Improving Performance with Interrupt Coalescing for Virtual Machine Disk IO in VMware ESX Server

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

Abstract

Interrupt coalescing is a proven technique for reducing CPU utilization when processing high IO rates in storage and networking controllers. Virtualization introduces a layer of virtual hardware whose interrupt rate can be controlled by the hypervisor. In this paper, we present the design and implementation of a virtual interrupt coalescing scheme for virtual SCSI hardware controllers in a hypervisor. We use the number of *commands in flight* from the guest to dynamically set our interrupt coalescing rate. 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 disk IO controllers which, unlike networking, don't receive anonymous traffic. We have built a prototype of virtual interrupt coalescing on the VMware ESX Server hypervisor and we provide preliminary experimental data on various workloads and show performance improvements of up to 18%.

1 Introduction

Many important datacenter 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, increasing deployments of high performance consolidated storage devices using Storage Area Network (SAN) or Network-Attached Storage (NAS) hardware and wider adoption of solid-state disks.

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 [3, 6]. 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. Traditionally, interrupt coalescing or moderation has been used in storage controller cards to limit the number of times application execution is interrupted by the device to handle IO completions. This technique has 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 hard and 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 our trusted computing base to a manageable size. We treat the virtual machine workload as an unmodifiable and opaque black box and we assume based on earlier work that guest workloads can change their behavior very quickly [2].

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 that one can 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 rate of interrupts based on the current number of commands in flight (CIF) from the guest OS. The rate, 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 for that guest. Note that $0 < R \leq 1$. Lower values of R denote higher degree of coalescing. We increase

R when CIF is low and decrease the delivery rate *R* for higher values of CIF. Unlike network IO, CIF can be used directly only for storage controllers because each completed request has a corresponding command in flight. Also as we show later, it is important to maintain certain number of commands in flight to efficiently utilize the underlying storage device. Many important applications issue synchronous IOs and delaying the completion of prior IOs can delay the issue of future ones.

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 might be running for notification purposes. As processor core density increases, it becomes more likely that hardware interrupts will be received on processors not running the target VM, thus increasing the number of times the IPIs need to be issued. We provide a mechanism to reduce the number of IPIs issued using the time-stamp 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 techniques in VMware ESX Server. Our experimentation with a set of micro-benchmarks and real workloads shows that our virtual interrupt coalescing (vIC) techniques can improve both workload throughput and CPU overheads related to IO processing by up to 18%.

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

2 System Model

Our system model consists of two components in the VMware ESX Server: ESX VMKernel and the virtual machine monitor (VMM). The VMKernel is a hypervisor server which is a thin layer of software controlling access to physical resources among virtual machines. ESX server provides isolation and resource allocation among virtual machines running on top of it. The virtual machine monitor is responsible for correct and efficient virtualization of the *x*86 instruction set architecture as well as common, high performance devices made available to the guest. 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 this, the VMKernel will take one of two actions. It will schedule the VM if it happened to have been 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 is purely an optimization to provide low latency for IO completions to the guest. For example, the guest might be mostly doing user space operations which would result in a long delay till the VMM naturally ends up taking execution control. 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 guest for every interrupt 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 can keep track of rate at which interrupts are being picked up by the VM monitor. All this needs to be done without the help of fine grained timers because they are prohibitively expensive in a hypervisor.

So the main challenges can be reduced to:

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

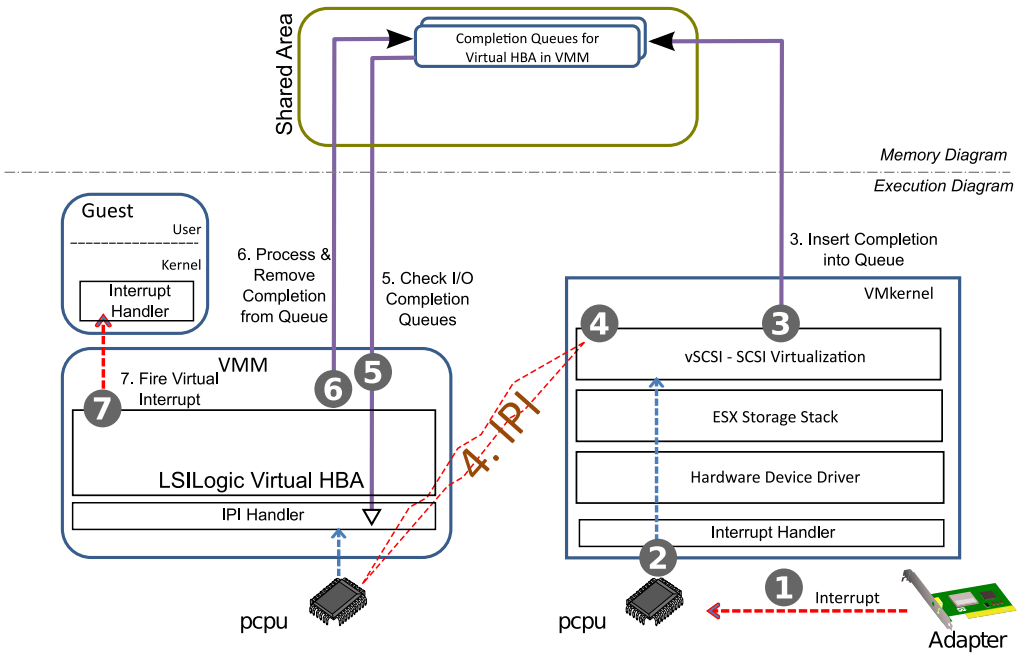

Figure 1: Virtual Interrupt Delivery Mechanism. When a disk IO completes, an interrupt is fired (1) from a physical adapter to a particular Physical CPU (PCPU) where the interrupt handler of the hypervisor delivers it to the appropriate device driver (2). Higher layers of the hypervisor storage stack 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 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).

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 higher 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 a high resolution timer is feasible in a dedicated IO processor 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 a hypervisor, we don't have access to such high resolution timing as a matter of course. 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 at least 100 μs resolution which would have prohibitive performance impact on real application workloads running in a guest OS (VM). As a comparison, VMware ESX typically sets up its timers to

go off anywhere between every 1 *ms* and 10 *ms*, up to two orders of magnitude less resolution than in hardware controller firmware.

3.2 Our approach

In our design, we define a parameter called interrupt delivery rate *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 rate implies a higher degree of coalescing. We dynamically set our interrupt delivery rate, *R*, in a way that will provide coalescing benefits for CPU and tightly control any extra vIC-related latency. This is done using commands in flight (CIF) as the main parameter and IOPS rate as a secondary control.

At a high level, if the IOPS rate 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 rate changes drastically or when the number of issued commands is very low. To control that 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 rate *R* in inverse proportion of the CIF value. Using both CIF values and estimated IOPS rate we are able to provide effective coalescing for a wide variety of workloads.

There are three main parameters used in our algorithm:

- iopsThreshold: IOPS rate 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 rate, in order to react to the change in the workload.

The algorithm operates in one of the three modes:
(1) **vIC low-IOPS:** We turn off vIC if the achieved throughput of a workload ever drops below the *iopsThreshold*. Recall that we don't have a high resolution timer. If we did, whenever it would fire, it would allow us to determine if we've held on to an IO completion 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 rate of 20,000 means that on average there will be a completion returned every 50 μs. Our default iopsThreshold is 2000 which 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) **vIC low-CIF:** We turn off vIC whenever the number of outstanding IOs (CIF) drops below a configurable parameter *cifThreshold*. Our interrupt coalescing 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, has no benefit and can only possibly add 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 interrupt coalescing rate (*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 achieved throughput. We discuss our solution next.

3.2.1 Dynamic Adjustment of Delivery Rate *R*

Which rate 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 can't 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 rate 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 have found that to be able to handle a multitude of situations where we deliver anywhere from 100% down to 6.25% of the incoming IO completions as in-

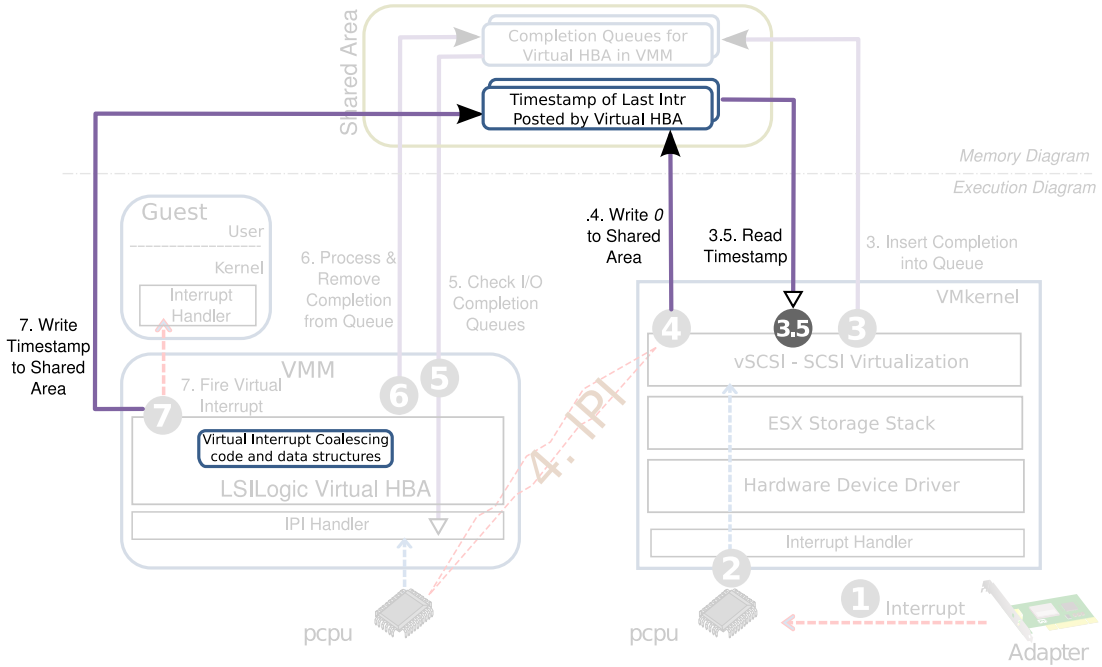

Figure 2: Virtual Interrupt Delivery Steps. In addition to Figure 1, vIC adds a new shared area object tracking the last time that the VMM fired an interrupt. Before sending the IPI, vSCSI checks to ensure that time since the last VMM-induced virtual interrupt is less than a configurable threshold. If not so, an IPI is still fired, otherwise, it is deferred. In the VMM, an interrupt coalescing scheme is introduced. Note that we didn't 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.

terrupts. 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.* $R = countUp/skipUp$. As illustration, 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 rates between 100% and 50%, we can dial down to a negligible level the throughput loss at smaller CIF.

The exact values of R are determined both based on experimentation and also to support the efficient implementation in a VMM. Figure 2 shows the additional new subsystem in the LSI Logic emulation code in the VMM. See Algorithm 1 for how the coalescing rate is determined. Next we will discuss the exact 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 pseudocode in Algorithm 2. First, 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 pseudocode 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 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.

In Algorithm 2, *counter* is an abstract number which 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's 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⟩.

Algorithm 1: Delivery Rate Determination

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

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 ⟵ currTime() − epochStart;
if diff > epochPeriod then
    IntrCoalesceRecalc(cif);
if cif < cifThreshold then
    counter ⟵ 1;
    deliverIntr();
else if counter < countUp then
    counter++;
    deliverIntr();
else if counter >= skipUp then
    counter ⟵ 1;
    deliverIntr();
else
    counter++;
    /* don't deliver */
if Interrupt Was Delivered then
    SharedArea.timeStamp ⟵ currTime();
```

CIF	Intr Delivery Rate R
1-3	100%
4-7	80%
8-11	75%
12-15	66%
CIF >= 16	8/CIF
e.g. CIF == 64	12%

Table 1: Interrupt Deliver Rate (R) as a function of CIF. cifThreshold is set to 4.

Whereas for $countUp/skipUp$ of $1/5$:
⟨1, no⟩, ⟨2, no⟩, ⟨3, no⟩, ⟨4, no⟩, ⟨5, yes⟩.

Algorithm 2 shows pseudo code for this per-IO decision. Finally, we update a time-stamp corresponding to the delivery time in a memory area shared between VMM and ESX VMKernel. Figure 2 shows this additional new operation (7) used to optimize the number of IPIs that are sent, as discussed in the next section.

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 heavy task which may not result in the VMM getting execution control till the next timer interrupt which might be several milliseconds away on average. 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 introduce 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 certain threshold, 100 μs by default, we

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

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

post the IPI. Otherwise, we give the VMM an opportunity to notice IO completions in due course on its own.

4 Preliminary Evaluation

To evaluate our vIC approach, we have examined several micro-benchmark and macro-benchmark workloads and compared against the existing case with no interrupt coalescing. In each case we have seen a reduction in CPU overhead, often associated with an increase in throughput (IOPS). A more comprehensive evaluation is being done as part of future work. For all of the experiments, the parameters are set as follows: $cifThreshold$ = 4, $iopsThreshold$ = 2000 and $epochPeriod$ = 200 ms.

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 test is a 4KB sequential IO reads with one worker thread running on a fully cached Logical Unit (LUN). In other words, IOs are hitting in the array cache instead of going to the disk. We varied the number of outstanding IOs to see the improvement over baseline. Table 2 shows the full matrix of our test results whereas Table 3 summarizes the percentage differences with and without coalescing. The column labeled R is the average rate 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 interrupt coalescing rate, 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 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%.

Our second workload is the same as the previous except we now used 8KB sequential IOs. In Tables 4 and 5

OIO	IOPS %diff	CPU cost %diff	Int/sec Guest %diff
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.

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

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

we see for the 64 CIF case the same interrupt coalescing rate of 1/6 with now a 7% efficiency improvement. The interrupt per second in the guest 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 our 8KB sequential read Iometer workload with 64 outstanding IOs we examined the breakdown between the VMM and guest OS CPU usage. In Table 6 we see our monitor's abridged kstats. The `VMK_VCPU_HALT` statistics is the percent of time that the guest was idle. We see 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 is measured by `device_Priv_Lsilogic_IO` measurement has decreased from 5.0% to 4.3%. The IO com-

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.

	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.

pletion work done in the VMM is part of a generic message delivery handler function and is measured by DrainMonitorActions in the profile. We see a slight increase from 0.5% to 0.7% of CPU consumption due to the management of the interrupt coalescing rate. 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 [5] 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 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.

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

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

5 Related Work

Smotherman [9] provides an interesting history of the evolution of interrupts and their usage in various computer systems starting from UNIVAC (1951). With increasing bandwidth of both network and 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 have been increasing to keep up with these devices, the motivation to reduce overall overhead of interrupt handling is still strong. Interrupt coalescing has been very successfully deployed in various hardware controllers to mitigate the CPU overhead. There is a lot of prior work both in terms of patents and publications with proposed mechanisms to perform interrupt coalescing both for network and storage hardware controllers.

Patent [7] 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. Maximum wait time parameter ensures a bounded delay on the latency of the packet delivery. Another patent [4] uses a single counter to keep track of 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, 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.

Marco et al. [10] 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. They provided and analyzed the relation-

ship between T_d and steady state TCP throughput.

Salah et al. [8] 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 a adaptive mechanism that can adjust to the current interrupt arrival rate and other workload parameters. Mogul and Ramakrishnan [6] 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 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. 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 don't have the luxury to change the guest behavior in terms of interrupts vs polling because the guest OS is like a black box to VMM.

6 Conclusions

In this paper, we studied the problem of doing 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 coalescing rate in fine-grained steps and to use future IO events to avoid the need of high-resolution. We also designed a technique to reduce the number of inter-process interrupts while keeping the latency bounded. Our prototype implementation in ESX server 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). As part of future work, we are looking into adapting our techniques to other devices such as NIC controllers and evaluating it for a more diverse set of workloads.

7 Some Open Issues

In addition to a more thorough evaluation, including against workloads with highly variable IO rates, there are several some open issues that we'd like to explore. Currently, we have hard-coded the best CIF-to-*R* mappings based on experimentation. We would like to explore dynamic adaptation of that mapping. We believe that OLTP workloads like TPC-C would benefit immensely from our optimization and would like to run and tune for some real benchmarks.

At first blush, networking controllers do not 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) don't know the semantics of higher layers. Still, there may be some inference techniques that can be applied to do aggressive coalescing without loss of throughput in context of high-bandwidth TCP connections.

Acknowledgements

We would like to thank Davide Bergamasco, Vincent Lin and Reza Taheri for help with experimental validation of our work. Many thanks to Ole Agesen, Mallik Mahalingham, Tim Mann, Glen McCready and Carl Waldspurger for reviews, valuable discussions and feedback.

References

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

[2] 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.

[3] X. Chang, J. Muppala, Z. Han, and J. Liu. Analysis of interrupt coalescing schemes for receive-livelock problem in gigabit ethernet network hosts. pages 1835–1839, May 2008.

[4] 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.

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

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

[7] C. Musumeci, Gian-paolo D. (San Francisco. System and method for dynamically tuning interrupt coalescing parameters. (US PTO 6889277), May 2005.

[8] 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.

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

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