vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core

Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, Dongyan Xu
Department of Computer Science, Purdue University

Abstract

In a virtual machine (VM) consolidation environment, it has been observed that CPU sharing among multiple VMs will lead to I/O processing latency because of the CPU access latency experienced by each VM. In this paper, we present vTurbo, a system that accelerates I/O processing for VMs by offloading I/O processing to a designated core. More specifically, the designated core – called turbo core – runs with a much smaller time slice (e.g., 0.1ms) than the cores shared by production VMs. Most of the I/O IRQs for the production VMs will be delegated to the turbo core for more timely processing, hence accelerating the I/O processing for the production VMs. Our experiments show that vTurbo significantly improves the VMs' network and disk I/O throughput, which consequently translates into application-level performance improvement.

1 Introduction

Cloud computing is arguably one of the most transformative trends in recent times. Many enterprises and businesses are increasingly migrating their applications to public cloud offerings such as Amazon EC2 [1] and Microsoft Azure [8]. By purchasing or leasing cloud servers with a pay-as-you-go charging model, enterprises benefit from significant cost savings in running their applications, both in terms of capital expenditure (*e.g.*, reduced server costs) as well as operational expenditure (*e.g.*, management staff). On the other hand, cloud providers generate revenue by achieving good performance for their "tenants" while maintaining reasonable cost of operation.

One of the key factors influencing the cost of cloud platforms is *server consolidation*—the ability to host multiple virtual machines (VM) in the same physical server. If the cloud providers can increase the level of server consolidation, *i.e.*, pack more VMs in each physical machine, they can generate more revenue from their infrastructure investment and possibly pass cost savings on to their customers. Two main resources that typically dictate the level of server consolidation, memory and CPU. Memory is strictly partitioned across VMs, although there are techniques (*e.g.*, memory ballooning [29]) for dynamically adjusting the amount of memory available to each VM. CPU can also be strictly partitioned across VMs, with the trend of ever increasing number of cores per physical host. However, given that each core is quite powerful, another major scaling factor comes by allocating multiple VMs per core. While the ever increasing core count in modern systems may suggest the possibility of a dedicated core per VM, it is not likely to happen any time soon, as evidenced in current cloud computing environments such as Amazon EC2, where a 3GHz CPU may be shared by three small instances.

In practice, CPU sharing among VMs can be quite complicated. Each VM is typically assigned one or more virtual CPUs (vCPUs) which are scheduled by the hypervisor on to physical CPUs (pCPUs) ensuring proportional fairness. The number of vCPUs is usually larger than the number of pCPUs, which means that, even if a vCPU is ready for execution, it may not find a free pCPU immediately and thus needs to wait for its turn, causing *CPU access latency*. If a VM is running I/O-intensive applications, this latency can have a significant negative impact on application performance. While several efforts [30, 12, 17, 28, 19, 25] have made this observation in the past, and have in fact provided solutions to improve VMs' I/O performance, the improvements are still moderate compared to the available I/O capacity because, they do not explicitly focus on reducing the most important and common component of I/O processing workflow—namely *IRQ processing latency*.

To explain this more clearly, let us look at I/O processing in modern OSes today. There are two basic stages involved typically. (1) Device interrupts are processed *synchronously* in an IRQ context in the kernel and the data (*e.g.*, network data, disk reads) is buffered in kernel buffers; (2) The application eventually copies the data from kernel buffer to its user-level buffer in an *asynchronous* fashion whenever it gets scheduled by the process scheduler. If the OS were running directly on a physical machine, or if there were a dedicated CPU for a given VM, the IRQ processing component gets scheduled almost instantaneously by preempting the currently running process. However, for a VM that shares CPU with other VMs, the IRQ processing may be significantly delayed because the VM may not be running when the I/O event (*e.g.*, network packet arrival) occurs.

IRQ processing delay can affect both network and disk I/O performance significantly. For example, in the case of TCP, incoming packets are staged in the shared memory between the hypervisor (or privileged domain) and the guest OS, which delays the ACK generation and can result in significant throughput degradation. For UDP flows, there is no such time-sensitive ACK generation that governs the throughput. However, since there is limited buffer space in the shared memory (ring buffer) between the guest OS and the hypervisor, it may fill up quickly leading to packet loss. IRQ processing delay can also impact disk write performance. Applications often just write to memory buffers and return. The kernel threads handling disk I/O will flush the data in memory to the disk in the background. As soon as one block write is done, the IRQ handler will schedule the next write and so on. If the IRQ processing is delayed, write throughput will be significantly reduced.

Unfortunately, none of the existing efforts explicitly tackle this problem. Instead, they propose indirect approaches that moderately shorten IRQ processing latency hence achieving only modest improvement. Further, because of the specific design choices made in those approaches, the IRQ processing latency cannot be fundamentally eliminated (*e.g.*, made negligible) by any of the designs, meaning that they cannot achieve close to optimal performance. For instance, the vSlicer approach [30] schedules I/O-intensive VMs more frequently using smaller micro time slices, which implicitly lowers the IRQ processing latency, but not significantly. Also it does not work under all scenarios. For example, if two I/O latency-sensitive VMs and two non-latency-sensitive VMs share the same pCPU, the worst-case IRQ processing latency will be about 30ms, which is still non-trivial, even though it is better than without vSlicer (which would be 90ms). Similarly, another approach called vBalance [10] proposes routing the IRQ to the vCPU that is scheduled for the corresponding VM. This may work well for SMP VMs that have more than one vCPU, but will not improve performance for single vCPU VMs. Even in the SMP case, it improves the chances that at least one vCPU is scheduled; but fundamentally it does not eliminate IRQ processing latency because each vCPU is contending for the physical CPU independently.

To solve this problem more fundamentally, we aim to make the IRQ processing latency for a CPU-sharing VM almost similar to the scenario where the VM is given a dedicated core. To achieve this, we propose a new solution called vTurbo, that involves two basic ideas. First, we leverage the existence of multiple cores in modern processors to designate a specialized turbo-sliced core (or *turbo core* for short), for synchronous processing threads in the guest OS. In terms of actual hardware, the turbo core is no different from a regular core, except that the hypervisor-level scheduler schedules VMs on this core with *extremely small quantum* (*e.g.*, 0.1ms). Second, we expose this turbo-sliced core to each VM as a "co-processor" just dedicated to kernel threads that require synchronous processing, such as IRQ handling. The other regular kernel threads are scheduled on a regular core with regular slicing just like what exists today. Since the IRQ handlers are executed by the turbo core, they are handled almost synchronously with a magnitude smaller latency. For example, assuming 5 VMs and 0.1ms quantum for the turbo core, an IRQ request is processed within 0.4ms compared to 120 ms (assuming 30ms time slice for regular cores).

The turbo core is accessible to all VMs in the system. If a VM runs only CPU-bound processes, it may choose not to use this core since its performance is not likely to be good due to frequent context switches. Even if a VM chooses to schedule a CPU-bound process on the turbo core, it has virtually no impact on other VMs' turbo core access latency thus providing good isolation between VMs. We ensure fair-sharing among VMs with differential requirement between regular/turbo cores because, otherwise, it would motivate VMs to push more processing to the turbo core. Thus, for example, if there are two VMs—VM1 requesting 100% of the regular core, and VM2 requesting 50% regular and 50% turbo cores, the regular core will be split 75-25% while VM2 obtains the full 50% of the turbo core, thus equalizing the total CPU usage for both VMs. We also note that, while we mention one turbo core in the system, our design seamlessly allows multiple turbo cores in the system driven by I/O processing load of all VMs in the host. This makes our design extensible to higher bandwidth networks (10Gbps and beyond) and higher disk I/O bandwidths that require significant IRQ processing beyond what a single core can provide.

To summarize, the main contributions of this paper are as follows:

(1) We propose a new class of high-frequency scheduling CPU core named turbo core and a new class of co-vCPU called turbo vCPU. The turbo vCPU pinned on turbo core(s) is used for timely processing of the I/O IRQs thus accelerating I/O processing for VMs in the same physical host.

(2) We develop a simple but effective VM scheduling policy giving general CPU cores and turbo core magnitudes different time-slice. The very small CPU time-slice on turbo cores grants VM low scheduling delay and low I/O IRQ processing latency.

(3) We have implemented a prototype of vTurbo based on Xen. Various evaluations prove the effectiveness of vTurbo. Our micro-benchmark results show that vTurbo can significantly improve the TCP throughput (up to 3×), UDP throughput (up to 4×), and disk write throughput (up to 2×). Our evaluation with application-level benchmarks shows that vTurbo improves application-specific performance as well. For example, Olio's throughput is increased by 38.7%. NFS' throughput is improved by up to 2×.

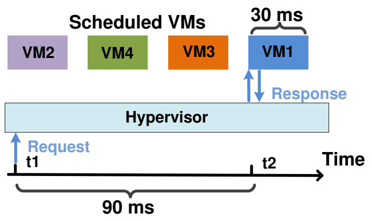
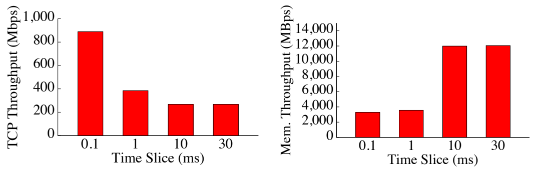


Figure 1: Impact of VM CPU sharing on I/O processing.



(a) TCP throughput (b) Memory throughput

Figure 2: Impact of micro-timeslice on TCP throughput and memory throughput

The rest of the paper is organized as follows. We motivate the problem in detail in Section 2 followed by the vTurbo design in Section 3. In Section 4 we describe the implementation of vTurbo prototype based on Xen. Section 5 presents evaluation results, followed by related work and conclusions.

2 Motivation

Let us first focus on receive-side I/O processing. In a non-virtualized system, all receive-side I/O events (*e.g.*, network packet arrival) are typically handled by specific IRQ routines corresponding to each device (*i.e.*, disk controller or NIC) in the OS kernel. The data is stored in a kernel buffer first, and once the user process is scheduled, it copies the data from the kernel buffer to the user buffer. Since I/O-bound processes usually have higher priority, they get scheduled relatively quickly and the data is subsequently processed by the application thus achieving high I/O throughput. However, in a virtualized system with several VMs sharing a physical CPU, each VM gets only a slice of the physical CPU, which means the incoming I/O event will need to wait until the VM gets access to the CPU. Such a CPU access latency will significantly affect the timeliness of IRQ processing, resulting in low I/O throughput.

We illustrate this negative effect using an example shown in Figure 1. In this example, 4 VMs share a physical CPU. VM1 runs a mixed workload that includes both CPU-bound tasks and I/O-bound applications, while VM2 to VM4 run only CPU-bound applications. Assuming a proportional-share VM scheduling policy (adopted by Xen and VMware ESX), VM1 gets only 25% of CPU when all VMs are busy, which means that roughly 75% of time, VM1 has to wait in *runqueue* and cannot process I/O events immediately. When an I/O request for VM1 reaches the hypervisor at t_1, VM1 cannot process this request and respond until t_2. If the I/O-bound application in VM1 is a TCP server, for instance, the client will stop sending data to the server once the client's TCP window is full, due to lack of acknowledgments from the server while VM1 is in *runqueue*. If VM1 runs a UDP server, even though the client can continue to send data to the server without getting responses, the packets will be dropped by the hypervisor once the shared buffers (between the hypervisor and guest OS) are full. As a result, throughput of either TCP or UDP for VM1 would be much lower than the available capacity.

In the reverse direction (*i.e.*, when a process sends packets or writes to the disk), the user process first copies data to the kernel buffer associated with the particular output (*e.g.*, socket, file descriptor). For some I/O mechanisms such as asynchronous network packet sends and disk writes, the call to output the data will return to the user process immediately after the data is copied to the kernel buffer. The kernel components associated with the corresponding device will asynchronously write the data to the device. However, this task cannot be continued efficiently if the hypervisor schedules the vCPU out while the kernel component is waiting for the completion of the write to the device, resulting in low throughput.

There are other sources of delay for interrupt processing even after the I/O event reaches the VM. These include long periods in which, the VM runs with interrupts disabled, locking conflicts for shared data structures (such as TCP accept queue [26]) and overhead of dispatching interrupts in virtualized environments [13]. However, most of these latencies lie within sub-millisecond range in the average case [20, 18], while the scheduling delay causes the interrupt processing to be delayed for tens of milliseconds (in our example, the average scheduling delay is about 35ms for Xen VMM).

Symmetric multi-processing (SMP) VMs can take advantage of a multi-core architecture to execute many different applications in parallel and improve the overall system throughput. In an SMP-VM, two or more allocated vCPUs are scheduled by the hypervisor scheduler on any available pCPUs and thus, each vCPU has a higher chance to get scheduled. However, the SMP-VM may still suffer from scheduling delays, if none of the vCPUs can be scheduled in because the pCPUs are all busy executing other vCPUs. Thus, we cannot guarantee that the vCPU running an IRQ gets scheduled in time when a target VM receives an I/O request.

2.1 Existing Approaches

Now we discuss several existing approaches addressing the problem of CPU sharing impacting I/O performance of VMs and discuss why they do not work well.

Reducing CPU time-slice. One intuitive approach to solve the scheduling latency problem is to uniformly reduce the VM scheduling time-slice [15]. In proportional-share scheduling, the worst-case scheduling delay of each VM is *(Number of sharing VMs -1)* $\times$ *time-slice*. A small scheduling time-slice enables VMs to get scheduled more frequently thus improving the I/O throughput of VMs. However, the short time-slice results in more frequent context switches which may hurt the performance of memory-intensive or CPU-bound applications. We conduct a simple experiment to demonstrate this problem.

In our experiment, 4 single vCPU VMs share one physical CPU. One VM hosts a TCP server, the client is running in another physical machine in the same LAN. Iperf [5] is used to measure the server's TCP throughput. We vary the scheduling time-slice from 0.1ms to 30ms, which is the default time-slice of Xen. From Figure 2(a) we can find that, smaller time-slice leads to higher TCP throughput. Especially, with a 0.1ms time-slice, the average TCP throughput is up to 900 Mbps which is close to the bandwidth of 1Gbps network card used in our experiment. However, the performance of memory/CPU bound applications degrades under smaller time-slice as shown in Figure 2(b). Here, we run STREAM [6] benchmark in one of the 4 VMs[1]. So, simply reducing the CPU time-slice cannot simultaneously benefit both I/O-intensive applications and CPU-intensive applications. Hence this approach is not suitable for cloud environments where mixed workloads are common.

Sending I/O interrupts to active vCPU To reduce the IRQ processing delay and improve I/O throughput for SMP-VMs, a recent approach called vBalance [10] sends I/O interrupts to the active vCPU of the target VM. In this way, I/O interrupts can be processed in a more timely fashion and I/O throughput may be improved. However, there are still several issues with this method. As discussed before, an SMP-VM may have increased chances to get scheduled because of the multiple vCPUs assigned to it. But there is no fundamental guarantee that the SMP-VM have at least one vCPU running at any time. If none of the vCPUs is running, an I/O interrupt still cannot be processed in time. Besides, even if the I/O interrupt is sent to an active vCPU successfully, the I/O cannot be finished if the vCPU executing the I/O application is not running simultaneously. This specifically impacts TCP, where the application vCPU may be in the *runqueue* holding the ownership of the lock structure, hence the kernel-level TCP processing cannot generate an ACK in time for incoming TCP packets. We suspect this is the main reason [10] only reports 400Mbps TCP throughput in a 1Gbps LAN environment.

Differentiated VM scheduling Tuning VM scheduling policy is another method to speed up I/O processing. vSlicer [30] schedules each latency-sensitive VM (LSVM) more frequently with a smaller micro time-slice, which enables more timely processing of I/O events by LSVMs. There are two caveats of this approach. First, we need to know which VMs are LSVMs running latency-sensitive applications in advance and adjust the VM scheduler configuration accordingly. Second, vSlicer reduces the scheduling delay but does not completely eliminate it, as discussed earlier. It, therefore, does not improve the TCP/UDP throughput significantly, although it does reduce application-perceived I/O latency.

[1]We conducted a similar experiment in [30]. But here we set even smaller time-slice (0.1ms) and contrast TCP and memory throughput under such a time-slice.

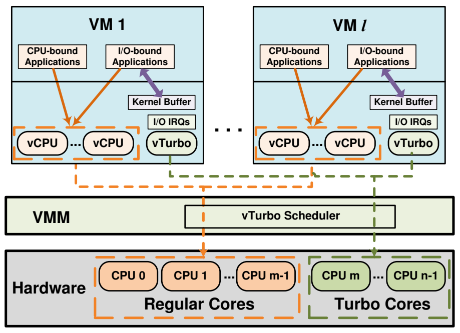


Figure 3: Architecture of vTurbo

3 Design

The discussion in the previous section suggests that if we use a very small value as the CPU time-slice, I/O performance of CPU-sharing VMs can be significantly improved. However, we also showed that such an approach may hurt the performance of CPU-bound VMs, for which larger time-slice is desirable. To address this dilemma, we leverage one *key degree of freedom* that has not been exploited hitherto: The CPU time-slice for each core may *not* be the same for a multi-cores system.

Thus, in our approach called vTurbo, we designate one (or more) core(s) in the system as what we call a *turbo core*, which is just any regular physical core, except that we set a very small (*e.g.*, 0.1ms) CPU time-slice for it. We expose the turbo core to each VM *in addition to* the regular cores, and allow the guest OS to schedule I/O-bound threads (*e.g.*, IRQ handling) in the turbo core thus speeding up I/O processing significantly. The guest OS still schedules CPU-bound workloads on cores with the regular time-slice. As such vTurbo achieves I/O processing speedup without impacting CPU-bound workloads.

In effect, vTurbo focuses on re-factoring the interface between the hypervisor and guest OS, with the new abstraction of turbo core. This approach is completely transparent to applications running in VMs, a key advantage of practicality. Another benefit of vTurbo is that it does not require classification of VMs into I/O- or CPU-intensive VMs, as required by some solutions such as vSlicer [30]. Such classification is difficult as most VMs in practice run a combination of I/O and CPU workloads. Of course, the guest OS now needs to identify I/O-bound threads such as IRQ processing and schedule them on the turbo core. But that is not hard as there are only a handful of such threads. Our approach also guarantees CPU *fairness* among all VMs. Any VM using the turbo core will essentially not obtain any "extra" CPU beyond its fair share—an important property in multi-tenancy clouds.

The architecture of vTurbo requires changes to both the hypervisor and the guest OS. At the hypervisor level, the VM scheduler needs to accommodate the new turbo core abstraction. At the guest kernel-level, we need to modify the VM process scheduler to pin certain threads to the turbo core in addition to a few changes to the TCP protocol stack. In the following subsections, we discuss these in more detail.

3.1 Modifications to Hypervisor

We mainly need to modify the VM scheduler in the hypervisor to support the turbo core abstraction. Upon host initialization, we designate a set of cores in the host as turbo cores. The number of turbo cores is configurable, and our current version statically assign turbo cores based on user configuration. However, we believe that our system can be improved by having a dynamic method to assign turbo cores based on the available machine capacity (*i.e.*, total number of cores), number of VMs, demand for the turbo core, and overall I/O intensity (*e.g.*, a host with multiple active NICs or 10GB/s NICs may require more turbo cores). One can also dynamically change the number of turbo cores via administrative tools (such as *xm* tools in Xen). While the current implementation of vTurbo randomly selects the turbo cores, we can incorporate parameters such as cache affinity to further improve their performance.

In vTurbo, each VM is assigned a *turbo vCPU* in addition to its regular vCPUs. The turbo vCPU is assigned to one of the turbo cores in the host. This step is performed during VM initialization. For instance, if a user launches an SMP-VM configured with 2 vCPUs, the VM will have 3 vCPUs after initialization. Among these, the 0^{th} vCPU is the turbo vCPU, whereas the 1^{st} and 2^{nd} vCPUs are regular vCPUs.

Based on our empirical study (discussed in Section 2), we set 0.1ms as the CPU scheduling time-slice for turbo cores (as it enables the VM to reach up to 900Mbps TCP throughput for a 4 CPU-sharing VMs scenario). Since only interrupt processing runs on turbo cores, frequent context switches caused by the small turbo core time-slice does not affect the performance of interrupt processing much because of the very short duration of the processing. According to our measurements (Figure 4), when 4 VMs each running an iperf server share one turbo core, the order of magnitude of cache miss per second on

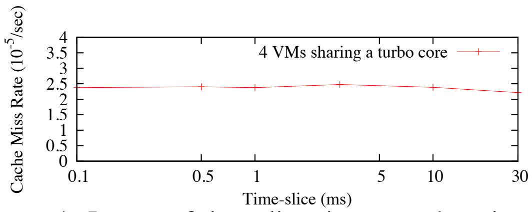


Figure 4: Impact of time-slice size on cache misses on turbo cores.

the turbo core is only 10^{-5}, which is negligible. The CPU time-slice for regular cores is set be a much larger value — 30ms in the current implementation which is the default time-slice of Xen. The vTurbo VM scheduler uses per-core scheduling timer to trigger scheduling code to select the next vCPU from the *runqueue*. We achieve CPU time-slice differentiation by setting these timers to 0.1ms for turbo cores and 30ms for regular cores.

Once the vCPUs are assigned to turbo cores and regular cores, our next concern is to correctly handle vCPU migration in the presence of turbo cores. vCPU migration allows to balance the CPU load among the available cores in the system. However, if we let the vCPUs to migrate freely among available cores, there is a possibility that a regular vCPU be migrated to a turbo core making undesirable effects. To solve this, we restrict migration of regular vCPUs to only among all regular cores and migration of turbo vCPUs only among turbo cores. We do not allow a turbo vCPU to migrate to a regular core or vice versa. This is done by changing each vCPU's affinity to the corresponding set of cores. Hence vTurbo scheduler not only determines the appropriate mapping between vCPUs and physical cores, but also ensures fair CPU sharing among all VMs.

VM scheduling policy Since we intend to use the turbo core only for I/O activity, we cannot treat it as a regular core and apply the existing scheduling policy to guarantee fair sharing among VMs. The challenge is to determine the CPU share of VMs for turbo and regular cores in the presence of heterogeneous workloads (*i.e.*, when a VM is CPU intensive, I/O-intensive, or both).

Current schedulers (*e.g.*, Xen's) use simple credit-based scheduling algorithm for achieving global load balancing and work-conservation. For instance, in Xen's credit scheduler, a VM is assigned some amount of credits periodically based on the priority of the VM. As the vCPUs belonging to a particular VM run on physical CPUs, credits are deducted from that VM. When the scheduler needs to make a decision, it uses the amount of available credits for each VM to decide which vCPU will run on the physical CPU. To accommodate turbo cores in our system, we mainly need to modify the credit assignment portion of the credit scheduling algorithm to account for the turbo vCPU execution time.

Specifically, assume l VMs are sharing an n-core host with m regular cores and n-m turbo cores. Let rd_i denote the percentage demand for regular cores (CPU-bound component) and let td_i denote the percentage demand for turbo cores (I/O-bound component) for VM_i. We assume the demand for regular core and turbo core in two consecutive scheduling periods does not change much (if it does, we account for and adjust it in future rounds). So both rd_i and td_i are calculated based on the consumed CPU cycles by the VM in the previous scheduling period. Since our scheduler is work-conserving, the division of the total capacity among the regular and turbo cores is determined by the following:

$$C_{tot}^{R} = \sum_{i=1}^{l} rd_i \quad and \quad C_{tot}^{T} = \sum_{i=1}^{l} td_i$$

The total capacity demand of the system is:

$$C_{tot} = C_{tot}^{R} + C_{tot}^{T}$$

The fraction of CPU allocated for a VM out of this total capacity is determined by its assigned weight wt_i. Hence each VM's fair share (FS_i) of CPU is given by:

$$FS_i = (C_{tot} \times wt_i)/(\sum_{j=1}^{l} wt_j)$$

In vTurbo, we first allocate turbo core capacity fairly among VMs, as all of the VMs' IRQ processing is performed by the turbo vCPUs and starvation of turbo vCPUs (even for CPU-bound VMs) will result in application performance hit. So VM_i's fair share of the turbo core (FS_i^T) is calculated as:

$$FS_i^T = (C_{tot}^{T} \times wt_i)/(\sum_{j=1}^{l} wt_j)$$

Once VM_i's turbo core share is determined, we allocate the rest of its CPU share from the regular cores. The fraction of the allocation is given by:

$$FS_i^R = FS_i - \hat{FS}_i^T$$

where $\hat{FS}_i^T$ denotes the *actual* usage of the turbo core by VM_i in the previous scheduling period. We use FS_i^T and FS_i^R to determine the proportion of credits given to VMs out of total credits in the turbo core pool and regular core pool, for the next scheduling period. Table 1 shows the CPU allocation results from experiments with our prototype, where two VMs—with equal weight—share one regular core and one turbo core, under various workload demands. Columns 2 and 3 of the table indicate the CPU demand of each VM (*i.e.*, CPU utilization if they were run without CPU sharing); Columns 4 and 5 indicate measured consumption in the previous scheduling period; Columns 6 and 7 indicate the allocated shares of regular and turbo cores based on our policy; and Columns 8 and 9 show the *measured* consumption of regular ($\hat{FS}_i^R$) and turbo ($\hat{FS}_i^T$) core capacity in the next scheduling period. The results confirm that our policy allocates CPU with proportional fairness.

	Demand		Measured		Allocated		Consumed	
	Reg.	Turbo	rd_i	td_i	FS_i^R	FS_i^T	$\hat{FS}_i^R$	$\hat{FS}_i^T$
VM1	100	0	50	0	50	0	50	0
VM2	100	0	50	0	50	0	50	0
VM1	100	0	50	0	100	0	100	0
VM2	100	100	50	100	0	100	0	100
VM1	100	100	50	50	50	50	50	50
VM2	100	100	50	50	50	50	50	50
VM1	100	15	50	15	70	35	70	15
VM2	100	55	50	55	30	35	30	55

Table 1: VMs' CPU demand and allocated CPU shares under different scenarios

3.2 Modifications to Guest OS

Process scheduler As noted before, if CPU-bound workload were scheduled on the turbo cores, its performance would degrade due to frequent context switches. Since process scheduling inside the VM is transparent to the hypervisor's VM scheduler, we should make the guest OS's process scheduler aware of the turbo core to prevent user processes and non-I/O-related kernel threads from being scheduled on the turbo core. This can be achieved by setting scheduler affinity rule which sets the affinity of the non-I/O related threads to regular vCPUs. In Linux, this can be easily done by a scheduling mechanism known as Linux CPU isolation [3] (by setting a kernel parameter).

I/O buffers in guest OS With the above change, we can reduce IRQ processing delay to extremely small values. However, low IRQ processing delay by itself does not automatically translate into high I/O throughput, because of a critical *locking behavior* between the kernel and application threads as we explain below. The network receive path in typical OSes (*e.g.*, Linux) consists of two main steps: (1) Processing IRQ in kernel and buffering data in kernel buffer; (2) Application reading the data from kernel buffer and clearing it. Since the CPU time-slice of regular cores is still 30ms in vTurbo, the CPU access delay on the regular core will make the kernel buffer full very quickly and stop the IRQ threads from buffering more data, which would lead to poor I/O performance.

To address this problem and to keep the turbo vCPU busy processing IRQs, we need to tune the kernel buffer to store more received data while the application running on regular vCPU is blocked. As an example when 4 single-vCPU (excluding turbo vCPU) VMs are sharing one regular core, the CPU access delay is up to 90ms (*(4 - 1) × 30ms*). To keep the IRQ threads on turbo vCPU busy, all data received during this period need to be buffered. So if the bandwidth of NIC is B_N, the minimum kernel buffer required (B_{min}) is: $B_{min} = B_N \times Scheduling\_Delay$ (*i.e.*, the required kernel buffer is proportional to the number of VMs sharing the same CPU core). In fact, the real kernel buffer we need is almost always much larger than B_{min}. For example, in our experimental environment with 1Gbps NICs, if 4 VMs share one CPU, the kernel buffer for UDP should be around 11.25MB. However, we did not obtain high throughput (more than 900Mbps) until we set the UDP kernel buffer (*net.core.rmem_max*) to about 40MB.

Algorithm 1 Generating ACK for Backlog Queue

```
1: $rcv.nxt$ is the seq. number of expected packet for receive queue
2: $bl.nxt$ is the seq. number of expected packet for backlog queue
3: $seq$ is the seq. number of received packet
4: if $backlog\_queue$ is empty then
5:     if $rcv.nxt \geq bl.nxt$ then
6:         /* initial status or packets in backlog queue are all acked by
           process context */
7:         $bl.nxt = rcv.nxt$;
8:         $bl.online = 1$; /* enable ACK generation */
9: else
10:    if $bl.online == 0$ and $bl.nxt \leq rcv\_nxt$ then
11:        /* packets in backlog queue are acked by process context */
12:        $bl.online = 1$; /* enable ACK generation */
13:        $bl.nxt = rcv.nxt$;
14: if $bl.online == 1$ then
15:    if $bl.nxt == seq$ then
16:        /* packet to be added to backlog queue is in order */
17:        update(bl.nxt);
18:    else
19:        /* stop ACK generation due to out-of-order packet */
20:        $bl.online = 0$;
21: if $add\_backlog()$ is successful and $bl.online == 1$ then
22:    $tcp\_ack\_backlog()$; /* generate and send ACK */
```

Modifications to VM's TCP stack While simply setting the guest kernel buffer to a high value ensured good UDP performance, it did not improve TCP throughput at all. Upon a deeper investigation, we found the following problem: In TCP, when a data segment is received, the receiver generates an ACK to inform the reception of the segment. The sender uses this ACK to confirm the reception of data as well as for congestion control. Now, using the turbo core, we eliminate the long delay for processing incoming data segments. With our additional I/O buffering enabled, the IRQ context now buffers all these data packets. However, the locking behavior in the VM's TCP stack still prevents the ACK generation in a timely manner, hence reducing TCP throughput significantly.

Specifically, when the user process is calling function *recv()*, it locks the socket to prevent the IRQ threads from modifying the socket structure while it is reading from the socket buffers. If a new data segment arrives during this period, the IRQ process will queue it in the *backlog* queue without generating an ACK. When the receiving process engages in a tight receiving loop, the socket gets locked frequently by the process context. Moreover, the process can get scheduled out of the regular core while it is holding the lock. When this happens, ACKs will not be generated for a long period (until the process gets scheduled and releases the lock), even though the turbo core can accept and buffer TCP segments from the network. As a result, the sender will throttle down the sending rate leading to sub-par TCP throughput.

We make a simple modification to the VM's TCP stack to enable ACK generation from the IRQ context running in the turbo core, even when the socket structure is locked by the user process. The high-level steps performed by our modification are shown in Algorithm 1 which runs in the *softIRQ* context just before queuing the packet in the TCP backlog queue. Here, when the IRQ thread discovers that the socket is locked by the user process, it checks whether the new data segment is in-order. If so, an ACK is generated for the data packet, which will then be marked as acknowledged and queued. Note that we are not modifying the socket structure as it is currently owned by the process context. This is somewhat similar to vSnoop [17], although vSnoop is implemented purely in the driver domain whereas the ACK generation here is from within the guest VM. Thus we have access to VM's TCP information and can afford much larger buffers (compared to the limited ring buffer space in vSnoop). If a flow encounters an out-of-order packet, we disable this ACK generation until the missing segments are recovered by the usual slow path of TCP processing. This small modification helps achieve TCP throughput close to the line rate.

4 Implementation

We have implemented a prototype of vTurbo based on Xen 4.1.2. vTurbo only requires small modifications to the VM scheduler in hypervisor (about 400 lines of code) and guest OS kernel (less than 200 lines of code).

Hypervisor To differentiate between regular cores and turbo cores, we added a field to the per-core data structure *schedule_data*, to indicate the CPU time-slice for the specific core–30ms for regular cores and 0.1ms for turbo cores. Our implementation allows the flexibility of changing these values dynamically via *xm* tools.

Our vTurbo scheduler inherits most of its functionality from Xen's credit scheduler which provides the proportional fairness and work-conserving properties. We added and modified functionality of the main scheduler code of the credit scheduler to accommodate turbo cores and turbo vCPUs. Specifically, we modified function *csched_schedule()*, which is responsible for selecting vCPUs from the *runqueue* to run on physical cores and setting the scheduling timer of turbo cores to 0.1ms.

We assign each VM a turbo vCPU by modifying the VM's configuration so that an extra vCPU is added during the configuration parsing step of VM initialization performed by the Xen tools. Also during this step, the turbo vCPUs are pinned to the set of turbo cores and regular vCPUs are pinned to the regular cores by modifying the loaded VM's configuration. By doing this, we do not have to modify the scheduler code to prevent undesirable vCPU migrations (discussed in Section 3), because the credit scheduler will adhere to the CPU affinity rules set in the configuration.

Algorithm 2 vTurbo accounting algorithm

```
Require: $num\_tcore \geq 1$
Require: $num\_rcore \geq 1$
Require: $num\_vm \geq 1$

  $Regular\_accounting$ triggered every 30ms:
  $tcore\_usage = get\_rcore\_usage()$; /* $C^R_{tot}$ */
  $rcore\_usage = get\_tcore\_usage()$; /* $C^T_{tot}$ */
  for $vm$ in $vm\_list$ do
      $vm.credits = vm.weight \times$
      $(tcore\_usage + rcore\_usage)/vm\_weight\_sum$;
      $vm.tcredits = get\_turbo\_core\_usage(vm)$;
      $vm.rcredits = vm.credits - vm.tcredits$;
      $ratio = 300$; /* = 30/0.1 */
      $vm.vturbo\_slice = vm\_vcredits/ratio$;
      $update\_rcredits(vm.rcredits)$;

  $vTurbo\_accounting$ triggered every 0.1ms:
  for $vm$ in $vm\_list$ do
      $update\_tcredits(vm.vturbo\_slice)$;
```

CPU accounting is conducted by function *csched_acct()* in the credit scheduler. We extended this function by implementing two accounting routines for regular and turbo vCPUs individually as shown in Algorithm 2. They run at different frequencies in accordance with CPU scheduling frequencies (e.g., 30ms for regular vCPUs and 0.1ms for turbo vCPUs), because updating credits faster or slower than the scheduling frequency would cause inaccurate state of vCPUs in terms of *OVER* and *UNDER* priorities in Xen. The vTurbo accounting routines are simple, incurring very low overhead considering the high frequency of their execution. Functions *get_rcore_usage()* and *get_tcore_usage()* retrieve the consumed clock cycles by regular vCPUs and turbo vCPUs of all VMs respectively; while functions *update_rcredits()* and *update_tcredits()* set the calculated credits for regular cores and turbo cores for the next scheduling period. Function *get_turbo_core_usage()* retrieves the the clock cycle usage by the turbo vCPU of a given VM. We do not change method *burning_credits()* in the credit scheduler, which deducts credits from the VMs based on their running time on the cores. Instead we implement a new method for vTurbo credit deduction.

Guest OS Our modification to the TCP stack, to generate early ACKs for packets buffered in backlog queue, is mainly in function *tcp_v4_rcv()*. There are 3 kernel buffers to buffer received TCP packets: (1) receive queue, (2) prequeue, and (3) backlog queue. When a socket is not locked, received packets are buffered in receive queue. However, if the application process locks the socket while fetching data from the kernel, packets received during that period will be buffered in backlog queue. We modified the backlog queuing path of function *tcp_v4_rcv()* to verify a received packet is "expected" and if so, call function *tcp_ack()* to generate an ACK for the received packet. Since very few packets (less than 0.1%) go to prequeue in CPU sharing VMs, we disable prequeue in vTurbo to simplify our implementation.

5 Evaluation

We first evaluate the effectiveness of vTurbo for different types of I/O operations via a series of micro-benchmarks. We then use NFS, SCP, and Apache Olio [2] to evaluate the application-level performance improvement by vTurbo.

Experimental setup Our testbed consists of servers with quad-core 3.2GHz Intel Xeon CPUs and 16GB of RAM. They are connected via Gigabit Ethernet, except for the experiments with 10Gbps Ethernet. These servers run Xen 4.1.2 as hypervisor and Linux 3.2 in both domain0 and guest VMs. We pin domain0 to one of the cores in all our experiments.

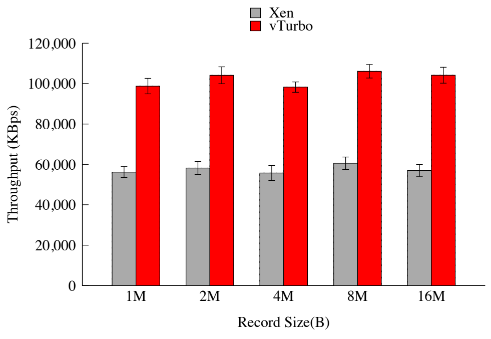
5.1 Micro-Benchmark Results

In this section we evaluate the performance of vTurbo for various types of I/O. We use *lookbusy* [7] to keep the CPU utilization at determined levels during experiments.

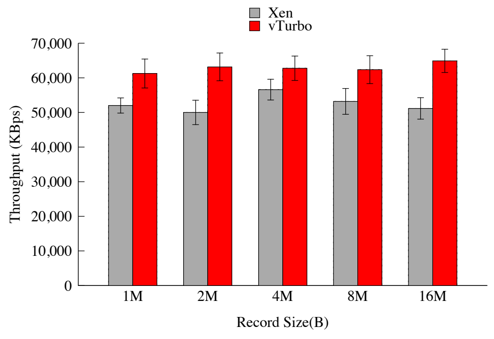
File read and write We use IOzone benchmark [4] to read/write a 1GB file from/to disk and measure the read/write throughput. Figure 5 shows the read and write throughput—in comparison with the vanilla Xen–when we vary the record size from 1MB to 16MB.

From Figure 5(a) we see that the disk write throughput is improved significantly (by 75% to 82%); whereas the disk read throughput (Figure 5(b)) sees less improvement (only up to 26%). The main reason is that, when the process performs a write, the data is immediately written to the file system cache and the *write()* call returns. So the process can keep writing while the regular vCPU is scheduled. The dirty pages of the disk cache are flushed to the disk by a kernel thread executed by the turbo vCPU. Therefore with vTurbo, disk write throughput is greatly improved. However, when the process performs a read for a fresh block from the disk, it gets blocked until the actual data blocks are read from the disk. Meanwhile the hypervisor may schedule other vCPUs on the regular core. The turbo vCPU will be able to handle the disk read completion interrupt and place the data in the process' buffer while the regular vCPU is scheduled out. But the process will not be able to make further read requests until it is scheduled again. Hence in this case, vTurbo achieves less throughput improvement than in the case of disk write.

UDP throughput To measure the benefit of vTurbo to network I/O we first measure the UDP throughput improvement achieved by vTurbo. In these experiments, we use iperf to send a stream of UDP packets for 10 seconds to a VM sharing a core with 2, 3, or 4 other VMs. The average throughput (averaged over 10 runs) observed at the

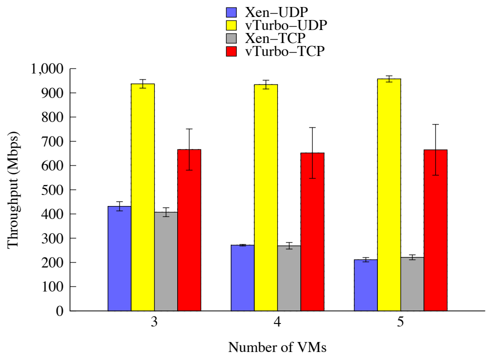


(a) Write

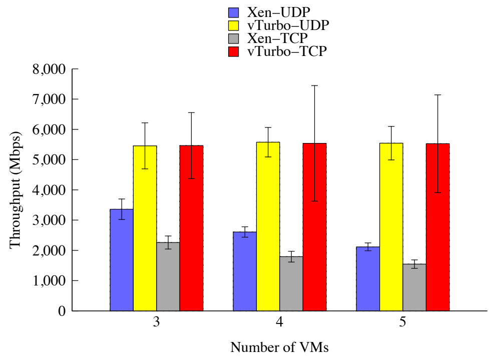


(b) Read

Figure 5: File read/write throughput.



(a) 1Gbps Network



(b) 10Gbps Network

Figure 6: TCP and UDP throughput.

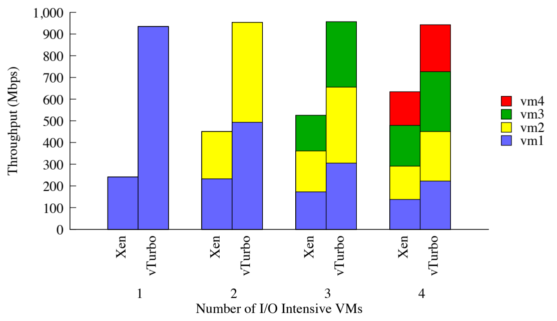


Figure 7: UDP throughput: multiple I/O-intensive VMs.

VM on vanilla Xen and vTurbo is shown in Figure 6(a) by blue and yellow bars, respectively. With vanilla Xen, the UDP throughput starts to decrease when the number of VMs sharing the core increases. This is because, when UDP packets arrive at domain0, the target VM may not be scheduled and the packets have to be buffered in domain0. But the space in domain0 is limited and hence once this buffer fills up, packets will be dropped causing the throughput to go down. With vTurbo, the target VM's network IRQ processing threads get scheduled frequently and hence the buffer in domain0 can be drained frequently. This leads to much less packet drops thereby achieving close-to full network bandwidth (1Gbps).

Next, we evaluate the impact of sharing the turbo core among multiple I/O-intensive VMs. We reuse the setup in the previous experiment. But instead of 1 VM receiving a UDP packet stream, we increase the number of VMs receiving UDP streams from 1 to 4. Figure 7 shows the aggregate throughput achieved as well as the throughput seen by individual VMs. In both vanilla Xen and Xen with vTurbo configurations, we see that the I/O bandwidth is fairly shared among VMs. However, vTurbo achieves (close to) wire speed and outperforms vanilla Xen irrespective of the number of I/O-intensive VMs.

TCP throughput We use a setup similar to the UDP experiments to measure the TCP throughput improvement achieved by vTurbo. In this experiment, we send a 200MB file using iperf to a VM from another server and we vary the number of VMs sharing the same core with the receiving VM. Figure 6(a) shows the TCP throughput on vanilla Xen and Xen with vTurbo by grey and red bars, respectively. Recall that with vTurbo, the TCP stack is modified to generate ACKs when the regular vCPU is holding the socket ownership and scheduled out. As the figure shows, vTurbo improves TCP throughput significantly (by 63% - 200%). However the TCP throughput achieved by vTurbo still does not reach the full available network bandwidth. The reason is, even with our modification, if a packet loss happens, we have to resort to the (usual) slow code path where packet loss recovery is subject to regular vCPU scheduling delay,

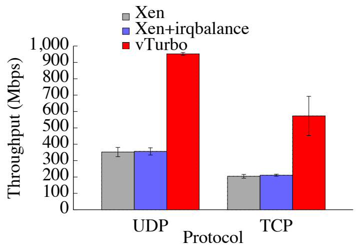
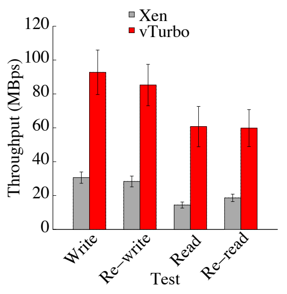


Figure 8: UDP and TCP throughput for VSMP VMs.

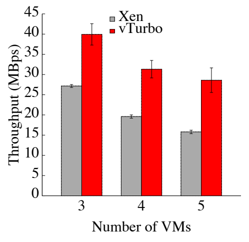
which negatively affects the TCP throughput.

10Gbps Ethernet To evaluate the benefit of vTurbo with 10Gbps Ethernet, we repeat the UDP and TCP experiments. In our setup, two physical servers are connected via 10Gbps Ethernet. In the UDP experiment, we use netperf [9] to send a 10-second UDP stream to the target VM sharing a core with 2 to 4 other VMs. In the TCP experiment, we send a 500MB file using iperf from one physical server to a VM running in the other server, varying the number of VMs sharing the same core with the receiving VM. The results in Figure 6(b) indicate that, in a 10Gbps network, vTurbo achieves a pattern of improvement for both UDP and TCP throughput similar to that in the 1Gbps network. However, since the regular core is shared by multiple VMs, the application does not get enough CPU cycles to copy the buffered data from kernel space to user space, hence we can not achieve line speed.

Benefit of vTurbo to VSMP VMs To show the benefit of vTurbo to SMP VMs, we use iperf to send TCP and UDP traffic (in different runs) to a VM which is assigned 2 vCPUs. In this experiment, we run 4 VMs each with 2 vCPUs. These vCPUs are restricted to run in the first 2 cores of the quad-core processor, but are allowed to migrate between the two cores. Similar to previous experiments, we pin domain0 to the 3rd core and, for vTurbo, we use the 4th core as the turbo core. In the vanilla Xen configuration, we first disable *irqbalance* in VM and allow the interrupts to be directed only to vCPU0 of the VM. Next we enable *irqbalance* so that the interrupts can be balanced between the two vCPUs. In the vTurbo configuration, interrupts are routed to the turbo vCPU. Figure 8 shows the TCP and UDP throughput when transferring 200MB of data to the VSMP VM. vTurbo vastly outperforms both *irqbalance*-on and *irqbalance*-off configurations. However, the TCP throughput is lower than that under the “4 single-vCPU VMs” configuration (for both vanilla Xen and vTurbo configurations – see Figure 6(a)) . We conjecture that this is due to the vCPU migrations between the two physical cores and the iperf receiver process migrations between the two vCPUs of the VSMP VM.



(a) NFS read/write throughput.



(b) SCP throughput.

Figure 9: SCP and NFS throughput.

5.2 Application-Level Results

NFS server throughput NFS uses TCP to transport commands and data blocks between the NFS client and server. We use the NFS tests in IOzone to evaluate the benefit of vTurbo to the NFS server. We export a directory of a VM using NFS and run IOzone in another server which mounts this exported directory. We pin the NFS server VM’s vCPU to a single core shared by three other VMs with 30% CPU utilization. Figure 9(a) shows the file read and write throughput (file size: 1GB). vTurbo significantly outperforms vanilla Xen for all types of operations. The results for “Read” and “Re-read” operations are especially interesting (and somewhat surprising). Recall that, for file read/write micro-benchmarks, vTurbo does *not* improve disk read throughput much. Yet we observe significant improvement in NFS read and re-read throughput. After some investigation, we figure out the reasons for the improvements here: First, NFS utilizes *pre-fetching* for sequential read operations where multiple read operations are issued in advance. Second, Linux NFS implementation uses *in-kernel* data transfer from files to sockets. As such, the server process is able to process many read requests while the regular vCPU is scheduled and to delegate the actual file block transfer operations to the kernel threads run by the turbo vCPU, hence achieving much higher throughput.

Secure copy (SCP) throughput SCP involves both CPU activity (for encryption and decryption of data) and I/O activity. We copy a 1GB file using SCP from a client to a VM sharing a core with 2, 3, or 4 other VMs. In this experiment the *sshd* process which is receiving the file is scheduled at the regular vCPU while *both* TCP processing and disk I/O handling threads are scheduled at the turbo vCPU. Figure 9(b) shows that vTurbo improves SCP throughput by 53% to 66%.

Apache Olio To assess the benefit of vTurbo to a cloud application, we use Apache Olio, an event calender developed using Web 2.0 technologies. The Apache Olio benchmark consists of 3 components: (1) a web server to process user requests, (2) a MySQL database server to

Operation	Single Instance		Two Simultaneous Instances			
			Instance 1		Instance 2	
	Count Vanilla Xen	Count vTurbo	Count Vanilla Xen	Count vTurbo	Count Vanilla Xen	Count vTurbo
HomePage	4028	5602	3918	5334	3839	5311
Login	1629	2190	1524	2121	1540	2109
TagSearch	5183	7198	4888	6822	4892	6778
EventDetail	3856	5274	3701	5075	3630	5013
PersonDetail	405	562	379	550	381	508
AddPerson	127	178	131	177	120	167
AddEvent	300	402	280	416	279	413
Total	15528	21406	14821	20495	14681	20299
Rate(ops/sec)	51.8	71.3	49.4	68.3	48.9	67.7
Improvement (%)	-	**37.6%**	-	**38.2%**	-	**38.4%**

Table 2: Results from Apache Olio experiment (single- and two-instance)

store user profiles and event information, and (3) an NFS server to store images and documents specific to events. We use the PHP version of the benchmark.

In our setup, we host the 3 Olio components in 3 different VMs each in a separate physical host. In each host we pin the Olio VM's vCPU to a single core, which is shared by 3 other VMs having 20% of CPU load. We stress the Olio service with 400 client threads generating requests using the Faban client simulator for 6 minutes.

In Table 2, the "Single Instance" (2^{nd} and 3^{rd}) columns show the breakdown of total operations (averaged over 3 runs) performed by Olio on vanilla Xen and on vTurbo, respectively. vTurbo achieves higher operation counts than vanilla Xen for all types of operations during the same period. This is because vTurbo improves communication performance among the three Olio components as well as file write performance of MySQL and NFS servers. With vTurbo the overall throughput of the Olio service is improved from 51.8 ops/second to 71.3 ops/second – a 37.6% improvement.

Next, we evaluate the performance of *two* simultaneous instances of Olio, with the same set of components hosted by the same physical servers. In this experiment, of the 4 CPU cores of each server, we dedicate one core to domain0 and one core as the turbo core shared by all VMs. In our replicated Olio configuration, we pin the two copies of each Olio component to the 2 remaining cores respectively, with each core shared by 3 other VMs. Columns 4, 5, 6, 7 of Table 2 show the breakdown of total operations performed by the two Olio instances, which are started at the same time and run for the same 6-minute period. Compared with the "Single Instance" results, most rows see a slight reduction of operation throughput for both vanilla Xen and vTurbo configurations. We believe this is due to the sharing of resources such as the disk and network. However, we observe that with vTurbo, the overall Olio throughput is increased by 38.2% and 38.4% for instances 1 and 2, respectively.

6 Related Work

We have discussed some of the recent and most related efforts in optimizing I/O processing for virtualized systems in Section 1 and Section 2. In this section, we discuss other related work in the same problem space. These efforts can be categorized into two categories: I/O path tuning and VM scheduling optimization.

I/O path improvements vSnoop [17] and vFlood [12] are two related efforts to improve TCP throughput of VMs. vSnoop offloads ACK generation from a VM to its driver domain to hide the VM scheduling delay from the TCP sender. We adopt a similar idea in vTurbo*inside* the guest OS to generate ACKs while the receiving process on the regular core has locked the socket. vSnoop only benefits TCP receiving, it does not benefit UDP or disk I/O. Moreover, due to the limited shared buffer space in the driver domain, vSnoop can only accelerate *small* TCP flows. On the other hand vTurbo can improve throughput of TCP/UDP receive—regardless of flow size—and disk write. It can also benefit disk read if data pre-fetching is used by applications (as shown by the NFS throughput results in Section 5.2). One can consider vTurbo as an alternative to vSnoop with extra features. vFlood offloads TCP congestion control to driver domain to hide VM scheduling delay from receiver thus improving the performance of TCP send. vFlood has the same problem as vSnoop: It only works for small TCP flows. IsoStack [27] offloads the entire TCP processing engine to a dedicated core. The main advantage of IsoStack is that, it can reduce cache misses and reduce synchronized accesses to shared state of the TCP stack by multiple cores (*e.g.*, socket structures). vTurbo in spirit offloads IRQ processing (only) to a separate core, with the goal of mitigating the impact of VMs' regular core access latency. In [24, 22, 23], Menon *et al.*propose optimizations for device virtualization using techniques such as checksum offload, segmentation offload, packet coalescing, scatter/gather I/O, and offloading device driver functionality. SR-IOV [11] devices and IOMMUs such as Intel VT-d [14] enable the hypervisor to directly assign devices to guests. This allows the guest to directly interact with the device eliminating the virtulization overhead. However, even in this case, scheduling delays still impact the interrupt processing delay. We believe that vTurbo is complementary to both SR-IOV and VT-d, since it enables the processing of the interrupt and data associated with the interrupt as soon as the interrupt is delivered to the VM.

VM scheduling optimization Adapting the scheduling policy in the hypervisor-level VM scheduler can also improve I/O performance perceived by applications running in VMs. vSlicer [30] reduces CPU scheduling delay and hence the application-perceived latency—to a certain degree by setting smaller time-slice for latency-sensitive VMs. Still, that time-slice is not small enough to improve TCP/UDP throughput in LAN/datacenter environments. With vTurbo, IRQ processing delay is reduced to sub-millisecond. And the concurrent I/O processing on regular and turbo cores brings significant I/O throughput improvement. We note that vTurbo and vSlicer can be *integrated* to achieve both low latency and high throughput for VM I/O. A soft-realtime VM scheduler is proposed in [21] to reduce response time of I/O requests thus improving the performance of soft-realtime applications such as media servers. But its preemption-based scheduling policy may violate CPU share fairness when a VM is performing heavy I/O activities. MRG [16] is a VM scheduler to improve I/O performance for MapReduce jobs. This scheduler facilities MapReduce job fairness by introducing a two-level group credit-based scheduling policy. Through batching of I/O requests within a group the efficiency of map and reduce tasks is improved and superfluous context switches are eliminated. However MRG is a MapReduce-specific scheduler; and it works well only when the VMs and the driver domain share the same CPU core.

7 Conclusion

We have presented vTurbo, a system that aims at accelerating I/O processing for VMs sharing the same core in a multi-core host. More specifically, vTurbo significantly reduces IRQ processing latency by dedicating one or more turbo core(s) to IRQ processing for all hosted VMs. The time-slice of a turbo core is magnitudes smaller than that of a regular core hence achieving negligible IRQ processing latency. vTurbo involves a VM scheduling policy that enforces fair sharing of both regular and turbo cores among VMs. Our evaluation of a vTurbo prototype shows that it vastly improves network and disk I/O throughput and consequently application-level performance for hosted VMs.

8 Acknowledgments

We thank our shepherd, Jonathan Walpole and the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by NSF grants 0855141, 1054788, and 1219004. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

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

[3] CPU isolation extensions. `http://lwn.net/Articles/270623/`.

[4] IOzone Filesystem Benchmark. `http://www.iozone.org/`.

[5] The Iperf Benchmark. `http://www.noc.ucf.edu/Tools/Iperf/`.

[6] J. McCalpin. The STREAM benchmark. `http://www.cs.virginia.edu/stream/`.

[7] Lookbusy-a synthetic load generator. `http://www.devin.com/lookbusy/`.

[8] Microsoft Cloud Platform (Microsoft Azure). `http://www.windowsazure.com/`.

[9] The Netperf Benchmark. `http://www.netperf.org`.

[10] CHENG, L., AND WANG, C.-L. vbalance: Using interrupt load balance to improve i/o performance for smp virtual machines. In *ACM SoCC* (2012).

[11] DONG, Y., YU, Z., AND ROSE, G. SR-IOV networking in Xen: architecture, design and implementation. In *WIOV* (2008).

[12] GAMAGE, S., KANGARLOU, A., KOMPELLA, R. R., AND XU, D. Opportunistic flooding to improve TCP transmit performance in virtualized clouds. In *ACM SoCC* (2011).

[13] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: bare-metal performance for I/O virtualization. In *ACM ASPLOS* (2012).

[14] HIREMANE, R. Intel virtualization technology for directed I/O (Intel VT-d). *Technology@ Intel Magazine 4*, 10 (2007).

[15] HU, Y., LONG, X., ZHANG, J., HE, J., AND XIA, L. I/o scheduling model of virtual machine based on multi-core dynamical partitioning. In *ACM HPDC* (2010).

[16] KANG, H., CHEN, Y., WONG, J. L., SION, R., AND WU, J. Enhancement of Xen's scheduler for MapReduce workloads. In *ACM HPDC'11* (2011).

[17] KANGARLOU, A., GAMAGE, S., KOMPELLA, R. R., AND XU, D. vSnoop: Improving TCP throughput in virtualized environments via acknowledgement offload. In *ACM/IEEE SC* (2010).

[18] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: predictable low latency for data center applications. In *ACM SoCC* (2012).

[19] KESAVAN, M., GAVRILOVSKA, A., AND SCHWAN, K. Differential Virtual Time (DVT): Rethinking I/O service differentiation for virtual machines. In *ACM SoCC* (2010).

[20] LARSEN, S., SARANGAM, P., HUGGAHALLI, R., AND KULKARNI, S. Architectural breakdown of end-to-end latency in a tcp/ip network. *International Journal of Parallel Programming 37*, 6 (2009), 556–571.

[21] LEE, M., KRISHNAKUMAR, A. S., KRISHNAN, P., SINGH, N., AND YAJNIK, S. Supporting soft real-time tasks in the Xen hypervisor. In *ACM VEE* (2010).

[22] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing network virtualization in Xen. In *USENIX ATC* (2006).

[23] MENON, A., SCHUBERT, S., AND ZWAENEPOEL, W. Twin-Drivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers. In *ACM ASPLOS* (2009).

[24] MENON, A., AND ZWAENEPOEL, W. Optimizing TCP receive performance. In *USENIX ATC* (2008).

[25] PATNAIK, D., KRISHNAKUMAR, A., KRISHNAN, P., SINGH, N., AND YAJNIK, S. Performance implications of hosting enterprise telephony applications on virtualized multi-core platforms. Tech. rep., IPTComm, 2009.

[26] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving network connection locality on multicore systems. In *ACM EuroSys* (2012).

[27] SHALEV, L., SATRAN, J., BOROVIK, E., AND BEN-YEHUDA, M. IsoStack: Highly efficient network processing on dedicated cores. In *USENIX ATC* (2010).

[28] WALDSPURGER, C., AND ROSENBLUM, M. I/O virtulization. In *Communications of the ACM* (2012).

[29] WALDSPURGER, C. A. Memory resource management in VMware ESX server. In *USENIX OSDI* (2002).

[30] XU, C., GAMAGE, S., RAO, P. N., KANGARLOU, A., KOMPELLA, R. R., AND XU, D. vslicer: Latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *ACM HPDC* (2012).