Interrupt Coalescing in Xen with Scheduler Awareness

Michael Peirce and Kevin Boos

April 29, 2016

1 Introduction

Whole-system virtualization causes interrupt handling to be much more difficult than that of operating systems running directly on the hardware. In particular, interrupt delivery and handling is crucial for timer accuracy, I/O performance, and CPU efficiency, among other facets of the system. Substantial efforts have been undertaken to improve interrupt performance on virtual machines, including software techniques such as interrupt coalescing that combines multiple interrupts into one, and paravirtualization that reduces the number of interrupts and exceptions in the first place.

While robust solutions exist for interrupt-related timer problems, I/O performance remains one of the most significant challenges for virtualization. A variety of work has explored using interrupt coalescing to improve I/O performance for the networking stack [3] and block devices [1], but these works often have undesirable tradeoffs. For example, VMware's vIC work [1] does indeed improve block device throughput, but at the cost of increased latency.

In this paper, we describe our experience adding interrupt coalescing to the Xen virtualization system, which does not currently offer coalescing support. Furthermore, we improve upon vIC, the existing state-of-the-art coalescing policy for block devices, by combating its increased latency without reducing its increased throughput. We observe that vIC, despite its best efforts, often coalesces interrupts too aggressively and ends up delivering them to guest VMs at inappropriate times, causing high latency. Our key insight is that a coalescing policy can make better interrupt delivery decisions if it stays informed of each guest's runstate in the scheduler, enabling it to avoid delivering interrupts to guests that are not running. We refer to this as a *scheduler-aware* coalescing policy and explore the various design decisions and benefits of such a policy, guided by the following hypothesis.

Hypothesis

The objective of this work is to implement interrupt coalescing for block devices in Domain0 of the Xen virtualization system. Our hypothesis for the expected results of this implementation has two distinct parts:

1. We believe that by adding the coalescing policy described in VMware's vIC paper to Xen, we can achieve increased block I/O throughput with a minor increase in latency.

2. Furthermore, we posit that adding scheduler awareness to the above coalescing policy will reduce its increase in latency without sacrificing much of the corresponding throughput gain.

In the next section, we give an overview of Xen's device model and describe how block drivers pass interrupts between guest domains through event channels. In Sections 3 and 4, we detail how we implemented coalescing similar to vIC and coalescing based on scheduler-awareness, respectively. Section 5 presents how we evaluated the performance of these policies, and Section 6 concludes the paper with a discussion of future work.

2 Background: Xen Block Driver Model

The Xen Project [2] is a virtualization system that subscribes to the microkernel philosophy: a small privileged hypervisor multiplexes hardware access among multiple guest operating systems, with all other features implemented outside of the hypervisor. Xen adopts a unique driver model in which device drivers live in their own separate domain, commonly referred to as Domain0 or *dom0*, not in the guest *domU* domains and certainly not in the hypervisor.

In this paper, we examine and focus our efforts on Xen's block driver architecture, though other driver subsystems follow a similar model. Figure 1 provides a high-level view of block drivers in Xen. Xen employs a split driver model in which the legacy block device driver lives entirely in dom0, prevent-

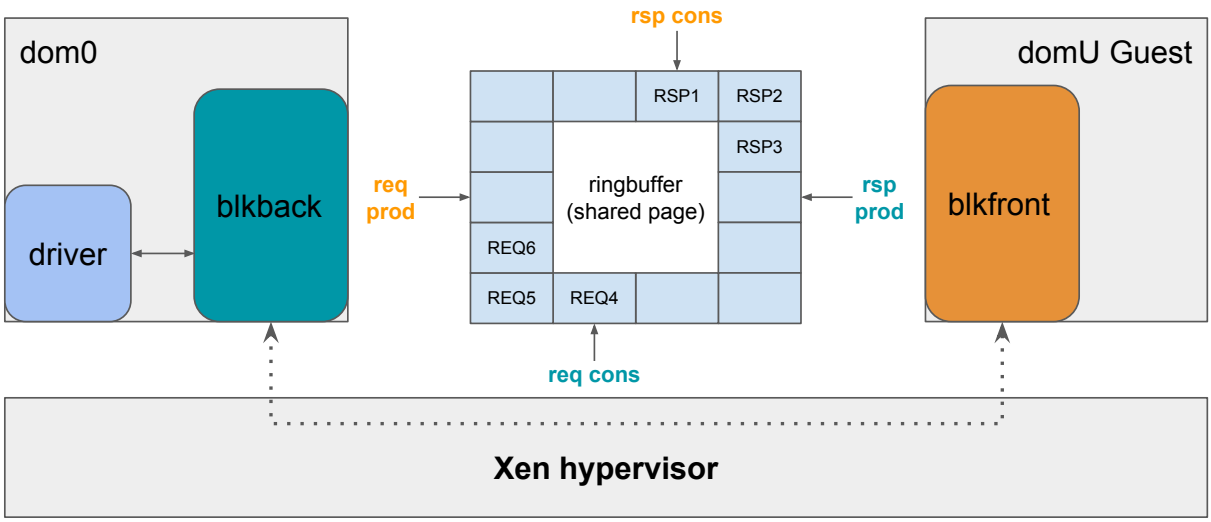

Figure 1: A high-level view of block drivers in Xen.

ing guests from directly accessing the device. Instead, guest domains issue block I/O requests to a paravirtual frontend driver, `xen_blkfront`, which forwards the requests to the `xen_blkback` backend. The backend then communicates directly with the native device driver to perform the actual I/O operation.

In Xen's split driver model, the frontend and backend communicate asynchronously via a ring buffer stored in a shared page that is mapped into both domains. [4] The ring buffer essentially comprises two FIFO queues: one for block I/O requests and one for responses. The frontend produces block I/O requests and consumes the corresponding responses from the backend; the backend does the inverse, consuming requests from the frontend and producing responses whenever it receives a completion event from the actual driver. Thus, the frontend manages the *request producer* and *response consumer* pointers into the ring buffer, and the backend manages the corresponding *request consumer* and *response producer* pointers (Figure 1). However, both the frontend and backend can read (but not write) the pointers managed by each other in order to carefully avoid overwriting any pending requests or responses.

Interrupts occur when the frontend has produced a request and needs to inform the backend of that pending request, or when the backend has pushed a response onto the queue and wants the frontend to acknowledge the completion of that block I/O event. For example, if the ring buffer is full, the frontend may issue an interrupt telling the backend to pop some requests off of the ring buffer before it can submit more. In normal execution, an interrupt is sent from the backend to the frontend whenever there are pending completion responses and the frontend is not currently processing them.

Xen implements these interrupts with event channels, analogous to sockets, between the frontend and the backend. As of this writing, Xen does not support interrupt coalescing in any part of its driver model, either from the backend or from the frontend[1]. In this paper, we focus exclusively on coalescing interrupts in the backend of dom0 as they are received from the device driver and routed to the appropriate guest domain. We are only concerned with managing the delivery of interrupts outgoing from the backend in dom0 that inform the frontend of newly available block I/O completion responses.

3 Conventional Coalescing Design

The coalescing policy described in the vIC paper is based around the number of commands in flight (CIF) and IO operations per second (IOPS) [1]. When these values reach a certain threshold, it makes sense to coalesce interrupts. The delivery ratio, R, is chosen based on how many commands in flight there are compared to the threshold. The ratio dictates how many interrupts to coalesce into a single interrupt to be forwarded.

The backed driver in dom0 stores the information related to a each domain in a `struct blkif`. This structure already had a field called `inflight` for

[1]We note that Xen does offer support for configuring hardware coalescing in most NICs, but software coalescing in other parts of the system is wholly absent.

Algorithm 1: ShouldSendNow

Input: *ci*, the coalescing information related to this domain; *cif*, the number of commands in flight; *endTime*, when the current time slice is expected to end, in nanoseconds

Output: *true* if the interrupt should be sent now, or *false* if it should be coalesced

1 $now \leftarrow$ the current time in nanoseconds;
`/* endTime = 0 if we do not know when the current timeslice ends */`
2 **if** $endTime \leq 0$ **then**
3 $\quad remaining \leftarrow -1$;
4 **else**
5 $\quad remaining \leftarrow endTime - now$;
`/* Check for end of timeslice */`
6 **if** CurrIops(ci) $\neq 0$ **then**
7 $\quad nsecPerIo \leftarrow \frac{1,000,000,000}{\text{CurrIops}(ci)}$;
8 $\quad$ **if** SkipUp(ci) $< 2 \cdot$ CountUp(ci) **then**
9 $\quad\quad ioPerEvent \leftarrow 2$;
10 $\quad$ **else**
11 $\quad\quad ioPerEvent \leftarrow$ SkipUp(ci);
12 $\quad nsecPerEvent \leftarrow nsecPerIo \cdot ioPerEvent$;
13 $\quad$ **if** $0 < remaining < nsecPerEvent$ **then**
14 $\quad\quad$ **return** *true*;
`/* This is all vIC-style coalescing */`
15 $epochSoFar \leftarrow now -$ EpochStart(ci);
16 **if** $epochSoFar < epochPeriod$ **then**
$\quad$ `/* Described in vIC paper */`
17 $\quad$ CoalesceRecalc(ci, cif);
18 $\quad$ SetEpochStart(ci, now);
19 **if** $cif < cifThreshold$ **then**
20 $\quad$ SetCounter(ci, 1);
21 **else if** Counter(ci) $<$ CountUp(ci) **then**
22 $\quad$ SetCounter(ci, Counter(ci) + 1)
23 **else if** Counter(ci) $\geq$ SkipUp(ci) **then**
24 $\quad$ SetCounter(ci, 1);
25 **else**
26 $\quad$ SetCounter(ci, 1);
27 $\quad$ **return** *false*;
28 **return** *true*;

```
struct coalesce_info {
  unsigned short count_up;
  unsigned short skip_up;
  unsigned short counter;
  long long epoch_start;
  unsigned int curr_iops;
  atomic_t next_iops;
  spinlock_t recalc_lock;
};
```

Listing 1: The data structure of coalescing details added to each domain's blkif structure.

keeping track of the number of commands in flight for that particular domain, so we did not need to add it ourselves. For the remaining parameters, we created a `struct coalesce_info` that we added to the `struct blkif`, as shown in Listing 1.

The `count_up`, `skip_up`, and `counter` fields are all directly associated with their uses in the vIC paper. The `epoch_start` field keeps track of when the last recalculation epoch started, in nanoseconds. The `curr_iops` field gives the measured IOPS during the last epoch, while the `next_iops` gives the measured IOPS so far in the current epoch. While the names do not make much sense in that context, they make sense when one considers how they are used. When deciding the coalescing policy for the current epoch, we use the calculation from the previous epoch. The `recalc_lock` is used to prevent multiple threads from trying to recalculate the parameters at the same time, which would cause `curr_iops` to be set to near-zero.

We use these parameters in the exact same manner as described in the vIC paper. We also included configurable parameters for `iops_threshold`, `cif_threshold`, and `epoch_period`. We defaulted to most of the same values as the paper, with a epoch period of 200ms and a cif threshold of 4, though we moved the IOPS threshold to 100 instead of 2,000 because our system is much smaller than the datacenter-class servers used by VMware.

The algorithm used is shown in lines 15 onward in Algorithm 1

4 Scheduler-aware Coalescing

The second aspect of our hypothesis proposes that adding scheduler awareness to our coalescing protocol will reduce the latency of traditional coalescing, i.e., vIC, while maintaining the same level of increased throughput. The reasoning behind this is that vIC's coalescing policy is oblivious to a given guest's behavior, so it may end up delivering a poorly-timed interrupt, such as when that guest is

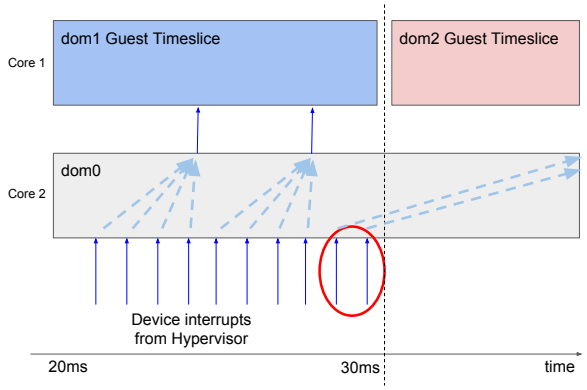

Figure 2: Conventional interrupt coalescing (vIC) sometimes results in poorly-timed interrupt delivery, causing huge interrupt latency particularly towards the end of a guest's timeslice.

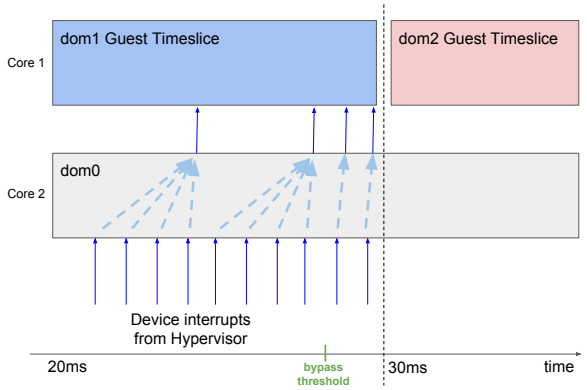

Figure 3: Scheduler-aware coalescing reduces latency by delivering more interrupts towards the end of a guest's timeslice.

not running.

Figure 2 shows an example of such poorly-timed interrupts. Towards the bottom of the figure, the device interrupts coming from the hypervisor are all destined for the dom1 guest, and we coalesce them according to the aforementioned vIC policy. In this example, the current IOPS and CIF parameters have resulted in a 0.25 delivery ratio, meaning that four interrupts must be received by dom0 before one interrupt is sent from dom0 to dom1. As such, the first four interrupts are delivered (after a minor coalescing delay) while dom1 is still running, as are the next four. However, the final two interrupts are not delivered to dom1 before it stops running, as the current coalescing policy dictates that four interrupts must be received before one can be sent. The remaining two necessary interrupts are not received until sometime in the future, e.g., during another guest's timeslice when dom1 is not running. Therefore, the two interrupts that *did* arrive while dom1 was running, circled in red in Figure 2, will experience high latency — they cannot be delivered to dom1 until its next timeslice is underway, after dom2 finishes its own timeslice.

We aim to reduce the delivery latency of such interrupts by equipping our coalescing decision algorithm with the scheduling details of guest domains. When dom0 is aware of a guest's progress through its timeslice, it can take this into account in order to avoid delivering an interrupt at an inopportune time. Specifically, towards the end of a guest's timeslice, we force delivery of interrupts at a higher delivery ratio than that specified by vIC; in other words, we send interrupts no matter what vIC's parameters would recommend. As seen in Figure 3, this results in an overall latency reduction as compared to vIC-style coalescing.

We initially attempted to eschew the vIC policy entirely in favor of a separate policy based on scheduler information alone. This took the form of a graduated increase in interrupt delivery ratio as the timeslice progressed. At the beginning of the timeslice, we start with a small delivery ratio to coalesce more interrupts, ideally reducing the overhead of many unnecessary interrupt deliveries since we expect additional interrupts to arrive during the later portions of the timeslice. Towards the middle of the timeslice, we gradually increase the ratio to coalesce fewer and fewer interrupts, and then disable coalescing to deliver every interrupt immediately near the end of the timeslice.

However, we found that ignoring the I/O-specific parameters that factor into vIC's policy ultimately proved detrimental to the overall throughput, as the above timeslice-based decisions were too coarse-grained to outperform vIC. Thus, we adopt a hybrid approach that reverts to the conventional vIC policy for the beginning and middle of the timeslice, and then delivers more interrupts towards the end of the timeslice regardless of vIC's delivery ratio. In the latter case (end of timeslice), the only information used from vIC is the IOPS rate and delivery ratio, which help define the cutoff threshold for bypassing coalescing, as discussed below. Though this hybrid approach is currently more beneficial than a separate scheduler-only policy, in the future we could revisit this with different policy settings, systems configurations, and test cases.

The cutoff threshold for bypassing interrupt delivery can either be statically defined, e.g., "always deliver interrupts when there is less than 1ms re-

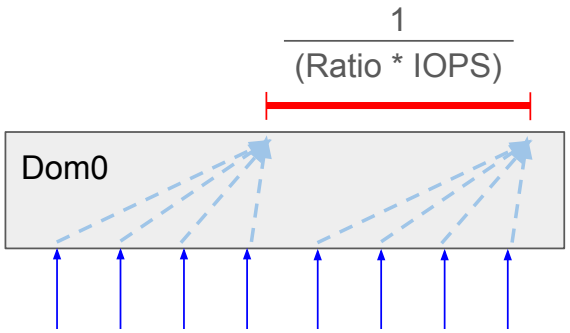

Figure 4: We dynamically calculate the coalescing bypass threshold as the time interval between when the vIC policy would deliver coalesced interrupts.

maining in the timeslice," or better yet, dynamically calculated based on parameters used in vIC. We use the following equation to dynamically choose an appropriate threshold after which interrupts will be immediately delivered (no coalescing):

$$t_r < \frac{1}{R \cdot IOPS}$$

in which t_r is the time remaining in the current guest's timeslice, $IOPS$ is the I/O operations per second as previously defined, and R is the delivery ratio defined by vIC. The logic behind this equation is that the fraction on the right side represents the time interval between two adjacent interrupt delivery events that would occur according to vIC's policy. Figure 4 further illustrates this time interval and demonstrates that we chose this policy because it adapts to the current vIC parameters. This ensures that we avoid spurious interrupts by not activating the "always-deliver policy" too soon, i.e., when vIC's policy would have delivered the interrupt anyway before the timeslice ended. This equation is a slight oversimplification, however. When $1/2 < R < 1$, interrupts can either be paired or sent alone. In this case, we assume the worst — that this interrupt would be paired, and that the next interrupt will not happen in time – and thus replace R in this equation with $1/2$. As such, a dynamic coalescing bypass threshold maximizes throughput by minimizing the number of non-vIC interrupts, yet minimizes latency by guaranteeing that interrupts necessarily bypass the vIC policy at the end of a guest's timeslice.

The algorithm for implementing scheduler-awareness is shown in lines 1 through 15 in Algorithm 1

	Pros	**Cons**
Hyper-call	Info generated on demand; easy to implement.	High overhead, long hypercall latency causes stale info.
Shared Page	Info is fresh and quickly available via memory access.	Info is constantly updated even if unused; difficult to implement.

Table 1: A comparison of implementation choices in exposing scheduler information from the hypervisor to dom0.

```
struct shared_info {
  struct vcpu_info vcpu_info[MAX_VCPUS];
  ulong evtchn_pending[8];
  ulong evtchn_mask[8];
  uint wc_sec, wc_nsec;
  struct arch_shared_info arch;
  struct shared_sched_info sched_infos[32];
};

struct shared_sched_info {
  int domain_id;
  int runstate;
  long timeslice_end_time;
  int latest_vcpu_id;
};
```

Listing 2: The data structure stored in each domain's shared page, with our additional structure containing shared scheduler information (in blue).

4.1 Exposing Scheduler Information from the Hypervisor to Domain0

When designing an interface to allow dom0 to access the guests' scheduling details maintained in the hypervisor, we considered two options as described in Table 1. For the reasons listed in the table, primarily the need for very fresh, constantly updated scheduler data, we chose to expose hypervisor scheduler data via a memory page shared with dom0 rather than a hypercall.

When Xen boots up a new guest domain, the hypervisor allocates a new memory page shared with that domain that includes one instance of the `struct shared_info` shown in Listing 2. This structure includes virtual processor information, interrupt masks and bit vectors for polling, wall-clock time synchronization details, architecture-specific information, and miscellaneous boot information (omitted from listing). The newly-booted guest domain will utilize the information in the shared page to initialize itself, and this information is only available to the guest's kernel. Therefore, it is trivial for

the backend block driver in dom0's kernel to quickly access any information in the shared page, making it the ideal medium for dom0-hypervisor communication.

Instead of wastefully allocating a new shared page just for a few bytes of scheduler information, we append it to the existing shared page because the `shared_info` structure only occupies about 2/3 of a standard 4KB page. There exists one instance of `shared_sched_info` per domain, which contains that domain's unique Xen-assigned ID, its runstate (e.g., running, blocked, stopped), the wall-clock time at which its current timeslice will end, and the ID of the virtual CPU on which it last ran. We choose wall-clock time as the synchronized clock because it is available in both guest domains and in the hypervisor. Alternative time values like boot time and CPU time are not directly exposed to guest domains, including dom0, because that would violate the isolation between the hypervisor and guests.

In order to update the scheduler information in the shared page, we instrument the `schedule()` routine in the hypervisor, which executes any time the hypervisor needs to context switch between guest domains. This results in very fresh information always readily available to dom0 for its coalescing decisions because `schedule()` is invoked very often, updating the scheduling data 2-3 orders of magnitude more frequently than dom0 uses it. Though this shared page exists in all domains, both dom0 and guest domUs, we ensure that the hypervisor only writes to dom0's copy of the shared page. This prevents the massive security hole of private scheduling information leaking into untrusted guest domains.

An interesting challenge we encountered involved a lack of clock synchronization between the hypervisor and dom0. Initially, we observed clock error on the order of dozens of milliseconds, meaning that dom0's understanding of the current wall-clock time severely lagged behind that of the hypervisor. This caused interrupts to be delivered at the wrong time, as the coalescing mechanism in dom0 thought there was still a few milliseconds remaining in a guest's timeslice, when in reality the hypervisor had already scheduled out that guest domain several milliseconds prior. Unfortunately, we were unable to influence the hypervisor's view of wall-clock time because it typically reads such values directly from a hardware register; thus, we were relegated to modifying dom0's clock. Through a combination of updating NTP and selecting different clock sources for dom0, i.e., `tsc` instead of the default `xen` clock source, we achieved a synchronized clock with fewer

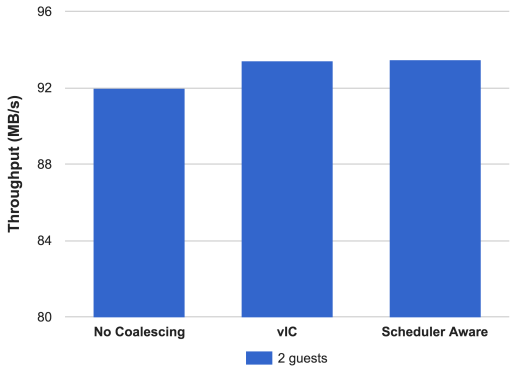

Figure 5: Throughput when running `dd` on one guest while one other guest runs a CPU-intensive infinite loop.

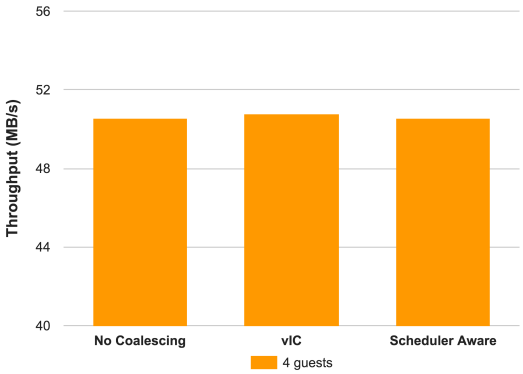

Figure 6: Throughput when running `dd` on one guest while three other guests run a CPU-intensive infinite loop.

than 200μs of error, more than accurate enough for our remaining timeslice estimation. The coalescing policy takes this into account by refraining from influencing vIC's policy if the remaining timeslice is within the margin of error.

Another challenge is that sometimes a guest is not actually scheduled out when its timeslice ends. If there are no waiting guests to take its place, it just continues running, though its `runstate` is switched to "stopped" and its expected `end_time` is not updated. When this happens, we forego scheduler-aware coalescing and revert back to regular coalescing. At that point, there is a decent chance that no other guests are competing for this CPU (e.g., if this guest alone has been pinned to a particular CPU), so there is no purpose in trying to guess when its extended timeslice will end.

Figure 7: Throughput when running `dd` on both guests simultaneously, with both guests also running a CPU-intensive infinite loop.

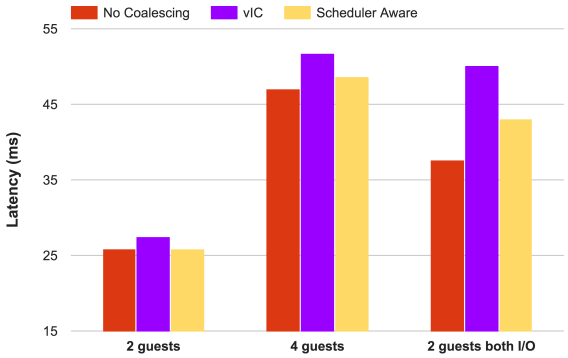

Figure 8: The end-to-end latency from when a block I/O request is submitted from the frontend to when the associated response is received back by the frontend. Each column group corresponds with the above three throughput tests.

5 Experimental Evaluation

5.1 Evaluation Setup

For all experiments described herein, we pin dom0 to two CPU cores and reserve those cores for dom0 alone, simply to ensure that anything executed in dom0 is not CPU-bound. All other guests are pinned to a single shared core to force scheduling conflicts. This also simulates the high-density environment typically found in today's virtualization servers, in which a large number of guests are contending for time on a smaller number of CPUs.

We conducted our experiments by running `dd` from inside a guest, which reads from a file and writes the contents to another file in small chunks. We choose to read from `/dev/zero` and write to a new file in the `root` directory. This meant that the reading performed by `dd` did not need to go to disk, allowing us to measure the effect of coalescing on writing independently from its effect on reading.

All benchmarks were performed on each of the three stages: no coalescing, coalescing using the vIC policy, and coalescing with additional scheduler awareness.

5.2 One Guest Performing I/O

To measure performance when only one guest performs IO, we ran two or four guests. All but the first guest ran a simple busy loop to fully maximize CPU utilization. The first guest ran `dd` to move 1 GB from `/dev/zero` to a local file repeatedly. We ran this for fifty iterations on two guests, and twenty iterations on four guests.

The throughput results for two guests are shown in Figure 5, and the latency results are in the furthest left column group of Figure 8. We can see that with the addition of coalescing, the throughput increased by 2.5%, and the latency increased by 6.7%. When scheduler-awareness was also added, the throughput remained at the same higher level that coalescing achieved, while the latency was reduced to that which was achieved with no coalescing.

The results for four guests are shown in Figure 6 and the middle of Figure 8. The throughput from all three policies are virtually the same, while the regular coalescing policy has a 10% increase in latency and the scheduler-aware coalescing policy has a 3.5% increase in latency.

5.3 Multiple Guests Competing for CPU

Additionally, we ran benchmarks where two guests each ran two processes: `dd` as described before and an infinite CPU-intensive loop. This caused more competition for the CPU within each guest's timeslice. The results from this test are shown in Figure 7 and the right-most column group of Figure 8. In this benchmark, regular coalescing increased throughput by 6.1%. Scheduler-awareness increased the throughput by 10% over the original, despite our prediction that it would slightly decrease it. As for latency, regular coalescing increased latency by 33% and scheduler-aware coalescing increased it by only 15%, an overall 14% reduction in latency compared to vIC.

5.4 Interrupt Interjection

Before we actually ran the above evaluations, we were uncertain whether we would be able to see the effects of our scheduler awareness policy. If the time period that we considered to be "at the end of the timeslice" was too small, the impact on our evalu-

ation could end up being statistically insignificant for any practical number of trials.

To prepare for that scenario, we added to dom0's back-end driver the ability to indefinitely delay both requests and responses. Then, instead of forwarding them to the disk to perform the read or write operation, the driver sets a timer for when to send back an artificial response. By using information gained from the scheduler, the driver could set the timer to send back the response right when the time slice is ending, thus letting us measure performance effects of a last-minute interrupt flood. This proved to be unnecessary, as we were able to see the impact of coalescing from our ordinary tests using real-world tools like `dd`.

6 Conclusion

In summary, we have verified both aspects of our initial hypothesis. First, we have shown that adding conventional interrupt coalescing (vIC) to Domain0 tends to increase the throughput and latency. Second, the addition of scheduler awareness consistently decreases the latency of interrupts, and may actually increase the throughput rather than decrease it.

We did not expect scheduler awareness to result in any throughput improvements, but it is a pleasant surprise. Our best explanation of this phenomenon is that when interrupts are coalesced and not sent until the next timeslice, the application often treats them as "timed out" and discards the completion event. When the guest starts running again, it realizes that it has not yet received those interrupts, and sends out repeated requests for them. The scheduler-aware coalescing algorithm avoids this in most cases, reducing the frequency of re-issued I/O requests and thus increasing throughput.

The modifications made to Domain0's kernel can be found in our GitHub repository at `https://github.com/mongoose700/xen-coalesce-kernel`. Our modifications to the Xen hypervisor are located at `https://github.com/mongoose700/xen-coalesce`.

Future Work

None of the benchmarks that we ran would have been impacted by the IOPS threshold below which no coalescing is done, as they were all designed to have a high throughput. Further benchmarks on a wide variety of applications that we did not manage to do would allow for this.

Additionally, the calculation of current IOPS itself should be modified to be scheduler-aware. As currently calculated, it does not take into account the fact that the guest may have been scheduled out for a large portion of the last epoch. Ideally, the driver would measure IOPS on a guest-specific basis, only when a particular guest is running.

Guests may also desire to configure the parameters (`cif_threshold`, `iops_threshold`, and `epoch_period`) separately. Currently, the values are set once for all guests in the kernel of dom0. To enable per-guest parameters, the guests could each send their desired parameters along with their requests, though this would consume space and reduce (barely) the number of requests and responses that could be put in the buffer.

We have only implemented and tested scheduler awareness for the default scheduler in Xen, the Credit scheduler. However, our implementation is scheduler-agnostic and can be easily applied to other schedulers, such as Xen's Borrowed Virtual Time and Simplest Earliest Deadline First schedulers. It would indeed be interesting to explore how interrupt coalescing is affected by other classes of schedulers, such as preemptive ones, in which it is more difficult to accurately predict exactly when a guest domain will be scheduled out.

References

[1] I. Ahmad, A. Gulati, and A. Mashtizadeh. vic: Interrupt coalescing for virtual machine storage device io. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 4–4, Berkeley, CA, USA, 2011. USENIX Association.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[3] Y. Dong, D. Xu, Y. Zhang, and G. Liao. Optimizing network i/o virtualization with efficient interrupt coalescing and virtual receive side scaling. In *2011 IEEE International Conference on Cluster Computing*, pages 26–34, Sept 2011.

[4] A. Warfield. How the blkif drivers work. `http://compbio.cs.toronto.edu/repos/snowflock/xen-3.0.3/docs/misc/blkif-drivers-explained.txt`. Online; accessed 16 April 2016.