vBalance: Using Interrupt Load Balance to Improve I/O Performance for SMP Virtual Machines

Luwei Cheng, Cho-Li Wang
Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong, P.R. China
{lwcheng, clwang}@cs.hku.hk

ABSTRACT

A Symmetric MultiProcessing (SMP) virtual machine (VM) enables users to take advantage of a multiprocessor infrastructure in supporting scalable job throughput and request responsiveness. It is known that hypervisor scheduling activities can heavily degrade a VM's I/O performance, as the scheduling latencies of the virtual CPU (vCPU) eventually translates into the processing delays of the VM's I/O events. As for a UniProcessor (UP) VM, since all its interrupts are bound to the only vCPU, it completely relies on the hypervisor's help to shorten I/O processing delays, making the hypervisor increasingly complicated. Regarding SMP-VMs, most researches ignore the fact that the problem can be greatly mitigated at the level of guest OS, instead of imposing all scheduling pressure on the hypervisor.

In this paper, we present *vBalance*, a cross-layer software solution to substantially improve the I/O performance for SMP-VMs. Under the principle of keeping hypervisor scheduler's simplicity and efficiency, vBalance only requires very limited help in the hypervisor layer. In the guest OS, vBalance can dynamically and adaptively migrate the interrupts from a preempted vCPU to a running one, and hence avoids interrupt processing delays. The prototype of vBalance is implemented in Xen 4.1.2 hypervisor, with Linux 3.2.2 as the guest. The evaluation results of both micro-level and application-level benchmarks prove the effectiveness and lightweightness of our solution.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design Studies; D.4.4 [**Operating Systems**]: Communications Management—*Input/Output*

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Virtualization, Xen, SMP, Cloud Computing

1. INTRODUCTION

Virtualization technology allows running multiple VMs on one physical host by multiplexing the underlying physical resources. Modern cloud data centers are increasingly adopting virtualization software such as VMware [41], Xen [15], KVM [26] and Hyper-V [9], for the purpose of server consolidation, flexible resource management, better fault tolerance, etc. Each VM is given the illusion of owning dedicated computing resources. A VM can be easily configured with different settings, such as the amount of CPU cycles, network bandwidth, memory size and disk size. A virtual SMP infrastructure can be easily created by assigning the VM more than one vCPU. Compared with UP-VMs which have only one vCPU, a SMP-VM allows

SOCC'12, October 14-17, 2012, San Jose, CA USA

running multi-threaded or multi-processed applications, by moving tasks among available vCPUs to balance the workload and thus more fully utilizes the processing power. It is particularly attractive for enterprise-class applications such as databases, mail servers, content delivery network, etc.

Virtualization can cause performance problems which do not exist in a non-virtualized environment. It is already known that the hypervisor scheduler can significantly affect a VM's I/O performance, as the vCPU's scheduling latencies actually translate into the processing delays of the VM's I/O requests. The evaluation results [16, 42] have revealed the very unpredictable network behaviors on Amazon's EC2 [1] cloud platform. The negative effect reflects as degraded I/O throughput, as well as longer and unstable I/O latency for applications. Most researches focus on improving the I/O performance for UP-VMs [20, 21, 22, 25, 27, 32, 34, 45], whereas little has been done to SMP-VMs, ignoring the fact that the I/O problem in SMP-VMs is substantially different from that in UP-VMs.

For a UP-VM, since all interrupts need to be processed by the only one vCPU, once external events arrive, the only way to avoid performance drop is to force the hypervisor to schedule its vCPU as soon as possible. As a result, the pressure of responding to I/O is completely imposed on the hypervisor scheduler, leading to increasing complexity and context switch overhead. SMP-VMs allow more flexible interrupt assignment to vCPUs and are not that demanding for the hypervisor's help to process I/O requests. From the perspective of the hypervisor, since a SMP-VM has multiple vCPUs, it is likely that when one vCPU is descheduled another vCPU is still running; therefore, if the guest OS can adaptively migrate the interrupt workload from the preempted vCPU to a running one, it is unnecessary to bother the hypervisor scheduler, saving a lot of context switch overhead. From the perspective of SMP guest OS, even though the whole VM gets CPU cycles, if the vCPU that is responsible for the interrupts is not scheduled by the hypervisor instead of other vCPUs, I/O processing delays can still happen. Inappropriate interrupt mapping inside the guest OS can also cause performance problems: 1) if the interrupts are statically mapped to a specific vCPU all the time, that vCPU can be easily overloaded when I/O workload is heavy; 2) since the hypervisor scheduler treats each vCPU fairly by allocating them an *equal* amount of CPU cycles, unbalanced interrupt load among vCPUs results in an *unequal* use of them, leading to suboptimal resource utilization.

In this paper, we will present *vBalance*, a cross-layer software solution that can substantially improve the I/O performance for SMP-VMs. vBalance bridges the knowledge gap between the guest OS and the hypervisor scheduler: the hypervisor dynamically informs the SMP-VM the scheduling status of each vCPU, and the guest OS is always trying to assign its interrupts to the running vCPUs. Unlike traditional approaches which totally count on altering the hypervisor scheduler, vBalance maximally exploits the guest-level capability to accelerate I/O speed, and only needs very limited help from the hypervisor. As the hypervisor is expected to be as lean as possible to keep its efficiency and scalability, we believe it is the trend to paravirtualize the guest OS more and more to adapt to the hypervisor.

The contributions of this paper can be summarized as: (1) our design *vBalance* demonstrates that for SMP-VMs, the guest OS has great potential to help improve the I/O performance, leaving the hypervisor scheduler unbothered most of the time; (2) we implement our vBalance prototype in Xen 4.1.2 hypervisor and Linux 3.2.2 guest OS, involving less than 450 lines of code; (3) we thoroughly evaluate the effectiveness of vBalance using both micro-benchmarks and cloud-style application benchmarks. The results show that vBalance brings significant improvement to both I/O throughput and I/O responsiveness, in sacrifice of very limited context switch overhead in the hypervisor.

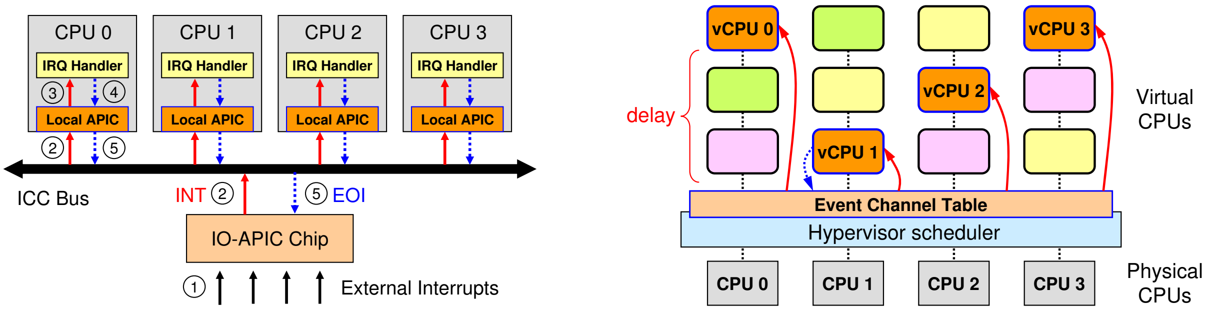
The rest of the paper is organized as follows. Section 2 presents a detailed analysis of the problem and discusses the possible approaches. Section 3 presents the design principles and vBalance's architecture. The implementation is introduced in Section 4. The solution is evaluated in section 5. Section 6 describes the related work. Section 7 discusses the future work. We conclude our research in Section 8.

2. PROBLEM ANALYSIS

In this section, we first illustrate the problem by comparing the physical SMP with the virtual SMP. We then discuss the possible approaches to address the problem.

2.1 IO-APIC vs. Event Channel

In the physical world, SMP system features an IO-APIC (I/O Advanced Programmable Interrupt Controller) chip, to which all processors are connected via an ICC (Interrupt Controller Communication) bus, as shown in Figure 1 (a). An SMP-aware OS enables a process/thread to run on any CPU, whether it is a part of the kernel or part of a user application. IO-APIC contains a *redirection table*, which routes specific interrupts to a specific core or a set of cores, by writing an *interrupt vector* to the Local-APICs. The OS receives the interrupts from the Local-APICs and does not communicate with IO-APIC until it sends an EOI (End Of Interrupt) notification. IO-APIC is capable of delivering the interrupts to any of the cores and even



(a) IO-APIC for SMP physical machine

(b) Event channels for SMP virtual machine

Figure 1: Comparison of I/O mechanisms between physical SMP and virtual SMP

perform load balancing among them. By default it delivers all interrupts to core 0. Multiple interrupts will keep one of the cores overloaded while the others remain relatively free. The OS scheduler has no idea about this state of affairs, and assumes that all interrupt handling cores are as busy as any other core. It is the task of the interrupt balancing software (such as Linux's irqbalance [7]) to distribute this workload more evenly across the cores: to determine which interrupts should go to which core, and then fill this table for IO-APIC chipset to use. If care is not taken to redistribute the interrupts properly, it could lead to a decrease in the overall system performance by overloading some processors and by not optimally utilizing the remaining processors.

In the virtualization world, para-virtualization has been adopted (e.g. in Xen [15]) to significantly shrink the cost and complexity of I/O handling compared to using full device emulation (e.g. QEMU [17]). In Xen's para-virtualized execution environments, the functionality of IO-APIC chip for VMs is totally replaced by a software *event channel* mechanism, as shown in Figure 1 (b). For each VM, there is a global event channel table to record the detailed information of each channel, such as event type, event state, notified vCPU, etc. The guest OS can correlate these events with its standard interrupt dispatch mechanisms. The events are distributed among all vCPUs, and each vCPU maintains its own event selection table (acting as the role of Local-APIC). The guest OS informs the hypervisor event binding information, e.g., which vCPU is responsible for network devices, and which vCPU takes care of block devices, so that the hypervisor can notify the corresponding vCPU when the relative events arrive. After a vCPU is selected by the hypervisor to run, it will check its own event selection table to see whether there are pending events. If yes, the corresponding interrupt handlers will be called to process the events, by copying data from shared memory and acknowledging the backend (similar to sending EOI notification).

There are three main differences between the physical SMP and the virtual SMP. First, the physical cores are always "online", which means once an interrupt is received by the local-APIC, the CPU immediately jumps to the interrupt gate and fetch the IRQ handler from the IDT (Interrupt Descriptor Table) to run, with almost no delay; however, in a virtual SMP system, after an event is delivered to a specific vCPU, it still needs to wait for the hypervisor's schedule to process the interrupts. The scheduling delays are in nature inevitably as there are multiple vCPUs sharing the same physical core. Second, most IO-APIC chips can be configured to route an interrupt to *a set of cores* (one to many), enabling more than one option for the interrupt's delivery. However, current event channel mechanism [15] can only deliver the events to a *specific* vCPU (one to one), thereby limits the hypervisor's delivering flexibility. Third, unlike the idle process in a traditional OS which consumes all the unused CPU cycles, the idle process in a paravirtualized guest OS will cause the vCPU to voluntarily yield up its CPU cycles to the hypervisor. As such, if the guest OS can not evenly distribute the workload among all available vCPUs to optimally utilize the CPU cycles, the resources allocated to the whole VM will be wasted. Since OS schedulers are mostly unaware of the interrupt load on each processor, an extra effort must be paid to balance the interrupt load.

Due to insufficient understanding of the problems mentioned above, the seemingly faithful design of current I/O virtualization techniques can result in very poor performance under heavy I/O workload. In the physical SMP system, interrupts are migrated to prevent a specific processor from being overloaded. In the virtual SMP system, interrupts *have to* be migrated to avoid the scheduling delays from the hypervisor.

2.2 Possible Approaches

We group the possible approaches into two categories, depending on the layer where the approach resides.

2.2.1 *Hypervisor-level Solution*

A non-intrusive approach at the hypervisor layer could be to modify the hypervisor scheduler, by immediately scheduling the vCPU that receives external events, regardless of its priority, state, credits, etc.

This approach can mitigate the problem to some extent, as it eliminates the scheduling delays of the targeted vCPU. Unfortunately, it also brings several critical problems: first, without interrupt load balancing from the guest OS, all events will be delivered to only one vCPU (vCPU0 by default), or some specific vCPUs, making them easily overloaded; second, the interrupt-bind vCPUs will consume much more CPU cycles than the others, leading to non-optimal resource utilization; through we can modify the hypervisor scheduler to give the targeted vCPU more CPU cycles than the others (in this way, the vCPUs of the same SMP-VM are *not* symmetric at all), how to determine the disproportion parameter is also a problem, as the interrupt load is highly unpredictable; third, the context switch overhead will substantially increase, because the hypervisor scheduler needs to keep swapping the vCPUs in response to the incoming events, making it too expensive in practice.

The hypervisor scheduler treats each vCPU as *symmetric*, in the expectation that all vCPUs can *behave symmetrically* and utilize the resources in a balanced manner. Without the help from the guest OS, the hypervisor scheduler is not able to fully address the I/O problem.

2.2.2 *Guest-level Solution*

At the guest layer, one approach is to monitor the interrupt load of each vCPU, and periodically reassign all the interrupts to achieve the load balance, like irqbalance [7].

This approach can easily make each vCPU consume approximately an equal amount of CPU cycles. However, the main problem is that there is no way to make *accurate* decisions: at which moment the interrupts should go to which vCPU. Since the vCPUs can be preempted by the hypervisor at any time which is totally transparent to the guest OS, if the interrupts are assigned to an offline vCPU, the processing will be delayed. The knowledge gap between the guest and the hypervisor eliminates the effectiveness of this approach. For a virtual SMP system, the responsibility of interrupt migration is *not only* to achieve load balancing, but more importantly, to avoid the hypervisor scheduling delays imposed on the interrupts' processing. Therefore, without the scheduling status information of each vCPU from the hypervisor, it is impossible for the guest OS to precisely determine interrupt migration strategies.

3. DESIGN

In this section, we describe the design goals of vBalance, and present its components in detail.

3.1 Design Goals

The followings are the design goals of vBalance to enable greater applicability as well as ease of use:

- **High performance**: the design should efficiently utilize the advantage of SMP-VMs, that is, the ability of leveraging more than one vCPU to process interrupts, and therefore more easily to achieve high-throughput and low-latency communication.

- **Light-weight**: to make it scalable to many guest domains, the design must be quite light-weight in the hypervisor layer. Also for ease of development, the modifications required to the existing software must be as few as possible.

- **Application-level transparent**: the deployment of vBalance should not require any change to the applications, so that they can transparently enjoy the performance benefits.

3.2 Main Concerns and Principles

Base on the discussion in Section 2.2 and the design goals in Section 3.1, we prefer a solution that (1) maximally seeks help from the guest OS to avoid interrupt processing delays, (2) bothers the hypervisor as little as possbile, (3) makes all vCPUs share the interrupt load evenly. By clearly identifying the responsibilities of the hypervisor scheduler and the guest OS to handle interrupts, we are able to derive the design principles of vBalance.

For hypervisor scheduler, since it is used very frequently to manage all the guest domains, it is so important that the design and implementation must be very lean and scalable. Therefore it should be made as *simple* as possible. We argue that the main responsibility of the hypervisor scheduler is to perform CPU time proportional sharing among VMs. It should not assume too much about the guest's interrupt processing characteristics, e.g., in favor of scheduling one or some specific vCPUs. For a SMP-VM, the hypervisor should guarantee that each vCPU receives relatively the same amount of CPU cycles, so that all vCPUs look “symmetric” to the guest OS. This is the “mechanism” that the hypervisor should provide to the SMP guest domains.

Based on the illusion that all vCPUs are “symmetric”, various “policy-level” load balancing strategies can be applied in guest domains. Since the capability of one vCPU to process interrupts is limited, the guest OS should endeavor to utilize all vCPUs to handle interrupts. Unbalanced use of vCPUs can cause low resource utilization, because the idle process for each vCPU in the guest OS will voluntarily yield up the CPU cycles to the hypervisor, and they are eventually reallocated to other domains or consumed by the idle domain. Once a vCPU is descheduled, the guest OS should be capable to migrate the interrupts to a running vCPU to avoid processing delay. So it is a must for the hypervisor scheduler to pass the necessary information to the guest OS, at least the scheduling status of each vCPU. However, the guest OS should not make any assumption about the scheduling strategies of the underlying hypervisor, or instruct the hypervisor to schedule a specific vCPU. The hypervisor always has a higher privilege than the guest OS. For security reasons, the control flow can only go from the hypervisor to the guest OS, instead of the opposite direction.

3.3 vBalance Architecture

vBalance is a cross-layer design residing in both guest OS layer and hypervisor layer, as shown in Figure 2. The component in hypervisor layer is responsible to inform the guest kernel the scheduling status of each vCPU. The interrupt remapping module in the guest OS always tries to bind the interrupts to a running vCPU, meanwhile achieves balanced interrupt load. vBalance does not violate Xen’s *split-driver* model [15], in which the frontend communicates with a counterpart backend driver in the driver domain, via shared memory and event channels. The frontend driver in the guest domain can be netfront, blkfront, etc. In the driver domain, there may exists data multiplexing/demultiplexing activities between the backend and hardware device driver. For example, the netback driver usually adopts software network bridge [8] or virtual switch utilities [12, 18] to multiplex the network packets from/to guest domains. Since vBalance does not modify the driver domain, we do not explicitly show the multiplexer component in the figure.

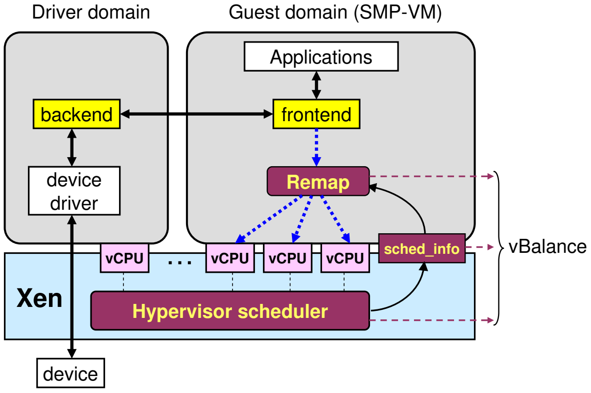


Figure 2: vBalance architecture

3.3.1 *Hypervisor Scheduler*

Xen uses a credit scheduler [4] to allocate CPU time proportionally according to each VM’s *weight*. The vCPUs on each physical core are sorted according to their priorities and remaining credit. If a vCPU has consumed more than its allocation, it will get an `OVER` priority, otherwise it keeps an `UNDER` priority. Each vCPU will receive a 30ms time slice once it gets scheduled. In order to improve VM’s I/O performance, the credit scheduler introduces a *boost* mechanism. The basic idea is to temporarily give the vCPU that receives external events a `BOOST` priority with preemption, which is higher than other vCPUs in `UNDER` and `OVER` state. Credit scheduler supports automatic load balancing to distribute vCPUs across all available physical cores.

Normal Path. In most cases, the external events for SMP-VMs can be properly delivered and timely processed by the targeted vCPU, as the guest OS guarantees that the receiver is always one of the running vCPUs. So for a SMP-VM, as long as there is at least one vCPU running on the physical core when external events arrive, the processing delay will not happen. We call it *normal path* as no extra reschedule operations are needed, and the current hypervisor scheduler can easily satisfy this scenario without any modification. Since a SMP-VM has more than one vCPU, the likelihood that all vCPUs are offline at the same moment is much smaller than that of a UP-VM.

Fast Path. The only circumstance the hypervisor scheduler needs to consider is that, for a SMP-VM, it is possible that all vCPUs are waiting in the runqueues when external events arrive. In this rare case, the hypervisor scheduler needs to explicitly force a reschedule to give the targeted vCPU the opportunity to run. We call it *fast path*. Of course, the more vCPUs a SMP-VM has, the more probably the event handling follows the normal path. It also depends on the proportion of CPU time that is allocated to the VM, and the density of co-located VMs.

The limitation of current *boost* mechanism is that it only boosts the *blocked* vCPU with the condition that it has not used up its credit. This will introduce scheduling delay for the vCPUs which are already *waiting* in the runqueue. Based on the credit scheduler, we introduce another priority `SMP_BOOST` with preemption, which is higher than all the other priorities. For a SMP-VM, only when none of its vCPUs can get a scheduling opportunity from the normal path, the targeted vCPU receives a `SMP_BOOST` priority and get scheduled at once.

3.3.2 Interrupt Remap Module in the Guest OS

The remap module of vBalance resides in the guest OS, which is detailedly shown in Figure 3. Generally, it contains three components: the *IRQ Load Monitor* is responsible to collect interrupt load statistics of each vCPU; the *Balance Analyzer* reads scheduling states of all vCPUs, and uses the IRQ statistics to determine whether an interrupt imbalance has happened; once an imbalance is identified, the interrupts will be migrated to another online vCPU, and the *IRQ Map Manager* will take the remap action.

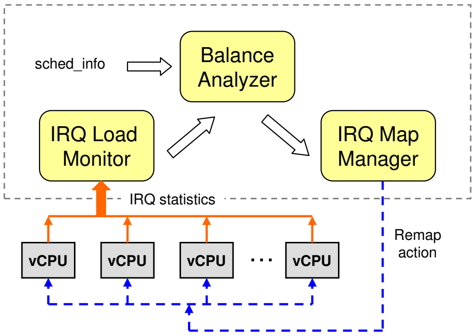


Figure 3: vBalance *remap* module in guest OS

Time-based interrupt load balance heuristic. The method for balancing interrupt load among multiple vCPUs basically involves generating a heuristic to assess the interrupt imbalance. It can be based on one or more interrupt sources such as network load or disk load. When the heuristic satisfies a certain criterion, the interrupt imbalance is identified and a new mapping of interrupts to vCPUs is generated. The heuristic can be generated based on one or more quantities, including but not limited to, the average load per interrupt on vCPUs, or the total load of each vCPU. Since the interrupt load on each vCPU changes along with time, the system heuristic must be determined at different times, resulting in a time-based system heuristic. A baseline heuristic is used to compare a given value to determine whether a sufficiently large interrupt imbalance has occurred.

Measure an interrupt imbalance. In the guest OS, using interrupt load data for vCPUs, a measurement of how balanced the interrupts are across them can be determined. Suppose that a SMP-VM has n vCPUs, denoted as the set $VC = \{vc_0, vc_1, ..., vc_{n-1}\}$. For each vCPU $vc_i \in VC(0 \le i \le n-1)$, let $ld(vc_i)_t$ denote its interrupt load at time t. We use the average interrupt load of all vCPUs as the baseline, which can be expressed and calculated in equation 1.

$$ld_{avg}(vc)_t = avg(ld(vc_0)_t, ld(vc_1)_t, ..., ld(vc_{n-1})_t) \tag{1}$$

There are k online vCPUs $(0 \le k \le n)$ at time t, denoted as $OVC_t = \{ovc_0, ovc_1, ..., ovc_{k-1}\}, OVC_t \subseteq VC$. If a rebalance operation is needed, these online vCPUs are actually the eligible vCPUs for interrupts to be mapped to. The online vCPUs are classified based on their current interrupt load. Figure 4 shows the state transition diagram of each vCPU. Generally there are two states for each vCPU: `online` and `offline`, depending on whether the vCPU is running on the physical core or not. To manage the online vCPUs more efficiently, each `online` vCPU has three sub-states: `HOLD`, `LOW` and `HIGH`. The vCPU that is holding the interrupts is labeled as HOLD; for the others, if its current interrupt load is above the average level, it is labeled as HIGH; otherwise, it is labeled as LOW. In this way, the vCPUs can be sorted in a more efficient way. The average load $ld_{avg}(vc)_t$ can be updated less frequently.

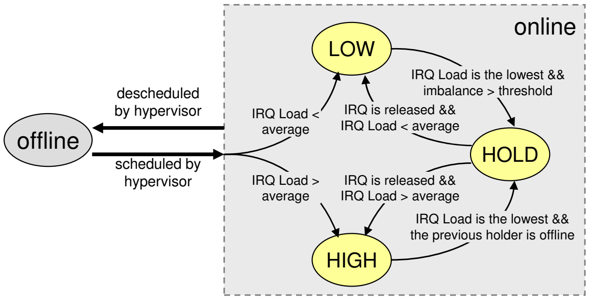

online
LOW
HOLD
HIGH
offline
descheduled by hypervisor
scheduled by hypervisor
IRQ Load < average
IRQ Load > average
IRQ Load is the lowest && imbalance > threshold
IRQ is released && IRQ Load < average
IRQ is released && IRQ Load > average
IRQ Load is the lowest && the previous holder is offline

Figure 4: vCPU state machine diagram

To remap the interrupts from one vCPU to another, there are two scenarios that need to be considered: first, the original interrupt holder is already descheduled, then there must be a remap operation; second, the original interrupt holder is still running on the physical core, however, a sufficiently large interrupt imbalance has occurred, thereby triggering an interrupt migration. To determine an imbalanced condition, we define an *imbalance ratio* R:

$$R_t = \frac{ld_{max}(ovc)_t}{ld_{min}(ovc)_t} \tag{2}$$

If R_t is detected to exceed a predefined *threshold* value, the interrupts will be migrated to the online vCPU which has the minimum interrupt load. Once the new mapping is generated, the hypervisor is notified through hypercalls to rebind the event channels to the new vCPU. The detailed operations are presented in Algorithm 1.

4. IMPLEMENTATION

We have implemented a prototype of vBalance using 64-bit Xen 4.1.2 as the hypervisor and 64-bit Linux 3.2.2 as the paravirtualized guest operating system. The design of vBalance is generic and hence applicable to many other hypervisors (e.g. KVM [26], VMware ESX [41] and Hyper-V [9]). Our implementation aims to minimize the footprint of vBalance by reusing existing Xen and Linux code as much as possible. vBalance only requires a few modifications to the hypervisor layer, involving about 150 lines of code. The implementation in Linux guest is highly modular, containing less than 300 lines of code.

4.1 Xen Hypervisor

The first modification is in shared_info data structure. The shared_info page is accessed throughout the runtime by both Xen and the guest OS. It is used to pass information relating to the vCPUs and VM states between the guest OS and the hypervisor, such as memory page tables, event status, wallclock time, etc. So it is natural that we put the vCPU scheduling status information in shared_info data structure and pass it to the guest OS at runtime. The sched_info simply includes an unsigned long integer type, out of which each bit represents the scheduling status of one vCPU (0 – offline, 1 – online).

```
# scheduling status of smp vcpus
typedef struct {
    unsigned long vcpu_mask;
} sched_info;
```

The value of sched_info will be updated when the function schedule() is called in the hypervisor scheduler. If the vCPU does not belong to the idle domain, the corresponding bit of vcpu_mask is updated before the context switch.

The second modification is to add *fast path* scheduling support for SMP guest domains. This involves a small change to Xen's credit scheduler [4], by adding another wake up branch for SMP vCPUs. We do not make any modification to Xen's split driver model and event channel mechanism. Specifically, in evtchn_send function, when it finds that the value of vcpu_mask is zero which means that there are no online vCPUs, the targeted vCPU will enter fast path to wake up. Eventually, it will

Algorithm 1: vBalance algorithm

```
Let prev_vc denote the previous vcpu that holds the IRQs;
Let next_vc denote the next vcpu that will hold the IRQs;
$ld_{avg}(vc)_t \leftarrow avg(ld(vc_0)_t, ld(vc_1)_t, ..., ld(vc_{n-1})_t)$;      /* precompute the average IRQ load of all vcpus */
Get online vcpus set $OVC_t$;
Sort online vcpus by their interrupt loads $ld(ovc)_t$;
$ld_{max}(ovc)_t \leftarrow max(ld(ovc_0)_t, ld(ovc_1)_t, ..., ld(ovc_{k-1})_t)$;      /* get the maximum IRQ load of onine vcpus */
$ld_{min}(ovc)_t \leftarrow min(ld(ovc_0)_t, ld(ovc_1)_t, ..., ld(ovc_{k-1})_t)$;      /* get the minimum IRQ load of onine vcpus */
for each online vcpu $ovc_i$ do
    if $ld(ovc_i)_t > ld_{avg}(vc)_t$ then
        $ovc_i.state \leftarrow HIGH$;
    else
        $ovc_i.state \leftarrow LOW$;
    end
end
if prev_vc is not online then
    next_vc $\leftarrow \{ovc \mid ld(ovc)_t = ld_{min}(ovc)_t\}$;      /* must map the irq to an online vcpu */
else
    if prev_vc.state is $HIGH$ then
        if $ld_{max}(ovc)_t > ld_{min}(ovc)_t * threshold$ then
            next_vc $\leftarrow \{ovc \mid ld(ovc)_t = ld_{min}(ovc)_t\}$;      /* the imbalance threshold is reached */
        else
            next_vc $\leftarrow$ prev_vc;      /* no need to remap */
        end
    else      /* no need to remap */
        next_vc $\leftarrow$ prev_vc;
    end
end
if next_vc != prev_vc then
    rebind irq to next_vc; ;      /* final operation */
end
```

be given a `SMP_BOOST` priority with preemption, behaves as the following.

```
# fast path for smp vcpus to wake up
static void csched_smp_vcpu_wake (struct vcpu *vc)
{
     struct csched_vcpu * const svc = CSCHED_VCPU(vc);
     const unsigned int cpu = vc->processor;
     if (__vcpu_on_runq(svc)) {
          __runq_remove(svc);
     }
     svc->pri = CSCHED_PRI_TS_SMP_BOOST;
     __runq_insert(cpu, svc);
     # tickle the cpu to schedule svc using softirq
     __runq_tickle(cpu, svc);
}
```

4.2 Guest Operating System

For the implementation in the guest OS, there are two technical challenges: (1) should vBalance be implemented as a user-level daemon or a part of kernel code? (2) how to determine the interrupt load of each vCPU?

4.2.1 User-level or Kernel-level

One possible approach is to implement vBalance as a user-level LSB (Linux Standard Base) daemon. The scheduling status information of vCPUs `vcpu_mask` can be accessed by a user-level program through a pseudo device residing in `/dev` or `/proc` file system. In this way, the footprint at the kernel-level is minimized. The most appealing benefit of this approach is high flexibility. Programmers can easily change the load balancing policy, function parameters, etc. The effort needed to develop and debug the program is also much less than that of kernel programming. For example, programmers can read interrupt binding information directly from `/proc/interrupts` and change them by simply writing to `/proc/irq/irq#/smp_affinity`.

A very important problem of this approach is how to determine *rebalance interval.* Linux irqbalance [7] uses a 10 seconds rebalance interval, which may not be appropriate in virtual machines, because the program needs to know which vCPUs are online and which are not, in a real-time manner. Of course the user-level program can periodically read the pseudo device to get the scheduling status information of each vCPU. However, since the hypervisor can update `vcpu_mask` in the magnitude of milliseconds and it is totally transparent to the user space, the program must poll the pseudo device very frequently to sense the change, which will consume a large amount of computing resources. Besides, if the vCPU that the program is running on was just preempted by the hypervisor, the rebalance function would be stalled.

Therefore, we argue that the implementation should reside in the kernel space. For paravirtualized guest kernel, once the vCPU is scheduled, an *upcall* will be executed to check the pending events. In Linux guest for Xen, it is `xen_evtchn_do_upcall`. Though the vCPU is not able to know when it will be preempted, it exactly knows when it is scheduled again, and thus gets the updated scheduling status at the right time.

4.2.2 Determine the Interrupt Load of vCPUs

There are two types of interrupt load statistics for each processor: (1) the number of interrupts received, (2) the time spent on processing all received interrupts. In Linux, the results can be obtained by reading `/proc/interrupts` and `/proc/stat` directly. [1] In most cases, there are actually only two interrupt sources that need to be considered: network interrupt and disk interrupt. For the other interrupts like virtual timer, virtual console, etc., the CPU cycles they consume can be nearly ignored. On the micro level, the CPU time that two interrupts consume may be different, even out of the same type. This is because the data each interrupt carries can be of very different sizes. But from a long-term view, if the two vCPUs have processed relatively the same amount of interrupts, the CPU cycles they consume should be approximately the same. So both types of interrupt load statistics can be used as the measurement method. For simplicity, we choose the second type which counts the CPU cycles the processor has spent on handling all interrupts received.

4.2.3 vBalance Components

vBalance contains several components in Linux guest kernel, with highly modular implementation. We will introduce them in detail one by one.

`vBalance_init()` is used to obtain the interrupt numbers from specific interrupt handlers, as they are dynamically allocated in guest kernel. For network interrupt, the IRQ handler is `xennet_interrupt`; for disk interrupt, the IRQ handler is `blkif_interrupt`. As a prototype implementation, we do not consider the VM that is configured with multiple virtual NICs or multiple virtual disks, which is likely to be our future research direction.

`vBalance_monitor()` reads `sched_info` from the shared memory page, and collects interrupt statistics of each vCPU from time to time. In Linux kernel 3.2.2, ten types of statistics are supported for each processor, including the time of normal processes executing in user mode (`user`), the time of processes executing in kernel mode (`system`), etc. We only use two types of them that are directly related to interrupt processing: the time of servicing hardware interrupts (`irq`) and the time of servicing softirqs (`softirq`).

`vBalance_analyze()` determines whether there is a need to remap the interrupts to another vCPU, based on the information from `vBalance_monitor()`. There are two conditions: (1) if the interrupt-bind vCPU is already offline, there must be a remap operation, and we simply select the vCPU that currently has the lowest interrupt load; (2) if the interrupt-bind vCPU is still online, the remap operation is done only when a severe imbalance among vCPUs is detected, as stated in Section 3.2.2. The *imbalance ratio* R_t is compared with a predefined *threshold* value. Currently we define *thredhold* to be 1.5, which means that the maximum interrupt load of one vCPU will not exceed 1.5 times of the minimum interrupt load.

`vBalance_remap()` is responsible to carry out the final interrupt remap operation. There are two existing functions `irq_set_affinity()` and `rebind_irq_to_cpu()` that can be used directly. The first one is quite high-level for general-purpose use, while the second one is relatively low-level and designed particularly for paravirtualized guest. For efficiency, we use `rebind_irq_to_cpu` to inform the hypervisor the change of event-notified vCPU. There is a corner case that must be handled: since vBalance is executed with the interrupts disabled, if an external event arrives before the interrupts are enabled again, the corresponding interrupt handler will not be invoked. So after the function finishes, we need to check the event status again and see whether it is needed to explicitly call the handler.

[1] To enable interrupt time accounting, the option `CONFIG_IRQ_TIME_ACCOUNTING` must be opened when compiling Linux 3.2.2 kernel source code. Other Unix-like operating systems also have similar IRQ statistics functionalities.

5. EVALUATION

In this section, we present the evaluation of our Xen-based vBalance prototype in detail. Both micro-benchmarks and application-level benchmarks are used to evaluate the effectiveness of vBalance.

5.1 System Configuration

Our experiments are performed on several Dell PowerEdge M1000e blade servers, connected with a Gigabit Ethernet switch. Each server is equipped with two quad-core 2.53GHz Intel Xeon 5540 CPUs, 16GB physical memory, two GbE network cards and two 250GB SATA hard disks (RAID-1 configured). We use a 64-bit Xen hypervisor (version 4.1.2) and a 64-bit Linux kernel (version 3.2.2) as the guest OS.

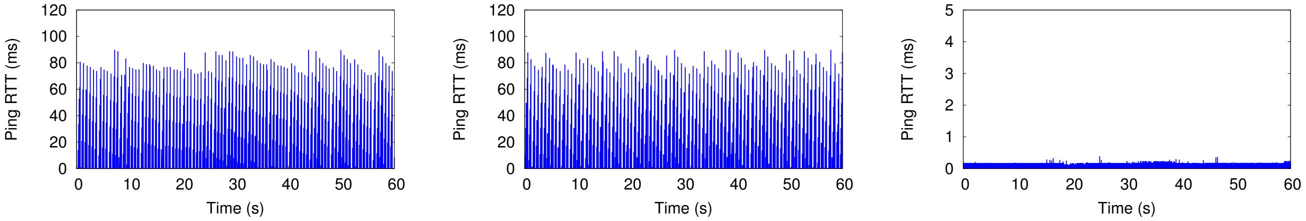
We run a set of experiments on the following three systems for the purpose of comparison:

- **Xen-native**: unmodified Xen 4.1.2 hypervisor with vanilla Linux 3.2.2 guest OS, as the baseline.
- **Irqbalance**: unmodified Xen 4.1.2 hypervisor with vanilla Linux 3.2.2 guest OS, running a traditional irqbalance [7] daemon inside.
- **vBalance**: with vBalance deployed in both Xen 4.1.2 hypervisor and Linux 3.2.2 guest OS.

5.2 Micro-level Benchmarks

This section presents the evaluation results of network performance experiments using micro-level benchmarks. The goal of the tests is to answer one question: *even if the SMP-VM gets CPU resource, can CPU cycles be properly used to serve I/O requests*? The SMP-VM under test was configured with 4 vCPUs and 2GB physical memory. We created an extreme case for stress tests *by pining all four vCPUs to one physical core*, to see how serious the problem is and how much improvement vBalance can achieve. The setup guarantees that: (1) the SMP-VM can always get CPU cycles by owning a dedicated core; (2) the vCPUs are stacked in one runqueue so that every vCPU will suffer the queueing delays. To trigger the vCPU scheduling in the hypervisor, we ran a four-threaded CPU burn script in the guest to make all vCPUs runnable. The driver domain ran on separated cores so that it will not affect the guest domains.

5.2.1 Network Latency



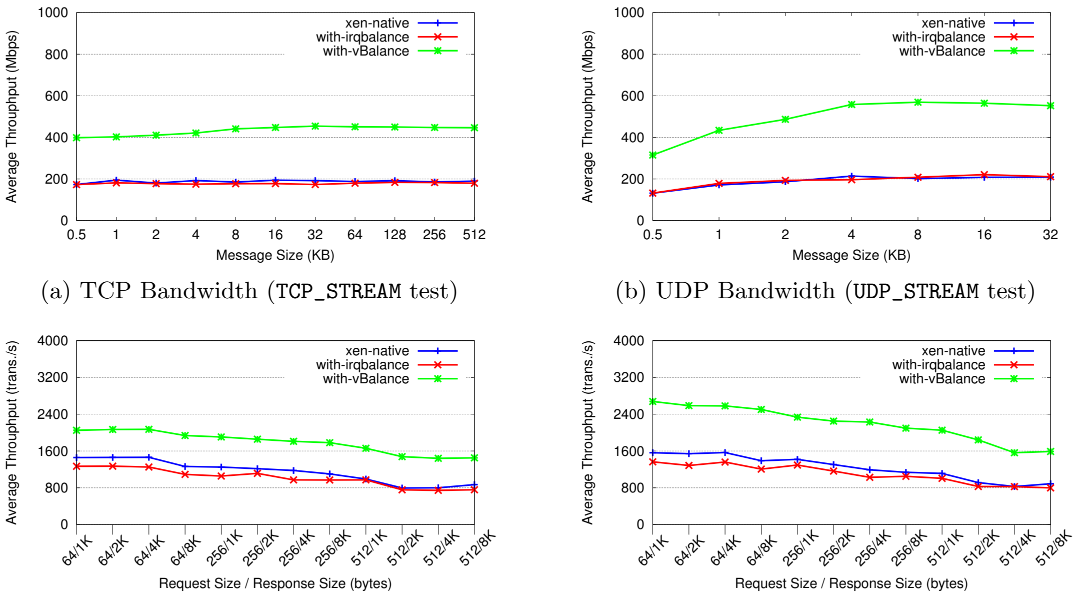
(a) Xen native (b) with irqbalance (c) with vBalance

Figure 5: Ping RTT evaluation results

To gain a micro-scope view of network latency behavior, we used *ping* with 0.1 second interval (10 ping operations per second) to measure the round trip time from one physical server to the SMP-VM. Each test lasted for 60 seconds. Figure 5 shows the testing results of RTTs. It can be seen that with default Xen and Linux irqbalance, the RTT largely varies, with 90ms peak reached. This can be explained that Xen's credit scheduler use a 30ms time slice, so the maximum scheduling latency of each vCPU of a 4vCPU-VM is 3 × 30 ms. vBalance can effectively reduce the network RTTs to less than 0.5ms, with no spikes observed. We can draw the conclusion that with Xen's credit scheduler: (1) even the whole SMP-VM always gets CPU resource, the large network latencies can still happen if the vCPU responsible for network interrupt can not get CPU cycles when I/O events arrive; (2) Linux irqbalance running in the VM does not show any benefit in reducing network latency.

5.2.2 Network Throughput

We used *netperf* [10] to evaluate both TCP and UDP network throughput. The SMP-VM hosted a *netserver* process and we ran a *netperf* process as the client on another physical machine. We wrote a script to automate the testing process. Each test lasted for 60 seconds and was repeated for three times to calculate the average value.



(a) TCP Bandwidth (`TCP_STREAM` test)

(b) UDP Bandwidth (`UDP_STREAM` test)

(c) TCP Transactions (`TCP_RR` test)

(d) UDP Transactions (`UDP_RR` test)

Figure 6: Netperf throughput evaluation results

Figure 6 (a) and (b) presents bandwidth testing results, using varied message sizes. The `TCP_STREAM` evaluation results show that vBalance achieves more than twice throughput compared with default Xen, while irqbalance shows no apparent performance improvement. The results of `UDP_STREAM` is even better than that of `TCP_STREAM`, with maximum performance reaching 2.8 times of default Xen. This is because TCP has to generate bidirectional traffic (SYN and ACK packets) to maintain reliable communication, whereas UDP traffic is unidirectional, therefore it can transmit more data. Figure 6 (c) and (d) show the transaction results for varied sizes of request/response combinations. The combinations are designed to reflect the fact that the sizes of network request packets are usually small whereas the replied data from the servers can be quite big. The transaction rate decreases along with the increase of request packet size and response packet size. Our solution significantly outperforms both default Xen and irqbalance in all test cases. Compared with default Xen, in `TCP_RR` tests, the maximum improvement of vBalance is 86%, reached at "512/2K" test case; in `UDP_RR` tests, vBalance maximally achieves 102% improvement, also reached at "512/2K" test case.

5.3 Application-level Benchmarks

In this section, we use two application-level benchmarks to measure the performance improvement of vBalance. The SMP-VM under test was configured with 4 vCPUs. This time, we allocated four physical cores to host the SMP-VM. To create the consolidate scenario, we ran eight CPU-intensive UP-VMs on the same four physical cores as the background. So on average, there were two co-located vCPUs per physical core to compete CPU cycles with the SMP-VM's vCPUs. All vCPUs were subjected to the load balancing activity of Xen's credit scheduler. As a routine, the driver domain ran on separate cores so that they would not affect the guest domains.

5.3.1 Apache Olio

Apache Olio [2, 37] is a Web 2.0 toolkit to help evaluate the suitability and performance of web technologies. It features a social-event calendar, where users can create and tag events, search, send comments, etc. Apache Olio includes a Markov-based distributed workload generator and data collection tools. We use Apache Olio (version 2.0) PHP5 implementation for our experiments which contains four components: (1) an Apache web server acting as the web front-end to process requests,

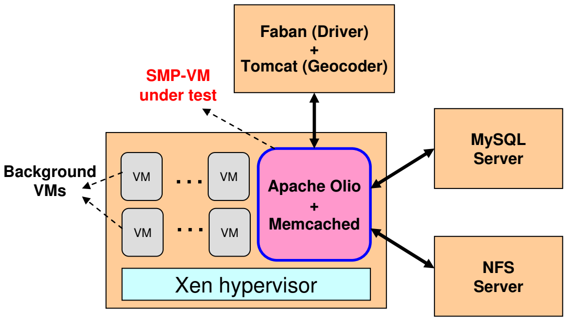


Figure 7: The experimental setup of Apache Olio benchmark

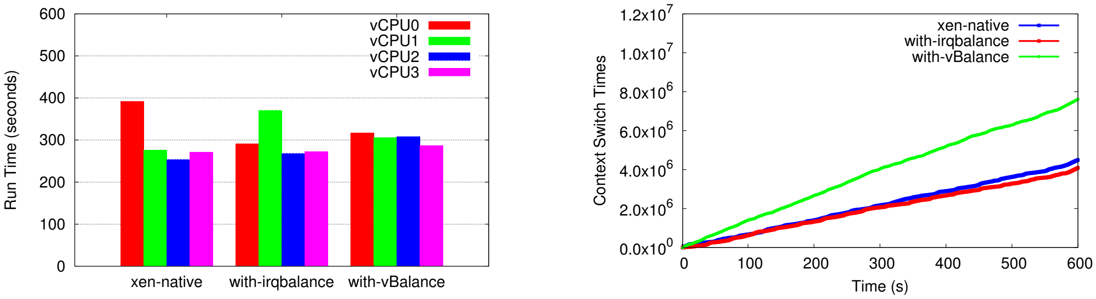
(2) a MySQL server that stores user profiles and event details, (3) an NFS server that stores user files and event specific data, and (4) a Faban [6] driver that generates the workload to simulate the external users' requests.

Table 1: Apache Olio benchmark results – throughput (count: success/failure)

Operation	1. Xen native	2. with irqbalance	with vBalance	**Improv–1**	**Improv–2**
HomePage	11177/12	11043/12	12493/0	+ 11.8%	+ 13.1%
Login	4076/565	4227/382	4917/0	+ 20.6%	+ 16.3%
TagSearch	14271/8	14297/16	16142/0	+ 13.1%	+ 12.9%
EventDetail	10580/6	10424/9	11791/0	+ 11.4%	+ 13.1%
PersonDetail	1091/1	1110/3	1241/0	+ 13.7%	+ 11.8%
AddPersion	358/0	349/0	408/0	+ 14.0%	+ 16.9%
AddEvent	841/1	865/3	903/0	+ 7.4%	+ 4.4%
Rate(ops/sec)	70.657	70.525	79.825	+ 13.0%	+ 13.2%

Table 2: Apache Olio benchmark results – average response times (seconds)

Operation	1. Xen native	2. with irqbalance	with vBalance	**Reduct–1**	**Reduct–2**
HomePage	1.387	1.346	0.212	– 84.7%	– 84.2%
Login	0.958	1.002	0.090	– 90.6%	– 91.0%
TagSearch	1.657	1.736	0.306	– 81.5%	– 82.4%
EventDetail	1.438	1.462	0.244	– 83.0%	– 83.3%
PersonDetail	1.825	1.785	0.339	– 81.4%	– 81.0%
AddPersion	2.852	3.012	0.800	– 71.9%	– 73.4%
AddEvent	3.341	3.722	0.972	– 70.9%	– 73.9%



(a) The runtime of each vCPU

(b) Context switch overhead

Figure 8: The hypervisor statistics of Apache Olio benchmark

Figure 7 shows our testbed configuration. Except from Olio application and Memcached server, the other components were deployed on physical machines. The SMP-VM was configured with 4GB physical memory, with 1GB used for Memcached service. We configured Faban driver to run for 11 minutes, including 30-second ramp up, 600-second steady state and 30-second ramp down. Two agents were used to control 400 concurrent users to generate different types of requests, with each agent taking care of 200 users. For fairness, MySQL server was configured to reload database tables in every test. We evaluated the number of operations and average response times performed by Apache Olio.

Table 1 shows the total count of different operations performed by Apache Olio. Regarding the overall rate (ops/sec), vBalance achieves 13.0% and 13.2% performance improvement respectively, compared with default Xen and irqbalance. Regarding each single operation, the maximum improvement is 20.6% compared with default Xen (reached at *Login* operation), and 16.9% compared with irqbalance (reached at *AddPerson* operation). One important observation is that without vBalance, many operations are reported to be failure. For example, *Login* operation with default Xen reported *connection timeout error* for 565 times, taking up 13.9% out of all login trials. This should be caused by too long scheduling delays of the corresponding vCPU. With our vBalance, we saw no failure case.

The results of average response times of each operation are presented in Table 2. The significant improvements are not surprising in that the scheduling latencies of vCPUs from hypervisor will eventually translate into response delays to user requests. With vBalance, the response times were significantly reduced. The maximum improvements are observed at *Login* operation for both default Xen and irqbalance, 90.6% and 91.0% respectively. The results respond to the large number of *connection timeout error* of *Login* operation in Table 1.

Figure 8 (a) shows the CPU cycles each vCPU consumed during Apache Olio tests. It is foreseeable that default Xen utilizes vCPU0 much more than the other vCPUs, as all interrupts are delivered to vCPU0 by default. However, irqbalance also cannot achieved balanced load among vCPUs. This may be explained that irqbalance makes rebalance decision not only based on vCPU load statistics, but also on cache affinity, power consumption, etc. As the guest OS is not able to obtain these hardware-related information, the decisions that irqbalance makes are mostly inappropriate. Figure 8 (b) shows the context switch times happened on the four physical cores. The results of default Xen and irqbalance are very close. This makes sense as irqbalance wakes up every ten seconds, so its effect on context switch times can be nearly ignored. Since vBalance introduces a *fast path* mode for SMP-VM's vCPUs in hypervisor scheduler, it causes more context switches. Compared with default Xen, vBalance only introduced 69% more context switches, keeping 30ms time slice unchanged. This proves that the design of vBalance in the hypervisor level is really light-weight.

5.3.2 Dell DVDStore

Dell DVDStore [5] is an OLTP-style e-commerce application that simulates users browsing an online DVD store and purchasing DVDs. This application includes a back-end database component, a web application layer and a driver program. The driver program simulates users logging in, browsing for DVDs by title, actor or category, adding selected DVDs to their shopping cart, and then purchasing those DVDs. We chose DVDStore (version 2.1) PHP5 implementation to evaluate the effectiveness of vBalance. The setup is shown in Figure 9. We set MySQL database size to be 4GB. We started 16 threads simultaneously to saturate the application server's capability. Each test lasted for 11 minutes, including 1-minute warm up and 10-minute stable running. The driver reports testing results every 10 seconds.

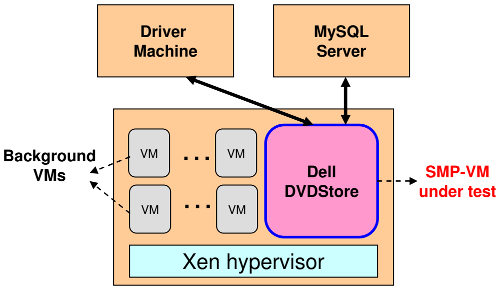
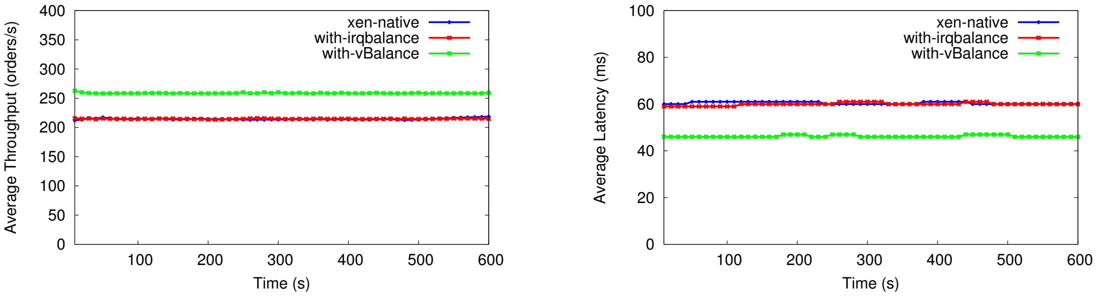


Figure 9: The experimental setup of Dell DVDStore benchmark

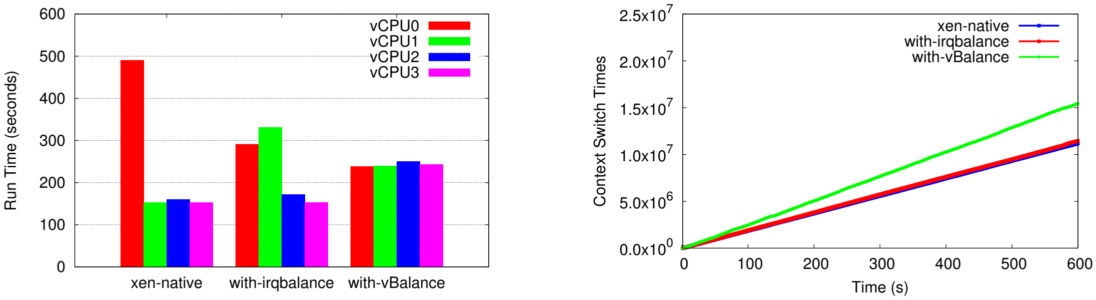
Figure 10 (a) shows the results of average throughput. Figure 10 (b) presents average request latency. vBalance improves the average throughput from ∼210 orders/sec to ∼260 orders/sec (about 24% improvement), and reduces the average request latencies from ∼60ms to ∼46ms (about 30% improvement). Figure 11 (a) shows the CPU cycles each vCPU consumed during the three tests. Compared with Apache Olio results in Figure 8 (a), the SMP-VM's unbalanced vCPU load is more serious. This is because: 1) Apache Olio tests can be configured to use a Memcached server to store recently used data, so the application does not need to read from NFS server and thus reduces network traffic; 2) Apache Olio can preload database tables (user profiles, events information, etc.) into memory before the Faban driver sends user requests; during the website operations, application server almost does not need to bother MySQL server and therefore also saves lots of network traffic. The situation of Dell DVDStore is quite different: every user request will trigger the database operations to MySQL server, and eventually translates into network traffic. The vCPU0 was bothered much more frequently by interrupts than that in Apache Olio tests. Figure 11 (b) shows the results of context switch times on the four physical cores hosting the experiments. The default Xen performed very similar to irqbalance. vBalance only introduced 35% more context switch overhead than default Xen.



(a) Average throughput

(b) Average request latency

Figure 10: The evaluation results of Dell DVDStore benchmark



(a) The runtime of each vCPU

(b) Context switch overhead

Figure 11: The hypervisor statistics of Dell DVDStore benchmark

6. RELATED WORK

6.1 Hypervisor Scheduling

6.1.1 Scheduling for I/O

Significant effort has been paid to address the VM's I/O performance problem in recent years. Task-level solutions [25, 34] map I/O bound tasks from the guest OS directly to physical CPUs. Since the characteristics of tasks may vary from time to time, the solution often needs additional efforts to predict the changes. Besides, the implementation involves heavy modifications from the guest OS scheduler to the hypervisor scheduler. Several non-intrusive approaches [22, 27, 32, 45] use real-time schedulers to shorten the vCPU scheduling latencies for event processing. First, they require complex configurations and careful parameter tuning and selection, which may not be feasible in a cloud environment with dynamic placement and migration of VMs. Our solution does not introduce any extra parameter or special configurations, thus does not bring additional management burden. Second, to meet the deadlines of some VMs, the time slice they use is very small (less than 1ms in most cases), which will excessively increase the context switch overhead. Our solution does not change the time slice (30ms) used in Xen's credit scheduler. Research [30] proposes to run a polling thread to monitor the event status, and schedules the vCPU as soon as possible once the events arrive. The drawback is that an additional physical core is needed to run the polling thread, resulting in low resource utilization. Research [23, 46] propose to dynamically reduce the time slice length used in the hypervisor scheduler, so as to decrease the waiting time of each vCPU in the runqueue. Apparently, these approaches will exponentially increases the context switch times among VMs. For instance, when using a time slice of 0.1ms as suggested in research [23], theoretically the context switch overhead becomes 300 times bigger than that of xen's credit scheduler (30ms)!

Another common problem of the above solutions is that none of them is designed for *SMP-VMs*. Without the cooperation from the guest OS to balance the interrupts load among multiple vCPUs, the potential to improve I/O performance at the hypervisor layer is quite limited.

6.1.2 Co-scheduling for SMP Virtual Machines

Most research papers with SMP-VMs focus on "synchronization" problem inside the guest OS. The lock contention problem among multiple processors is already a well-known problem, which will degrade the parallel application's performance by serializing its execution. However, with the OS running in a VM, the synchronization latency of spinlocks becomes more serious, as the vCPU holding the lock may be preempted by the hypervisor at any time. Research [40] identifies this problem as LHP (Lock-Holder Preemption): the vCPU holding a lock is preempted by the hypervisor, resulting in much longer waiting time for the other vCPUs to get the lock. They propose several techniques to address LHP problem, including intrusively augmenting the guest OS with a delayed preemption mechanism, non-intrusively monitoring all switches between user-level and kernel-level to determine the safe preemption, and making the hypervisor scheduler locking-aware by introducing a preemption window. Research [38] proposes "balance scheduling" by dynamically setting CPU affinity to guarantee that no vCPU siblings are in the same CPU's runqueue. Research [43] addresses this problem by detecting the occurrence of spinlocks with long waiting times and then determine co-scheduling of vCPUs. Research [14] uses gray-box knowledge to infer the concurrency and synchronization of guest tasks. VMware proposes the concept of "skew" to guarantee the synchronized execution rates between vCPUs [3].

All the above approaches are actually the variants of co-scheduling algorithm proposed in [33], which schedules concurrent threads simultaneously to reduce the synchronization latency. For SMP-VMs, the vCPUs from the same VM are co-scheduled. That is, if physical cores are available, the vCPUs are mapped one-to-one onto physical cores to run simultaneously. In other words, if one vCPU in the VM is running, a second vCPU is co-scheduled so that they execute nearly synchronously. Though it may mitigate the negative effect of spinlocks inside guest OS to some extent, it helps little to reduce interrupt processing delays, as vCPU scheduling latencies from the hypervisor are *inherently unavoidable*. Besides, the CPU fragmentation problem introduced can reduce CPU utilization [28]. To our best knowledge, we are the first one to use interrupt load balance technique to improve I/O performance for SMP-VMs.

6.2 Linux Irqbalnace

Irqbalance [7] is a Linux daemon that distributes interrupts over the processors and cores in SMP physical machines. The design goal is to find a balance between power savings and optimal performance. By periodically analyzing the interrupt load on each CPU (every 10 seconds by default), interrupts are reassigned among the eligible CPUs. Irqbalance also considers cache-domain affinity, and tries to make each interrupt stand a greater chance of having its interrupt handler be in cache. It automatically determines whether the system should work in power-mode or performance-mode, according to the system's workload.

As the underlying execution environment for OS significantly differs in virtualization world, the solution designed based on physical assumptions are not effective at all. First, irqbalance has no knowledge of the scheduling status of each vCPU, so it has no way to correctly determine which vCPUs are eligible ones; a rebalance interval of 10 seconds is too long, as the hypervisor schedules the vCPUs in the magnitude of milliseconds. Second, it is quite difficult to predict the cache behaviors even for the hypervisor, so it is infeasible for the guest OS to manage cache directly. Third, power saving will be much more powerful to work in the hypervisor, which accesses the hardware directly.

6.3 Hardware-based Solutions for Directed I/O

Hardware-based approaches assign dedicated devices to VMs and allow direct hardware access from within the guest OS. In this approaches, performance critical I/O operations can be carried out by interacting with the hardware directly from a guest VM. For example, Intel VT [39] (including VT-x, VT-d, VT-c, etc.) provides the platform hardware support for DMA and interrupt virtualization [13]. *DMA remapping* transforms the address in a DMA request issued by an I/O device, which uses

Machine Physical Address (MPA) to the VM's corresponding Guest Physical Address (GPA). *Interrupt remapping* hardware distinguishes interrupt requests from specific devices and routes them to the appropriate VMs to which the respective devices are assigned. By offloading many capabilities into hardware and simplifying the execution, it can greatly improve the I/O performance of VMs.

However, these approaches have several limitations. First, it sacrifices key advantages of a dedicated driver domain model: device driver isolation in a safe execution environment avoiding guest domain corruption by buggy drivers. Instead, a virtual machine would have to include device drivers for a large variety of devices, increasing their size, complexity, and maintainability. Second, it requires special support in both processors and I/O devices, such as self-virtualized PCI devices [11] which present multiple logical interfaces, thereby increases hardware cost. Third, using the intelligent hardware for VMs is already a very complicated task [19, 35], it also complicates many other functionalities such as safety protection [44], transparent live migration [24], checkpointing/restore, etc. Last but most importantly, even with hardware support like DMA-remapping and interrupt migration, the processing of a interrupt still relies on whether the targeted vCPU can be scheduled timely by the hypervisor.

7. DISCUSSIONS AND FUTURE WORK

Time Slice of the Hypervisor Scheduler. Since our solution mainly relies on the guest-level strategies to migrate interrupts among vCPUs, it is independent from the time slice used in the hypervisor scheduler. Therefore, using a longer time slice for SMP-VMs should cause less context switches. For the guest OS, since each vCPU can run longer at a time, the frequency of interrupt migration can be reduced. Research [23] divides the physical cores into several groups (e.g. normal cores, fast-tick cores), and uses dedicated fast-tick cores to schedule I/O bound vCPUs. We are inspired to use "slow-tick" cores to host SMP-VMs, with vBalance running in the guest OS. Considering the "synchronization" problem mentioned in the related work [14, 38, 40, 43], how this approach can effectively work with co-scheduling is worth investigating.

Extension of the Algorithm. The current load balance strategy migrates interrupts to only one vCPU (many-to-one). This simplifies the design and implementation, and works well when the number of interrupt sources is small (e.g. one NIC plus one disk). However, if the SMP-VM is configured with multiple NICs or disks, when applications simultaneously access them, the current solution may not optimally balance the workload. Therefore, a many-to-many interrupt mapping algorithm is desirable in the future. As such, the historical workload of each interrupt source may be used to predict its future pattern, and based on these heuristics the interrupts are distributed to multiple online vCPUs. The problem is complicated in that a tradeoff between fairness and efficiency needs to be achieved.

Cache-Awareness. Cache management is known to be a quite complicated problem for VMs, subject to vCPU migrations among physical cores and VM migrations among physical hosts. Modern CPU hardware is increasingly to be NUMA style, with each node owning a shared LLC (Last Level Cache) of very big size (e.g., Intel's Nehalem [36]). The future hypervisor scheduler should be cache-aware, making vCPUs fairly benefit the hardware cache. Cache partitioning [29] has been proposed to address the conflicting accesses in shared caches. Virtual Private Caches (VPCs) [31] provide hardware support for the Quality of Service (QoS) of Virtual Private Machines (VPM). For guest OS, it is expected to always migrate the interrupts to cache-hot vCPUs, instead of cache-hold ones. In order to make the guest OS aware of whether the vCPU is cache-hot or cache-cold, the CPU-tree information needs to be passed to the guest level. How interrupt load balancing inside guest OS can benefit from cache-awareness is a research problem.

8. CONCLUSION

High performance I/O virtualization is endlessly desirable in data-intensive cloud computing era. In this paper, we propose a new I/O virtualization approach called vBalance for SMP virtual machines. vBalance is a cross-layer solution, which takes advantage of the guest-level help to accelerate I/O speed by adaptively and dynamically migrating interrupts from a preempted vCPU to a running one. Unlike traditional approaches which impose all pressure on the hypervisor scheduler, vBalance exploits the potential of guest operating system and is quite light-weight in the hypervisor. vBalance is based on software and does not require special hardware support. To demonstrate the idea of vBalance, we developed a prototype using Xen 4.1.2 and Linux 3.2.2. Our stress tests with micro-level benchmarks show that vBalance significantly reduces network latency and greatly improves network throughput. The experimental results with representative cloud-style applications show that vBalance easily outperforms the original Xen, achieving much lower response time and higher throughput. Overall, leveraging vBalance makes SMP virtual machines more appealing for applications to deploy.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This research is supported by a Hong Kong RGC grant HKU 7167/12E and in part by a Hong Kong UGC Special Equipment Grant (SEG HKU09).

10. REFERENCES

[1] Amazon EC2: http://aws.amazon.com/ec2/.

[2] Apache Olio: http://incubator.apache.org/olio/.

[3] Co-scheduling SMP VMs in VMware ESX Server: http://communities.vmware.com/docs/doc-4960.

[4] Credit Scheduler: http://wiki.xensource.com/xenwiki/creditscheduler.

[5] Dell DVD Store: http://linux.dell.com/dvdstore/.

[6] Faban: http://java.net/projects/faban/.

[7] Irqbalance: http://www.irqbalance.org/.

[8] Linux bridge: http://www.losurs.org/docs/ldp/howto/pdf/bridge-stp-howto.pdf.

[9] Microsoft Hyper-V Server: http://www.microsoft.com/hyper-v-server/.

[10] Netperf: http://www.netperf.org/.

[11] PCI-SIG I/O virtualization specifications: http://www.pcisig.com/specifications/iov/.

[12] VMware Virtual Networking Concepts: http://www.vmware.com/files/pdf/virtual_network-ing_concepts.pdf.

[13] Intel virtualization technology for directed i/o. *Architecture Specification - Revision 1.3, Intel Corporation*, February 2011.

[14] Y. Bai, C. Xu, and Z. Li. Task-aware based co-scheduling for virtual machine system. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 181–188.

[15] 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, volume 37, pages 164–177, 2003.

[16] S. K. Barker and P. Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems (MMSys)*, pages 35–46, 2010.

[17] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference*, pages 41–41.

[18] P. Ben, P. Justin, K. Teemu, A. Keith, C. Martin, and S. Scott. Extending networking into the virtualization layer. In *HotNets-VIII*, 2009.

[19] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. V. Doorn. The price of safety: Evaluating iommu performance. In *The 2007 Ottawa Linux Symposium (OLS)*, pages 9–19.

[20] L. Cheng and C.-L. Wang. Network performance isolation for latency-sensitive cloud applications. *Future Generation Computer Systems*, 2012.

[21] L. Cheng, C.-L. Wang, and S. Di. Defeating network jitter for virtual machines. In *The 4th IEEE International Conference on Utility and Cloud Computing (UCC)*, pages 65–72, 2011.

[22] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE)*, pages 126–136, 2007.

[23] 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 *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, pages 142–154, 2010.

[24] A. Kadav and M. M. Swift. Live migration of direct-access devices. *ACM SIGOPS Operating Systems Review*, 43:95–104, July 2009.

[25] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for i/o performance. In *Proceedings of the 5th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 101–110, 2009.

[26] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *The 2007 Ottawa Linux Symposium (OLS)*, pages 225–230.

[27] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the xen hypervisor. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, volume 45, pages 97–108, 2010.

[28] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of i/o for gang scheduled workloads. In *Job Scheduling Strategies for Parallel Processing*, volume 1291, pages 215–237, 1997.

[29] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *IEEE 14th International symposium on High Performance Computer Architecture (HPCA)*, pages 367–378, 2008.

[30] J. Liu and B. Abali. Virtualization polling engine (vpe): using dedicated cpu cores to accelerate i/o virtualization. In *Proceedings of the 23rd International Conference on Supercomputing (ICS)*, pages 225–234, 2009.

[31] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*, pages 57–68, 2007.

[32] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 1–10, 2008.

[33] J. K. Ousterhout. Scheduling techniques for concurrebt systems. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30, 1982.

[34] R. Rivas, A. Arefin, and K. Nahrstedt. Janus: a cross-layer soft real-time architecture for virtualization. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, pages 676–683, 2010.

[35] J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, W. Zwaenepoel, and P. Willmann. Concurrent direct network access for virtual machine monitors. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 306–317, 2007.

[36] R. Singhal. Inside intel next generation nehalem microarchitecture. In *Intel Developer Forum (IDF) presentation*, August 2008.

[37] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *The First Workshop Cloud Computing and Its Applications (CCA)*, 2008.

[38] O. Sukwong and H. S. Kim. Is co-scheduling too expensive for smp vms? In *Proceedings of the sixth conference on Computer systems (EuroSys)*, pages 257–272, 2011.

[39] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38:48–56, 2005.

[40] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium (VM)- Volume 3*, pages 4–4, 2004.

[41] C. A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th symposium on Operating Systems Design and Implementation (OSDI)*, volume 36, pages 181–194, 2002.

[42] G. Wang and T. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *The 29th Annual IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–9, 2010.

[43] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic adaptive scheduling for virtual machines. In *Proceedings of the 20th international symposium on High Performance Distributed Computing (HPDC)*, pages 239–250, 2011.

[44] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized i/o devices. In *USENIX 2008 Annual Technical Conference*, pages 15–28.

[45] S. Xi, J. Wilson, C. Lu, and C. Gill. Rt-xen: towards real-time hypervisor scheduling in xen. In *Proceedings of the 9th ACM international conference on Embedded software*, pages 39–48, 2011.

[46] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. Kompella, and D. Xu. vslicer: Latency-aware virtual machine scheduling via differentiated frequency cpu slicing. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 3–14, 2012.