Offloading Interrupt Load Balancing from SMP Virtual Machines to the Hypervisor

Luwei Cheng, and Francis C.M. Lau, *Senior Member, IEEE*

Abstract—Cloud computing increasingly leverages SMP virtual machines (VMs) to host multi-threaded applications. Interrupt balancing as a problem becomes more challenging because VMs are subject to the hypervisor's scheduling. Since the scheduling delays are typically tens of milliseconds, when they are added to one VM's interrupt delivery, they can seriously degrade the VM's I/O performance. Traditional balancing techniques are designed for dedicated environments, which cannot work well in virtualized environments because VMs are disallowed to directly control the hardware in many cases. In this paper, we present *hBalance*, a very simple approach to offload interrupt load balancing from SMP-VMs to the hypervisor. To accelerate the interrupt processing, our approach does not require shortening the hypervisor's scheduling time slice, but dynamically redirects interrupts from preempted virtual CPUs to running ones in a balanced manner. hBalance supports both Fully Virtualiized (FV) guests and Para-Virtualized (PV) guests, and exhibits high portability among various hypervisors. With our prototype implementation in Xen, the experimental results with both micro-level and application-level benchmarks show that hBalance significantly improves SMP-VMs' I/O performance while introduces moderate overhead.

Index Terms—SMP Virtual Machines, Interrupt Load Balancing, I/O Performance

1 Introduction

Cloud platforms adopt VMs to provide on-demand computing services. To improve hardware resource utilization, independent workloads are often consolidated in the same machine using VMs. As multi-core computer systems become prevalent, SMP-VMs have been widely deployed to exploit their inherent parallelism to cope with heavy workloads. One useful feature of SMP-VMs is the ability to adapt to the changing resource demand. For example, when the workload increases, an SMP-VM can expand its computing capability by running on dedicated cores; while during off-peak periods, it can be simply consolidated with other VMs. In this consolidated environment, I/O performance is critical for communication-intensive applications.

To guarantee I/O performance, interrupts must be served as soon as possible. Interrupt processing in physical SMPs typically includes two stages: (1) when a core receives an interrupt, it must suspend the current task to fetch the data from the I/O device to the kernel stack; (2) after the data is passed to the user space, a number of child threads may be created or waked up by the corresponding application to process the data. This is true of many web server applications: in Apache HTTP server, which is a multi-process multi-threaded application, the daemon listens to a TCP port and after receiving a new request, it will spawn a copy of itself to process the request while the parent goes back to listening. Since each child inherits almost everything from its parent (e.g., memory address space, global variables, binary code and loaded libraries), initially they run on the same core. Regarding virtual SMP systems, the scenario is similar but starts to deviate when the hypervisor takes over the hardware and multiplexes the underlying resources among different VMs. Figure 1 clearly shows this difference: when there are multiple virtual CPUs (vCPUs) running on one physical CPU (pCPU), vCPU preemption is usually inevitable; such scheduling delays can seriously affect the timeliness of interrupt processing.

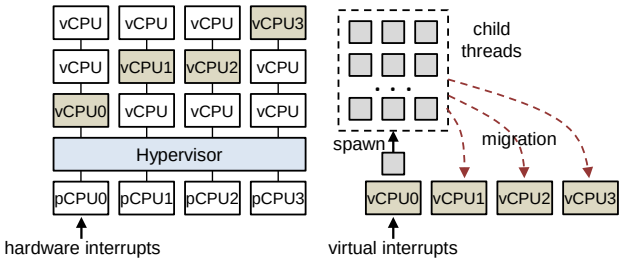


Fig. 1: In a virtualized system, the hypervisor translates hardware interrupts into virtual interrupts, and one pCPU is often time-shared by several vCPUs.

In a dedicated environment, to amortize the overhead of the first stage, the OS either relies on hardware chipsets to automatically distribute the interrupts to different cores, or adopts a user-level daemon (e.g., *irqbalance* [1] in Linux) to periodically change the interrupt-receiving core. To balance the spawned threads in the second stage, the process scheduler takes the role to migrate them among different cores. However, in a virtualized environment, the problem is more complicated in that *hardware interrupts* are served by the hypervisor which manages all the native device drivers, and VMs see and process only *virtual interrupts*. Therefore, the need for load balancing exists in three levels: hardware interrupts, virtual interrupts and the spawned threads.

Take Xen [11] for example, hardware interrupts are bal-

Luwei Cheng is now at Facebook (email: chengluwei@fb.com); Francis C.M. Lau is with the Department of Computer Science, the University of Hong Kong (HKU), (e-mail: fcmlau@cs.hku.hk). This work was partially done during Luwei Cheng's PhD study at HKU.

anced by the driver domain (dom0) which is out of any guest VM's concern (§2 has more details). As dom0 often runs on dedicated pCPUs, it is similar to a dedicated environment. However, for virtual interrupts, traditional OS-level methods are suboptimal in several aspects. *First*, vCPU preemption is totally transparent to VMs; if an interrupt is routed to a waiting vCPU, it will not be seen by the SMP-VM until that vCPU gets scheduled again. *Second*, since the newly created threads have many things in common with their parent, the process scheduler would tend to retain them on the same core as long as possible in order to maximize the CPU cache effect; this is different from `exec()` in which the child is replaced with *another* program, and the kernel will try to migrate the child to another core for load balancing because it has the smallest effective memory and cache footprint. *Third*, the process scheduler may attempt to use as fewer cores as possible, assuming that idle cores can stay in sleep mode to save power (§3 has more details); however, this assumption is not true for VMs because power is managed by the hypervisor which sits on the physical CPUs; on the other hand, as the hypervisor scheduler treats all vCPUs of an SMP-VM equally when allocating CPU quota, imbalanced load potentially causes wasted resources.

We observe that the vCPUs of an SMP-VM are mostly scheduled *independently* in each pCPU's run queue, so it is very likely that when one vCPU is waiting, some other is already running. Therefore, if interrupts can be dynamically migrated from preempted vCPUs to running ones: (1) they can be processed immediately; (2) *no* vCPU context switch will occur. In our prior work, we propose an *OS-level* solution called *vBalance* [13] to migrate virtual interrupts *within the guest*. However, this approach only works for PV guests, without any support for FV guests; besides, its portability is also very limited. In this paper, we argue that *the hypervisor* is actually at a vantage point to balance I/O for SMP-VMs, because it knows exactly about each vCPU's runtime scheduling state. To this end, we propose *hBalance* to offload such functionality *from the guest OS to the hypervisor*. Our new design is based on two representative interrupt delivery models: virtual APIC, adopted by most hypervisors; and event channel, introduced by Xen's PV technology. For APIC-based guests, hBalance simply reuses its logical destination mode, requiring no modifications to the OS; regarding PV guests, we introduce *OS-specific* event channel, which decouples I/O pertaining to a specific vCPU so that *all* vCPUs can process the virtual interrupts (more details are in §4). At the core of hBalance, a scheduling-aware routing algorithm is proposed to avoid vCPU scheduling delays as much as possible. We also optimize the current proportional-share scheduler for SMP-VMs, making it more I/O-friendly by periodically compensating vCPUs that have served I/O on behalf of the whole SMP-VM.

We have implemented a prototype of hBalance in Xen 4.2.2. The experimental results show that hBalance significantly reduces I/O latency and improves I/O throughput. In the SPECWeb tests, hBalance improves throughput by 48.1% for a FV guest and 15.5% for a PV guest, while reducing average response times by 48.5% and 64.2% respectively. Such benefit is obtained *without* shortening the vCPU scheduling time slice, but by appropriately routing virtual interrupts. When the current targeted vCPU is preempted, in the FV tests there are over 60% of cases where hBalance can find another running vCPU to receive virtual interrupts; in the PV tests, such preemption-free opportunities are above 80%. hBalance introduces acceptable overhead: only about 18% extra vCPU context switches in the FV tests and less than 50% in the PV tests.

The remainder of the paper is as follows. §2 makes plain the interrupt delivery procedures in SMP-VMs. §3 discusses the limitations of existing solutions. §4 introduces the principles and algorithms of our hBalance. We present the prototype implementation details in §5. Our solution is evaluated in §6. §7 discusses the related work. We conclude our work in §8.

2 Background

To understand how interrupts are delivered in virtualized environments, in this section, we detail two typical models for SMP-VMs: (a) "virtual APIC" model, which is broadly implemented in Xen [11], KVM [22], VMware [37] and Hyper-V [9] to support unmodified FV guests; (b) "event channel" model, which is specific to Xen's PV technology. Since Xen includes both models, in Figure 2, we use it as an example to illustrate their differences.

Xen provides basic mechanisms for its upper-layer domains, such as CPU proportional sharing, memory sharing and I/O device emulation. Hardware interrupts from I/O devices first arrive at the physical IOAPIC (pIOAPIC) which is responsible to redirect the signal to a physical Local APIC (pLAPIC) via the connected interrupt pin[1]. After that, the so-called *driver domain* which contains the real device driver[2], will process the interrupts in step 1. The requests are then forwarded to the targeted VM's virtual devices (fully-virtualized or para-virtualized).

2.1 Virtual APIC

To support FV guests, the hypervisor needs to present a similar interrupt processing architecture to VMs as that in dedicated environments. Xen adopts QEMU [12] to emulate I/O devices. Once the QEMU thread receives a virtual interrupt in step 2, it will invoke the VM's virtual IOAPIC (vIOAPIC). Depending on how the guest OS configures vIOAPIC, the interrupt will be injected into a certain virtual Local APIC (vLAPIC) in step 3. If the targeted vCPU runs on another pCPU, which is very likely because the driver domain often runs on dedicated pCPUs to guarantee I/O efficiency, Xen will send an IPI (Inter-Processor Interrupt) to let the remote pCPU trap into the hypervisor. Finally in step 5, the hypervisor scheduler will determine whether or not the targeted vCPU should get scheduled to process the interrupt.

Virtual APIC is commonly implemented according to the Intel 82093AA chipset specification [6]. Basically, there are two fields in the I/O redirection table (IOREDTBL) dictating

1. Modern PCIe devices are not pin-based, but use in-band MSI/MSI-X to directly interact with pLAPIC, bypassing pIOAPIC.

2. It is worth noting that some other hypervisors like VMware's ESX(i) contain device drivers as components (called *VMkernel*) [7], so sometimes the driver domain is also considered a secondary part of Xen hypervisor.

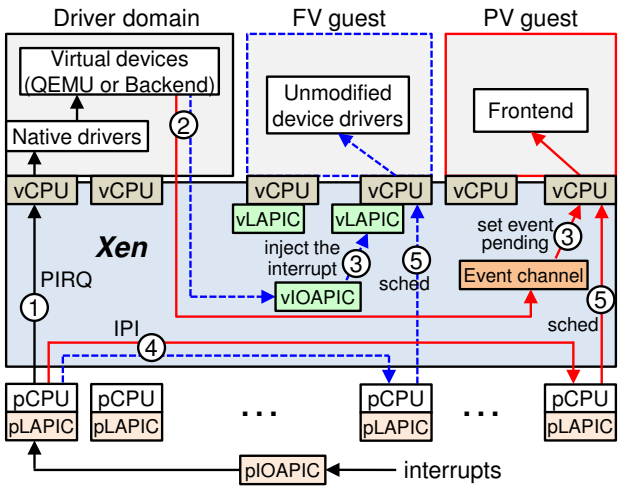


Fig. 2: Xen adopts virtual APIC to support FV guests and event channel to support PV guests.

how an I/O interrupt should be routed: *destination mode* and *delivery mode*. Figure 3 shows the options for the two modes. When *physical* destination mode is used, the interrupt will be delivered to one predefined core, regardless of the delivery mode; the guest OS also disables the other cores to serve this interrupt, and if accidentally another core receives this interrupt, it will be viewed as spurious and then discarded. When the destination mode is *logical*, more than one core would be allowed to receive this interrupt: in *Fixed* delivery mode, the interrupt will be broadcasted to all cores listed in the destination filed of IOREDTBL; in *Lowest Priority* delivery mode, the interrupt will be delivered to the core that has the lowest "task priority" at the moment, resulting in a round-robin (RR) fashion in practice. Linux does not use the Fixed delivery mode, as the broadcast operation is too expensive and also unnecessary.

2.2 Event Channel

Xen's PV technology uses *event* to abstract *interrupt*, and introduces a *split-driver* model for I/O communication: a *frontend* residing inside the guest OS communicates with its *backend* counterpart in the driver domain. Upon receiving data, the backend will put the data in the shared memory for the frontend to retrieve, and then notify a *predefined* guest vCPU via an event channel in step 2. For the notified vCPU, it will receive a pending event in step 3. Similarly in steps 4 and 5, after an IPI, the hypervisor will determine whether the targeted vCPU should be scheduled or not to process the event.

By comparison, Xen's event channel is actually another form of the *physical* destination mode of virtual APIC: each event channel is statically bound to *one* vCPU, while the other vCPUs are disallowed to see this type of event. For an SMP-VM, if the hypervisor redirects the event to another vCPU, the event cannot be processed because it has been masked out for the other vCPUs by the guest kernel. In the current implementation, all I/O events are delivered to vCPU0 by default, and this mapping can only be changed by the guest OS. For UniProcessor (UP) VMs, it makes no difference because all interrupts are bound to the only

Fig. 3: Intel IOAPIC uses the above two modes to determine how an I/O interrupt should be routed. It should be noted that in the delivery mode, there are other options (SMI, NMI, INIT and ExtINT) which are not used for I/O interrupts.

vCPU; but for SMP-VMs, virtual APIC is apparently more flexible because it allows multiple vCPUs to serve I/O in the *logical* destination mode.

3 Problems

3.1 Physical mode or Logical mode

In physical SMP systems, the RR routing of the logical destination mode can cause performance problems in some scenarios, because the change of the targeted core of every interrupt will reduce the CPU cache effect. Take network I/O for example, when one interrupt arrives, IOAPIC directs it to one of the cores; and next time, the interrupt will be directed to yet a different core; as a result, two different cores will work with the same TCP connection and both of them have to fetch its content into their own caches. As such, IOAPIC's RR routing is usually disabled by setting the interrupt's CPU affinity, resulting in the same effect as that in the physical destination mode. To achieve load balancing, Linux makes use of *irqbalance* [1] software daemon to periodically migrate interrupts from overloaded cores to underloaded cores, in a coarse-grained manner (e.g., every 10 seconds).

In virtualized SMP systems, irqbalance is still recommended to balance *hardware interrupts* in *the driver domain* when it runs on dedicated pCPUs [8]. But for guest domains, since there are often several vCPUs sharing one pCPU, vCPU scheduling delays are unavoidable. If the targeted vCPU has been preempted when an interrupt comes, I/O processing will be delayed, by typically 10× milliseconds (the default scheduling time slice is 30ms in Xen [5] and 50ms in VMware ESX(i) [10]). Due to the semantic gap, the guest OS has no knowledge of each vCPU's runtime state; therefore irqbalance has no way to properly respond to VM scheduling delays. Besides, its *second-level* IRQ remapping interval is also incapable to react timely to a vCPU's *millisecond-level* preemption.

In conclusion, the logical destination mode allows multiple cores to serve I/O, which is a useful feature, but its RR routing is not so desired; the physical destination mode plus irqbalance can effectively balance the interrupts when pCPUs are *dedicated* to vCPUs, but this combination is less effective *when one pCPU is time-shared by multiple vCPUs.*

3.2 Task Balancing in the Guest OS

To efficiently utilize CPU cycles in multi-core systems, contemporary OSes use per-core runqueues to schedule tasks, and tasks are dynamically migrated across different cores for load balancing. Take Linux's process scheduler

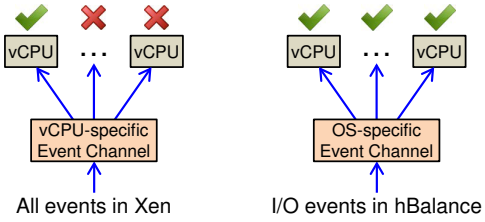


Fig. 4: For Xen's PV guests, we extend *I/O events* to be OS-specific, while the other vCPU-specific events are not affected.

(CFS) for example, task migration happens typically in three situations: (1) when one core goes idle, it will attempt to pull a task from the busiest runqueue if the average idle period is larger than the cost of migrating a task; (2) when a task wakes up or a new task is created, runqueue selection will consider the overall task balance of the system; (3) a periodic attempt will be made to balance the current scheduling domain if it has not been balanced for longer than a predefined time interval. However, there are also constraints on migrating tasks. In particular, the balancer resists migrating cache-hot tasks, and one task is prone to be scheduled on a particular core for as long as possible. Besides, power saving has also been taken into account: if tasks can be consolidated on fewer cores when the system is not heavily loaded, idle cores can enter low-power states [32]. In the scenario of Apache web server as shown in Figure 1, we observe that child threads are largely running on the interrupt-receiving core, while the other cores are much less loaded.

In virtualized environments, the above balancing strategies cannot work effectively in several aspects. First, the underlying pCPU topology is usually invisible to the guest OS; adding to the complication, one vCPU can be occasionally relocated to a different pCPU by the hypervisor scheduler for global load balancing. Second, power management is in fact an onus on the hypervisor which dictates the hardware (e.g., `xenpm` module in Xen), out of any VM's concern. Third, from the perspective of an SMP-VM, since each vCPU only gets a portion of CPU cycles in each allocation period, to better utilize the limited resource, it is more desired to spread the tasks across *all* vCPUs rather than using as fewer as possible. Lastly, Linux kernel relies on each core's historical utilization to determine whether the core has ever been overloaded or not; but when multiple vCPUs time-shares one pCPU, the measurements are not accurate so the kernel is unable to correctly estimate each vCPU's utilization, making CFS fail to migrate many tasks.

4 Solution

We propose *hBalance*, a shim layer residing in the *hypervisor* to balance interrupt workloads for SMP-VMs, in a very simple but effective way. Our solution supports both FV guests and PV guests, and has very high portability among various hypervisors.

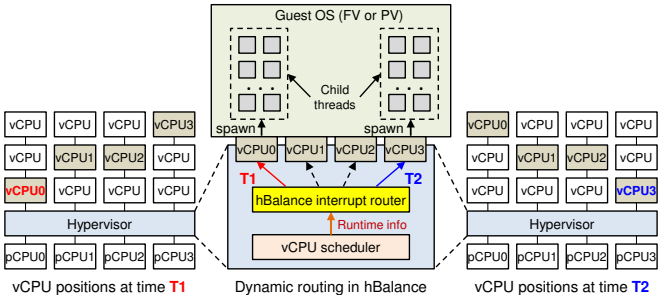


Fig. 5: hBalance dynamically routes virtual interrupts from a preempted vCPU to a running one. In this way, the spawned child threads would not crowd into a single interrupt-receiving vCPU.

4.1 Basic Principles

4.1.1 Enable the Flexibility – Decouple I/O from vCPU

In SMP systems, some interrupts are generated for *specific* CPUs, which *cannot* be redirected to elsewhere. For instance, periodic timer interrupts are generated for the local core to create timed events for many kernel services; performance monitoring interrupts report runtime information of the local core, which are critical for performance profiling tools; IPIs are largely used to wake up processes on remote CPUs, to pull a process remotely in an effort to spread the workload, and to synchronize the cache and the memory management unit between CPUs.

In contrast, interrupts from external *I/O devices* do not have bias towards a particular CPU; therefore they are *OS-specific* rather than *CPU-specific*. However, recall the two representative interrupt delivery models mentioned in §2, in vAPIC's physical mode and Xen's PV event channel, virtual interrupts are forcibly delivered to one *predefined* vCPU, preventing the hypervisor from redirecting them to other vCPUs. To be not bound by this limitation, with vAPIC, hBalance simply reuses logical mode's flexibility; with Xen's event channel, we introduce *OS-specific* event channel which allows *every* vCPU to access it, as shown in Figure 4. In this way, the hypervisor gains full flexibility to select vCPUs when delivering virtual interrupts, regardless of the VM's type. Note that our OS-specific event channel will not introduce contentions between different vCPUs, because microscopically one I/O interrupt is set as pending to only *one* vCPU at a time, therefore the guest OS will never observe two vCPUs accessing the same event channel simultaneously.

4.1.2 Offload the Responsibility – In-hypervisor Virtual Interrupt Routing

With the routing capability enabled in the above, interrupt balancing can be offloaded from the guest OS to the hypervisor. The rationality behind our design is that: it is the hypervisor rather than the guest OS that schedules vCPUs, and virtual interrupts are initiated *by* the hypervisor and then applied *to* the guest OS. The hypervisor thereby has all the smartness to optimize the interrupt routing policies. Since interrupt redirection is totally transparent to VMs, the guest OS does not need to instrument each IRQ's CPU affinity. Furthermore, when interrupts are evenly

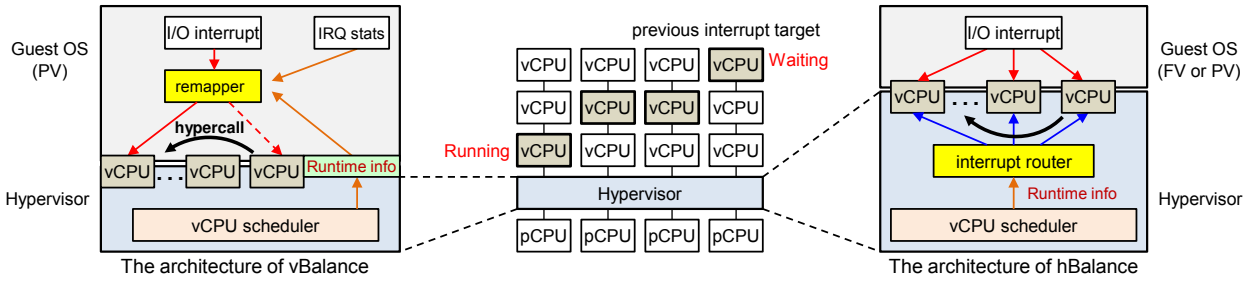


Fig. 6: vBalance [13] is a cross-layer design to migrate interrupts in the guest OS kernel, which is only applicable to PV guests. By comparison, hBalance offloads such functionality from the guest OS to the hypervisor, supporting both FV and PV guests.

TABLE 1: *Hypervisor-level* balancer is apparently more generic and neater than *OS-level* balancer.

	VM support	*Interrupt migration*	*Guest-hypervisor shared data*	*Interrupt load balancing*
vBalance	PV only	Use hypercall to sync	vCPU's scheduling status	Use IRQ statistics in the guest OS
hBalance	FV and PV	Directly in the hypervisor	Shareless	Scheduling-aware interrupt routing

distributed at the millisecond time granularity, the child threads triggered by interrupts can be uniformly created on *all* vCPUs as shown in Figure 5, without completely counting on the process scheduler to balance them. This is more efficient than to migrate the threads *after* they have all been spawned on *one* vCPU as shown in Figure 1. This way, we do not need to modify the process scheduler, which is not possible for commodity OSes and likely also lacks the general applicability for other scenarios.

4.1.3 hBalance vs. vBalance

Without the above two principles, as a comparison, our previous work vBalance [13] has to migrate interrupts *within the guest OS*. This is because Xen's vCPU-specific event channel forbids the hypervisor to redirect I/O events, so the guest OS has to perform this duty. To obtain each vCPU's scheduling status, vBalance requires the hypervisor to pass such runtime information to the guest space, using an additional communication channel. This OS-level method, however, has many limitations. First, it is tightly coupled with Xen's PV implementation, leaving all FV guests (which are more widely deployed) unsupported. Second, when vBalance remaps an interrupt to another vCPU, the guest OS must trap into the hypervisor to synchronize this change, via a hypercall. Although hypercall has been optimized to be very light-weight, it still incurs certain CPU overhead when it happens too frequently, e.g., every 10×ms as introduced by VM scheduling delays. More importantly, this approach has no portability for other hypervisors.

In contrast, hBalance is *in-hypervisor* so it can directly access all vCPUs' runtime information. Figure 6 illustrates the primary differences between hBalance and vBalance. hBalance requires *no* interactions between the guest OS and the hypervisor. This feature is important in that FV guests would not allow any modification to the OS kernel. Even for PV guests, with the use of the OS-specific event channel, *no* hypercall will be involved when interrupt migration happens. All these advantages are obtained by just enabling the routing flexibility in the hypervisor. Despite its simplicity, it is very powerful to unlock the performance. Table 1 summarizes the advantages of hypervisor-level balancer over OS-level balancer. From the perspective of the design and implementation, hypervisor-level solution is obviously a much simpler approach.

4.2 Components

4.2.1 Scheduling-Aware Interrupt Routing

To balance I/O interrupts, hBalance does not rely on vCPUs' IRQ statistics, which is unavailable in the hypervisor and also too heavy-weight to implement. Instead, we adopt a statistics-free method by *circularly* delivering interrupts to all vCPUs. The details are described in Algorithm 1. Each time when the current vCPU is preempted, hBalance will select a new target starting from the one that is logically *next* to the current one. In this way, all vCPUs are treated *equally*.

Compared with the RR routing of IOAPIC's *logical* destination mode, which changes the targeted vCPU for *every* interrupt, hBalance is different in two ways: (1) small batching: interrupts will not be redirected to another vCPU until the current vCPU has used up its time slice so that the system can be used more effectively by maintaining the cache effect; (2) when interrupts have to be migrated, hBalance will try selecting a vCPU that has unused CPU quota rather than in the native RR way. Specifically, hBalance will find a *running* vCPU first so that no vCPU context switch will be introduced; if there are no running vCPUs, a *blocked* vCPU will be considered as a second choice. Although both blocked and waiting vCPUs need to be rescheduled to process interrupts, blocked vCPUs serve better for load balance in that a vCPU stays in blocked state primarily because it has *no* tasks running and thereby voluntarily yields CPU control to the hypervisor; in contrast, a waiting CPU still has tasks in need of CPU cycles. There is also a concern about resource utilization: if a blocked vCPU cannot be woken up to use up its quota within the resource refill interval, the unused allocation will be consumed by other co-located vCPUs without reimbursement in the future. When all vCPUs of the SMP-VM are in waiting state, hBalance simply selects the current vCPU's logical neighbor to balance the load. hBalance allows a blocked or waiting vCPU to preempt the

Algorithm 1: Scheduling-Aware Interrupt Routing

```
Data: N, the number of vCPUs of the SMP-VM;
      run_bitmap, all running vCPUs at the moment;
      blk_bitmap, all blocked vCPUs at the moment;
      int.cur_vcpu, the current notified vCPU;
      int.nxt_vcpu, the next notified vCPU;
for each virtual interrupt "int" of the SMP-VM do
    if int.cur_vcpu is waiting in the runqueue then
        /* Redirect the interrupt */
        k = int.cur_vcpu.id;
        if run_bitmap != 0 then
            /* Select a running vCPU */
            Search from (k+1)th bit of run_bitmap, find
            the first marked bit, j;
            int.nxt_vcpu.id = j;
        else        /* Wake up a blocked vCPU */
            if blk_bitmap != 0 then
                Search from (k+1)th bit of blk_bitmap,
                find the first marked bit, j;
            else    /* All are waiting, let the
            neighbor balance the load. */
                j = (k+1)%N;
            end
            int.nxt_vcpu.id = j;
            if the SMP-VM has unused quota then
                int.nxt_vcpu.priority = HighestPrio;
            end
        end
        int.cur_vcpu ← int.nxt_vcpu;
    end
    Inject the virtual interrupt into int.cur_vcpu;
end
```

current running vCPU, but only when the whole SMP-VM still has unused quota. In this way, CPU fairness between different VMs will not be compromised.

For cache effectiveness, when a schedulable entity (either a task or a vCPU) begins to run on a physical CPU, in practice, the cache becomes hot in around 1ms. For example, Linux CFS scheduler sets this value to 0.5ms (in `sysctl_sched_migration_cost`), while Xen's credit scheduler adopts 1ms (in `vcpu_migration_delay`). In our scenario, since interrupts are migrated at the same pace with that of vCPU preemption, we believe cache effectiveness can be maintained in the same degree with the hypervisor scheduler.

4.2.2 I/O-Friendly CPU Quota Allocation

Cloud computing features a pay-per-use usage model where the service level is priced by the provided resources. For CPU resources, proportional share (PS) based schedulers have been largely implemented to allocate CPU cycles, such as Xen's credit scheduler [5], KVM's CFS scheduler [35] and VMware's stride scheduler [38]. Based on each VM's given *share*, all VMs periodically receive certain *quota* from the hypervisor scheduler, and inter-VM fairness is ensured in this level. For an SMP-VM with N vCPUs, since commodity OSes typically assume that all cores have identical computing power, to create this *symmetric* illusion, the hypervisor scheduler simply divides the VM's quota into *equal* shares:

$$Quota(vcpu_i) \quad += \quad \frac{new\_quota}{N} \quad (0 \leq i < N) \qquad (1)$$

Algorithm 2: I/O-Friendly CPU Quota Allocation

```
Data: N, the number of vCPUs of the SMP-VM;
      new_quota, the VM's allocation in each quota refill
period;
for every quota refill period do
        /* Aggregate all remaining quota */
    Quota(smp-vm) = Σ_{i=0}^{N-1} Quota(vcpu_i);
    for each vCPU of the SMP-VM do
        /* Rebalance: make all vCPUs have the
        same quota at the end */
        Quota(vcpu_i) = (1/N)(Quota(smp-vm) + new_quota);
    end
end
```

This simple scheme, however, could potentially cause inter-vCPU unfairness, because the quota may *not* be equally utilized by all vCPUs. Note that in virtual APIC's physical destination mode and Xen's event channel, I/O interrupts are bound to only one vCPU (vCPU0 by default). The incurred interrupt receiving overhead can sometimes dominate the CPU consumption of the whole SMP-VM. As a result, the vCPU ($vcpu_i$) that has served I/O would have much less quota than other vCPUs ($vcpu_j$):

$$Quota(vcpu_i) \ll Quota(vcpu_j) \ \ (0 \leq i, j < N) \qquad (2)$$

With the resource allocation scheme in Equation 1, the quota imbalance between vCPUs will carry over to every next refill period. As a result, the vCPU with fewer quotas will get fewer opportunities to run. Worse still, if an upper bound is set for the VM (e.g., "cap" mechanism in Xen's credit scheduler or "CPU bandwidth limit" in Linux CFS), the I/O-bound vCPU will be throttled much earlier than other vCPUs, introducing extra delays to interrupt processing. On the other hand, if one vCPU has received too many credits which exceed a predefined upper bound, the excess will be automatically deducted, resulting in wasted allocation for the whole SMP-VM.

Although this problem is less severe when virtual interrupts can be fairly distributed to all vCPUs, the vCPU in question actually performs I/O *on behalf of the whole SMP-VM*, so it is unfair to charge all the consumption to only one vCPU. In our new allocation scheme described in Algorithm 2, in every quota refill period, we rebalance the quota among all vCPUs. By compensating the vCPU which has served I/O in the passed period, the interrupt receiving overhead is implicitly charged to all vCPUs at a very fine-grained time interval. In the long run, this method will not compromise vCPU fairness because virtual interrupts have been uniformly distributed.

Alternatively, one would be tempted to try assigning CPU quota asymmetrically, e.g., giving the interrupt-receiving vCPU (vCPU0) more CPU quota so that fewer scheduling delays happen to it. However, we find this approach can bring serious starvation to the other vCPUs, which causes the kernel to hang quickly. In Linux, there are many per-CPU services that must be alive, such as the scheduling queue and various kernel threads. Simply asking the hypervisor to starve one vCPU will make all these services stall unexpectedly.

5 IMPLEMENTATION

We have implemented a prototype of hBalance in Xen 4.2.2. For FV guests, we do not make any modification to the guest OS. For PV guests, Linux 3.10.0 is slightly modified to use the *OS-specific* event channel.

5.1 Modifications to Xen Hypervisor

Virtual APIC for FV Guests. Virtual interrupts are delivered in the function `vioapic_deliver()`. With the logical destination mode, if the *LowestPrio* delivery mode is used, the function `vlapic_lowest_prio()` will compare PPR (Processor Priority Register) values of all vCPUs to select the vCPU that currently has the lowest task priority. hBalance replaces this selection algorithm with Algorithm 1. If the *Fixed* delivery mode is used, a virtual interrupt will be injected into *all* vLAPICs. This is very costly because every vCPU will be kicked by the hypervisor scheduler, resulting in many expensive guest-hypervisor switches (i.e., `VMExit` instruction in Intel-VT platform). In fact, only one vCPU needs to execute the interrupt handler, so it is unnecessary to bother all vCPUs. Even in non-virtualized environments, Linux does not support the use of the Fixed delivery mode. To this end, hBalance disables this broadcast operation and just kicks the vCPU selected by Algorithm 1. After that, the function `ioapic_inj_irq()` will inject the interrupt into the targeted vCPU. Finally, the corresponding bit of IRR (Interrupt Request Register) is set in the function `vlapic_set_irq()`. In order to record vCPU ID of the current interrupt receiver, a new variable `notify_vcpu_id` is added to `struct hvm_irq`.

Event Channel for PV Guests. Xen categorizes "event" into four types: (1) PIRQs, used by the driver domain to send and receive hardware interrupts; (2) IPIs, for inter-vCPU communication; (3) VIRQs, typically used for per-vCPU events such as local timers; (4) inter-domain notifications, driven by frontends and backends. *I/O events* of guest domains are all of *inter-domain* type, so we do not need to explicitly differentiate them from other events. Our interrupt routing algorithm is implemented as a subfunction called by the function `evtchn_send()`.

Hypervisor Scheduler. When migrating interrupts in Algorithm 1, if there was no auxiliary method, all vCPUs have to be visited in the worst case. Fortunately, Xen assigns each vCPU a *priority* according to its remaining CPU resource: UNDER means the vCPU has unused CPU quota while OVER means the vCPU has consumed more than its allocation. This priority is updated mainly in two places: 1) Xen's periodic accounting, and 2) when vCPU preemption happens. In our approach, we simply record the priority changes, so we can quickly find an UNDER vCPU.

Two bitmaps (`run_bitmap` and `blk_bitmap`) are added to `struct domain` to record the scheduling statuses of all vCPUs of the SMP-VM. When one vCPU is selected to run in the function `schedule()`, the corresponding bit of `run_bitmap` will be updated. Likewise, when one vCPU is blocked in the function `do_block()` or wakes up in the function `vcpu_wake()`, a particular bit of `blk_bitmap` will be set or cleared. Recall that when a blocked or a waiting vCPU is selected to receive interrupts, if the SMP-VM still has unused credits, the vCPU is allowed to preempt the current running vCPU by getting the highest priority. In Xen's credit scheduler [5], we introduce SMP_BOOST priority which is higher than all the other priorities. Regarding Algorithm 2, we track both vCPU-level and VM-level credit usage in the function `burn_credits()`; and in the function `csched_acct()`, if one VM is an SMP-VM, we will recalculate its total credits and then rebalance the credits among all its vCPUs.

5.2 Modifications to Xen's PV Guest OS

The status of each event channel is stored in the `shared_info` structure, which is implemented as a shared memory page between guests and the hypervisor for passing runtime information. In the current implementation, when a vCPU checks its pending events in `__xen_evtchn_do_upcall()`, it will first call `active_evtchns()` to mask out the events that do not belong to it. If the guest OS changes the interrupt mapping in `set_affinity_irq()`, only the *first* bit of the CPU affinity value is used as the event notifier. This is how Xen's *vCPU-specific* event channel is implemented. To replace it with our *OS-specific* event channel, we introduce the `bind_evtchn_to_all_vcpus()` function to enable I/O events to be visible to *every* vCPU. This change is only applied to the network and disk interrupt. We obtain their IRQ numbers from the frontend handlers, `xennet_interrupt()` and `blkif_interrupt()`. Other interrupts such as local timer interrupt and IPI still use Xen's vCPU-specific event channel.

5.3 Portability of hBalance

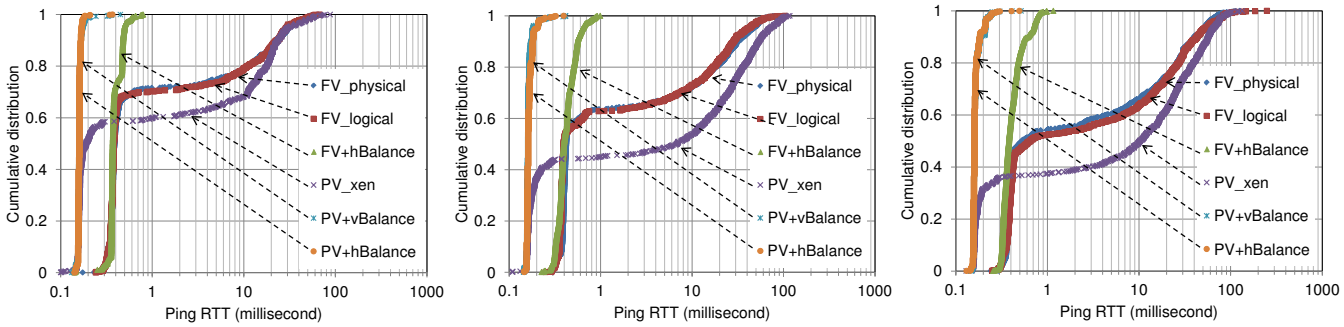
We take KVM [22] as an example to discuss the portability of our design. KVM adopts virtual APIC model to deliver interrupts for FV guests. It is worth mentioning that KVM also has PV drivers [30], and there are two different implementations: (1) a user-space implementation in QEMU (*qemu-virtio*), and (2) a kernel module implementation (*vhost*). vhost performs better than qemu-virtio because it avoids data copy between the user space and the kernel space [34]. Both implementations share the same guest OS frontend driver, and also comply with virtual APIC. Therefore, there is no problem with implementing dynamic interrupt routing in KVM, i.e., in the function `kvm_irq_delivery_to_apic()`.

Regarding CPU resource allocation, KVM relies on Linux's CFS group scheduling (`cgroup`) to schedule the vCPUs of an SMP-VM. Algorithm 2 can be implemented by adjusting each vCPU's share within the group, without affecting other VMs' CPU budgets.

6 EVALUATION

We conduct our experiments on several Dell PowerEdge M1000e blade servers, connected by a Brocade FastIron SuperX GbE switch. Each server is equipped with two quad-core 2.53GHz Intel Xeon 5540 CPUs, 16GB physical memory, and two 250GB SATA disks.

For the FV tests, three different settings are evaluated: (1) the physical destination mode, with vCPU0 being the interrupt receiver; (2) the logical destination mode, in which



(a) 2 vCPUs per pCPU on average. (b) 3 vCPUs per pCPU on average. (c) 4 vCPUs per pCPU on average.

Fig. 7: The CDF diagrams of ping experimental results with different vCPU densities. Note that to clearly show the performance differences, we use log scale on the x-axis.

I/O interrupts are routed to all vCPUs in a RR fashion; (3) hBalance, which replaces the RR routing of the logical destination mode with its own algorithms. The FV SMP-VM under test is equipped with an emulated e1000 network adapter using QEMU. For the PV tests, we compare hBalance with both vanilla Xen and our previous OS-level approach vBalance [13]. Vanilla Xen sets vCPU0 as the default interrupt receiver, similar to FV's physical destination mode. The hypervisor's scheduling time slice is not changed (30ms).

6.1 Micro-level Benchmarks

We use a set of I/O benchmarks to evaluate hBalance. The SMP-VM under test is configured with 4vCPUs and 4GB memory. To observe hBalance's effectiveness under different VM consolidation levels, we vary the number of background VMs running on the same set of pCPUs. The pCPUs are shared fairly among all vCPUs by properly setting the 'weight' of each VM. In the experiments, we use *lookbusy* [2] tool to keep the CPU load of each VM at a desired level.

6.1.1 Network RTT

We use Linux's *ping* to evaluate the network RTT. Since *ping* consumes very few CPU cycles, to trigger the hypervisor scheduling, the SMP-VM also runs the same CPU load as background VMs. With various VM densities, we ping the SMP-VM for 1000 times and show the results in Figure 7. In the FV tests, RTT varies largely in both the physical mode and the logical mode. In contrast, hBalance keeps RTT mostly within 1 millisecond. Similarly in the PV tests, a large number of long delays are observed in vanilla Xen, while both vBalance and hBalance can maintain the RTT at around 0.2 millisecond, with sporadic ones approaching but not exceeding 0.5 millisecond. Compared with the PV's split-driver model, the QEMU-based FV device emulation incurs much more overhead. Regarding the tails in the hBalance's tests, we speculate they are caused by Xen's credit scheduler's global load balancing: when sometimes a virtual interrupt arrives just before a vCPU is migrated to another pCPU, both the migration cost and the vCPU switching cost are potentially included in RTTs.

6.1.2 Network Throughput

We measure the benefit of hBalance to network throughput using *iperf*. As iperf tests would bring certain CPU load

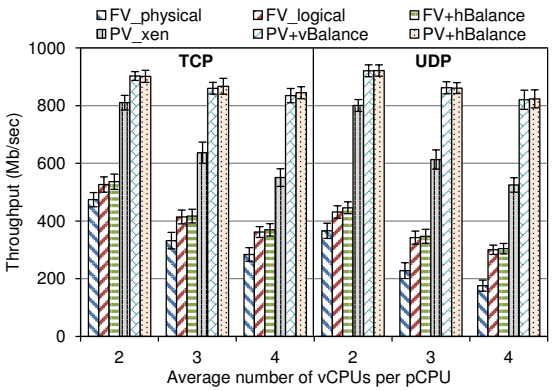


Fig. 8: The results of iperf throughput experiments.

to the SMP-VM, we do not run *lookbusy* as we did in the above. Each test lasts for 10 seconds and we repeat the test for ten times to average out the results. Figure 8 shows both TCP and UDP throughput results. In the FV tests, the logical mode performs similarly to hBalance, but clearly outperforms the physical mode. This makes sense because iperf is single-threaded, which invokes only one vCPU at a time. With the RR routing in the logical mode, since each vCPU serves only one interrupt at a time, it consumes very few credits and then goes idle (in the "blocked" state). Xen's boost mechanism [27] guarantees that blocked vCPUs with unused quota can be scheduled immediately when it receives another interrupt next time. hBalance differs from the RR routing in that it will not turn to another vCPU until the current one is preempted. In the PV tests, the network throughput with vanilla Xen degrades along with the increased number of background VMs, because once the targeted vCPU (vCPU0) is preempted, the iperf session has to be delayed until vCPU0 gets CPU cycles again. Both vBalance and hBalance can adapt to this situation by migrating interrupts to another vCPU so that the iperf server can be resumed to receive data.

6.1.3 HTTP Performance

We use *httperf* to measure the HTTP performance of the SMP-VM, which runs the Apache web server. In the experiments, we set four vCPUs per pCPU and then vary the requested file size. Httperf reports the reply rate every 5

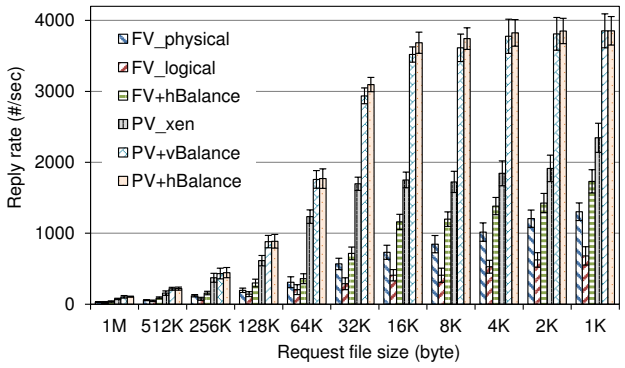


Fig. 9: The results of httperf experiments.

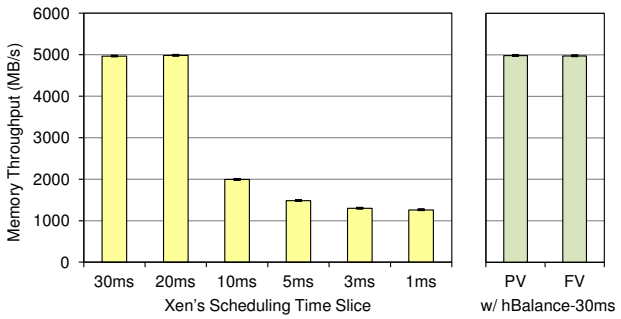


Fig. 10: The results of STREAM experiments.

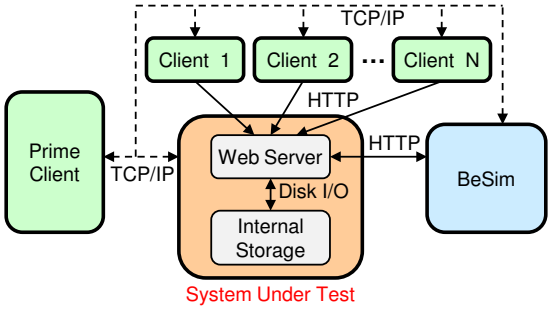


Fig. 11: The settings of SPECweb experiments.

seconds and we run each test for one minute. Quite different from the iperf tests, the results in Figure 9 show that FV's logical mode performs much worse than the physical mode. This is because iperf only triggers *one* vCPU at a time and the other vCPUs stay in the blocked state; while Apache web server is a multi-process multi-threaded application, a large number of child threads are created, so *all* vCPUs are active to concurrently process the requests. In the physical mode, the requests can be accepted in a batched manner by vCPU0 within each scheduling timeslice, so the scheduling delays would not happen to every request. However, in the logical mode, the targeted vCPU is changed for *every* request, although the current one may still be running; since all vCPUs are active, it is very possible that the next target is a waiting vCPU which cannot process the request until it gets scheduled again. hBalance considers each vCPU's scheduling status when routing the interrupts, and therefore substantially improves the reply rate. The improvement for PV guests is more apparent than FV guests due to the high performance of Xen's split-driver model. In the PV tests, we find that hBalance is particularly suitable for small files: when the file size is between 1KB and 32KB, hBalance achieves an improvement from 68.4% to 117.2%. This can be explained simply: under the fixed network bandwidth, the smaller the file is, the more requests can be simultaneously served and thereby more CPU cycles are consumed; when such CPU demand exceeds one vCPU's capability, vanilla Xen shows limited scalability because only vCPU0 is able to receive interrupts; in comparison, hBalance is able to utilize all vCPUs to process the requests.

6.1.4 Cache Effectiveness

To investigate whether hBalance would degrade cache's performance or not, we use *STREAM* benchmark [4] to measure the VM's memory bandwidth. STREAM is specifically designed to work with arrays much larger than the available cache so that most of their time is spent on waiting for cache misses to be satisfied. First, we set up 4 single-vCPU VMs to fairly share one pCPU, and then reduce the time slice of Xen's credit scheduler from 30ms to 1ms. Though a smaller time slice can potentially reduce VM scheduling delays, the results in Figure 10 (left) show that it seriously decreases memory access efficiency because the increased the number of VM context switches induces more cache flushes. Second, with the same vCPU density, we rerun the httperf experiments with 16KB file size (in which hBalance achieves the highest performance improvement), but with one background VM running STREAM benchmark. The results in Figure 10 (right) indicate that the background VM's cache effectiveness is not affected by hBalance's interrupt migrations, because (1) the scheduling time slice is not shortened, and (2) interrupts will not be redirected to another vCPU until the current vCPU has used up its whole time slice.

6.2 Application-level Benchmarks

We adopt the PHP implementation of SPECweb2009 Banking (v1.2) for evaluation. This benchmark simulates online banking operations and exhibits a mixed workload patterns: it has 16 different operations such as login/logout, bank balance inquiry, money transfers, show and modify the user profile, etc, which can generate various pressures on both CPU and I/O. Each test lasts for more than 30 minutes.

Figure 11 shows the application's architecture, consisting of a prime client, a certain number of agent clients, a web request processing unit and a backend (BeSim). The primary client drives multiple agent clients to generate HTTPS requests, and the web server communicates with the backend to retrieve specific information needed to dynamically construct responses. The experimental setting is the same as that in the httperf tests, having four vCPUs per pCPU on average. The clients and the backend run on dedicated machines, which communicate with the web server via our hardware switch. We launch 400 simultaneous sessions in the PV tests to saturate the SMP-VM; in the FV tests, we find the SMP-VM is unable to sustain given such a workload due to its higher device emulation overhead, so we reduce the number of simultaneous sessions to 200.

Table 2 shows the overall experimental results. In the FV tests, the physical mode again outperforms the logical destination mode, similar to that in the httperf tests. It

TABLE 2: The overall results of SPECweb experiments. FV tests: 200 sessions; PV tests: 400 sessions.

	FV_phyical	FV_logical	FV+hBalance	PV	PV+vBalance	PV+hBalance
Total Finished Requests (#)	27,837	24,538	**36,352**	94,704	108,111	**109,351**
Improvement w/ hBalance	**+30.6%**	**+48.1%**	–	**+15.5%**	**+1.1%**	–
Avg. Response Times (sec)	3.053	3.443	**1.772**	1.108	0.442	**0.397**
Reduction w/ hBalance	**-42.0%**	**-48.5%**	–	**-64.2%**	**-10.2%**	–

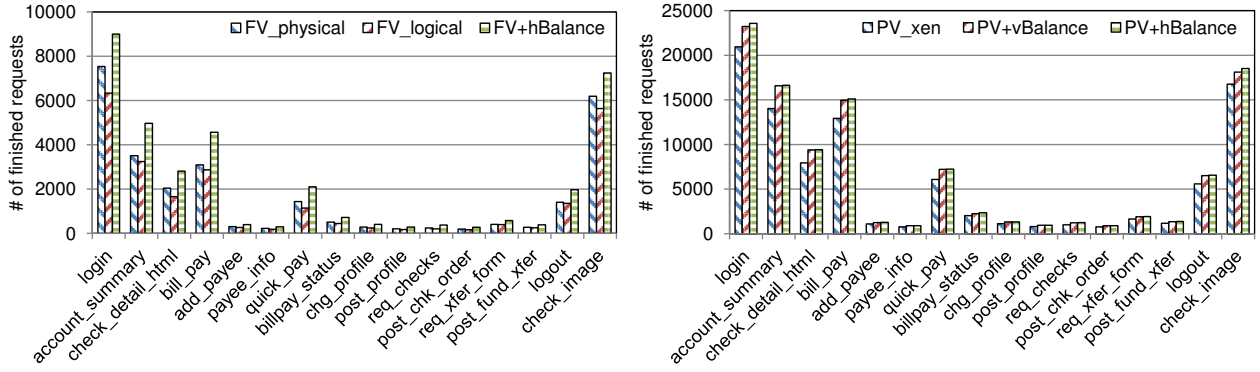


Fig. 12: The breakdown of the throughput results in the SPECWeb2009 experiments.

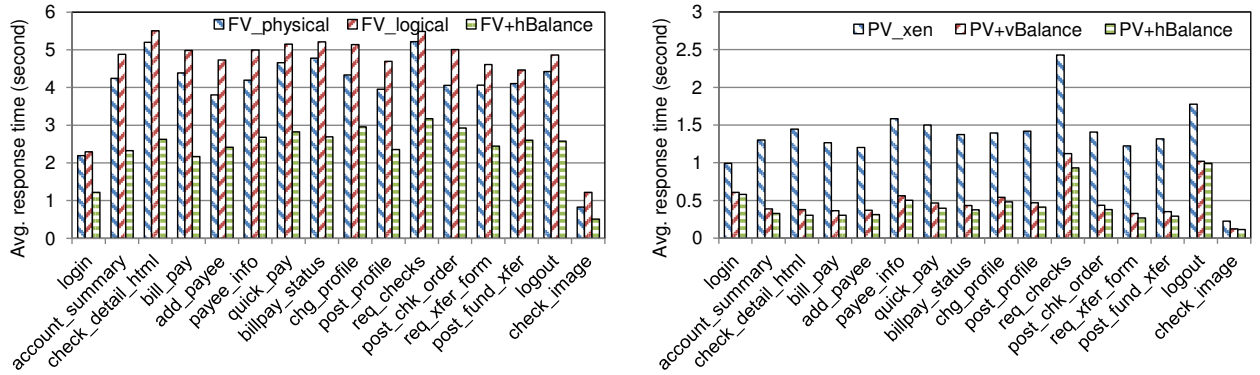


Fig. 13: The breakdown of the average response times in the SPECWeb2009 experiments.

proves that when the workload is high, RR routing is very unsuitable for SMP-VMs because it is oblivious of the targeted vCPU's scheduling status. With hBalance in the FV tests, the performance is significantly improved – 30.6% and 48.1% more throughput respectively, and 42.0% and 48.5% reduction in average response times. In the PV tests, when comparing with vanilla Xen, hBalance improves the throughput by 15.5% and reduces the average response times by as much as 64.2%. Even compared with vBalance, hBalance achieves 1.1% more throughput and 10.2% less average response times. We attribute the improvement to the avoidance of hypercalls when migrating interrupts. Figure 12 and Figure 13 show the breakdown of the testing results. It can be seen that hBalance achieves highest performance in nearly all types of requests.

To better understand the experimental results, we record all interrupt migrations in hBalance. We use Xen's `debug-key` to periodically dump the statistical data for analysis. Figure 14 shows that when the current notified vCPU is preempted, in the FV tests, in as many as 65.2% of the cases that hBalance is capable to find another running vCPU as the next target without preempting other vCPUs; in about 28.1% of the cases a blocked vCPU is selected and only 6.7% of the cases have all vCPUs are waiting in the queue. The results in the PV tests are more encouraging: there are about 83.8% of the cases in which virtual interrupts are migrated to another running vCPU, and about 12.0% of the cases where a blocked vCPU is selected and only 4.2% of the cases have a waiting vCPU being selected. From the statistics it can be seen that the hypervisor-level balancer is very advantageous because it can directly obtain each vCPU's runtime state to make optimal interrupt routing decisions.

We also examine the overhead of hBalance. Figure 15 shows the number of vCPU context switches per pCPU on average. In the FV tests, hBalance introduces only 17.7% more context switches than the physical mode. In the logical mode, there are much less context switches because it often routes the interrupts to waiting vCPUs, causing I/O activities to temporarily stall; whereas in the physical mode, we observe that vCPU0 can frequently migrate a certain number of tasks to other vCPUs, bringing more vCPU preemptions. Within one time slice, hBalance behaves similarly to the physical mode in that it routes all interrupts to only one vCPU; while when that vCPU has been preempted, hBalance can immediately redirect the interrupts to other

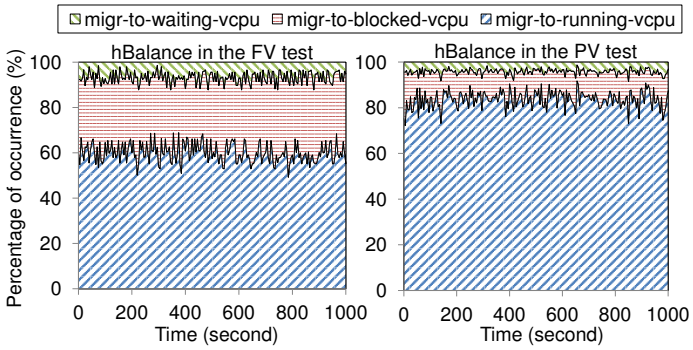


Fig. 14: The statistics of interrupt migration using hBalance. Note that "migr-to-running-vcpu" does not bring vCPU preemptions, while "migr-to-waiting-vcpu" and "migr-to-blocked-vcpu" will potentially trigger vCPU scheduling, because the selected vCPU will receive the SMP_BOOST priority if the SMP-VM has not used up its credits.

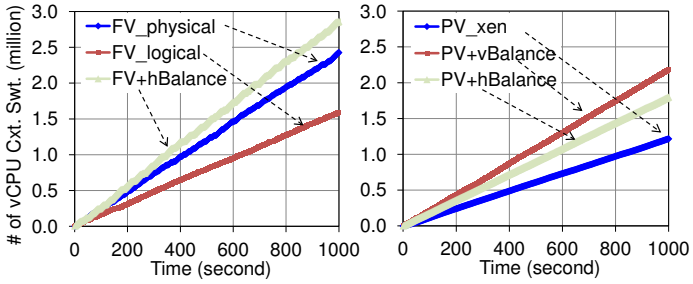


Fig. 15: vCPU context switch times per pCPU. The results are obtained via Xen's performance counters.

vCPUs. Extra context switches should be caused by the wakeups of *blocked* vCPUs as illustrated in Figure 14 and the SMP_BOOST priority we introduce in hBalance. In the PV tests, hBalance brings 47.9% more context switches than vanilla Xen, but 17.6% less than vBalance. We believe this is achieved by completely avoiding the guest-hypervisor race condition in vBalance when passing vCPU runtime states, proving that hypervisor-level migration is more accurate and light-weight than OS-level migration. Considering the benefit hBalance brings, we argue that its overhead is acceptable. Recall that different from other works [17], [42] which use a very small vCPU scheduling timeslice (e.g., 0.1ms), hBalance does not change it (30ms).

Figure 16 shows the balancing effect for the vCPUs. In FV's physical mode and PV's vanilla Xen, since all interrupts go to vCPU0, it consumes many more CPU cycles than the other vCPUs, resulting in an asymmetric use of the allocated resources. Although FV's logical mode can utilize CPU cycles as balanced as vBalance and hBalance, it does not benefit I/O performance because of its RR routing policy.

7 Related Work

7.1 VM Scheduling for I/O

The "boost" mechanism [27] is introduced to allow "wakeup preemption" for blocked vCPUs; "partial boost" is proposed to prevent CPU-bound vCPUs from compromising CPU fairness [21]. vSlicer [43] reduces Xen's scheduling time slice

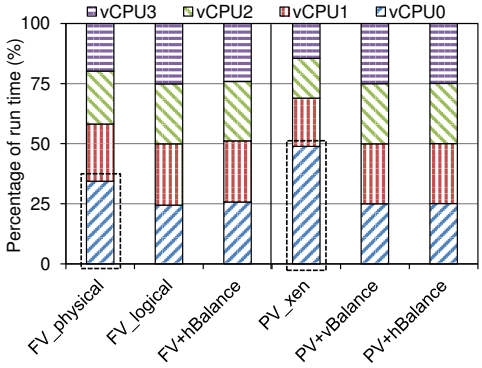


Fig. 16: Proportion of each vCPU's run time.

from 30ms to 10ms for latency-sensitive VMs, but it requires users to explicitly specify such VMs which is difficult in practice as many VMs run a combination of I/O and CPU workloads. Soft real-time methods [25], [41] can improve I/O responsiveness, but maintaining CPU fairness becomes can be quite challenging because an I/O-intensive VM can frequently preempt other VMs.

Dynamically partitioning pCPUs into "fast-tick cores" and "general cores" is proposed in [17], which requires the hypervisor to predict each VM's I/O workload. vAMP [19] proposes asymmetric scheduling for user-interactive workloads, but it requires identifying the workload's characteristics in the hypervisor. Janus [29] categorizes vCPUs into "real-time" type and "best-effort" type, and the schedules them differentially. vTurbo [42] binds I/O tasks to an extra "turbo vCPU" which is created by the hypervisor for all VMs, and schedules all turbo-vCPUs on designated pCPUs using 0.1ms time slice. Actually, the essence of these approaches is to *move* the extra vCPU context switches, caused by preempting vCPUs to quickly serve I/O, to *fewer* cores rather than avoiding them. Moreover, instrumenting tasks' CPU affinities is inconvenient when tasks are dynamically created. Therefore, the practicality of these intrusive approaches is probably limited.

The side-core approach [23] uses dedicated pCPUs for costly operations, e.g., to let a remote core carry out privileged hypervisor instructions so as to avoid the expensive guest-hypervisor swapping on the local core. VPE [26] runs dedicated polling threads on dedicated pCPUs to help with I/O device virtualization. SplitX [24] relies on conceptual hardware to split the execution of guests and the hypervisor. ELI [15] and ELVIS [16] remove the virtualization overhead caused by exits/entries during interrupt handling, by polling I/O cores in the guest. The common problem of these approaches is the lack of general applicability. Though poll-driven I/O is more efficient than interrupt-driven I/O when handling extremely high-rate I/O, such as 10Gb Ethernet and very fast SSDs, it leads to wasted CPU cycles and longer latencies when I/O rate decreases. Besides, the dedicated cores are often seriously under-utilized. In practice, a hybrid approach is often adopted for the system to adaptively switch between polling and interrupt.

We argue that, for SMP-VMs, accelerating I/O by preempting vCPUs should *not* be the only choice. hBalance explores another opportunity: the *scheduling asynchronism* of vCPUs. Within an SMP-VM, if virtual interrupts can be

redirected from a preempted vCPU to a running one, I/O processing will not be delayed and no vCPU context switch will be introduced. As our approach is independent of the scheduling time slice, it is complementary to other time slice based solutions. Another merit of hBalance is that it takes advantages of *all* vCPUs to process interrupts, which is very useful when to serve a certain I/O workload is beyond one vCPU's capability.

7.2 VM Scheduling for Synchronization

Uncoordinated vCPU scheduling can decrease the performance of multi-threaded applications running in an SMP-VM: if the vCPU that holds a contended spinlock is preempted, other vCPUs have to wait for a longer time, which is known as Lock-Holder Preemption (LHP). To deal with it, VMware ESX 2.x adopts strict co-scheduling to make all vCPUs progress at similar rates, but it introduces CPU fragmentation problem. Relaxed co-scheduling only tracks the slowest vCPU and lets each vCPU make co-scheduling decisions independently [10], which is introduced in ESX 3.x and later versions. This technique is further refined in [40], [44] by detecting long-lived lock contention. Balance scheduling [33] places vCPU siblings in different runqueues, and similar idea has been incorporated in VMware ESXi 5.x [10]. More adaptively, demand-based scheduling [20] identifies TLB shutdown and reschedule IPI as two main sources for vCPU coordination. vCPU ballooning [31] alleviates LHP by dynamically adjusting the number of vCPUs according to available CPU cycles of the SMP-VM.

There are also optimizations at the guest level. In [36], an OS-informed approach is proposed to ask the hypervisor not to preempt lock-holder vCPUs until the lock is released. Linux's PV spinlock [14] allows LHP, but prevents long active waiting: specifically, when one vCPU has been busy-waiting for more than a predefined time threshold, it will return the CPU control to the hypervisor. For ticket spinlock, authors in [28] point out that if the waiters are not scheduled in the same order as they require the lock, waiters in the tail will be delayed for longer. They propose to set the waiters' sleep times to be proportional to their positions in the lock waiting queue, so that the hypervisor can schedule them properly.

Hardware-assisted approaches include Intel's Pause Loop Exiting (PLE) and AMD's Pause Filter (PF). The basic idea is to let the hypervisor take over CPU control after the guest has executed a certain number of `pause` instructions when spinning. In [39], the authors observe that a spinning thread makes very few modifications to the program state (e.g., the `store` instruction that changes variables in memory), and they implement Spin Detection Buffer (SDB) to indicate such busy-waiting, which can assist the hypervisor to schedule vCPUs more wisely.

In general, most concurrencies inside SMP-VMs can be inferred by the hypervisor, informed by the guest OS or detected by the hardware. Co-scheduling is only needed temporarily and selectively for certain vCPUs. Most of the time, the vCPUs of an SMP-VM are scheduled *independently* in their own runqueues, leaving much space for hBalance to explore the scheduling asynchronism for better interrupt routing.

7.3 Receiver-side Network Balancing

At the end host, network processing includes interrupt processing and protocol processing. In physical SMPs, Receive Side Scaling (RSS) [3] allows a multi-queue NIC to distribute packets of each flow to a separate CPU to balance the load. However, in virtualized environments, since virtual NIC is currently implemented as a mono-queue software entity, RSS does not apply here. As for the protocol processing, Receive Packet Steering (RPS) [3] is a kernel approach to distribute packets to sibling vCPUs, *after* the packets have been copied into the VM. Compared to kernel operations (e.g., interrupt execution) which are mostly at the cost of microseconds, in consolidated VMs, the dominant delay actually comes from *interrupt delivery* which can be tens of milliseconds when the targeted vCPU has been preempted.

8 Conclusion and Future Work

Traditional interrupt balancing techniques are built upon the assumption that the OS runs on dedicated pCPUs. For SMP-VMs, this assumption is not true when one pCPU is time-shared by multiple vCPUs. In this paper, we revisit two existing interrupt delivery models and point out their limitations. We present hBalance to offload interrupt balancing from the guest OS to the hypervisor. hBalance mainly seeks for preemption-free opportunities when migrating interrupts. Our approach can support both FV and PV guests and has very high portability among various hypervisor. The evaluation with SPECWeb application and several micro-benchmarks shows that hBalance considerably improves network performance with moderate overhead.

It will be meaningful to investigate to what extent our solution can benefit other applications. For example, video streaming applications have very different traffic patterns and scheduling requirements. In the past, there have already been a few works studying soft real-time virtual machine scheduling [18], [45]. The problem space of our paper is different from theirs in that our approach aims to be agnostic of specific VM schedulers, but to explore preemption-free interrupt delivery opportunities to assist them. Meanwhile, since video streaming quality is also affected by other factors, such as datacenter-level caching policy and client-side buffering policy, we view video streaming applications a good genre in which to investigate how various layers can interact with each other.

9 Acknowledgments

We thank the anonymous reviewers for comments that improved this paper. This work was supported in part by a Hong Kong RGC CRF grant (No. C7036-15G).

References

[1] irqbalance: https://code.google.com/p/irqbalance.
[2] lookbusy – a synthetic load generator: http://www.devin.com/lookbusy/.
[3] Scaling in the linux networking stack. *Linux Kernel Document*.
[4] STREAM bechmark: https://www.cs.virginia.edu/stream/.
[5] Xen's Credit Scheduler: http://wiki.xen.org/wiki/credit_scheduler.
[6] 82093AA I/O ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (IOAPIC). *Intel*, May 1996.

[7] The architecture of VMware ESXi. *VMware Technical White Paper*, 2008.

[8] Achieving a fair distribution of the processing of guest network traffic over available physical CPUs. *Citrix Technical White Paper*, 2011.

[9] Server Virtualization, Windows Server 2012. *Microsoft Technical White Paper*, 2012.

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

[11] 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 *SOSP*, 2003.

[12] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX ATC*, 2005.

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

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

[15] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. ELI: Bare-metal performance for I/O virtualization. In *ASPLOS*, 2012.

[16] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky. Efficient and scalable paravirtual I/O system. In *USENIX ATC*, 2013.

[17] 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 *HPDC*, 2010.

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

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

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

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

[22] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *The Ottawa Linux Symposium*, 2007.

[23] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-architecting VMMs for multicore systems: The sidecore approach. In *WIOSCA*, 2007.

[24] A. Landau, M. Ben-Yehuda, and A. Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *WIOV*, 2011.

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

[26] J. Liu and B. Abali. Virtualization polling engine (VPE): using dedicated CPU cores to accelerate I/O virtualization. In *ICS*, 2009.

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

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

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

[30] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

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

[32] V. Srinivasan, G. R. Shenoy, S. Vaddagiri, D. Sarma, and V. Pallipadi. Energy-aware task and interrupt management in Linux. In *Ottawa Linux Symposium*, 2008.

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

[34] M. S. Tsirkin. vhost-net and virtio-net: need for speed. In *KVM Forum*, 2010.

[35] P. Turner, B. B. Rao, and N. Rao. CPU bandwidth control for CFS. In *Linux Symposium*, volume 10, pages 245–254, 2010.

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

[37] C. A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI*, 2002.

[38] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical report, MIT Laboratory for Computer Science, 1995.

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

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

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

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

[43] 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 *HPDC*, 2012.

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

[45] L. Zhou, S. Wu, H. Sun, H. Jin, and X. Shi. Virtual machine scheduling for parallel soft real-time applications. In *MASCOTS*, 2013.

Luwei Cheng received his PhD in computer science from the University of Hong Kong in 2015. He is currently a Research Scientist at Facebook. His research interests are mainly on performance problems in cloud datacenters, including operating system, networking and distributed storage. He received Best Student Paper Award in UCC 2011 conference, Hong Kong PhD Fellowship in 2012 and Microsoft Research Asia Fellowship in 2013.

Francis C.M. Lau received his PhD in computer science from the University of Waterloo in 1986. He has been a faculty member of the Department of Computer Science, The University of Hong Kong since 1987, where he served as the department chair from 2000 to 2005. He is now Associate Dean of Faculty of Engineering, the University of Hong Kong. He was a honorary chair professor in the Institute of Theoretical Computer Science of Tsinghua University from 2007 to 2010. His research interests include computer systems and networking, algorithms, HCI, and application of IT to arts. He is the editor-in-chief of the Journal of Interconnection Networks.