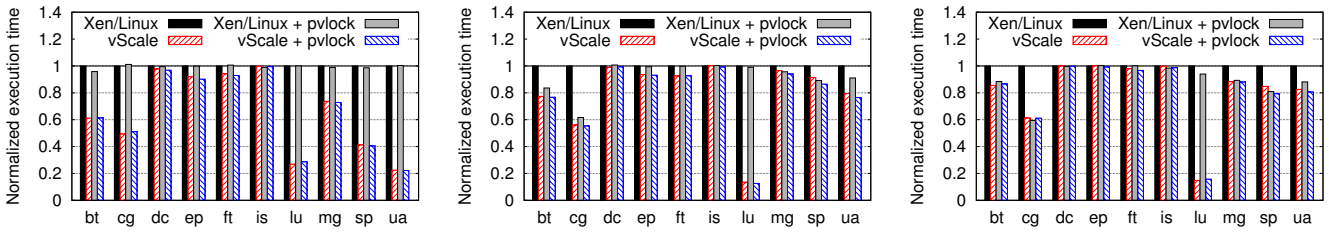
Aside from the default Xen/Linux, we also investigate the performance of Linux kernel's pv-spinlock [20]. It is important to note that though both pv-spinlock and vScale can mitigate LHP problem, they work in different layers so they can actually be integrated seamlessly. Overall, we have 4 different settings for comparison: a) vanilla Xen/Linux, which serves as the baseline; b) Xen/Linux with pv-spinlock; c) vScale; d) vScale with pv-spinlock.

5.2.2 NPB-OMP Results

In OpenMP, programmers can specify how long a thread spins before giving up CPU control. This can be done by configuring `OMP_WAIT_POLICY` and `GOMP_SPINCOUNT`. By default, `GOMP_SPINCOUNT` is automatically determined by `OMP_WAIT_POLICY`: it is 30 billion when the policy is `ACTIVE`, 0 when the policy is `PASSIVE` and 300K when the policy is undefined. In each spin operation, the program checks whether the assumed synchronization variable has reached a desired value or not; if yes, it returns directly; otherwise, it executes `cpu_relax()` function which is implemented as a compiler barrier in GCC-OpenMP, and then starts the next spinning. If the number of times of spinning has exceeded `GOMP_SPINCOUNT`, the program will give up CPU control via `sys_futex` system call ("futex" is Linux's kernel mechanism to support application-level asynchronous communication via sleep-and-wakeup), and count on the kernel's process scheduler to wake it up. Though busy-waiting wastes CPU cycles, it avoids the overhead of context switches in the kernel and the delay of virtual IPIs in the hypervisor. Since it is difficult or even impossible to determine an optimal spin threshold that fits all applications, because it depends on the running environment as well as workload patterns, we evaluate vScale with all the aforementioned values for `GOMP_SPINCOUNT`: 30 billion, 300K and 0.

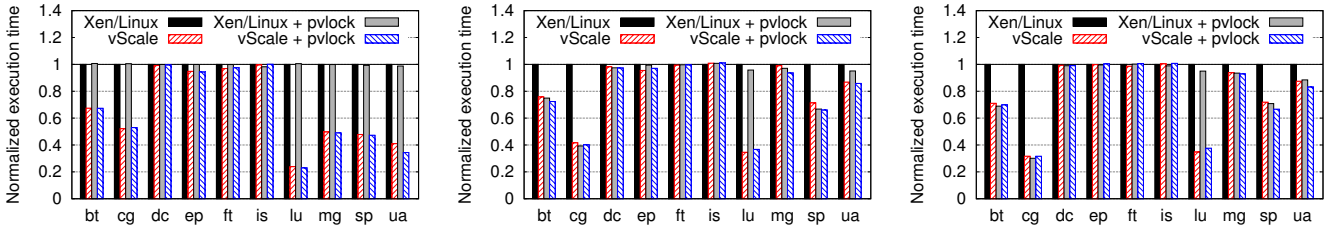
Figure 6 reports the normalized execution time with a 4-vCPU VM, while Figure 7 reports similar results with an 8-vCPU VM, out of three runs (error bars are not shown as the variance is small). Take the cases of 4-vCPU VM for example: when the spinning is very heavy, pv-spinlock has little effect, as shown in Figure 6(a), because the spinning happens in the user space rather than the kernel space; in contrast, vScale significantly reduces the execution time for many applications, e.g., 39% for `bt`, 51% for `cg`, 73% for `lu`, 59% for `sp` and 78% for `ua`. When the spin threshold decreases to 300K as in Figure 6(b), although pv-spinlock shows some benefit, e.g., in `bt` and `cg`, vScale still apparently outperforms it. When the wait policy is `PASSIVE`, as in Figure 6(c), vScale performs closely but still slightly better than pv-spinlock in most applications. The most interesting results are in the `lu` tests: vScale's performance improvement is constantly over 60%, regardless of OpenMP's waiting policy. We check the source code and find that `lu` implements its own synchronization primitives via busy-waiting, beyond the control of OpenMP.

Figure 8 presents a trace we capture of the number of active vCPUs when running `bt` in the 4-vCPU VM and the 8-vCPU VM respectively. By default, OpenMP starts a number of worker threads based on the number of online vCPUs (Linux provides this information via `cpu_online_mask`). With vScale, the SMP-VM is bestowed the ability to adap-



(a) GOMP_SPINCOUNT = 30 billion. (b) GOMP_SPINCOUNT = 300K. (c) GOMP_SPINCOUNT = 0.

Figure 6: Performance results in NPB-OMP suite with different user-level spinning counts. The SMP-VM has 4 vCPUs.



(a) GOMP_SPINCOUNT = 30 billion. (b) GOMP_SPINCOUNT = 300K. (c) GOMP_SPINCOUNT = 0.

Figure 7: Performance results in NPB-OMP suite with different user-level spinning counts. The SMP-VM has 8 vCPUs.

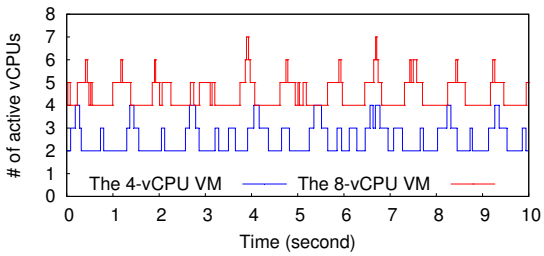


Figure 8: The change of active vCPUs when running `bt` application with vScale enabled in a 4-vCPU VM and an 8-vCPU VM respectively.

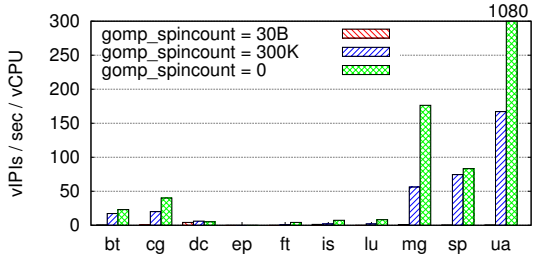


Figure 10: The average IPIs each vCPU received per second under different spinning policies in NPB-OMP experiments. The results correspond to Figure 6's "Xen/Linux" tests.

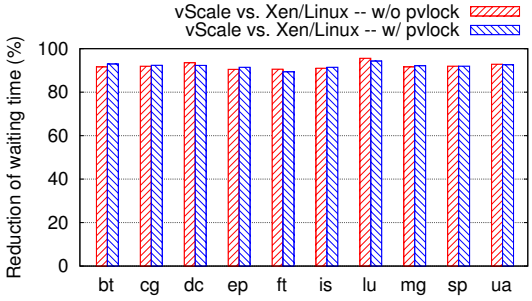


Figure 9: With vScale, VM's waiting time is significantly reduced, no matter pv-spinlock is enabled or not. The results are obtained via an extra command implemented in `libxl`.

tively change the number of its vCPUs according to the underlying pCPU availability. The benefit of such vCPU-level scaling can be clearly seen from Figure 9: regardless of whether pv-spinlock is used or not, in all applications, vCPUs' waiting time (i.e., delay) in their pCPUs' scheduling queues is reduced by over 90%. This means that all delay-sensitive components in the VM can directly enjoy this benefit without being modified.

In order to clearly identify the source of performance degradation under different spinning policies, in the hypervisor, we profile virtual IPIs for different applications. Figure 10 shows that virtual IPI's intensity is highly related to the spinning degree. When spinning is very heavy, very few virtual IPIs are triggered because it does not require thread wakeup operations. This implies that IPI-driven scheduling [27] could not be effective to deal with user-level LHP. Figure 10 also explains why vScale's benefit is limited in `ep`, `ft` and `is`, because they require little synchronization and therefore are insensitive to scheduling delays. For applications like `mg`, `sp` and `ua`, the less they spin, the more they rely on futex for thread synchronization. This explains why when `GOMP_SPINCOUNT` is 0 in Figure 6(c), vScale can also reduce the execution time for them, even without the integration of pv-spinlock. In the face of scheduling delays,

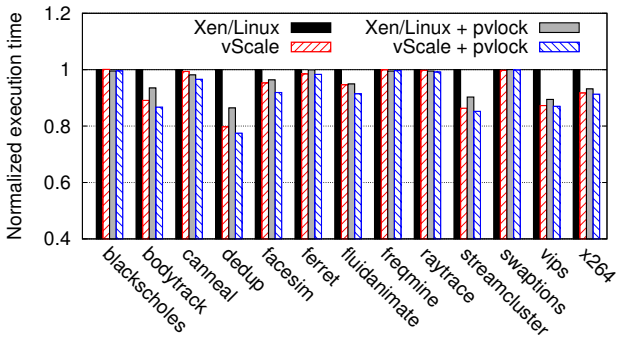


Figure 11: Performance results in PARSEC experiments with a 4-vCPU VM.

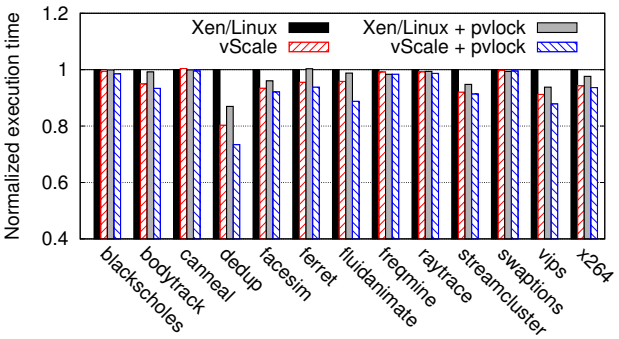


Figure 12: Performance results in PARSEC experiments with an 8-vCPU VM..

the efficiency of both active spinning and asynchronous IPIs are largely degraded. With vScale, no matter what policy is adopted in the guest and where the spinning happens, the guest OS always tries to avoid delays as much as possible by keeping a reasonable number of vCPUs.

5.2.3 PARSEC Results

The PARSEC suite contains 13 applications from different areas including computer vision, video encoding, image processing, data mining and animation physics. Except `freqmine` which is written with OpenMP, the others are all compiled with `Pthread`. In this model, thread synchronization is implemented in *sleep-then-wakeup* style. Commonly used primitives are mutex (`pthread_mutex_lock/unlock`) and conditional variable (`pthread_cond_wait/signal`). In Linux, these calls are eventually translated into kernel functions `futex_wait` and `futex_wake`. When waking up a thread, if the wakee thread resides on a different core from the signalling thread, which is quite possible in parallel applications, the kernel must notify the remote core by means of reschedule IPI. In virtualized environments, IPI latency heavily depends on the hypervisor's scheduling policy. For example, in Xen's credit scheduler, an IPI will be delayed if the target vCPU is already waiting or its credit level is very low, leading to tens of millisecond synchronization latency.

Figure 11 and Figure 12 report the results with a 4-vCPU VM and an 8-vCPU VM respectively, averaged out of 3 runs. In general, more than half of applications can benefit from vScale, but the gains are quite diversified. Take Figure 11 for example, the most obvious improvement is observed in `dedup` application: over 20%. In other 3 cases (`bodytrack`, `streamcluster` and `vips`), vScale reduces their execution time by over 10%. However, in `ferret`, `freqmine`, `raytrace` and `swaptions`, the benefit is marginal. Regarding pv-spinlock, it also has certain effect because it avoids kernel-level LHP, but the gap with vScale is still visible in a few cases: e.g., in `dedup`, such a gap is 11%.

In Figure 13, we take a closer look at the performance by profiling virtual IPIs in the hypervisor. We find that these applications exhibit very diverse characteristics. In

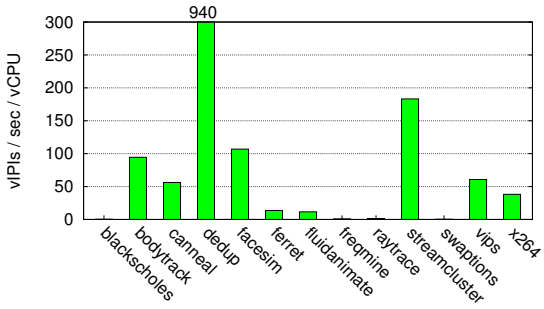
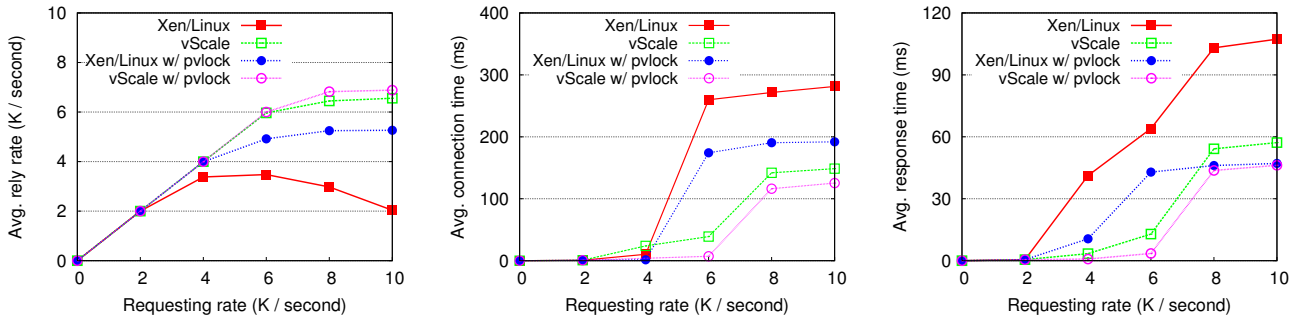


Figure 13: The average IPIs each vCPU received per second in PARSEC experiments, corresponding to Figure 11's tests.

`dedup`, a significant amount of reschedule IPIs are observed: 940/vCPU/second. Further investigation indicates `dedup` imposes heavy pressure on memory operations; in Linux kernel, `mm_struct`'s shared address space is protected by a semaphore, so it is unsurprising to see so many virtual IPIs. In `streamcluster`, we observe 183 IPIs/vCPU/second; we find the application implements its own barrier above mutex and conditional variable, where each thread waits for the next stage until all threads have arrived at a synchronization point. In `blackscholes`, `freqmine` and `raytrace`, few virtual IPIs are observed, probably because their well-partitioned data seldom need to be synchronized during execution. For `swaptions`, it does not include any synchronization primitive. It can be concluded that communication-driven applications are very susceptible to scheduling delays. With vScale, since the VM can adaptively freeze vCPUs when pCPU competition is heavy, the delays to virtual IPIs are largely avoided. Meanwhile, part of inter-vCPU communication is converted into intra-vCPU communication which is much cheaper in virtualized environments.

5.2.4 Apache Web Server Results

We use two machines to evaluate Apache web server's performance: one machine hosts the SMP-VM under test running Apache `httpd`, while the second machine runs *httperf* [1] to request for a 16KB file at a constant rate. We



(a) Average reply rate (higher is better). (b) Average connection time (lower is better). (c) Average response time (lower is better).

Figure 14: Performance results in Apache Web Server, with a 4-vCPU VM. The client constantly requests for a 16KB file from the SMP-VM via a 1GbE link with different requesting rates..

vary the request sending rate and run for 1 minute to obtain the average result. Performance is measured in reply rate, connection time and response time. Figure 14 depicts the results with a 4-vCPU VM (error bars are not shown because the deviation is small). We did not test with an 8-vCPU VM because we find 4 vCPUs are already enough to fully utilize the 1Gbps network link.

With Xen/Linux, in Figure 14 (a), the VM can only sustain low request rate: when there are less than 4K requests per second, the reply rate increases linearly along with it; however, after the requesting rate exceeds 6K/s, the reply rate gradually drops. Meanwhile, in Figure 14(b) and (c), the connection time which reflects the delay of I/O interrupt and the response time which reflects the VM's processing capability also significantly increase. When pv-spinlock is enabled, though performance break is avoided, its peak rate (5.3K/s) is still much lower than the rate that can saturate the network link (which is around 7K/s).

There are two major reasons for the performance problem: i) when every vCPU is active, each vCPU can only get a portion of pCPU, so I/O processing is inevitably delayed on the interrupt-receiving vCPU; ii) inter-vCPU interrupts can also be delayed, preventing threads on different vCPUs from communicating with each other in a timely manner. In our experiments, when the requesting rate is 6K/s, the VM observes 11.8K network receive interrupts per second and 1.7K IPIs/vCPU/second (not shown in the figure). With vScale (w/o pv-spinlock), the VM achieves 6.6K/s at the peak, 3.2 times higher than the baseline. While when vScale is integrated with pv-spinlock, the peak throughput is 6.9K/s which is very close to the optimal performance (i.e., the rate to saturate the link). At the same time, vScale achieves the lowest connection time and response time in all group tests.

6. Related Work

It is meaningful to revisit Linux's CPU hotplug [33], which has been in the kernel for over a decade. It was first designed to allow failing hardware to be removed from a running kernel (which should happen infrequently), but nowadays people use it for energy management and even to achieve real-time response [23]. Linux implements CPU hotplug using *notifiers*: each subsystem registers their own callbacks that will be executed in a certain order when a CPU comes and goes. What is most troublesome is to safely remove per-CPU kthreads, which requires global operations [31]. To synchronize all online CPUs, Linux kernel calls `__stop_machine()` to halt an application's execution for an extended period of time with interrupts disabled, and then runs `take_cpu_down()` to execute the `CPU_DYING` class of notifiers under this special context. However, stop_machine() has very heavy and disruptive atomicity (hundreds of milliseconds), which is equivalent to grabbing every spinlock in the kernel. Bhat [11] made an effort to remove CPU hotplug's dependence on stop_machine(), by introducing per-CPU rwlocks. Unfortunately, the design brings with it many subtle lock dependency problems and also performance regression [12]. Actually, in *virtualized* environments, even if the guest OS evicts everything from a vCPU, the hypervisor *never* tears it down by only putting it in the offline state. In our scenario, we only expect the target vCPU to not participate in pCPU competition, without caring whether it has been destroyed or not.

In physical environments, the OS can also reconfigure the processors via power management, e.g., low-power mode (P-states) and sleep mode (C-states). However, in virtualized systems, power management is actually an onus on the hypervisor which dictates the hardware (e.g., `xenpm` module in Xen), out of any VM's concern. In fact, a vCPU that has no active workload consumes no power at all because it is essentially a *software* entity residing in physical memory.

Chameleon [36] is a close relative to what we want to achieve. It embraces the idea of *processor proxy*: when a CPU intends to go offline, another CPU will immediately take over so that the costly global operations are smartly avoided. Efficient as it may be, the technique complicates the kernel's scheduling management. First, all per-CPU structures must be augmented to track the *proxying dependence*. Second, a new context abstraction called *proxy context* is

introduced. As a result, the *proxying CPU* has to frequently switch between its own context and the *proxy context*, thus having to deal with two different scheduling queues. This also complicates SMP load balancing as the scheduling groups/domains need to be aware of such proxy. Furthermore, if the *proxying CPU* also goes offline, a third CPU has to act as the proxy of the above two CPUs, possibly leading to *nested proxy context* which is a complex situation for the kernel to manage. In contrast, vScale only adds a single variable (`cpu_freeze_mask`) to the kernel and is tightly coupled with the existing scheduler's framework. Another difference is in handling I/O interrupts: Chameleon leverages IOAPIC's *broadcast* mode to enforce *logical address renaming*, but this method is not applicable to VMs because Xen does not support interrupt broadcasting. Fortunately, modifying event channel's vCPU affinity is very fast in Xen, as it is just a software entity rather than a separate hardware component on the motherboard. Overall, we believe Chameleon is more suited for physical environments while vScale by its design serve virtualized environments better.

Bolt [37] also aims to provide rapid CPU reconfiguration. Different from vScale that stays away from Linux's CPU hotplug, Bolt chooses to refactor it. Basically, Bolt classifies operations carried out during hotplug as *critical* and *non-critical*: only critical operations are performed immediately, while non-critical ones are done lazily hoping that they would be reused later on. Like Chameleon [36], Bolt is also designed to reconfigure *physical* CPUs, so it has to handle many hardware dependencies, such as CPU microcode and MTRR registers. In contrast, the hypervisor provides a much neater environment for vScale to reconfigure its *virtual* CPUs. As such, vScale incurs much lower overhead (i.e., microseconds latency) compared with the millisecond-level overhead of Bolt. Further, we show that vScale is effective in improving the performance of synchronization- and I/O-intensive workloads. Bolt was evaluated with a mostly idle system and it is unclear how application performance would benefit from the proposed new CPU hotplug mechanism.

Gleaner [17] studies the "blocked-waiter wakeup" problem, which focuses on the cost from the hypervisor's intervention into synchronization-induced vCPU idling. It also introduces a technique to migrate tasks among a varying number of vCPUs, by manipulating tasks' *processor affinity* in the user space. However, this is not easy to maintain at runtime because application threads are often dynamically launched and terminated which must be accurately tracked. Besides, once a new vCPU is activated or deactivated, all tasks' masks need to be modified accordingly, which could be very costly when there are too many threads. Our approach is transparent to applications and is much more lightweight as it does not need to tamper with any thread's affinity but just modifies a single kernel variable. We believe our mechanism can assist Gleaner to work more efficiently.

7. Conclusions and Future work

Although virtualization has been in development for years, many of its performance problems are still short of a satisfactory solution. In terms of latency, the abstraction of vCPU does not always match well with pCPUs for which applications and OSes are originally designed. *Dynamic vCPUs* is a promising approach for SMP-VMs, but this idea is unrealistic in current platforms due to the scheduling semantic gap between the guest OS and the hypervisor, as well as the lack of enough efficient knobs. In this paper, we propose vScale, a design using super light-weight mechanisms to help SMP guests adaptively scale their vCPUs in real time. In our future work, we look for a broader usage of vScale. Intuitively, it should be beneficial if applications can be made aware of the VM's real computing power. Therefore, it would be interesting to explore how vScale's interface can directly assist applications to optimize their policy-specific decisions.

8. Acknowledgments

We thank the anonymous reviewers and our shepherd, Prof. Edouard Bugnion, for comments that improved this paper. This work was supported in part by a Hong Kong RGC CRF grant (No. C7036-15G) and a U.S. National Science Foundation grant (No. CNS-1320122).

References

[1] Httperf. `http://www.hpl.hp.com/research/linux/httperf/`.

[2] Libxenlight (libxl). `http://wiki.xen.org/wiki/Choice_of_Toolstacks`.

[3] Netperf. `http://www.netperf.org/`.

[4] Xen Credit Scheduler. `http://wiki.xen.org/wiki/Credit_Scheduler`.

[5] Xen PCI Passthrough. `http://wiki.xen.org/wiki/Xen_PCI_Passthrough`.

[6] XenBus. `http://wiki.xen.org/wiki/XenBus`.

[7] Xen's x86 paravirtualised memory management. `http://wiki.xenproject.org/wiki/X86_Paravirtualised_Memory_Management`.

[8] The CPU scheduler in VMware vSphere 5.1. *VMware Technical White Paper*, 2013.

[9] VMware Horizon View Architecture Planning 6.0. *VMware Technical White Paper*, 2014.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. SOSP*, 2003.

[11] S. S. Bhat. CPU hotplug: stop_machine()-free CPU hotplug. `https://lkml.org/lkml/2013/1/22/44`.

[12] S. S. Bhat. percpu_rwlock: Implement the core design of per-CPU reader-writer locks. `https://lkml.org/lkml/2013/3/5/329`.

[13] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng. Group ratio round-robin: O(1) proportional share schedul-

ing for uniprocessor and multiprocessor systems. In *Proc. USENIX ATC*, 2005.

[14] L. Cheng and C.-L. Wang. vBalance: using interrupt load balance to improve I/O performance for SMP virtual machines. In *Proc. ACM SoCC*, 2012.

[15] L. Cheng, C.-L. Wang, and F. C. M. Lau. PVTCP: Towards practical and effective congestion control in virtualized datacenters. In *Proc. IEEE ICNP*, 2013.

[16] J. Corbet. CFS group scheduling. `http://lwn.net/Articles/240474/`, 2007.

[17] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *Proc. USENIX ATC*, 2014.

[18] Y. Dong, X. Zheng, X. Zhang, J. Dai, J. Li, X. Li, G. Zhai, and H. Guan. Improving virtualization performance and scalability with advanced hardware accelerations. In *Proc. IEEE IISWC*, 2010.

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

[20] T. Friebel and S. Biemueller. How to deal with lock holder preemption. In *Xen Developer Summit*, 2008.

[21] S. Gamage, A. Kangarlou, R. R. Kompella, and D. Xu. Opportunistic flooding to improve TCP transmit performance in virtualized clouds. In *Proc. ACM SoCC*, 2011.

[22] S. Gamage, C. Xu, R. R. Kompella, and D. Xu. vPipe: Piped I/O offloading for efficient data movement in virtualized clouds. In *Proc. ACM SoCC*, 2014.

[23] T. Gleixner, P. E. McKenney, and V. Guittot. Cleaning up linux's CPU hotplug for real time and energy management. *SIGBED Rev.*, 9(4):49–52, Nov 2012.

[24] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia. I/O scheduling model of virtual machine based on multi-core dynamic partitioning. In *Proc. HPDC*, 2010.

[25] A. Kangarlou, S. Gamage, R. R. Kompella, and D. Xu. vSnoop: Improving TCP throughput in virtualized environments via acknowledgement offload. In *Proc. SC*, 2010.

[26] H. Kim, S. Kim, J. Jeong, and J. Lee. Virtual asymmetric multiprocessor for interactive performance of consolidated desktops. In *Proc. VEE*, 2014.

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

[28] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv – optimizing the operating system for virtual machines. In *Proc. USENIX ATC*, 2014.

[29] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the Xen hypervisor. In *Proc. VEE*, 2010.

[30] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proc. ASPLOS*, 2013.

[31] P. McKenney. The linaro connect scheduler minisummit. `https://lwn.net/Articles/482344/`, 2012.

[32] P. McKenney. The new visibility of RCU processing. `https://lwn.net/Articles/518953/`, 2012.

[33] Z. Mwaikambo, A. Raj, R. Russell, J. Schopp, and S. Vaddagiri. Linux kernel hotplug CPU support. In *Proc. Linux Symposium*, 2004.

[34] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in virtual machine monitors. In *Proc. VEE*, 2008.

[35] J. Ouyang and J. R. Lange. Preemptable ticket spinlocks: improving consolidated performance in the cloud. In *Proc. VEE*, 2013.

[36] S. Panneerselvam and M. M. Swift. Chameleon: Operating system support for dynamic processors. In *Proc. ASPLOS*, 2012.

[37] S. Panneerselvam, M. M. Swift, and N. S. Kim. Bolt: Faster reconfiguration in operating systems. In *Proc. USENIX ATC*, 2015.

[38] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending networking into the virtualization layer. In *Proc. HotNets*, 2009.

[39] R. Rivas, A. Arefin, and K. Nahrstedt. Janus: a cross-layer soft real-time architecture for virtualization. In *Proc. HPDC*, 2012.

[40] X. Song, J. Shi, H. Chen, and B. Zang. Schedule processes, not VCPUs. In *Proc. APSys*, 2013.

[41] O. Sukwong and H. S. Kim. Is co-scheduling too expensive for SMP VMs? In *Proc. EuroSys*, 2011.

[42] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Virtual Machine Research and Technology Symposium*, 2004.

[43] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical report, Massachusetts Institute of Technology, 1995.

[44] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proc. PACT*, 2006.

[45] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic adaptive scheduling for virtual machines. In *Proc. HPDC*, 2011.

[46] C. Weng, Z. Wang, M. Li, and X. Lu. The hybrid scheduling framework for virtual machine systems. In *Proc. VEE*, 2009.

[47] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: towards real-time hypervisor scheduling in Xen. In *Proc. EMSOFT*, 2011.

[48] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Proc. OSDI*, 2010.

[49] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu. vTurbo: Accelerating virtual machine I/O processing using designated turbo-sliced core. In *Proc. USENIX ATC*, 2013.

[50] 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 *Proc. HPDC*, 2012.

[51] L. Zhang, Y. Chen, Y. Dong, and C. Liu. Lock-Visor: An efficient transitory co-scheduling for MP guest. In *Proc. ICPP*, 2012.