# Fip-see: A Low Latency, High Throughput IPC Mechanism

Scott Bauer

October 27, 2016

## 1 Introduction

Modern storage stacks designed by NetApp, EMC, provide complex software designed to provide data replication, versioning of data, caching of data, etc . In a modern virtual datacenters the storage stack can be colocated with applications. If the storage stack was written for a different operating system it can no longer be colocated with the application.

We explore and implement a novel cross-VM communication mechanism, we have the storage stack sit in a Xen virtual machine and the customer's operating system in another VM. This allows storage stacks to be colocated with applications running on different operating systemes, but opens up a series of performance issues as cross-VM communication can be extremely expensive.

For low latency high throughput NVMe devices the Xen paravirtualized multi-queue enabled block driver is a bottleneck and has undesirable throughput and latency. The throughput and latency will soon be such a bottleneck on faster devices that colocation through VMs won't be a feasable option.

We design, implement and evaluate Fip-see a low-latency, high throughput IPC mechanism. We replaced the legacy Xen ring buffer IPC that is currently used in the paravirtualized block driver with Fip-See. We argue that its faster and cheaper to slightly modify the block drivers for a few operating systems than completely re-writing the storage solution.

With Fip-see and a hack to share memory across virtual machines, it gets near host OS performance in throughput and latency. Fip-see achieves 90% of the read IOPs you would get running by running on the host OS, 89% of the write IOPs. With latency Fip-see brings us within 8 microseconds of host performance. More importantly, Fip-see has significant throughput and latency improvements over the original multi-queue enabled paravirtualized block drivers. In the best tests Fip-see provides 583k more IOPs a 3.38x throughput increase over the original driver and a reduction in latency of 28.57 microseconds.

## 2 Fip-see Design

**Fip-see Transport Medium**

Fip-see is built using two ring buffers sitting in coherent memory. The two ring buffers are called an IPC Channel. Each ring buffer is an order of 2 sized pages, thus a ring can be 2, 4, 8, 16 pages. I typically select a ring size of 4 pages. The ring is then split into a series of 64 byte IPC messages which is strategically selected to be the size of a cache line on x86 processors.

Each ring is uni-directional meaning the ring either only has requests or responses, not both. Say we have an IPC Channel and name the internal rings A and B, if a process wants to call function foo it will marshal all the required information into an IPC message and place it in ring buffer A. When a response is ready the response will be marshaled into an IPC message and placed in ring buffer B. Requests and responses are transferred on different rings. This design alleviates some cache coherence issues as well as maintenance issues which cause performance loss.

### IPC Channel Maintenance data

In Xen's IPC mechanism when a thread wants to send data to another domain it checks a shared cache line where a producer and consumer counter are located. Before the message can be written the code must assert there is room in the ring by reading the producer and consumer counter. If the consumer counter is equal to the producer counter all messages have been read by the other process and there are open slots available. If the producer and consumer counter differ by the total size of the ring then it is full. Once a message slot is available the thread will write a message and increment the producer counter stating there is a new message available. On the inverse side, the consumer will read the two counters and subtract the producer from the consumer the difference is the amount of available messages. This design has some faults which cause performance loss. Specifically there can be significant amounts of cache thrashing in the cache line that holds the shared counters. Each side of the IPC ring can at the same time be writing data into the ring, causing updates to the producer and consumer counters. Moreover, in the case of writing a message two sets of cache coherence traffic gets generated. One set of coherence traffic is generated to read and update the counters, and another set to fetch the message and write data to it.

In Fip-see there is no shared metadata for the ring. Each ring has its own private copy of data which describe the size of the ring, where in virtual memory it resides and the last message slot used. To denote whether a slot in the ring is available for consumption or is free to have data written into it we place a flag at the end of the cache-line sized IPC message. Instead of having a shared location that describe what slots are free and what slots are in use we bake the information directly into the IPC messages.

When a message is placed into the ring the thread will increment its private slots used variable and place a *MSG_READY* flag into the end of the IPC message. When the other thread is done using the data in the IPC message it will place a *SLOT_FREE* flag in the end of the message.

### Fip-see Notification Mechanism

Each pair of ring buffers has a dedicated receiving thread which we pin to a specific core. For each IPC Channel there will be two threads pinned to dedicated cores. Each side will be waiting for messages on their rx buffers.

Once a message is placed in the ring the transmitting side wants to notify the receiving end that a message is available for consumption. In Xen's IPC mechanism once the message is available for consumption the shared counter will be updated and a hypercall into Xen will occur. Xen will then send an inter processor interrupt (IPI) to the receiving VM. Once the interrupt gets to the handler on the other VM the code will toggle work ready on a kernel work queue or handle the message directly in interrupt context.

Fip-see's notification mechanism is designed to eliminate costly hypercalls into Xen and slow IPIs across cores or sockets. To do this we use an always on receiving thread which constantly looks for new messages. The hypercall and IPI were required to "transfer control" to another VM so it could schedule work or handle the message immediately. In Fip-see since the thread is always running and constantly looking for new messages no control transfer is required, eliminating thousands of cycles of latency.

### Putting it all together

For each Fip-see channel there are two threads pinned to specific cores that will busy wait loop on the current free message slot. When it notices the flag change from *SLOT_FREE* to *MSG_READY* it then knows it can unmarshal the data from the message and issue the proper command and response. Figure 1 Shows the legacy IPC mechanism and Figure 2 shows Fip-see.

## 3 Cache Coherence and Limits of Cross-Core Communication

### Cache coherence refresher

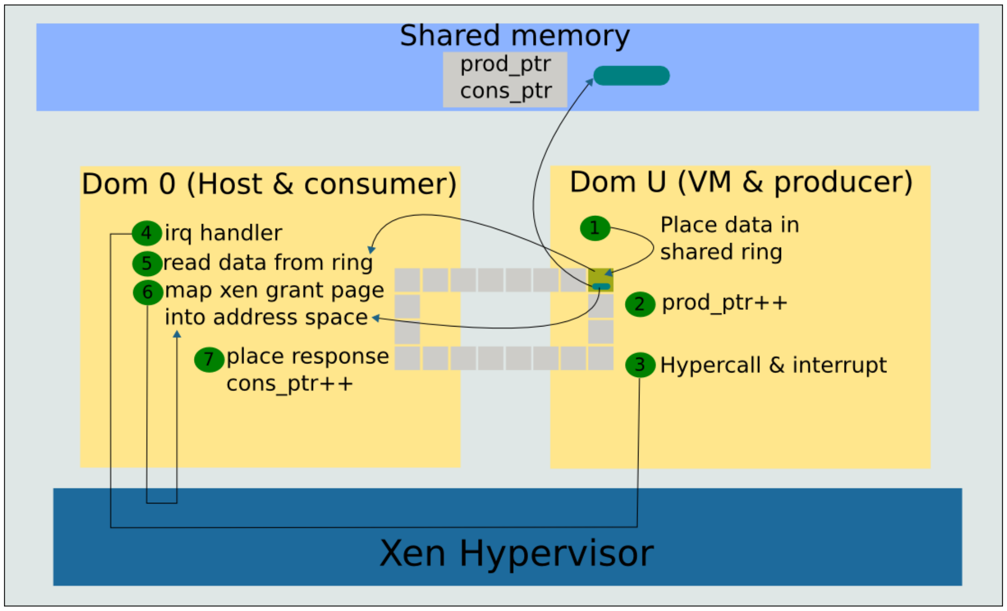


Figure 1: Legacy Xen Block IO IPC

To understand why Fip-see is fast we must understand the cache coherence traffic generated. Obviously processors have caches which reduce costly accesses to main system memory. The design of the caches with private L1 and L2 present an interesting problem of keeping data on multiple cores and sockets in a consistent state. Intel provides a consistency model MESI (Modified, Exclusive, Shared, Invalid) and MESIF (Forward) on NUMA systems.

1. A modified cache line is exactly what it sounds like, a cache line that a core has written fresh data to it. When a cache line is in the modified state all other copies of the cache line on other cores are invalid.

2. An exclusive cache line is an unmodified cache line that is only present in one core's L1 or L2 cache. The exclusive state can be achieved by issuing the *WBINVD* (Write Back Invalidate) instruction and re-reading the memory location. Or if all other caches evict their copy of the cache line the last remaining core with the line will have it in exclusive.

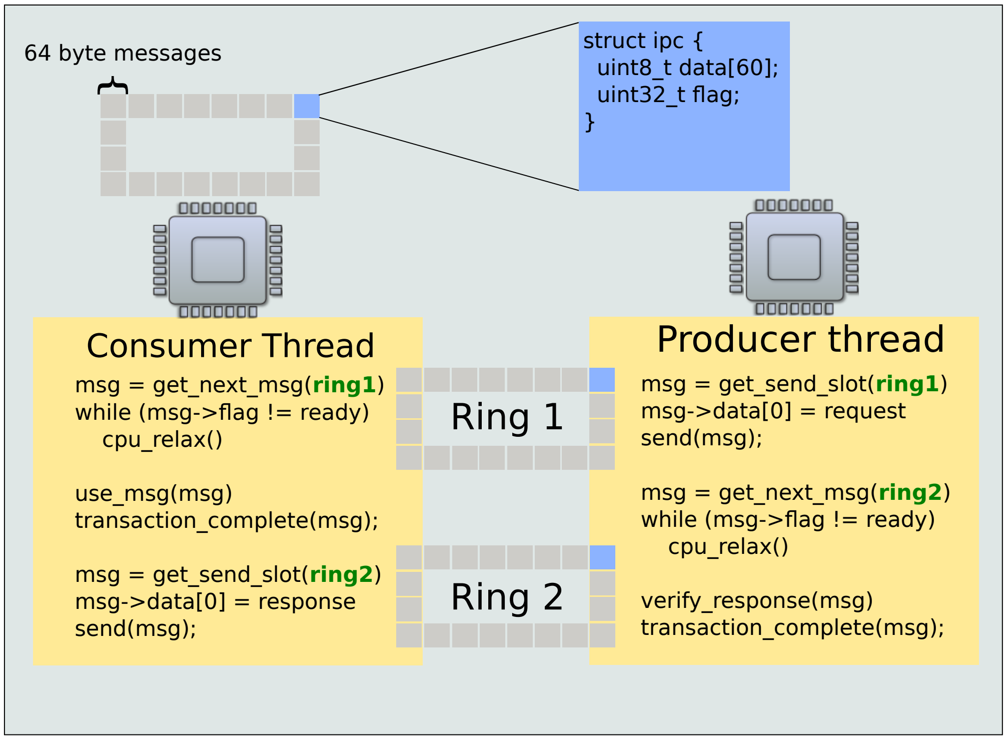
3. A shared cache line, like its name is a cache line that is resident in two or more core's L1 or L2 caches.

4. An invalid cache line is a line that has incorrect data. Any subsequent reads or writes to the cache line will generate coherence traffic and a read to memory to get the most up-to-date value.

5. A forward cache line is a special addition to the coherence model for NUMA systems. In order to prevent large amounts of coherence traffic on the QPI Link connecting multiple processors the forward state is used. When 2 or more cores have a cache line one core is designated the forwarder of the cache line. If any core attempts to read from the shared cache line instead of all cores responding, clogging the QPI Link only the forwarder will transmit the line.

**Internals of call/reply invocation**

The minimal send/receive or call/reply communication pattern consists of the following steps (Figure 3). The consumer sends a message to the producer by updating a 60 byte message and the 4 byte flag in the "call" cache line that is shared between consumer and producer. The flag is used to signal that the message is ready.



```
struct ipc {
  uint8_t data[60];
  uint32_t flag;
}
```

Consumer Thread:

```
msg = get_next_msg(ring1)
while (msg->flag != ready)
   cpu_relax()

use_msg(msg)
transaction_complete(msg);

msg = get_send_slot(ring2)
msg->data[0] = response
send(msg);
```

Producer thread:

```
msg = get_send_slot(ring1)
msg->data[0] = request
send(msg);

msg = get_next_msg(ring2)
while (msg->flag != ready)
   cpu_relax()

verify_response(msg)
transaction_complete(msg);
```

Figure 2: Fip-see Synchronous IPC

The producer waits for incoming messages from the consumer in a busy-wait loop that checks the state of the message flag in the message cache line. The moment the producer notices the change of the message flag from the consumer it replies to the consumer by updating the reply message and its flag in the “reply” cache line.

After sending the message, the consumer immediately proceeds to a busy-wait loop waiting for the reply message from the producer. The busy-wait loop polls on the message flag in the “reply” cache line.

**Cache coherence for call/reply**

What are theoretical limits of performing the call/reply invocation outlined above? To answer this question we need to understand behavior of cache coherence protocol on Intel architecture. On a single-core Intel implements a version of a snoop based protocol in which coherence traffic is sent in broadcast messages to all other cores.

There are two different scenarios we must model for the coherence traffic. First scenario is an initially empty ring buffer, meaning we’ve never wrapped on the ring buffer. The second scenario is when we’ve written enough message to wrap around to the start of the ring buffer.

## 3.1 Coherence traffic of an initially empty ring:

1. As the consumer is constantly polling on the “call” cache line in a tight busy-wait loop, the cache line is cached by the L1 cache of the consumer CPU in one of two states: shared or exclusive.

2. The producer core reads the status flag of the message. In doing so it generates a look up for the cache line in the local L1 and L2 caches of the core. The line will not be in the L1 or L2 because it has never been seen before. The Caching Agent (CA) on the producer core will send snoops to all other core’s

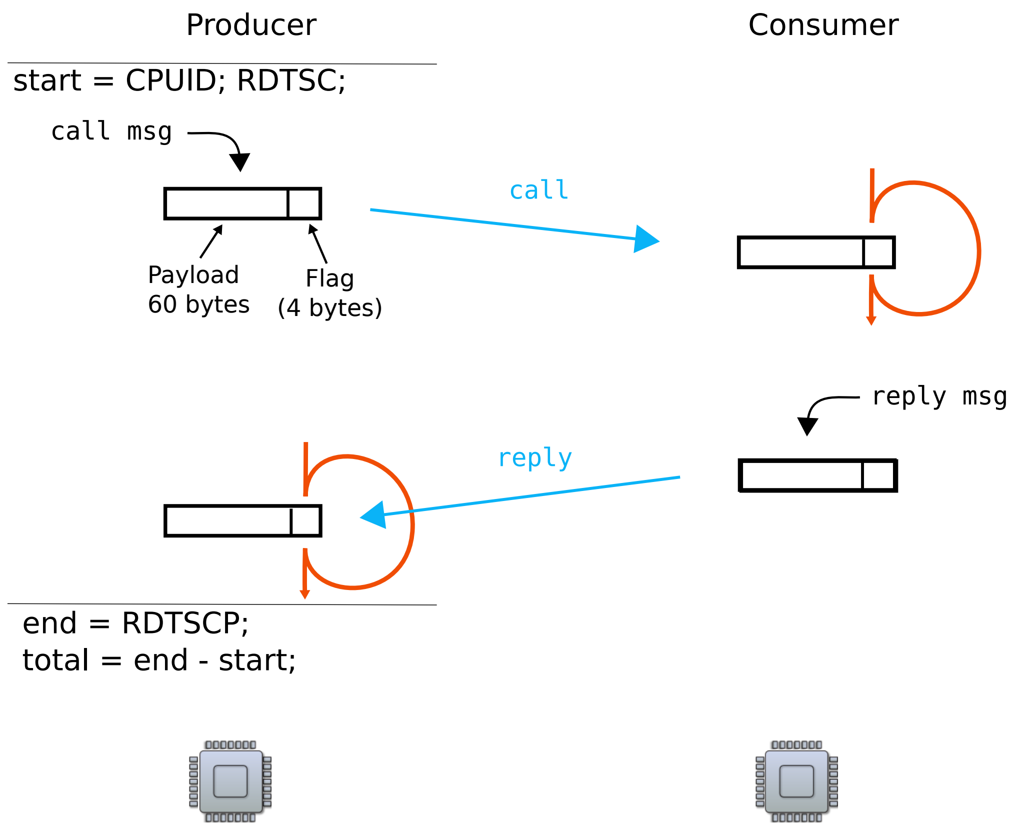


Figure 3: Call/reply invocation

Home Agent (HA) requesting the line. It will also, in parallel, send a request to the local HA which will communicate with the memory controller to get the line from memory. The consumer's HA will query its local L1 and L2 cache and return the cache line to the CA of the producer core. The CA on the consumer core also transitions its copy of the cache line to the shared state. All other cores will respond to the producer's CA with invalid meaning they do not have the cache line.

3. The producer core then immediately starts to write its data and sets the status flag to *MSG_AVAILABLE* causing the producer's CA to transition the cache line to the modified state. Since the cache line was in the shared state the CA sends invalidation snoops to all other cores.

4. The consumer core once again reads the status flag of the message to see if it is ready for consumption. Doing so generates coherence traffic from the consumer's CA to all other HAs. The consumer's CA sends a request for read, which will request the line in exclusive or shared.

5. Since the producer has the sole modified copy of the cache line (it wrote data and the flag) its CA transfers the data to the CA of the consumer core and transitions its copy of the line into the shared state.

6. The consumer core uses the data in the message. Once complete it acknowledges the message by writing *SLOT_FREE* to the status flag. Doing so generates invalidate snoops to all other cores. The CA of the consumer core will transition the line into the modified state.

   When the consumer core has a response to the message it does the same set of transactions as previously outlined, except it becomes the producer, and the producer becomes the consumer. It also places its message into the other ring, not the same one that it just received a message on.

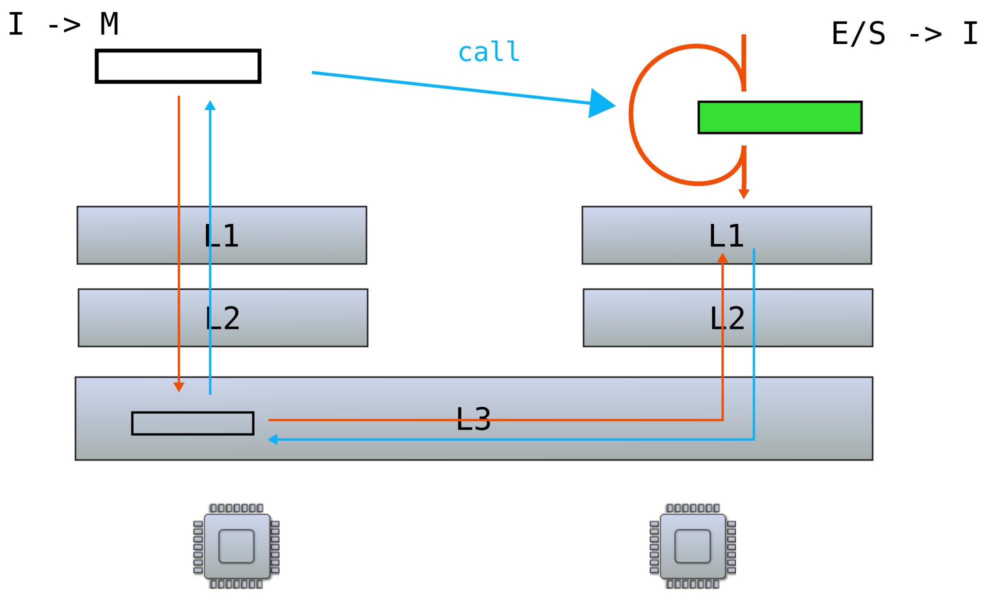


Figure 4: Cache transactions on the call path

### 3.2 Coherence traffic of a wrapped ring:

The only change between an empty ring and a wrapped ring is the initial state of the cache line in the consumer's L1/L2 cache. Remember after the the consumer has unblocked and used the data in the message, it must set the slot in the ring to free by changing the status of the flag. Doing so sets the cache line to modified and invalidates all other copies of the cache line on other cores. When the ring buffer wraps around and the producer core wants to reuse the message slot it reads the status flag generating coherence traffic as in number 2 above, but instead of the consumer's CA going from exclusive to shared it goes from modified to shared.

We use BenchIT CPU benchmark to measure the latency of the cache transactions described above (a similar analysis of Nehalem and Sandy Bridge CPUs can be found in [5, 6]). Results of the BenchIT test show that one cache transaction moving a cache line between two L1 caches on the same socket requires around 80 cycles to complete. Both call and reply parts of the invocation require two transactions each. Thus, a complete call/reply invocation requires four cache transactions or about 360 cycles on our Nehalem hardware.

## 4 Guaranteed memory consistency without memory barriers on x86

One of the benefits of using Fip-see is its guaranteed memory consistency without fences on x86 Intel processors. We must dig down into the total store order (TSO) of the x86 Intel processors in order to understand why Fip-see can operate without fences and guarantee consistent data across cores and sockets.

On x86 Intel processors there are store buffers that will accept store requests which alleviates unnecessary stalling while waiting for cache line transfers. The processor can simply issue a write to a cache line, and continue executing while the store buffer deals with actually writing the data to the cache.

On the other hand on x86 Intel processors there is no read buffer. In the case of reads, the processor will read an address and it will immediately wait for data stalling the pipeline until it is ready. This brings up an interesting scenario as some reads can be reordered around previous writes. The term reordered is used commonly in literature but is somewhat ambiguous and confusing. It's easier to say that some reads can finish before previous writes have been flushed from the store buffer. If your program sets a variable then reads a variable and they must occur in that order you must fence your write, which will stall all other memory accesses until the write has cleared the store buffer.

Below is a list of memory ordering rules for Intel x86 which are visible at the software level. [3]

1. Loads are not reordered with other loads.

2. Stores are not reordered with other stores.

3. Stores are not reordered with earlier loads.

4. Loads may be reordered with earlier stores to different locations but not with earlier stores to the same location.

5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

6. Loads and stores are not reordered with locked instructions.

   **In a multiple-processor system, the following ordering principles apply:**

7. Writes by a single processor are observed in the same order by all processors.

8. Writes from an individual processor are NOT ordered with respect to the writes from other processors.

9. In a multiprocessor system, locked instructions have a total order.

10. Individual processors use the same ordering principles as in a single-processor system

We guarantee on x86 that when a thread is polling on the flag and it transitions to *MSG_AVAILABLE* that the current message and all previous messages will have consistent and fresh data.

During a Fip-see transaction, the code writes data into one or more IPC messages. Once complete they toggle the flag(s) stating the messages are ready for consumption. Under rules 2 and 7 Fip-see is able to guarantee that the write to the flag(s) will become visible at the same time or after the data has been written. This is the most important guarantee. If it were possible for the flags to become visible before the data then the TSO rules from above would be broken. The consuming core continuously polls on the flag until the flag states there is data available to be read. Under rule 4 and 10 we see that this load instruction can occur before, during or after the writes to the messages and to the flag. The fact that the read from the other core or socket can come during any period of the write isn't a problem for Fip-see, as I explain below.

Consider the following scenario:
The producer core writes half the 60 byte IPC message then goes to sleep and finishes writing the data 2 seconds later. Meanwhile the consumer thread is polling on the flag. At some point while the producer is asleep the consumer thread will force a cache line transfer from the producer core. Now, the consuming thread has an IPC message in an incomplete state, the data in the message is inconsistent and incomplete. This occurs because of rules 4 and 10 the reads can get interspersed among the writes.

The fact that loads can occur when the data isn't consistent and complete forcing a cache transfer isn't a problem for Fip-see. The way the consuming thread notification is handled guarantees that we will only unblock and start reading the data in the IPC message is when the flag is set to *MSG_AVAILABLE*. Since the status flag is only written **after** all the data has been written and x86 cannot re-order stores we can guarantee that once the thread sees *MSG_AVAILABLE* it will see the correct data as well.

# 5 IPC timing/IOPS numbers

## 5.1 IPC latency tests

I performed numerous tests on our IPC code with different CPU features turned on and off as well as different IPC configurations as well. I was trying to identify what configurations lead to the fastest IPC transfers.

#### 5.1.1 Experiment: Ping-pong(kernel-driver) with different message queue size 1-64, cpu set at 2.4ghz, `pause` instruction is used

**Hardware configuration:** (a) Nahelem Intel Xeon CPU E5530 @ 2.40GHz; (b) Cache line size: 64 bytes; (c) Intel Hyper threading (**off**); (d) Adjacent cache pre-fetcher (**on**); (e) Hardware cache pre-fetcher (**on**); (f) Intel Turbo Mode (**off**); (g) C-States (**off**); (h) Producer pinned to cpu 0, consumer—cpu 3; (i) Ring buffer size 4096 bytes; (j) Producer buffer size 4096 bytes; (k) Consumer buffer size 4096 bytes;

**Experiment description:** The producer CPU has a parameter QUEUE-SIZE which is the number of messages it places in the ring buffer before getting the QUEUE-SIZE amount of responses back from the consumer. For each message the producer puts a 60 byte message into the ring buffer and updates the 4 byte flag to signal that the message is ready. After sending the QUEUE-SIZE number of messages, the producer immediately proceeds to waiting for reply messages from the consumer polling on the message flag in the consumer buffer.

The consumer starts by polling on the message flag in the producer buffer. The consumer notices the change from the producer and sets the 4 byte flag to `READ`. Then it replies to the producer by updating the flag of a message in the consumer buffer.

I time the latency of the round trip time of all the message. I send/reply 10,000 batches of messages, i.e., 10,000 batches of 2,4,8, etc. messages and responses. The maximum queue size is 64. Table 1 provides results for this experiment.

As we flood more messages the total cycles per round trip drops. The reason for this is two-fold. First, as we write messages the producing core is able to write directly into the store buffer which is extremely fast. Second, the hardware and adjacent line prefetcher on the consuming core will successfully pre-fetch lines and bring them into L1 prior to a real access. There is a cost for the first line transfer but some subsequent cache lines have been pre-fetched and are considered free in cost as they already reside in L1. On the producer core, when it goes to get the responses the prefetchers are able to bring in future lines that have already cleared the store buffer on the consumer core.

#### 5.1.2 Experiment: ping pong(kernel driver) MONITOR/MWAIT latencies with pause CPU set @ 2.4GHZ – Intel turbo on

**Hardware configuration:**
(a) Intel turbo boost (**on**) (b) Intel C-STATE 1Extended **on** (c) C-States **enabled**

**Experiment description:**

This experiment is designed to determine the latencies of using *MONITOR-MWAIT* instructions instead of the busy wait loop. Similar to the ping-pong experiment in subsubsection 5.1.1, the producer places a message in the queue and triggers the message ready by writing to the flag. The producer continues by busy-wait polling on a message flag in the consumer ring buffer. Once the producer detects a flag change it stops timing.

The consumer is monitor/mwaiting on the message flag location and once written is immediately woken up and places a message in the consumer queue to notify the producer. We run 10,000 ping pongs and

| Queue size | Min Cycles | Avg Cycles | Median Cycles | Cycles per 1 message (median) |
|---|---|---|---|---|
| 1 | 324 | 384 | 380 | 380 |
| 1 | 332 | 382 | 380 | 380 |
| 1 | 324 | 384 | 380 | 380 |
| 2 | 360 | 494 | 500 | 250 |
| 2 | 364 | 507 | 492 | 246 |
| 2 | 368 | 499 | 500 | 250 |
| 4 | 472 | 622 | 624 | 156 |
| 4 | 492 | 618 | 620 | 155 |
| 4 | 472 | 665 | 672 | 168 |
| 8 | 576 | 849 | 856 | 107 |
| 8 | 572 | 852 | 868 | 109 |
| 8 | 600 | 930 | 936 | 117 |
| 16 | 892 | 1476 | 1492 | 94 |
| 16 | 804 | 1381 | 1396 | 88 |
| 16 | 796 | 1367 | 1396 | 88 |
| 32 | 1696 | 2556 | 2548 | 80 |
| 32 | 1512 | 2275 | 2256 | 71 |
| 32 | 1500 | 2280 | 2264 | 71 |
| 64 | 2444 | 3820 | 3784 | 60 |
| 64 | 2264 | 3995 | 3956 | 62 |
| 64 | 2496 | 4001 | 3964 | 62 |

Table 1: Ping-pong test with various queue sizes.

take the minimum, average, and median. The CPU is pegged at 2.4ghz. Table 3 summarizes results for this experiment. Table 2 provides description of the C-States.[1]

| C-state | Name | Description |
|---|---|---|
| C1 | Halt | CPU main internal clocks are stopped. Bus interface unit and APIC are kept running at full speed. |
| C1E | Enhanced Halt | CPU main internal clocks are stopped and the CPU voltage is reduced. Bus interface unit and APIC are kept running. The frequency can be also reduced. |
| C3 | Deep Sleep | CPU internal and external clocks are stopped, L1/L2 cache can be flushed. |
| C6 | Deep Power Down | Core states are saved into memory with low power consumption. It can reduce the CPU internal voltage to any value, including 0 V. |

Table 2: Description of C-States.

For some users who may know the exact latency requirements of their call/reply invocation and who don't wish to waste energy spinning a CPU they can use the monitor/mwait instructions to save power.

[1]Source: http://pm-blog.yarda.eu/2011/10/deeper-c-states-and-increased-latency.html

| CSTATE | Min Cycles | Average Cycles | Median Cycles | Median ns |
|---|---|---|---|---|
| C1 | 546 | 750 | 726 | 303ns |
| C1 | 555 | 736 | 726 | 303ns |
| C1 | 552 | 742 | 726 | 303ns |
| C1E | 576 | 733 | 726 | 303ns |
| C1E | 432 | 752 | 726 | 303ns |
| C1E | 552 | 742 | 732 | 305ns |
| C3 | 2928 | 3048 | 3054 | 1273ns |
| C3 | 2583 | 3058 | 3054 | 1273ns |
| C3 | 2229 | 3059 | 3054 | 1273ns |
| C6 | 5724 | 6486 | 6447 | 2687ns |
| C6 | 5736 | 6510 | 6459 | 2692ns |
| C6 | 5742 | 6502 | 6459 | 2692ns |

Table 3: Monitor/mwait latency

#### 5.1.3 Experiment: ping pong(kernel driver) MONITOR/MWAIT latencies with pause CPU set @ 2.4GHZ – Intel turbo OFF

**Hardware configuration:**
(a) Intel C-STATE 1Extended **ON** (b) C-States **ENABLED**

**Experiment description:**

This experiment is designed to verify that the turbo-boost functionality is not significant for the latency of IPC. This experiment is identical to the experiment in subsubsection 5.1.2, except that turbo boost functionality is disabled. Table 4 summarizes results of this experiment.

| CSTATE | Min Cycles | Average Cycles | Median Cycles | Median ns |
|---|---|---|---|---|
| C1 | 580 | 755 | 760 | 317ns |
| C1 | 576 | 772 | 740 | 309ns |
| C1 | 736 | 789 | 764 | 319ns |
| C1E | 736 | 838 | 772 | 322ns |
| C1E | 736 | 859 | 776 | 324ns |
| C1E | 736 | 872 | 916 | 382ns |
| C3 | 3036 | 3159 | 3172 | 1322ns |
| C3 | 2952 | 3173 | 3176 | 1324ns |
| C3 | 2912 | 3158 | 3172 | 1322ns |
| C6 | 6748 | 6886 | 6876 | 2865ns |
| C6 | 6760 | 6912 | 6876 | 2865ns |
| C6 | 6760 | 6904 | 6876 | 2865ns |

Table 4: Monitor/mwait latency

## 5.2 The `pause` instruction is critical for performance

For a while we were not able to match the theoretical performance limits for the IPC, i.e., the cost of four cache line transfers plus several dozens cycles for the protocol overhead. We found later that the `pause` instruction is critical for the performance of tight busy-wait loops. Without the `pause` instruction read requests are constantly propagated to the CA causing large amounts of coherence traffic to be generated.

The `pause` instruction is a hint to the processor that the cpu should delay the next instruction by a finite amount of time. Intel states " a Pentium 4 or Intel Xeon processor suffers a severe performance penalty when exiting the [busy wait] loop because it detects a possible memory order violation." ... ". The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance". [4]

#### 5.2.1 Experiment: Message queue 1 (ping-pong), kernel driver, no CPU Features (Latency), no `pause` instruction

**Hardware configuration:**
(a) Nahelem Intel Xeon CPU E5530 @ 2.40GHz; (b) Cache line size: 64 bytes; (c) Intel Hyper-threading (**off**); (d) Adjacent cache pre-fetcher (**off**); (e) Hardware cache pre-fetcher (**off**); (f) Intel Turbo Mode (**off**); (g) C-States (**off**); (h) Producer pinned to cpu 0, consumer—cpu 3; (i) Ring buffer size 4096 bytes; (j) Producer buffer size 4096 bytes; (k) Consumer buffer size 4096 bytes;

**Experiment description:**

The producer places a 60 byte message into the queue and updates the 4 byte flag to signal that the message is ready. After sending the message, the producer immediately proceeds to waiting for a reply message from the consumer, polling on the message flag in the consumer buffer.

The consumer starts by polling on the message flag in the producer buffer. The consumer notices the change from the producer it sets the flag as `READ` and replies to the producer by updating the flag of a message in the consumer buffer.

The experiment sends 10,000 messages. We measure the latency for each round trip in cycles using the *RDTSCP* instruction. The table below provides results for four 10,000 message tests

| Minimum time in cycles | Average Time in cycles | Median time in cycles |
|:---|:---:|---:|
| 480 | 573 | 552 |
| 468 | 552 | 546 |
| 468 | 565 | 558 |

Table 5: Ping-pong test (queue size 1) without the *pause* instruction in the busy wait polling loop.

As you can see by omitting the *pause* instruction the median time jumps from around 380 cycles Table 1 to around 550 cycles.

#### 5.2.2 Experiment: ping-pong, various queue sizes (kernel driver), no `pause` instruction

**Hardware configuration:**

Same as experiment in subsubsection 5.1.1, but with the following changes: (a) Adjacent cache pre-fetcher **on** (b) Hardware cache pre-fetcher **on**

**Experiment description:**

This experiment is identical to the one described in subsubsection 5.1.1 but does not use the `pause` instruction in the busy-wait loop. The producer is configured with QUEUE-SIZE parameter which is the amount of message it places in the buffer before getting the QUEUE-SIZE amount of responses back from the consumer. The consumer on the other hand operates like a normal ping-pong. When it sees a new message it immediately replies. We time the latency of the round trip time of all the message. We send/reply 10,000 batches of messages, i.e., 10,000 batches of 2,4,8, etc. messages and responses. The maximum flood size is 64, as that is the entire ring buffer.

| Flood size | Min Cycles | Avg Cycles | Median Cycles | Cycles per 1 message (median) |
|---|---|---|---|---|
| 2 | 510 | 740 | 720 | 360 |
| 2 | 516 | 734 | 726 | 363 |
| 2 | 522 | 744 | 732 | 366 |
| 4 | 684 | 965 | 942 | 235 |
| 4 | 744 | 986 | 960 | 240 |
| 4 | 738 | 973 | 954 | 238 |
| 8 | 1020 | 1385 | 1320 | 165 |
| 8 | 1020 | 1445 | 1386 | 173 |
| 8 | 1008 | 1378 | 1329 | 165 |
| 16 | 1710 | 2206 | 2124 | 132 |
| 16 | 1728 | 2213 | 2118 | 132 |
| 16 | 1692 | 2264 | 2142 | 133 |
| 32 | 3096 | 3750 | 3666 | 114 |
| 32 | 3036 | 3741 | 3630 | 113 |
| 32 | 3444 | 3944 | 3846 | 120 |
| 64 | 6228 | 7171 | 7032 | 109 |
| 64 | 6252 | 7143 | 7044 | 110 |
| 64 | 6660 | 7416 | 7284 | 113 |

Table 6: Ping-pong test with various queue sizes, but without the *pause* instruction in the busy wait polling loop.

#### 5.2.3 Experiment: ping-pong(kernel driver) MONITOR/MWAIT latencies–no `pause` instruction in the busy wait loop

Same as experiment in subsubsection 5.1.1, but with the following changes: (a) Adjacent cache pre-fetcher **on** (b) Hardware cache pre-fetcher **on** (c) Intel C-STATE 1Extended **on** (d) C-States **enabled**

**Experiment description:** This is a simple PING-PONG experiment to determine the latencies of using *MONITOR-MWAIT* combo if the `pause` instruction is not used. The producer places a message in the queue and triggers the message ready by writing to the flag. The producer polls on the consumer flag, once it detects a change it stops timing. The consumer is monitor/mwaiting on that flag location and once written is immediately woken up and places a message in the consumer queue to notify the producer. We run 10,000 ping pongs and take the minimum-average-median. The C-States are described in subsubsection 5.1.2. The results are listed in Table 7.

# 6 Xen Multi-queue paravirtualized block drivers

### Paravirtualization

Paravirtualization is technique to trick a guest virtual machine into thinking it is running on actual hardware without doing any sort of emulation. In the case of the paravirtualized Xen block drivers the guest virtual machine will think it is talking directly to some physical device, but in reality is talking to a

| CSTATE | Min Cycles | Average Cycles | Median Cycles | Median ns |
|---|---|---|---|---|
| C1 | 846 | 1518 | 1476 | 615ns |
| C1 | 846 | 1521 | 1476 | 615ns |
| C1 | 846 | 1510 | 1476 | 615ns |
| C1 | 846 | 1492 | 1476 | 615ns |
| C1E | 840 | 1480 | 1476 | 615ns |
| C1E | 840 | 1501 | 1470 | 612ns |
| C1E | 846 | 1476 | 1476 | 615ns |
| C1E | 864 | 1487 | 1470 | 612ns |
| C3 | 888 | 4374 | 4224 | 1760ns |
| C3 | 876 | 4339 | 4224 | 1760ns |
| C3 | 858 | 4255 | 4224 | 1760ns |
| C3 | 864 | 4248 | 4224 | 1760ns |
| C6 | 888 | 9831 | 9648 | 4020ns |
| C6 | 828 | 9561 | 9654 | 4022ns |
| C6 | 810 | 9740 | 9636 | 4015ns |
| C6 | 816 | 9833 | 9684 | 4035ns |

Table 7: Monitor/mwait test without the *pause* instruction in the producer's busy wait polling loop.

device driver that marshals block requests through a ring buffer to the host operating system. Once at the host OS the driver does some sanity checks, unmarshals the data and forwards it on to the block subsystem. Once the request has been handled the data is placed back into the ring and the guest VM is notified of fresh data. The terms front-end, and block-front are used to describe the driver in the guest VM and back-end and block-back are used to describe the driver on the host OS which will talk with the underlying device.

**Multi-queue Block IO**

With the advent of high throughput low latency SSD and NVM (Non-volatile memory) storage devices bottlenecks in the OS block layer started to appear. These bottlenecks limited throughput to a point where the devices supported higher throughput than what was possible in software. On the traditional IO path multiple cores trying to submit IO requests would contend for a single lock and single submission queue to the device. With a shared lock to a single submission queue you can't utilize parallel submissions. Each core trying to submit requests must wait and contend for the lock before adding to the queue. Bjorling et al. designed and implemented, multi-queue a lock-less per-cpu submission queuing system. In this design each cpu has a submission queue where block requests can be placed into. From there the device driver for the device would merge submission queues into the actual hardware queues for the device. This design alleviated lock contention and scales extremely well as the number of cores in the system increases. [1]

**Xen Paravirtualized MQ Block driver design**

We took a stock 3.19 Linux kernel and applied a series of non-upstream patches which enable multi-queue in the Xen block-front and block-back driver. These patches created a set of legacy interrupt driven Xen IPC rings for each software queue in the block layer. The first set of changes was to rip out the legacy rings and replace them with Fip-see channels. For each software queue in the block layer there is a corresponding Fip-see channel connecting the front-end driver to the back-end driver. On the receiving end of every Fip-see ring, for latency and throughput, there is a dedicated thread pinned to a core, whose sole task is to remove messages and dispatch them to the block layer. Since software queues are per-cpu that would mean we would have to have every cpu dedicated to a ring, which is clearly unacceptable. To fix the issue we only allow up to 8 threads to be dedicated to the rings depending on work load. Each thread is pinned to a virtual CPU selected at boot up. The threads are set as real-time and kernel preemption has been turned on. For example, in a configuration where we have 16 virtual cpus for the front-end, there will be 16 software submission queues. For each one of those queues there will be a corresponding Fip-see channel to connect to the front-end and back-end. If we dedicate two vcpus on the front-end to the IPC channels the threads will iterate round-robin through 8 response rings each pulling the responses from the back-end, marshaling them back into bio structs and submitting them to the block layer.

The front-end driver is also tasked with communicating with Xen to setup the underlying IPC pages for sharing with the back-end. Once Xen has setup the pages for sharing the front-end driver will publish grant references in the Xenstore for the back-end. The front-end also will publish how many software queues it wants to use, based off how many vcpus it has.

The back-end driver during guest VM boot-up will read from the Xenstore and pull out how many software queues the front-end wants to use. If the host OS has enough cpus and thus enough software queues to handle the amount the front-end wants to use it begins pulling out the grants for each IPC channel. Once all the grants are pulled out it calls into Xen to have the pages mapped into its address range. If there aren't sufficient software queues on the back-end it will notify the front-end of the maximum amount it can support and they use that number. Like the front-end the back-end can dedicate between 1 and 8 threads to the IPC channels. Each thread will iterate through the request rings and pull out IPC messages which contain IO requests from the virtual machine. Figure 5 gives a good pictorial overview of the entire system, from the process in the VM generating IO requests to the physical device on the back-end.

**Life cycle of an IO request in this system**

To give a general idea of the life cycle of a block request we'll walk through a scenario starting at the Xen front-end driver. After some user space process has called write/read/pwrite/pread/aio_submit etc it walks the block layer until the multi-queue code calls a queue_rq function pointer. The front-end driver registers itself and exports this function pointer to the MQ block layer so it will be called. It should be noted we are still in the execution context of the process, not a dedicated kernel thread. Once inside the front-end implementation of queue_rq the code has to do two things. First it must grant access to the user-space pages to back-end domain through Xen. Then it must marshal grant references and other metadata about the IO request into an IPC message and send it to the receiving end. We must grant access to the user-space pages so the physical device can DMA data into or out of the pages based on the type of request in the IPC messages. Once all the data has been placed into the IPC message the flag will be set so the other end can read the data. The thread of execution walks out and returns to user space if its an async transaction or waits in the block layer for the response if its a synchronous transaction.

On the back-end driver one of the cpus tasked with polling on the ring with our request. It gets unblocked and drops into a dispatch function which figures out what type of block request its dealing with, read/write/flush etc. From there it pulls out the list of grants and hypercalls into Xen to map the pages the grants reference into its virtual address range. Once the mapping has completed it unmarshals all the metadata from the IPC message into a bio struct and submits it to the block layer. Along with all the data it includes a completion function pointer. When the device has completed the request it fires an interrupt which will either schedule a work queue or it will handle the completion in IRQ context. In either case, the completion function pointer gets called and a response IPC message is placed in one of the rings.

On the front-end one of the virtual cpus is then unblocked and unmarshals the data and tells the block layer the request is complete for the ID inside the IPC message.

**Optimizing the drivers with memshare**

By removing the legacy IPC adding Fip-see to the driver we have eliminated a hypercall and an IPI each time a message is placed in a ring. But there is still room for improvement. Remember above we mentioned that the front-end driver must grant access to user-space pages and the back-end drivers must map those pages by hypercalling into Xen. This occurs on every transaction and is a significant source of slowdown and latency. To solve this issue Matthew Hannon implemented and built-upon a portion of FIDO [2]. He designed device drivers that would map the entirety of DOMU's memory into DOM0's virtual address space. I took the standalone drivers and rewrote portion to be used as a library that Fip-see could interface with.

Now instead of the front-end hypercalling into Xen to grant access to user-space pages on every IO request, the front-end driver now simply finds the Physical Frame Number (PFN) of the pages and sends those to the back-end. In the back-end it will remove the list of PFNs and map the PFN to the physical page that those PFNs reference. Using this technique eliminates page table walks from mapping and unmapping pages into virtual memory as well as two hypercalls that were slowing down the block drivers.

## 6.1 Host OS Null Block test device

To test our implementation we utilize a null block driver on the back-end. The null block pretends to be a driver for a physical device that exports hardware queues. The driver has a series of modes you can place the "null device" into. You can turn on timers to simulate slower devices or have the driver complete the request immediately. Because we're interested in testing the limits of our design we place the null device into the fastest modes possible. The device upon receiving a request will immediately trigger a completion of that request to the block layer.

The source code is located in /drivers/block/null_blk.c
We instantiate the driver with:
*modprobe null_blk queue_mode=2 nr_devices=2 hw_queue_depth=64 irqmode=0 use_per_node_hctx=1 submit_queues=$(nproc)*

#### Exporting Null Block to DOM U

Normally in a guest VM's configuration you specify what devices you want to export by setting the following in /etx/xen/temp.cfg:
*phy:/dev/vg0/temp-disk,xvda2,w*
This statement says I want to export temp-disk to the guest VM and have a dev entry in the VM under 'xvda2', in this case temp-disk is the root partition for the VM. During development I was modifying the device drivers that both the null block and the root device were using. This was an interesting design choice on my part because the VM crashed more times than it booted correctly during initial development. But as it turned out it was a good method of testing the changes I was making. If the VM booted and didn't crash, randomly hang or corrupt data I was on the right path. Ultimately when the changes were complete we decided that it didn't make sense to have the Fip-see version of the drivers running for the root device. Once past VM boot up the root device was never used and the cores that both the front-end and back-end were using for the IPC channels were sitting useless and could be used for performance tuning and testing.

To alleviate this issue we made a new set of back-end and front-end drivers in the Linux source tree. The new drivers had the original code which we started with after applying the patches to the 3.19 tree. The other drivers had the Fip-see and memory sharing implementation. We did this so we could boot the VM on the old slow drivers that don't hog cores and we could use the new block drivers for fast NVMe devices or our null block driver.

In order to actually get this to work we had to hack the Xentools to inform it that there is a new type of physical device called a 'phy1'. Xentools during VM boot-up exports the physical device to the VM under the name in the configuration which triggers a probe for the correct device driver. Xentools then needs to probe in the back-end device driver for that device as well. Now, if you want to use the Fip-see drivers with your fast device, or your null block device you place the following in your VM configuration file:
*phy1:/dev/nullb0,fipc15,w*
The phy1 will tell Xentools that we want to use our custom driver and to export the null block device as /dev/fipc15 on the VM.

## 6.2 Results

We performed a series of tests on a Dell R820 4 socket 32 core Emulab node. We had Xen provision 16 cores for DOM0 and leave the rest for the guest VM. In the configuration file for the VM we black listed the cores used by DOM0 to make sure all computation would occur on a non-contested core. We performed tests using Flexible I/O Tester (Fio) which allows us great fine grained control over how many processes to use, what IO engine to use, the submission queue depth among other tuneables.

### 6.2.1 Host OS results, baseline

For our measurements we wanted to test a few different things. First is the throughput of the multi-queue block layer for reads and writes. Second is the latency of reads and writes to our null block. For all tests the following parameters were setup, unless otherwise specified.

(a) DIRECT mode on the device (Bypasses page cache)

(b) Exported hardware queues from null block: **16**

(c) Software queue depth: **64**

(d) Amount of IO requests to batch before submitting to the kernel: **64**

(e) Amount of IO requests to complete before requesting from the kernel: **64**

(f) Number of processes generating IO requests: **16**

(g) Run time: **30 seconds**

**IOPs on DOM0**

These tests are our baseline tests for how many IOPs are possible on our null block driver. We used Fio and the AIO (Asynchronous IO) engine to generate sequential IO requests to the block driver using different block sizes.

| Block Size | R/W | IOPs | Aggregated Bandwidth MB/sec |
|---|---|---|---|
| 4096 Bytes | Write | 936,531 | 3658.4 |
| 4096 Bytes | Read | 952,190 | 3719.6 |
| 512 Bytes | Write | 925,099 | 462.5 |
| 512 Bytes | Read | 943,663 | 471.8 |

Table 8: DOM0 IOPs throughput on different block sizes

**Read and Write Latency DOM0**

These tests are our baseline tests for how fast we can access the null block. We used Fio and the Sync (read/write System calls) engine to generate sequential IO requests to the block driver using different block sizes.

In this test, instead of 16 processes generating IO we use just one process. We use “taskset” to pin Fio to a specific core.

| Block Size | R/W | Latency in microseconds |
|---|---|---|
| 4096 Bytes | Write | 5.94 |
| 4096 Bytes | Read | 5.70 |
| 512 Bytes | Write | 6.01 |
| 512 Bytes | Read | 5.72 |

Table 9: DOM0 latency on different block sizes

#### 6.2.2 Guest VM results, original drivers

**IOPS of original MQ-Block Drivers**

| Block Size | R/W | IOPs | Aggregated Bandwidth MB/sec |
|---|---|---|---|
| 4096 Bytes | Write | 249,547 | 998.1 |
| 4096 Bytes | Read | 296,288 | 1157.4 |
| 512 Bytes | Write | 244,212 | 119.2 |
| 512 Bytes | Read | 285,161 | 139.2 |

Table 10: DOMU IOPs throughput on different block sizes, original drivers

**Read and Write latency DOMU**
Like above, we use one process and “taskset” it to a specific core for the duration of the experiment.

| Block Size | R/W | Latency in microseconds |
|---|---|---|
| 4096 Bytes | Write | 43.00 |
| 4096 Bytes | Read | 42.78 |
| 512 Bytes | Write | 43.23 |
| 512 Bytes | Read | 42.20 |

Table 11: DOMU latency on different block sizes, original drivers

#### 6.2.3 Guest VM results, Fip-see drivers

For the following tests we had Fip-see setup where both the front-end and back-end drivers had 8 pre-emptible kernel threads pinned to 8 cores. On the front-end each core was tasked with pulling IO responses off two response rings. On the back-end each core had two request rings to pull IO requests off from the front-end.

**IOPs on DOMU with the Fip-see driver**

| Block Size | R/W | IOPs | Aggregated Bandwidth MB/sec |
|---|---|---|---|
| 4096 Bytes | Write | 829,043 | 3238.5 |
| 4096 Bytes | Read | 865,857 | 3382.3 |
| 512 Bytes | Write | 827,799 | 413.8 |
| 512 Bytes | Read | 844,761 | 422.3 |

Table 12: DOMU IOPs throughput on different block sizes with Fip-see

**Latency on DOMU with Fip-see driver**

| Block Size | R/W | Latency in microseconds |
|---|---|---|
| 4096 Bytes | Write | 14.61 |
| 4096 Bytes | Read | 14.21 |
| 512 Bytes | Write | 14.57 |
| 512 Bytes | Read | 14.27 |

Table 13: DOMU latency on different block sizes, original drivers

### 6.2.4 Results analysis Host OS vs Fip-see performance

First we will explore how close our changes get us to Host OS performance.

| Block Size | R/W | Fip-see DOMU Latency | DOM0 (Host OS) Latency | Percentage of Host Latency |
|---|---|---|---|---|
| 4096 Bytes | Write | 14.61 | 5.94 | 40.6% |
| 4096 Bytes | Read | 14.21 | 5.70 | 40.1% |
| 512 Bytes | Write | 14.57 | 6.01 | 41.2% |
| 512 Bytes | Read | 14.27 | 6.72 | 47.0% |

Table 14: Percentages of Host OS Latency Performance

| Block Size | R/W | Fip-see DOMU IOPs | DOM0 (Host OS) IOPs | Percentage of Host IOPs |
|---|---|---|---|---|
| 4096 Bytes | Write | 829,043 | 936,531 | 88.5% |
| 4096 Bytes | Read | 865,857 | 952,190 | 90.9% |
| 512 Bytes | Write | 827,799 | 925,099 | 89.5% |
| 512 Bytes | Read | 844,761 | 943,663 | 89.5% |

Table 15: Percentages of Host OS IOPs Performance

Adding Fip-see significantly improved the performance of the IOPs numbers compared to the latency numbers. This isn't entirely surprising when you take into account of how the IO request is dealt with in the front-end, back-end interaction. If we refer to Figure 5 again we can see that in an IO request on the VM it must first traverse block layer in the VM's kernel, then get passed over to the back-end. In the back-end it must once again traverse the block layer. For synchronous IO requests (What we used for latency numbers) one IO request is submitted then waited on in the front-end until it is complete. For the IOPs however, we used AIO which will batch 64 IO requests and submit them at the same time. Therefore all 64 IO requests will traverse the same path at the same time. This means the double "work" (walking the block layer twice) is shared among 64 requests not just one, allowing better throughput and thus better numbers.

## 6.3 Fip-see vs Original driver performance

Now we will compare the performance of the original Multi-queue drivers vs the new Fip-see enabled drivers.

| Block Size | R/W | Fip-see DOMU Latency | DOMU Original Latency | Percent Reduction |
|---|---|---|---|---|
| 4096 Bytes | Write | 14.61 | 43.00 | 66% |
| 4096 Bytes | Read | 14.21 | 42.78 | 66.8% |
| 512 Bytes | Write | 14.57 | 43.23 | 66.3% |
| 512 Bytes | Read | 14.27 | 42.20 | 66.1% |

Table 16: Percentage Reduction of Original Driver Latency

| Block Size | R/W | Fip-see DOMU IOPs | DOMU Original IOPs | Percent Increase |
|---|---|---|---|---|
| 4096 Bytes | Write | 829,043 | 249,547 | 332% |
| 4096 Bytes | Read | 865,857 | 296,288 | 292% |
| 512 Bytes | Write | 827,799 | 244,212 | 338% |
| 512 Bytes | Read | 844,761 | 285,161 | 296% |

Table 17: Percentages of Host OS IOPs Performance

## 7 Conclusion

We set out to test whether it would be possible to get near Host OS performance on a block device from a virtual machine. To do so we developed a Low latency IPC mechanism named Fip-see and built it into the paravirtualized Linux block-front and block-back drivers for Xen. Doing so we were able to significantly improve the latency and throughput compared to the original drivers getting us near Host OS performance. Rewriting the drivers for Fip-see took a graduate student three and a half months mostly due to being unfamiliar with the code base. We believe the speed ups achieved and the low cost and time of rewriting the drivers would be a worth wile investment for a company trying to sell its storage software. The research presented in this report was supported by the National Science Foundation under the Grant No. 1319076, and by NetApp.

## 8 Future work

**Auto scaling of Back-end Front-end CPUs**
One of the biggest trivial things that can be implemented is auto scaling/pinning of threads to IPC channels. Right now I have to manually go in and change how many threads will be used, rebuild and install on both DOM0 and the VM. It wont be hard to implement load calculations that once hit will automatically spawn or destroy receiving kernel threads and hand off some IPC channels to the new/old threads.

**Access to Low latency NVMe devices**
5B

Although we got great speed-ups we're not anywhere near where we want to be in terms of latency. In order to get to Host-OS performance we want to try and bypass the guest VM kernel all together. We want to drop Fip-see directly into the NVMe driver and export an interface to processes through the ring buffers. Instead of interrupting into the kernel, walking the block layer, applications could setup an IPC channel with the NVMe driver and submit requests directly to the hardware queues. For applications that require direct low latency access to the devices we believe this is the best method as it eliminates the two most costly portions of the code, the syscall interrupt as well as the walking of the block layer.

**124 cycle ping-pongs**
During all of our testing for Fip-See we had hyperthreading off. During the development of the block drivers I had the revelation that hyperthreading may be extremely useful for certain scenarios. In Linux hyperthreads are treated as their own CPU and tasks can be scheduled on those CPUs as if they were physical CPUs. I turned on hyperthreading and ran our timing tests, on our Nahelem a full ping-pong on one CPU with two threads took 124 cycles. For applications that need a separation boundary it would be interesting to test out if removing a context switch with this type of notification would have any sort of benefit. The tasks on one side would have to be minimal, as the contested pipeline would slow down the other task. Anecdoteally I flipped hyperthreading on during some of our IOPs test and we saw a degradation of performance. Since AIO is asynchronous once the IO requests are submitted both side continue to do work meaning there is a bunch of work in the pipeline. Perhaps for the Synchronous case this 124 cycle round-trip time would be beneficial since one side could be waiting in a CPU-relax loop while the other is dealing with the IO request.

**Smart placement of threads on NUMA systems**
On our R820 we measured the cost of accessing modified cache lines across sockets and found something extremely interesting. The costs of transferring lines across sockets is asymmetric. For example transferring a cache line from Socket 0 to Socket 1 takes 369 Cycles, however transferring the same cache line (after its been placed in the modified state) from Socket 1 to Socket 0 takes 465 cycles. Figure 6 shows an overview of the costs of transferring Modified cache line around the sockets. Xen needs a mechanism to run tests like this on boot-up in order to determine which cores and sockets to assign to which domains. When Xen blindly assigns CPUs to domains its not accounting for where the CPUs are sitting and what features they have which can hurt performance.

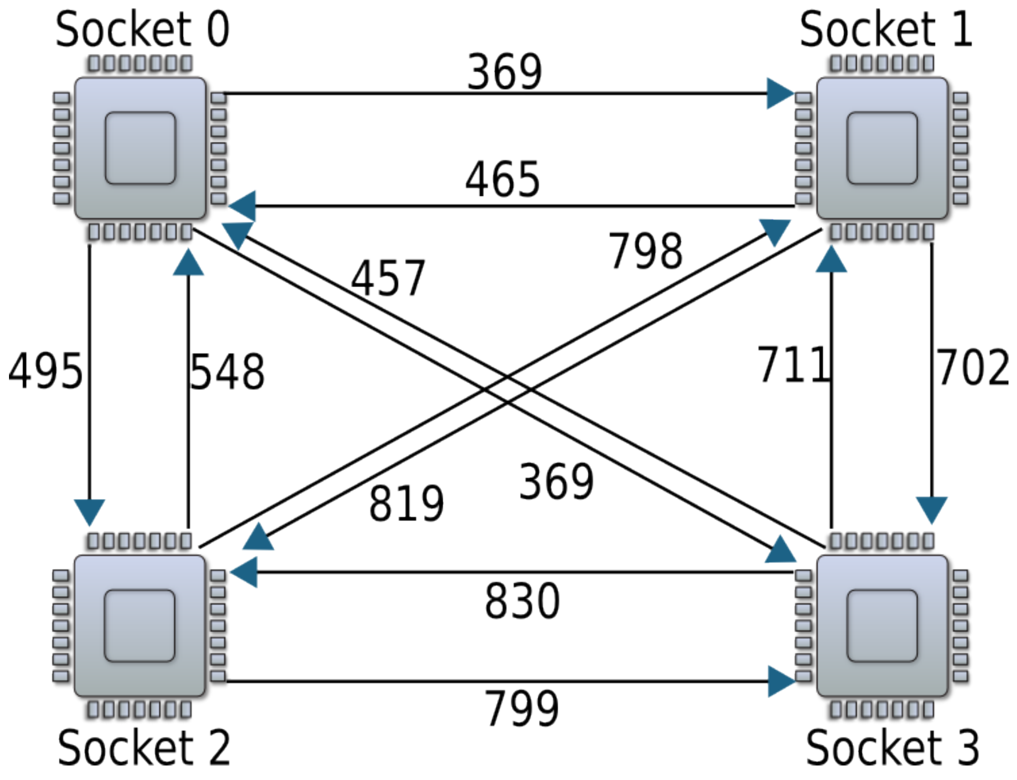


Figure 6: Latencies for accessing cache line in the “Modified state”

Our best explanation for why there is a large jump in latency for some sockets is because we believe only two sockets have Home Agents on the R820. When a socket without an HA wants to do a cache transfer it must interact with the cache directory which resides on another socket, thus it must traverse the QPI link. For a socket with an HA it can talk directly to the local HA and we see lower numbers.

## References

[1] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference*, page 22. ACM, 2013.

[2] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 25–25. USENIX Association, 2009. Also available as: `http://www.cs.utah.edu/~aburtsev/doc/fido-usenix09.pdf`.

[3] I. Corp. Intel 64 Architecture Memory Ordering White Paper . Technical report, Intel Corp, 8 2007.

[4] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018. March 2009.

[5] D. Molka, D. Hackenberg, and R. Schöne. Main memory and cache performance of intel sandy bridge and amd bulldozer. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 4:1–4:10, New York, NY, USA, 2014. ACM.

[6] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 261–270. IEEE, 2009.

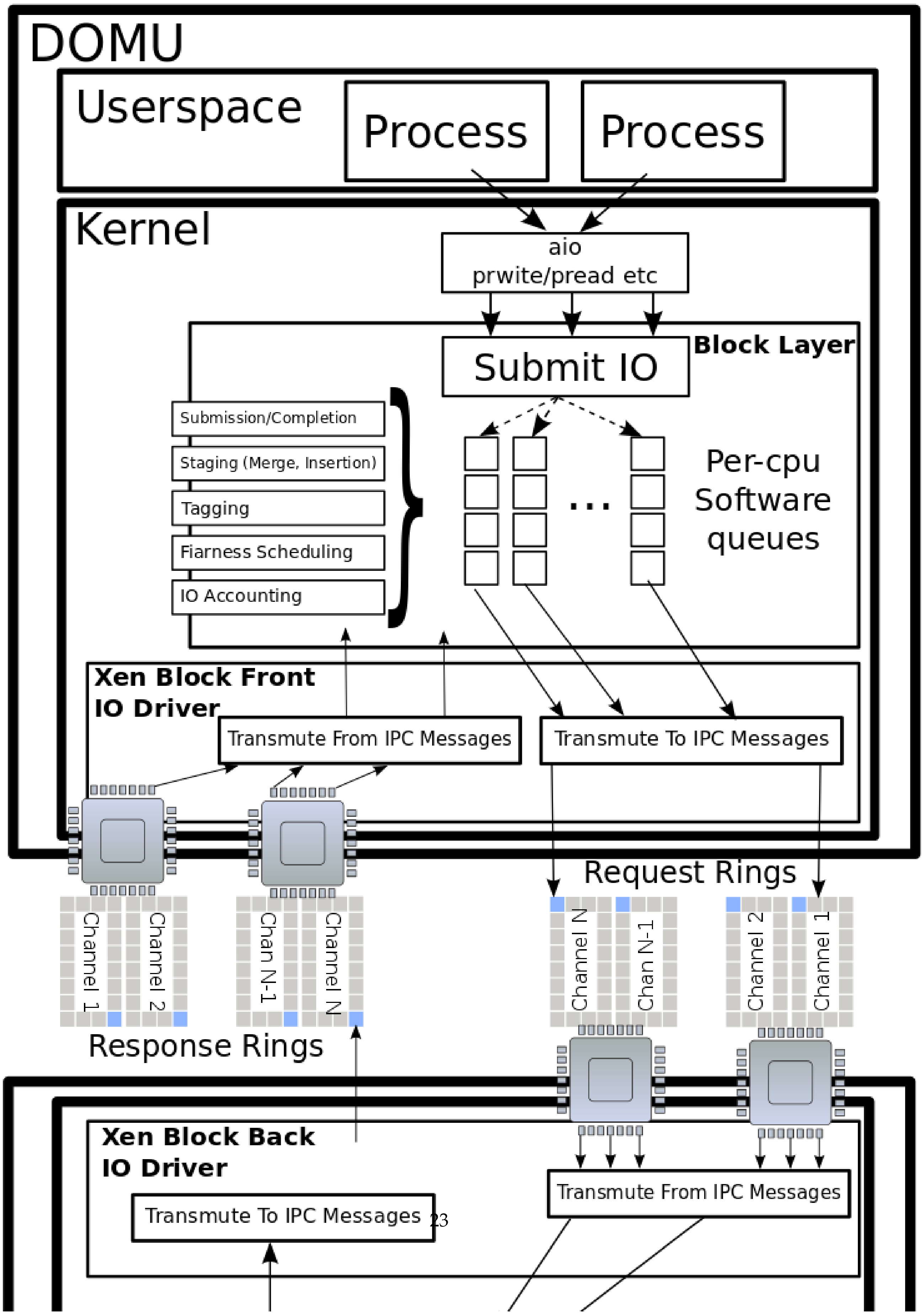


Figure 5: Figure Block IO Driver