vIC: Interrupt Coalescing for Virtual Machine Storage Device IO

Irfan Ahmad Ajay Gulati Ali Mashtizadeh
{irfan, agulati, ali}@vmware.com
VMware, Inc., Palo Alto, CA 94304

Abstract

Interrupt coalescing is a well known and proven technique for reducing CPU utilization when processing high IO rates in network and storage controllers. Virtualization introduces a layer of virtual hardware for the guest operating system, whose interrupt rate can be controlled by the hypervisor. Unfortunately, existing techniques based on high-resolution timers are not practical for virtual devices, due to their large overhead. In this paper, we present the design and implementation of a virtual interrupt coalescing (vIC) scheme for virtual SCSI hardware controllers in a hypervisor.

We use the number of *commands in flight* from the guest as well as the current *IO rate* to dynamically set the degree of interrupt coalescing. Compared to existing techniques in hardware, our work does not rely on high-resolution interrupt-delay timers and thus leads to a very efficient implementation in a hypervisor. Furthermore, our technique is generic and therefore applicable to all types of hardware storage IO controllers which, unlike networking, don't receive anonymous traffic. We also propose an optimization to reduce inter-processor interrupts (IPIs) resulting in better application performance during periods of high IO activity. Our implementation of virtual interrupt coalescing has been shipping with VMware ESX since 2009. We present our evaluation showing performance improvements in micro benchmarks of up to 18% and in TPC-C of up to 5%.

1 Introduction

The performance overhead of virtualization has decreased steadily in the last decade due to improved hardware support for hypervisors. This and other storage device optimizations have led to increasing deployments of IO intensive applications on virtualized hosts. Many important enterprise applications today exhibit high IO rates. For example, transaction processing loads can issue hundreds of very small IO operations in parallel resulting in tens of thousands of IOs per second (IOPS). Such high IOPS are now within reach of even more IT organizations with faster storage controllers, wider adoption of solid-state disks (SSDs) as front-end tier in storage arrays and increasing deployments of high performance consolidated storage devices using Storage Area Network (SAN) or Network-Attached Storage (NAS) protocols.

For high IO rates, the CPU overhead for handling all the interrupts can get very high and eventually lead to lack of CPU resources for the application itself [7, 14]. CPU overhead is even more of a problem in virtualization scenarios where we are trying to consolidate as many virtual machines into one physical box as possible. Freeing up CPU resources from one virtual machine (VM) will improve performance of other VMs on the same host. Traditionally, interrupt coalescing or moderation has been used in network and storage controller cards to limit the number of times that application execution is interrupted by the device to handle IO completions. Such coalescing techniques have to carefully balance an increase in IO latency with the improved execution efficiency due to fewer interrupts.

In hardware controllers, fine-grained timers are used in conjunction with interrupt coalescing to keep an upper bound on the latency of IO completion notifications. Such timers are inefficient to use in a hypervisor and one has to resort to other pieces of information to avoid longer delays. This problem is challenging for several other reasons, including the desire to maintain a small code size thus keeping the trusted computing base to a manageable size. We treat the virtual machine workload as unmodifiable and as an opaque black box. We also assume based on earlier work that guest workloads can change their behavior very quickly [6, 10].

In this paper, we target the problem of coalescing interrupts for virtual devices without assuming any support from hardware controllers and without using high resolution timers. Traditionally, there are two parameters that need to be balanced: maximum interrupt delivery latency (MIDL) and maximum coalesce count (MCC). The first parameter denotes the maximum time to wait before sending the interrupt and the second parameter denotes the number of accumulated completions before sending an interrupt to the operating system (OS). The OS is interrupted based on whichever parameter is hit first.

We propose a novel scheme to control for both MIDL and MCC implicitly by setting the *delivery ratio* of interrupts based on the current number of commands in flight (CIF) from the guest OS and overall IO completion rate. The ratio, denoted as R, is simply the ratio of how many virtual interrupts are sent to the guest divided by the number of actual IO completions received by the hypervisor on behalf of that guest. Note that $0 < R \leq 1$. Lower values of delivery ratio, R, denotes a higher degree of coalescing. We increase R when CIF is low and decrease the delivery rate R for higher values of CIF.

The key insight in the paper is that unlike network IO, CIF can be used directly for storage controllers because each request has a corresponding command in flight prior to completion. Also, based on the characteristics of storage devices, it is important to maintain certain number of commands in flight to efficiently utilize the underlying storage device [9, 11, 23]. The benefits of command queuing are well known and concurrent IOs are used in most storage arrays to maintain high utilization. Another challenge in coalescing interrupts for storage IO requests is that many important applications issue synchronous IOs. Delaying the completion of prior IOs can delay the issue of future ones, so one has to be very careful about minimizing the latency increase.

Another problem we address is specific to hypervisors, where the host storage stack has to receive and process an IO completion before routing it to the issuing VM. The hypervisor may need to send inter-processor interrupts (IPIs) from the CPU that received the hardware interrupt to the remote CPU where the VM is running for notification purposes. We provide an optimization to reduce the number of IPIs issued using the timestamp of the last interrupt that was sent to the guest OS. This reduces the overall number of IPIs while bounding the latency of notifying the guest OS about an IO completion.

We have implemented our virtual interrupt coalescing (vIC) techniques in the VMware ESX hypervisor [21] though they can be applied to any hypervisor including type 1 and type 2 as well as hardware storage controllers. Experimentation with a set of micro benchmarks shows that vIC techniques can improve both workload throughput and CPU overheads related to IO processing by up to 18%. We also evaluated vIC against the TPC-C workload and found improvements of up to 5%. The vIC implementation discussed here is being used by thousands of customers in the currently shipping ESX version.

The next section presents background on VMware ESX Server architecture and overall system model along with a more precise problem definition. Section 3 presents the design of our virtual interrupt coalescing mechanism along with a discussion of some practical concerns. An extensive evaluation of our implementation is presented in Section 4, followed by some lessons learned from our deployment experience in real world in Section 5. Section 6 presents an overview of related work followed by conclusions and directions for future work in Sections 7 and 8 respectively.

2 System Model

Our system model consists of two components in the VMware ESX hypervisor: VMkernel and the virtual machine monitor (VMM). The VMkernel is a hypervisor kernel, a thin layer of software controlling access to physical resources among virtual machines. The VMkernel provides isolation and resource allocation among virtual machines running on top of it. The VMM is responsible for correct and efficient virtualization of the x86 instruction set architecture as well as emulation of high performance virtual devices. It is also the conceptual equivalent of a "process" to the ESX VMkernel. The VMM intercepts all the privileged operations from a VM including IO and handles them in cooperation with the VMkernel.

Figure 1 shows the ESX VMkernel executing storage stack code on the CPU on the right and an example VM running on top of its virtual machine monitor (VMM) running on the left processor. In the figure, when an interrupt is received from a storage adapter (1), appropriate code in the VMkernel is executed to handle the IO completion (2) all the way up to the vSCSI subsystem which narrows the IO to a specific VM. Each VMM shares a common memory area with the ESX VMkernel, where the VMkernel posts IO completions in a queue (3) following which it may issue an inter-process interrupt or IPI (4) to notify the VMM. The VMM can pick up the completions on its next execution (5) and process them (6) resulting finally in the virtual interrupt being fired (7).

Without explicit interrupt coalescing, the VMM always asserts the level-triggered interrupt line for every IO. Level-triggered lines do some implicit coalescing already but that only helps if two IOs are completed back-to-back in the very short time window before the guest interrupt service routine has had the chance to deassert the line.

Only the VMM can assert the virtual interrupt line and it is possible after step 3 that the VMM may not get a chance to execute for a while. To limit any latency implications of a VM not entering into the VMM, the

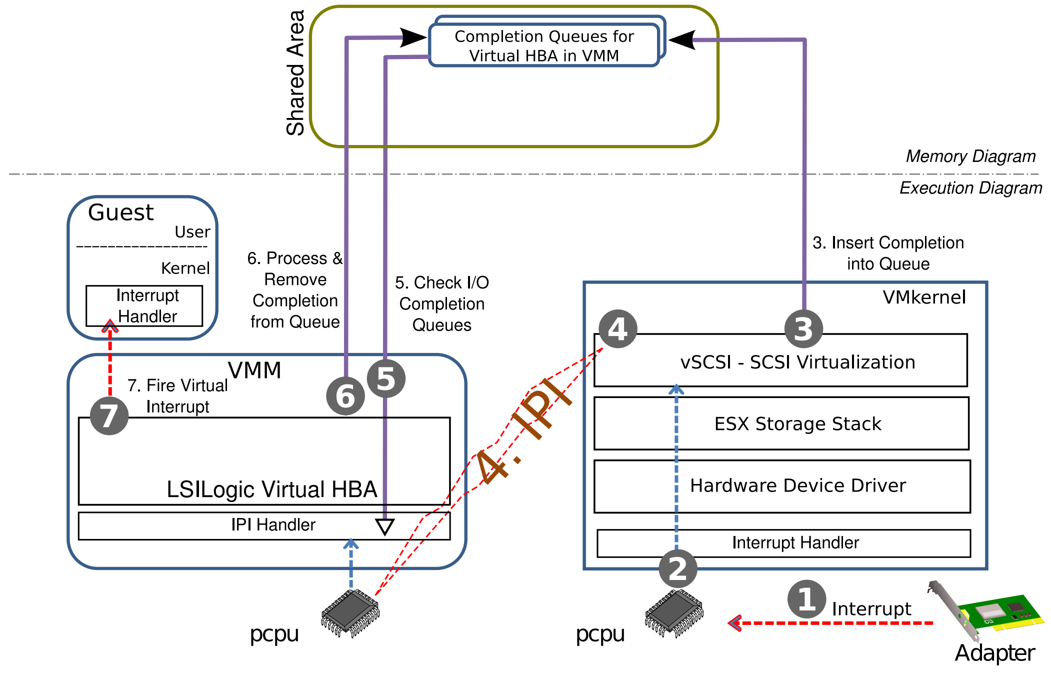

Figure 1: Virtual Interrupt Delivery Mechanism. When a disk IO completes, an interrupt is fired (1) from a physical adapter to a particular Physical CPU (PCPU) where the interrupt handler of the hypervisor delivers it to the appropriate device driver (2). Higher layers of the hypervisor storage stack process the completion until the IO is matched (vSCSI layer) to a particular Guest Operating System which issued the IO and its corresponding Virtual Machine Monitor (VMM). vSCSI then updates the shared completion queue for the VMM (3) and if the guest or VMM is currently executing, issues an inter-processor interrupt (IPI) to the target PCPU where the VMM is known to be running (4). The IPI is only a latency optimization since the VMM would have inspected the shared queues the next time the guest exited to the VMM anyway. The remote VMM's IPI handler takes the signal and (5) inspects the completion queues of its virtual SCSI host bus adapters (HBAs), processes and virtualizes the completions (6) and fires a virtual interrupt to be handled by the guest (7).

VMkernel will take one of two actions. It will schedule the VM if it is descheduled. Otherwise, if both the VM and the VMkernel are executing on separate cores at the same time, the VMkernel sends an IPI, in step 4 in the figure. This IPI is purely an optimization to provide lower latency IO completions to the guest. Without the IPI, guests may execute user level code for an extended period without triggering any hypervisor intercept that would allow for virtual interrupt delivery. Correctness guarantees can still be met even if the IPI isn't issued since the VMM will pickup the completion as a matter of course the next time that it gets invoked via a timer interrupt or a guest exiting into VMM mode due to a privileged operation.

Based on the design described above, there are two inefficiencies in the existing mechanism. First the VMM will interrupt the guest for every completion that it sees posted by the VMkernel. We would like to coalesce these to reduce the guest CPU overhead during high IO rates. Second, IPIs are very costly and are used mainly as a latency optimization. It would be desirable to dramatically reduce them if one could track the rate at which completions are being picked up by the VMM. All this needs to be done without the help of fine grained timers because they are prohibitively expensive in a hypervisor. Thus the main challenges in coalescing virtual interrupts can be summarized as:

1. How to control the rate of interrupt delivery from a VMM to a guest without loss of throughput?

2. How and when to delay the IPIs without inducing high IO latencies?

In the next section, we present our virtual interrupt coalescing mechanisms to efficiently resolve both of these challenges.

3 vIC Design

In this section, we first present some background on existing coalescing mechanisms and explain why they cannot be used in our environment. Next, we present our approach at a high level followed by the details of each component and a discussion of specific implementation issues.

3.1 Background

When implemented in physical hardware controllers, interrupt coalescing generally makes use of high resolution timers to cap the amount of extra latency that interrupt coalescing might introduce. Such timers allow the controllers to directly control MIDL (maximum interrupt delivery latency) and adapt MCC (maximum coalesce count) based on the current rate. For example, one can configure MCC based on a recent estimate of interrupt arrivals and put a hard cap on latency by using high resolution timers to control MIDL. Some devices are known to allow a configurable MIDL in increments of tens of microseconds.

Such high resolution timers are generally used in dedicated IO processors where the firmware timer handler overhead can be well contained and the hardware resources can be provisioned at design time to meet the overhead constraints. However, in any general purpose operating system or hypervisor, it is generally not considered feasible to program high resolution timing as a matter of course. The associated CPU overhead is simply too high.

If we were to try to directly map that MCC/MIDL solution to virtual interrupts, we would be forced to drive the system timer interrupt using resolutions as high as 100 μs. Such a high interrupt rate would have prohibitive performance impact on the overall system both in terms of the sheer CPU cost of running the software interrupt handler ten thousand times a second, as well as the first- and second-order context switching overhead associated with each of them. As a comparison, Microsoft Windows 7 typically sets up its timers to go off every 15.6 *ms* or down to 1 *ms* in special cases whereas VMware ESX configures timers in the range of 1 *ms* and 10 *ms* or even longer when using one-shot timers. This is orders of magnitude lower resolution than what is used by typical storage hardware controller firmware.

3.2 Our approach

In our design, we define a parameter called interrupt delivery ratio R, as the ratio of interrupts delivered to the guest and the actual number of interrupts received from the device for that guest. A lower delivery ratio implies a higher degree of coalescing. We dynamically set our interrupt delivery ratio, R, in a way that will provide coalescing benefits for CPU efficiency as well as tightly control any extra vIC-related latency. This is done using commands in flight (CIF) as the main parameter and IO completion rate (measured as IOs per sec or IOPS) as a secondary control.

At a high level, if IOPS is high, we can coalesce more interrupts within the same time period, thereby improving CPU efficiency. However, we still want to avoid and limit the increase in latency for cases when the IOPS changes drastically or when the number of issued commands is very low. SSDs can typically do tens of thousands of IOPS even with CIF = 1, but delaying IOs in this case would hurt overall performance.

To control IO delay, we use CIF as a guiding parameter, which determines the overall impact that the coalescing can have on the workload. For example, coalescing 4 IO completions out of 32 outstanding might not be a problem since we are able to keep the storage device busy with the remaining 28, whereas even a slight delay caused by coalescing 2 IOs out of 4 outstanding could result in the resources of the storage device not getting fully utilized. Thus we want to vary the delivery ratio R in inverse proportion of the CIF value. Using both CIF values and estimated IOPS value, we are able to provide effective coalescing for a wide variety of workloads.

There are three main parameters used in our algorithm:

- *iopsThreshold*: IOPS value below which no interrupt coalescing is done.
- *cifThreshold*: CIF value below which no interrupt coalescing is done.
- *epochPeriod*: Time interval after which we re-evaluate the delivery ratio, in order to react to the change in the VM workload.

The algorithm operates in one of the three modes:
(1) **Low-IOPS ($R = 1$):** We turn off vIC if the achieved throughput of a workload ever drops below the *iopsThreshold*. Recall that we do not have a high resolution timer. If we did, whenever it would fire, it would allow us to determine if we have held on to an IO completion for too long. A key insight for us is that instead of a timer, we can actually rely on *future* IO-completion events to give our code a chance to control extra latency.

For example, an IOPS value of 20,000 means that on average there will be a completion returned every 50 μs. Our default iopsThreshold is 2000 that implies a completion on average every 500 μs. Therefore, at worst, we can add that amount of latency. For higher IOPS, the extra latency only decreases. In order to do this, we keep an estimate of the current number of IOPS completed by the VM.

(2) **Low-CIF ($R = 1$):** We turn off vIC whenever the number of outstanding IOs (CIF) drops below a configurable parameter *cifThreshold*. Our interrupt coalescing

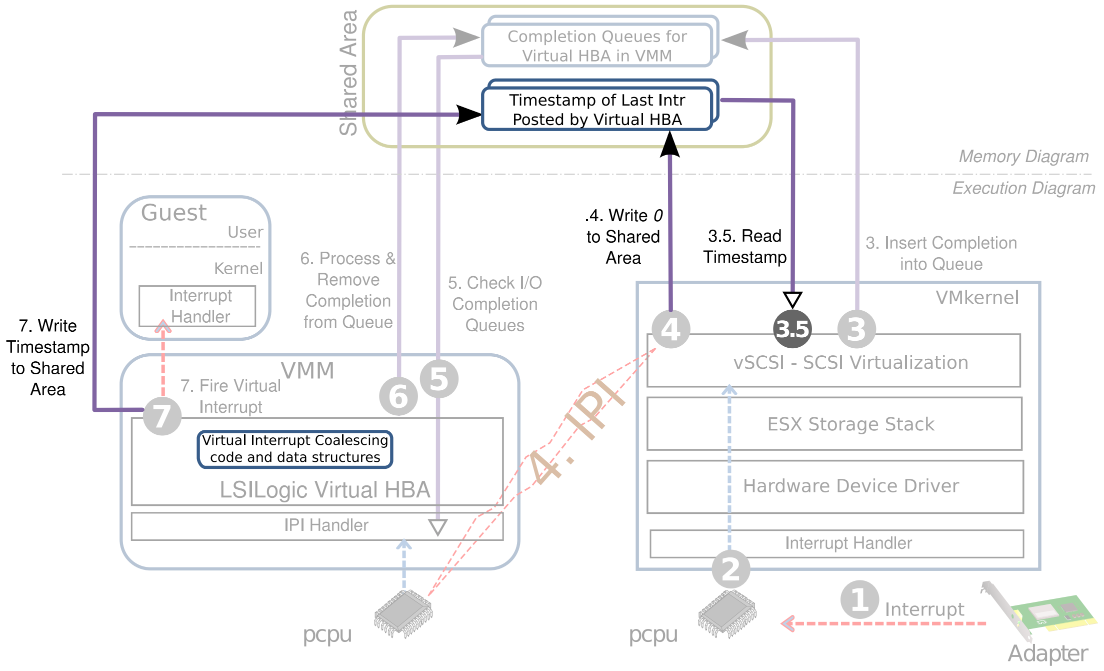
Shared Area
Completion Queues for Virtual HBA in VMM
Timestamp of Last Intr Posted by Virtual HBA
Memory Diagram
Execution Diagram
Guest
User
Kernel
Interrupt Handler
7. Write Timestamp to Shared Area
6. Process & Remove Completion from Queue
5. Check I/O Completion Queues
.4. Write 0 to Shared Area
3.5. Read Timestamp
3. Insert Completion into Queue
VMkernel
VMM
7. Fire Virtual Interrupt
Virtual Interrupt Coalescing code and data structures
LSILogic Virtual HBA
IPI Handler
4. IPI
vSCSI - SCSI Virtualization
ESX Storage Stack
Hardware Device Driver
Interrupt Handler
pcpu
pcpu
Interrupt
Adapter

Figure 2: Virtual Interrupt Delivery Steps. In addition to Figure 1, vIC adds a new shared area object tracking the last time that the VMM fired an interrupt. Before sending the IPI, vSCSI checks to ensure that time since the last VMM-induced virtual interrupt is less than a configurable threshold. If not so, an IPI is still fired, otherwise, it is deferred. In the VMM, an interrupt coalescing scheme is introduced. Note that we did not introduce a high-resolution timer and instead rely on the subsequent IO completions themselves to drive the vIC logic and to regulate the vIC related delay.

algorithm tries to be very conservative so as to not increase the application IO latency for trickle IO workloads. Such workloads have very strong IO inter dependencies and generally issue only a very small number of outstanding IOs.

A canonical example of an affected workload is `dd`, which issues *one* IO at a time. For `dd`, if we had coalesced an interrupt, it would actually hang forever. In fact, waiting is completely useless for such cases and it only adds extra latency. When only a small number of IOs (cifThreshold) remain outstanding on an adapter, we stop coalescing. Otherwise, there may be a throughput reduction because we are delaying a large percentage of IOs.

(3) **Variable R based on CIF:** Setting the delivery ratio (R) dynamically is challenging since we have to balance the CPU efficiency gained by coalescing against additional latency that may be added especially since that may in turn lower the achieved throughput. We discuss our computation of R next.

3.2.1 Dynamic Adjustment of Delivery Ratio R

Which ratio is picked depends upon the number of commands in flight (CIF) and the configuration option “cifThreshold”. As CIF increases, we have more room to coalesce. For workloads with multiple outstanding IOs, the extra delay works well since they are able to amortize the cost of the interrupt being delivered to process more than one IO. For example, if the CIF value is 24, even if we coalesce 3 IOs at a time, the application will have 21 other IOs pending at the storage device to keep it busy.

In deciding the value of R, we have two main issues to resolve. First we cannot choose an arbitrary fractional value of R and implement that because of the lack of floating point calculations in the VMM code. Second, a simple ratio of the form $1/x$ based on a counter x would imply that the only delivery-ratio options available to the algorithm would be (100%, 50%, 25%, 12.5%, ...). The jump from 100% down to 50% is actually too drastic. Instead, we found that to be able to handle a multitude of situations, we need to have delivery ratios, anywhere from 100% down to 6.25%. In order to do this we chose to set *two* fields, countUp and skipUp, dynamically to express the delivery ratios. Intuitively, we deliver (countUp) out of every (skipUp) interrupts, *i.e.* $R = countUp/skipUp$. For example, to deliver 80% of the interrupts, countUp = 4 and skipUp = 5 whereas for 6.25% countUp = 1 and skipUp = 16. Table 1 shows the full range of values as encoded in Algorithm 1. By allowing ratios between 100% and 50%, we can tightly

Algorithm 1: Delivery Ratio Determination

```
IntrCoalesceRecalc(int cif)
$currIOPS$ : Current throughput in IOs per sec;
$cif$ : Current # of commands in flight (CIF);
$cifThreshold$ : Configurable min CIF (default=4);
if $currIOPS < iopsThreshold \lor cif < cifThreshold$ then
    /* $R = 1$ */
    $countUp \longleftarrow 1$;
    $skipUp \longleftarrow 1$;
else if $cif < 2 * cifThreshold$ then
    /* $R = 0.8$ */
    $countUp \longleftarrow 4$;
    $skipUp \longleftarrow 5$;
else if $cif < 3 * cifThreshold$ then
    /* $R = 0.75$ */
    $countUp \longleftarrow 3$;
    $skipUp \longleftarrow 4$;
else if $cif < 4 * cifThreshold$ then
    /* $R = 0.66$ */
    $countUp \longleftarrow 2$;
    $skipUp \longleftarrow 3$;
else
    /* $R = 8/CIF$ */
    $countUp \longleftarrow 1$;
    $skipUp \longleftarrow cif/(2 * cifThreshold)$;
```

control the throughput loss at smaller CIF.

The exact values of *R* are determined based on experimentation and to support the efficient implementation in a VMM. Algorithm 1 shows the exact values of delivery ratio *R* as a function of CIF, cifThreshold and iopsThreshold. Next we will discuss the details of interrupt delivery mechanism and some optimizations in implementing this computation.

3.2.2 Delivering Interrupts

On any given IO completion, the VMM needs to decide whether to post an interrupt to the guest or to coalesce it with a future one. This decision logic is captured in pseudo code in Algorithm 2. First, at every "epoch period", which defaults to 200 *ms*, we reevaluate the vIC rate so we can react to changes in workloads. This is done in function $IntrCoalesceRecalc()$, the pseudo code for which is found in Algorithm 1.

Next, we check to see if the new CIF is below the cifThreshold. If such a condition happens, we immediately deliver the interrupt. The VMM is designed as a very high performance software system where we worry about code size (in terms of both lines of code (LoC) and bytes of `.text`). Ultimately, we have to calculate for *each* IO completion whether or not to deliver a virtual interrupt given the ratio $R = countUp/skipUp$. Since this decision is on the critical path of IO completion, we have designed a simple but very condensed logic to do so with the minimum number of LoC, which needs careful explanation.

Algorithm 2: VMM—IO Completion Handler

```
$cif$ : Current # of commands in flight (CIF);
$cifThreshold$ : Configurable min CIF (default=4);
$epochStart$ : Time at start of current epoch (global);
$epochPeriod$ : Duration of each epoch (global);
$diff \longleftarrow currTime() - epochStart$;
if $diff > epochPeriod$ then
    $IntrCoalesceRecalc(cif)$;
if $cif < cifThreshold$ then
    $counter \longleftarrow 1$;
    $deliverIntr()$;
else if $counter < countUp$ then
    $counter++$;
    $deliverIntr()$;
else if $counter >= skipUp$ then
    $counter \longleftarrow 1$;
    $deliverIntr()$;
else
    $counter++$;
    /* don't deliver */
if Interrupt Was Delivered then
    $SharedArea.timeStamp \longleftarrow currTime()$;
```

CIF	Intr Delivery Ratio *R* as %
1-3	100%
4-7	80%
8-11	75%
12-15	66%
CIF $\geq$ 16	8 / CIF
e.g., CIF = 64	12%

Table 1: Default interrupt delivery ratio (*R*) as a function of CIF. $cifThreshold$ is set to the default of 4.

In Algorithm 2, *counter* is an abstract number that counts up from 1 till $countUp - 1$ delivering an interrupt each time. It then continues to count up till $skipUp - 1$ while skipping each time. Finally, once *counter* reaches *skipUp*, it is reset back to 1 along with an interrupt delivery. Let us look at two examples of a series of *counter* values as more IOs come in, along with whether the algorithm delivers an interrupt as tuples of ⟨counter, deliver?⟩. For $countUp/skipUp$ ratio of 3/4, a series of IOs looks like:

⟨1, yes⟩, ⟨2, yes⟩, ⟨3, no⟩, ⟨4, yes⟩.

Whereas for *countUp/skipUp* of 1/5:

⟨1, no⟩, ⟨2, no⟩, ⟨3, no⟩, ⟨4, no⟩, ⟨5, yes⟩.

Next we look at the optimization related to reducing the number of IPIs sent to a VM during high IO rate.

3.3 Reducing IPIs

So far, we have described the mechanism for virtual interrupt coalescing inside the VMM. As mentioned in Section 2 and illustrated in Figure 1, another component involved in IO delivery is the ESX VMkernel. Recall that IO completions from hardware controllers are handled by this component and sent to the VMM, an operation that can require an IPI in case the guest is currently running on the remote processor. Since IPIs are expensive, we would like to avoid them or at the very least minimize their occurrence. Note that the IPI is a mechanism to force the VMM to wrest execution control away from the guest to process a completion. As such it is purely a latency optimization and correctness guarantees don't hinge on it since the VMM frequently gets control anyway and always checks for completions.

Figure 2 shows the additional data flow and computation in the system to accomplish our goal of reducing IPIs. The primary concern is that a guest OS might have scheduled a compute intensive task, which may result in the VMM not receiving an intercept. In the worst case, the VMM will wait until the next timer interrupt, which could be several milliseconds away, to receive a chance to execute and deliver virtual interrupts. So, our goal is to avoid delivering IPIs as much as possible while also bounding the extra latency increase.

We introduce as part of the shared area between the VMM and the VMkernel where completion queues are managed, a new time-stamp of the last time the VMM posted an IO completion virtual interrupt to the guest (see last line of Algorithm 2). We added a new step (3.5) in the VMkernel where before firing an IPI, we check the current time against what the VMM has posted to the shared area. If the time difference is greater than a configurable threshold, we post the IPI. Otherwise, we give the VMM an opportunity to notice IO completions in due course on its own. Section 4.5 provides experimental evaluation of the impact of IPI delay threshold values.

3.4 Implementation Cost

We took great care to minimize the cost of vIC and to make our design and implementation as portable as possible. A part of that was to refrain from using any floating point code. In the critical path code (Algorithm 2), we even avoid integer divisions. This should allow our design to be directly implementable in other hypervisors on any CPU architecture, and even in firmware or hardware of storage controllers. For reference, the increase in the 64-bit VMM `.text` section was only 400 bytes and the `.data` section grew by only 104 bytes. Our patch for the LSI Logic emulation in the VMM was less than 120 LoC. Similarly, the IPI coalescing logic in the VMkernel was implemented with just 50 LoC.

4 Experimental Evaluation

To evaluate our vIC approach, we have examined several micro-benchmark and macro-benchmark workloads and compared each workload with and without interrupt coalescing. In each case we have seen a reduction in CPU overhead, often associated with an increase in throughput (IOPS). For all of the experiments, unless otherwise indicated, the parameters are set as follows: $cifThreshold = 4$, $iopsThreshold = 2000$ and $epochPeriod = 200\,ms$. All but the TPC-C experiments were run on an HP Proliant DL-380 machine with 4 dual-core AMD 2.4 GHz processors. The attached storage array was an EMC CLARiiON CX3-40 with very small fully cached LUNs. The Fibre Channel HBA used was a dual-port QLogic 4Gb card.

In the next subsections, we first discuss the results for the Iometer micro benchmark in Section 4.1. Next, we cover the CPU utilization improvements of the Iometer benchmark and a detailed break-down of savings in Section 4.2. Section 4.3 presents our evaluation of vIC using two SQL IO simulators, namely SQLIOSim and GSBlaster. Finally, we present results for a complex TPC-C-like workload in Section 4.4. For each of the experiments, we have looked at the CPU savings along with the impact on throughput and latency of the workload.

4.1 Iometer Workload

We evaluated two Iometer [1] workloads running on a Microsoft Windows 2003 VM on top of an internal build of VMware ESX Server. The first workload consists of 4KB sequential IO reads issued by one worker thread running on a fully cached Logical Unit (LUN). In other words, all IO requests are hitting the array's cache instead of requiring disk access. The second workload is identical except for a different block size of 8KB.

For both workloads we varied the number of outstanding IOs to see the improvement over baseline. In Table 2, we show the full matrix of our test results for the 4KB workload. Furthermore, Table 3 summarizes the percentage improvements over the baseline where coalescing was disabled. The column labeled *R*, in Table 2, is the average ratio chosen by algorithm based on varying CIF over the course of the experiment; as expected, our algorithm coalesces more rigorously as the number of outstanding IOs is increased. Looking closely at the 64 CIF case, we can see that the dynamic delivery ratio, *R*, was found to be 1/6 on average. This means that one interrupt was delivered for every six IOs. The guest operating system reported a drop from 113K interrupts per

OIO	$\hat{R}$	IOPS	CPU cost cycles/ IO	Int/sec Guest	Baseline IOPS	Baseline CPU Cost	Baseline Int/sec Guest
8	4/5	31.2K	82.6K	49K	30.5K	84.2K	47K
16	2/3	38.9K	74.8K	58K	38.4K	77.0K	60K
32	1/3	48.3K	68.0K	69K	46.4K	70.5K	74K
64	1/6	53.1K	64.0K	34K	52.9K	78.4K	113K

Table 2: Iometer 4KB reads with one worker thread and a cached Logical Unit (LUN). $\hat{R}$ is the average delivery ratio set dynamically by the algorithm in this experiment. OIO is the number of outstanding IOs setting in Iometer. At runtime, CIF is often lower than the workload configured OIO as confirmed by $\hat{R}$ here being lower than the $R(OIO)$ from Table 1.

OIO	IOPS %diff	CPU cost %diff	Int/sec Guest %diff
8	2.3%	-1.9%	4.3%
16	1.3%	-2.8%	-3.3%
32	4.1%	-3.5%	-6.8%
64	0.4%	-18.4%	-66.4%

Table 3: Summary of improvements in key metrics with vIC. The experiments is the same as in Table 2.

second to 34K. The result of this is that the CPU cycles per IO have also dropped from 78.4K to 64.0K, which is an efficiency gain of 18%.

In Tables 4 and 5 we show the same results as before, but now with the 8KB IO workload. For the 64 CIF case, the algorithm results in the same interrupt coalescing ratio of 1/6 with now a 7% efficiency improvement over the baseline. The interrupt per second in the guest have dropped from 30K to 11K.

In both Table 2 and 4 we see a noticeable reduction in CPU cycles per IO whenever vIC has been enabled. We also would like to note that throughput never decreased and in many cases actually increased significantly.

4.2 Iometer CPU Usage Breakdown

For the 8KB sequential read Iometer workload with 64 outstanding IOs, we examined the breakdown between the VMM and guest OS CPU usage. Table 6 shows the monitor's abridged kstats. The `VMK_VCPU_HALT` statistic is the percent of time that the guest was idle. Notice that the guest idle time has increased which implies that the guest OS spent less time processing IO for the same effective throughput. The guest kernel runtime is measured by the amount of time we spent in the `TC64_IDENT`. Here we see a noticeable decrease in kernel mode execution time from 9.0% to 7.4%. The LSI Logic virtual SCSI adapter IO issuing time measured by `device_Priv_Lsilogic_IO` has decreased from

OIO	$\hat{R}$	IOPS	CPU cost cycles/ IO	Int/sec Guest	Baseline IOPS	Baseline CPU Cost	Baseline Int/sec Guest
8	4/5	31.2K	83.6K	48K	29.9K	88.2K	49K
16	2/3	39.3K	77.6K	61K	38.5K	81.3K	63K
32	1/3	41.5K	76.0K	60K	41.1K	77.1K	69K
64	1/6	41.5K	71.0K	11K	41.1K	75.7K	30K

Table 4: Iometer 8KB reads with one worker thread and a cached Logical Unit (LUN). Caption as in Table 2.

OIO	IOPS %diff	CPU cost %diff	Int/sec Guest %diff
8	4.3%	-5.2%	-2.0%
16	2.1%	-4.5%	-3.2%
32	1.0%	-1.5%	-13.0%
64	1.0%	-6.2%	-63.3%

Table 5: Summary of improvements in key metrics with vIC. The experiments is the same as in Table 4.

5.0% to 4.3%.

The IO completion work done in the VMM is part of a generic message delivery handler function and is measured by `DrainMonitorActions` in the profile. The table shows a slight increase from 0.5% to 0.7% of CPU consumption due to the management of the interrupt coalescing ratio.

The net savings gained by enabling virtual interrupt coalescing can be measured by looking at the guest idle time which is a significant 6.4% of a core. In a real workload which performs both IO and CPU-bound operations, this would result in an extra 6+% of available time for computation. We expect that some part of this gain also includes the reduction of the virtualization overhead as a result of vIC mostly consisting of second order effects related to virtual device emulation.

4.3 SQLIOSim and GSBlaster

We also examined the results from SQLIOSim [13] and GSBlaster. Both of these macro-benchmark workloads are designed to mimic the IO behavior of Microsoft SQL Server.

SQLIOSim is designed to target an "ideal" IO latency to tune for. That means that if the benchmark sees a higher IO latency it assumes that there are too many outstanding IOs and reduces that number. The reverse case is also true allowing the benchmark to tune for this preset optimal latency value. The user chooses this value to maximize their throughput and minimize their latency. In SQLIOSim we used the default value of 100ms.

GSBlaster is our own internal performance testing tool which behaves similar to SQLIOSim. It was designed as a simpler alternative to SQLIOSim which we could

	With vIC	Without vIC
VMK_VCPU_HALT	71.4%	65.0%
TC64_IDENT	7.4%	9.0%
device_Priv_Lsilogic_IO	4.3%	5.0%
DrainMonitorActions	0.7%	0.5%

Table 6: VMM profile for 8KB sequential read Iometer workload. Each row represents time spent in the related activity relative to a single core. The list is filtered down for space reasons to only the profile entries that changed.

	IOPS	CPU Cost	Baseline IOPS	Baseline CPU Cost	IOPS %diff	CPU Cost %diff
SQLIOSim	6282	$339K$	5327	$410K$	+17.9%	-17.4%
GSBlaster	24651	$126K$	20755	$151K$	+18.8%	-16.6%

Table 7: Performance improvements in SQLIOSim and GSBlaster. Improvements are seen both in IOPS and CPU efficiency.

understand and analyze in an easier manner. As opposed to SQLIOSim, when using GSBlaster we choose a fixed value for the number of outstanding IOs. It will then run the workload based on this configuration.

Table 7 shows the results of our optimization on both the target macro-benchmark workloads. We can see that the IOPS increased as a result of vIC by more than 17%. As in previous benchmarks, we also found that the CPU cost per IO decreased (by 17.4% in the case of SQLIOSim and 16.6% in the case of GSBlaster).

4.4 TPC-C Workload

The results seen so far have been for micro and macro benchmarks with relatively simple workload drivers. Whereas such data gives us insight into the upside potential of vIC, we have to put that in context of a large, complex application that performs computation as well as IO. We chose to evaluate our system on our internal TPC-C testbed[1]. The number of users is always kept large enough to fully utilize the CPU resources, such that adding more users won't increase the performance. Our TPC-C run was on a 4-way Opteron E based machine. The system was backed by a 45-disk EMC CX-3-20 storage array.

Table 8 shows results with and without interrupt coalescing with a *cifThreshold* range of 2–4. When vIC is enabled, we were able to increase the number of users from 80 to 90 at the CPU saturation point. This immediately demonstrates that vIC freed up more CPU for real workload computation. Increasing the number of users also helps increase the achieved, user-visible benchmark

[1]Non-comparable implementation; results not TPC-C^{TM}compliant; deviations include: batch benchmark, undersized database.

	T	T Diff	Users	IOPS	Intr/ Sec	Latency
No vIC	43.3		80	10.2K	9.9K	7.7ms
$cifT = 4$	44.6	+3.0%	90	10.4K	6.4K	8.5ms
$cifT = 2$	45.5	+5.1%	90	10.5K	5.8K	9.2ms

Table 8: TPC-C workload throughput (T) run with and without interrupt coalescing and with different cifThreshold (cifT) configuration parameter values.

	Diff vs. baseline	
	$cifT = 4$	$cifT = 2$
IDT_AfterInterrupt	-28%	-31%
device_Priv_Lsilogic_IO	-12%	-14%
DrainMonitorActions	-19%	-22%
Intr_Deliver	-25%	-30%
Intr_IOAPIC	-40%	-46%
device_SCSI_CmdComplete	-13%	-16%
Intr	-42%	-49%

Table 9: VMM profile for TPC-C. Each row represents improvements in time spent in the related activity. The list is filtered down for space reasons to only the profile entries that changed significantly.

metric of throughput or transactions per minute. Table 8 shows that the transcation rate increased by 3.0% and 5.1% for *cifThreshold* of 4 and 2, respectively. In our experience, optimizations for TPC-C are very difficult and significant investment is made for each fractional percent of improvement. As such, we consider an increase of 3.0%–5.1% to be very significant.

We also logged data for the algorithm-selected vIC rate, R, every 200 ms during the run. Figure 3 shows the histogram of the distribution of R for certain duration of the TPC-C run. It is interesting that R varies dramatically throughout the workload. The frequency distribution here is a function of the workload and not vIC.

Furthermore, in Figure 4 we show the same data plotted against time. The interrupt coalescing rate varies significantly because the TPC-C workload has periods of IO bursts. In fact, there are numerous times where the algorithm selects to deliver less than 1 out of every 12 interrupts. This illustrates how the self-adaptability and responsiveness of our algorithm is necessary to satisfy complex real-world workloads.

As a result of interrupt coalescing, we see a decrease in virtual interrupts per second delivered to the guest from 9.9K to 6.4K and 5.8K. We also noticed an increase in the IOPS achieved by the storage array. This can be explained by the fact that there is increased parallelism (more users) in the input workload. Such a change in workload has been demonstrated in earlier work to increase throughput [11].

Any increase in parallelism is also accompanied by an expected increase in average latency [11]. This explains

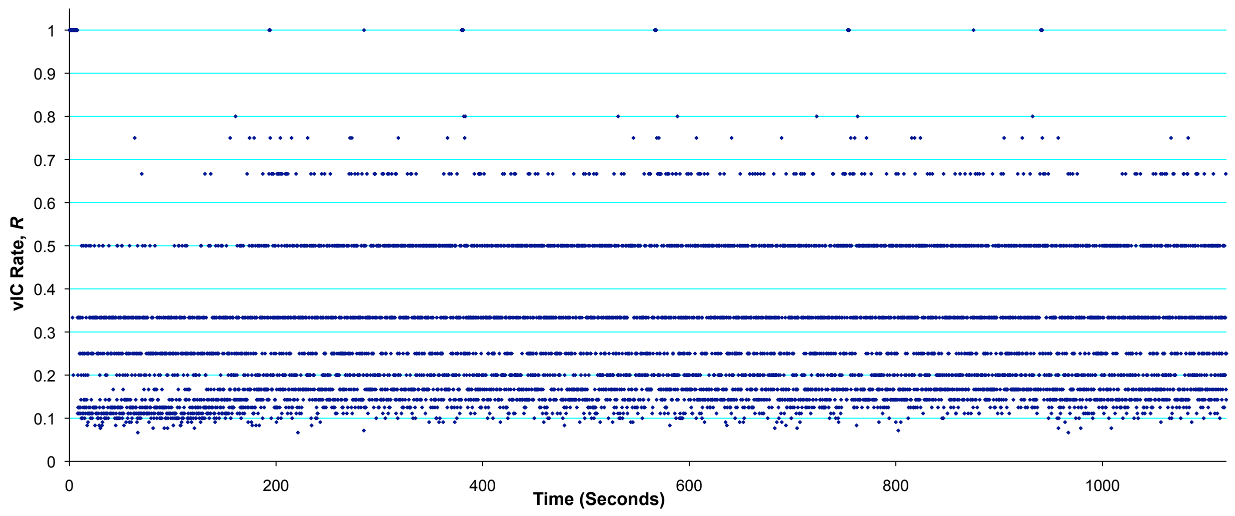

Figure 4: The algorithm selected virtual interrupt coalescing rate, R, over time for TPC-C. The high dynamic range illustrates the burstiness in outstanding IOs of the workload and the resulting online adaptation by vIC.

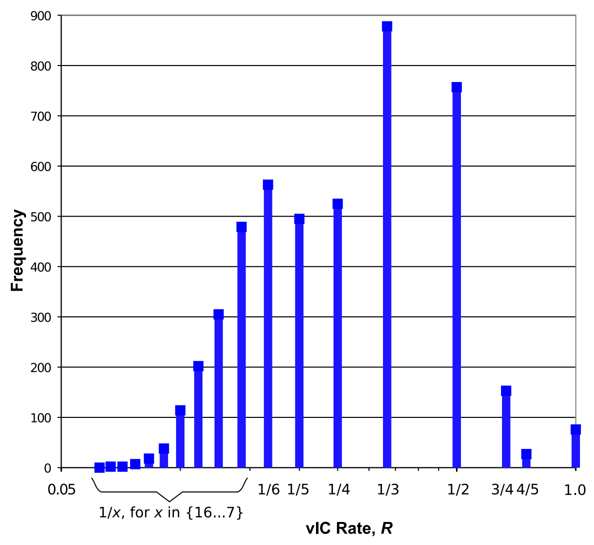

Figure 3: Histogram of dynamically selected virtual interrupt coalescing rates, R, during our TPC-C run. The x-axis is log-scale.

the bulk of the increase between the no-vIC and the vIC-enabled rows in Table 8. However, one would expect a certain increase in latency from any interrupt coalescing algorithm. In our case, we expect latency increases to be less than a few hundred microseconds. Using instantaneous IOPS, CIF and R, we can calculate the mean delay from vIC. For instance, at the median delivery rate $R = 1/3$, at the achieved mean IOPS of 10K for this experiment, the increase would be 200 μs.

In Table 9 we show a profile of percentage reduction in CPU utilization of several key VMM components as a result of interrupt coalescing. IOs are processed as part of the VMM's action processing queue. The reduction in the queue processing is between 19% and 22% for the CIF thresholds of 4 and 2 respectively. Fewer interrupts means that the guest operating system is performing fewer register operations on the virtual LSI Logic controller, shown by `device_Priv_Lsilogic_IO`. The net reduction in device operations translated to a 12% and 14% reduction, respectively, in CPU usage relative to using the virtual device without vIC. We also measured an approximately 30% reduction in the monitor's interrupt delivery function `Intr_Deliver`.

4.5 IPI interference: CPU-bound loads

Recall that we described an optimization of not posting IPIs in all cases using a threshold delay, in order to lower the impact of IPIs on VMs workload (Section 3.3). In this section, we first motivate our optimization by showing that the impact on CPU-bound applications can be very high. We show nevertheless that sending at least some IPIs is essential as an IO latency and throughput optimization. We then provide data to illustrate the difficult trade-off between the two concerns.

We ran two workloads, one IO bound and the other CPU bound, on the *same* virtual machine running the Microsoft Windows 2003 operating system. The IO-bound workload is running an Iometer, 1 worker benchmark, doing 8 OIO, 8K Read from a fully cached small LUN. The CPU-bound workload is an SMP version of Hyper Pi for which the run time in seconds to calculate 1M digits of π on two virtual processors is plotted as triangles (average of 6 runs). Hyper Pi is fully-backlogged and we run

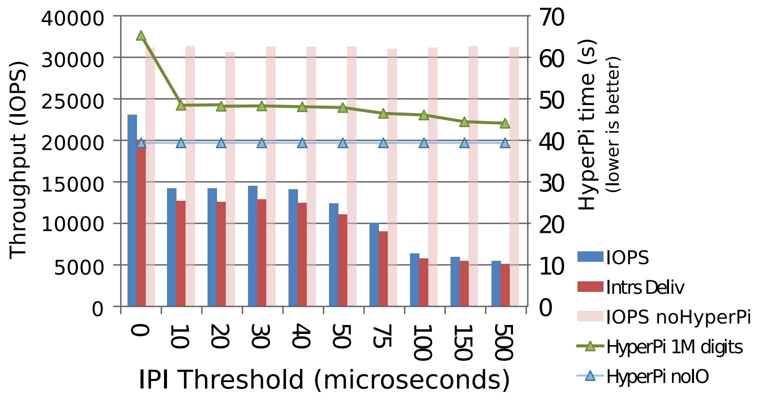

Figure 5: Effect of different IPI send thresholds. This plot illustrates the trade-off in IO throughput and CPU efficiency of two co-running benchmarks, as we vary the key parameter: the IPI send threshold. The workloads are run on the same 2-vCPU Windows 2003 guest. The IO workload performance is shown as vertical bars for an Iometer, 1 worker, 8 OIO, 8K Read from a fully cached small LUN. Each IO data point is the average of 50 consecutive samples. The CPU-bound workload is an SMP version of Hyper Pi for which the run time in seconds to calculate 1M digits of *pi* on two virtual processors is plotted as triangles (average of 6 runs). For each workload, both the co-running and independent scores are plotted. No delay in sending an IPI results in the highest IO throughput whereas waiting 500 microseconds to send an IPI results in the highest performance of the CPU-bound workload.

it at the `idle` priority class effectively giving Iometer higher priority while ensuring that the guest never halts due to idling.

Figure 5 shows the effect of different IPI send thresholds on performance. This plot illustrates the trade-off in IO throughput and CPU efficiency of two co-running benchmarks respectively, as we vary the IPI send threshold from VMkernel. For each workload, Figure 5 shows both the co-running and independent scores. The IO workload performance is shown as vertical bars in terms of IO throughput observed by Iometer. Each IO data point is the average of 50 consecutive samples. Performance of Hyper Pi workload is shown in terms of time to completion of 1 million digit computation for π.

First, notice that the throughput for "IOPS no Hyper Pi" bars do not change much with the IPI send threshold. This is because the guest is largely idle and frequently entering the CPU halt state. Whenever the guest halts, the VMM gains execution control and has the chance to check for pending completions from the VMkernel. As such, sending IPIs more or less frequently has hardly any bearing on the latency of the VMM noticing and delivering an interrupt to the guest. Similarly, the blue triangle case of "Hyper Pi no IO" shows no sensitivity to our parameter. This is obvious: no IO activity means that IPIs are out of the picture anyway.

As soon as we run those two benchmarks together, they severely interfere with each other. This is due to the fact that Hyper Pi is keeping the CPU busy implying that the guest never enters halt. Therefore, the VMM only has rare opportunities to check for pending IO completions (*e.g.*, when a timer interrupt is delivered). Hyper Pi sees its best performance at high IPI-send thresholds (right end of the x-axis) since it gets to run uninterrupted for longer periods of time. However, IO throughput (see the blue bars) suffers quite a bit from the increased latency. Conversely, if no delay is used (left end of the x-axis), the impact on the performance of Hyper Pi is severe. As expected, IO performance does really well in such situations.

It should be noted that the two workloads, though running on the same virtual machine, are not tied to each other. In other words, there is no closed loop between them as is often the case with enterprise applications. Still, the setup is illustrative in showing the impact of IPIs on IO and CPU bound workloads.

These results indicate that setting a good IPI send threshold is important but nontrivial and even workload dependent. We consider automatic setting of this parameter to be highly interesting future work. For ESX, we have currently set the default to be 100 microseconds to favor CPU bound parts of workloads based on experiments on closed loop workloads run against real disks.

5 Deployment Experience

Our implementation of vIC as described in this paper has been the default for VMware's LSI Logic virtual adapter in the ESX hypervisor since version 4.0, which was released in the second quarter of 2009. As such, we have significant field experience with deployments into the tens of thousands of virtual machines. Since then, we have not received any performance bug reports on the virtual interrupt coalescing algorithm.

On the other hand, the `pvscsi` virtual adapter which first shipped in ESX 4.0 did not initially have some of the optimizations that we developed for the LSI Logic virtual interrupt coalescing. In particular, although it had variable interrupt coalescing rates depending on CIF, it was missing the *iopsThreshold* and was not capable of setting the coalescing rate, R, between $1/2$ and 1. As a result, several performance bugs were reported. We triaged these bugs to be related to the missing optimizations in `pvscsi` which are now fixed in the subsequent ESX 4.1 release. We feel that this experience further validates the completeness of our virtual interrupt coalescing approach as a successful, practical technique for significantly improving performance in one dimension (lower CPU cost of IO) without sacrificing others (throughput or latency).

The key issue with any interrupt coalescing scheme is the potential for increases in IO response times which we have studied above. At high IO rates, some application *IO* threads might be blocked a little longer due to coalescing. In our case, this delay is strictly bound by the $1/iopsThreshold$. Our solution is significantly better than other coalescing techniques since it explicitly takes CIF into account. In our experience, vIC lets *compute* threads of real applications run longer before getting interrupted. Increased execution time can reduce overhead from sources such as having the application's working set evicted from the CPU caches, etc. Interrupt coalescing is all about the trade off between CPU efficiency and IO latencies. Hence, we provide parameters to adjust that tradeoff if necessary, though the default settings have been tuned using a variety of workloads.

6 Related Work

Interrupts have been in active use since early days of computers to handle input-output devices. Smotherman [19] provides an interesting history of the evolution of interrupts and their usage in various computer systems starting from UNIVAC (1951). With increasing network bandwidth and IO throughput for storage devices, the rate of interrupts and thus CPU overhead to handle them has been increasing pretty much since the interrupt model was first developed. Although processor speeds and number of cores have been increasing to keep up with these devices, the motivation to reduce overall CPU overhead of interrupt handling has remained strong. Interrupt coalescing has been very successfully deployed in various hardware controllers to mitigate the CPU overhead. Many patents and papers have been published on performing interrupt coalescing for network and storage hardware controllers.

Gian-Paolo's patent [15] provides a method for dynamic adjustment of maximum frame count and maximum wait time parameters for sending the interrupts from a communication interface to a host processor. The packet count parameter is increased when the rate of arrivals is high and decreased when the interrupt arrival rate gets low. The maximum wait time parameter ensures a bounded delay on the latency of the packet delivery. Hickerson and Mccombs's patent [12] uses a single counter to keep track of the number of initiated tasks. The counter is decremented on the task completion event and it is incremented when the task is initiated. A delay timer is set using the counter value. An interrupt is generated either when the delay timer is fired or the counter value is less than a certain threshold. In contrast to both of these patented techniques, our mechanism adjusts the delivery rate itself based on CIF and does not rely on any delay timers. It should be noted, however, that our approach is complementary to interrupt coalescing optimizations done in the hardware controllers since they can benefit in lowering the load on the hypervisor host, in our case the ESX VMkernel.

QLogic [4] and Emulex [5] have also implemented interrupt coalescing in their storage HBAs but the details of their implementation are not publicly available. The knowledge base article for a Qlogic driver [3] suggests the use of a delay parameter, *ql2xintrdelaytimer* which is used as a wait time for firmware before generating an interrupt. This is again dependent on high resolution timers and delaying the interrupts by a certain amount. Online documentation suggests that the QLogic interrupt delay timer can be set in increments of 100 μs. Interrupt coalescing can be disabled by another parameter called *ql2xoperationmode*. Interestingly, this driver parameter allows two modes of interrupt coalescing distinguished by whether an interrupt is fired if CIF drops to 0. A similar document related to Emulex device driver for VMware [2] suggests the use of statically defined delay and IO count thresholds, *lpfc_cr_delay, lpfc_cr_count*, for interrupt coalescing.

Stodolsky, Chen and Bershad [20] describe an optimistic scheme to reduce the cost of interrupt masking by deferring the processing ("continuation") of any interrupt that arrives during a critical section to a later time. Many operating systems now use similar techniques to handle the scheduling of deferred processing for interrupts. The paper also suggests that interrupts be masked at the time of deferral so that the critical section can continue without further interruptions. Level-triggered interrupts like the ones described in our work are another way of accomplishing the same thing. Both of these techniques from the Bershad paper are complementary to the idea of coalescing which is more concerned with the *delay* of interrupt delivery.

Zec et al. [22] study the impact of generic interrupt coalescing implementation in 4.4BSD on the steady state TCP throughput. They modified the *fxp* driver in FreeBSD and controlled only the delay parameter T_d, which specifies the time duration between the arrival of first packet and the time at which hardware interrupt is sent to the OS. Similarly, Dong et al. recently studied [8] the CPU impact of interrupts and proposed an adaptive interrupt coalescing scheme for a network controller.

Mogul and Ramakrishnan [14] studied the problem of receive livelock, where the system is busy processing interrupts all the time and other necessary tasks are starved to the most part. To avoid this problem they suggested the hybrid mechanism of polling under high load and using regular interrupts for lighter loads. Polling can increase the latency for IO completions, thereby affecting the overall application behavior. They optimized their system by using various techniques to initiate polling and enable interrupts under specific conditions. They also proposed round robin polling to fairly allocate resources among various sources.

Salah et al. [17] did an analysis of various interrupt handling schemes such as polling, regular interrupts, interrupt coalescing, and disabling and enabling of interrupts. Their study concludes that no single scheme is good under all traffic conditions. This further motivates the need for an adaptive mechanism that can adjust to the current interrupt arrival rate and other workload parameters. Salah [16] performed an analytical and simulation study of the relative benefit of time-based versus number-of-packets based interrupt coalescing in context of networking. More recently Salah and Qahtan [18] implemented and evaluated a different hybrid interrupt handling scheme for Gigabit NICs in Linux kernel 2.6.15. Their hybrid scheme switches between interrupt disabling-enabling (DE) and polling.

Our approach, instead of switching to polling, adjusts the overall interrupt delivery rate during high load. We believe this is more flexible and adapts well to drastic changes in guest workload. We also use CIF which is available only in context of storage controllers but allows us to solve this problem more efficiently. Furthermore, we do not have the luxury to change the guest behavior in terms of interrupts vs polling because the guest OS is like a black box to virtualization hypervisors.

7 Conclusions

In this paper, we studied the problem of efficient virtual interrupt coalescing in context of virtual hardware controllers implemented by a hypervisor. We proposed the novel techniques of using the number of commands in flight to dynamically adjust the interrupt delivery ratio in fine-grained steps and to use future IO events to avoid the need of high-resolution timers. We also designed a technique to reduce the number of inter-processor interrupts while keeping the latency bounded. Our prototype implementation in the VMware ESX hypervisor showed that we are able to improve application throughput (IOPS) by up to 19% and improve CPU efficiency up to 17% (for the GSBlaster and SQLIOSim workloads respectively). When tested against our TPC-C workload, vIC improved the workload performance by 5.1% and demonstrated the ability of our algorithm to adapt quickly to changes in the workload. Our technique is equally applicable to hypervisors and hardware storage controllers; we hope that our work spurs further work in this area.

8 Open Problems

There are some open problems which deserve further exploration by our fellow researchers and practitioners. Firmware implementations of vIC could lower the cost of hardware controllers and provide tighter latency control than what is available today. Currently, our vIC implementation hard-codes the best CIF-to-*R* mappings based on extensive experimentation. Dynamic adaptation of that mapping appears to be an interesting problem. In some architectures, PCI devices are directly passed-through to VMs. Interrupt coalescing in this context is worthy of investigation.

At first blush, networking controllers do not appear to lend themselves to a CIF-based approach since the protocol layering in the stack means that the lower layers (where interrupt posting decisions are made) do not know the semantics of higher layers. Still, we speculate that inference techniques might be applicable to do aggressive coalescing without loss of throughput in context of high-bandwidth TCP connections using window size-based techniques.

Acknowledgements

We would like to thank Maxime Austruy who worked on the `pvscsi` virtual adapter and co-invented some of the interrupt coalescing techniques discussed here. Many thanks to Davide Bergamasco, Jinpyo Kim, Vincent Lin and Reza Taheri for help with experimental validation of our work and to our shepherd, Muli Ben-Yehuda, for detailed comments and guidance. Finally, this work would not have been possible without support, encouragement and feedback from Ole Agesen, Mateen Ahmad, Keerti Garg, Mallik Mahalingham, Tim Mann, Glen McCready and Carl Waldspurger.

References

[1] Iometer. http://www.iometer.org.

[2] *Emulex Driver for VMware ESX*. August 2007. `www-dl.emulex.com/support/vmware/732/vmware.pdf`.

[3] *QLogic: Advanced Parameters For Driver Modules Under VMware*. October 2009. `http://kb.qlogic.com/KanisaPlatform/Publishing/548/1492_f.SAL_Public.html`.

[4] *QLogic: QLE8152 datasheet*. 2009. `http://www.starline.de/fileadmin/images/produkte/qlogic/QLogic_QLE8152.pdf`.

[5] *Emulex: OneCommand Manager*. June 2010. `http://www.emulex.com/artifacts/ad19cc4e-870a-42e9-a4b2-bcaa70e2afd6/elx_rc_all_onecommand_efficiency_qlogic.pdf`.

[6] I. Ahmad. Easy and Efficient Disk I/O Workload Characterization in VMware ESX Server. *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 149–158, Sept. 2007.

[7] X. Chang, J. Muppala, Z. Han, and J. Liu. Analysis of interrupt coalescing schemes for receive-livelock problem in gigabit ethernet network hosts. In *IEEE International Conference on Communications(ICC)*, pages 1835–1839, May 2008.

[8] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. In *HPCA*, pages 1–10, 2010.

[9] A. Gulati, I. Ahmad, and C. Waldspurger. PARDA: Proportionate Allocation of Resources for Distributed Storage Access. In *Proc. Conference on File and Storage Technology (FAST '09)*, Feb. 2009.

[10] A. Gulati, C. Kumar, and I. Ahmad. Storage Workload Characterization and Consolidation in Virtualized Environments. In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009.

[11] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. BASIL: Automated IO Load Balancing across Storage Devices. In *Proc. Conference on File and Storage Technology (FAST '10)*, Feb. 2010.

[12] R. Hickerson and C. C. Mccombs. Method and apparatus for coalescing i/o interrupts that efficiently balances performance and latency. (US PTO 6065089), May 2000.

[13] Microsoft. How to use the sqliosim utility to simulate sql server activity on a disk subsystem, 2009. `http://support.microsoft.com/kb/231619`.

[14] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.

[15] G. paolo D. Musumeci. System and method for dynamically tuning interrupt coalescing parameters. (US PTO 6889277), May 2005.

[16] K. Salah. To coalesce or not to coalesce. *Intl. J. of Elec. and Comm.*, pages 215–225, 2007.

[17] K. Salah, K. El-Badawi, and F. Haidari. Performance analysis and comparison of interrupt-handling schemes in gigabit networks. *Comput. Commun.*, 30(17):3425–3441, 2007.

[18] K. Salah and A. Qahtan. Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Comput. Commun.*, 32(1):179–188, 2009.

[19] M. Smotherman. Interrupts, 2008. `http://www.cs.clemson.edu/~mark/interrupts.html`.

[20] D. Stodolsky, J. B. Chen, and B. N. Bershad. Fast interrupt priority management in operating system kernels. In *moas'93: USENIX Symposium on USENIX Microkernels and Other Kernel Architectures Symposium*, pages 9–9, Berkeley, CA, USA, 1993. USENIX Association.

[21] VMware, Inc. *Introduction to VMware Infrastructure*. 2010. http://www.vmware.com/support/pubs/.

[22] M. Zec, M. Mikuc, and M. Zagar. Estimating the impact of interrupt coalescing delays on steady state tcp throughput. *Tenth SoftCOM 2002 conference*, 2002.

[23] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. In *Proc. of MASCOTS*, Sep 2005.