Xen and XenServer
Storage Performance
Low Latency Virtualisation Challenges

Dr Felipe Franciosi
XenServer Engineering Performance Team

e-mail: felipe.franciosi@citrix.com
freenode: felipef #xen-api
twitter: @franciozzy
Agenda

• Where do Xen and XenServer stand?
  • When is the virtualisation overhead most noticeable?

• Current implementation:
  • blkfront, blkback, blktap2+tapdisk, blktap3, qemu-qdisk

• Measurements over different back ends
  • Throughput and latency analysis
  • Latency breakdown: where are we losing time when virtualising?

• Proposals for improving
Where do Xen and XenServer Stand?

When is the virtualisation overhead noticeable?
Where do Xen and XenServer stand?

- What kind of throughput can I get from dom0 to my device?
  - Using 1 MiB reads, this host reports 118 MB/s from dom0.
Where do Xen and XenServer stand?

- What kind of throughput can I get from domU to my device?
  - Using 1 MiB reads, this host reports 117 MB/s from a VM

**IMPERCEPTIBLE virtualisation overhead**
Where do Xen and XenServer stand?

• That’s not always the case...
  ※ Same test on different hardware (from dom0)

```bash
[root@xrtuk-08-06 ~]# dd if=/dev/sda of=/dev/null bs=1M count=500 iflag=direct
500+ records in
500+ records out
524288000 bytes (524 MB) copied, 0.762199 seconds, 688 MB/s
```

my disks can do 700 MB/s !!!!
Where do Xen and XenServer stand?

• That’s not always the case...
  ○ Same test on different hardware (from domU)

why is my VM only doing 300 MB/s ???
Current Implementation
How we virtualise storage with Xen
Current Implementation (bare metal)

• How does that compare to storage performance again?
  • There are different ways a user application can do storage I/O
    • We will use simple `read()` and `write()` libc wrappers as examples

```c
1. char buf[4096];
2. int fd = open("/dev/sda", O_RDONLY | O_DIRECT);
3. read(fd, buf, 4096);
4. buf now has the data!
```
Current Implementation (Xen)

- The “Upstream Xen” use case
  - The virtual device in the guest is implemented by blkfront
  - Blkfront connects to blkback, which handles the I/O in dom0

The virtual device in the guest is implemented by blkfront. Blkfront connects to blkback, which handles the I/O in dom0.
Current Implementation (Xen)

• The XenServer 6.2.0 use case
  • XenServer provides thin provisioning, snapshot, clones, etc. hello VHD
  • This is easily implemented in user space. hello TAPDISK
Current Implementation (Xen)

- The blktap3 and qemu-qdisk use case
  - Have the entire back end in user space
Measurements Over Different Back Ends

Throughput and latency analysis
Measurement Over Different Back Ends

• Same host, different RAID0 logical volumes on a PERC H700
  • All have 64 KiB stripes, adaptive read-ahead and write-back cache enabled
• dom0 had:
  - 4 vCPUs pinned
  - 4 GB of RAM

• It becomes visible that certain back ends cope much better with larger block sizes.

• This controller supports up to 128 KiB per request.

• Above that, the Linux block layer splits the requests.

Dell PowerEdge R815 (512 GB DRAM)
AMD Opteron 6272 @2.1 GHz (64 Cores - 8 NUMA Nodes - 4 Sockets)
CentOS 6.4 + Kernel 3.11.0 + Xen 4.3.0
Read operations with varying block sizes
• Seagate ST (SAS)

• blkback is slower, but it catches up with big enough requests.
• Seagate ST (SAS)

• User space back ends are so slow they never catch up, even with bigger requests.

• This is not always true: if the disks were slower, they would catch up.
• Intel DC S3700 (SSD)

• When the disks are really fast, none of the technologies catch up.
Measurement Over Different Back Ends

• There is another way to look at the data:

\[
\frac{1}{\text{Throughput (data/time)}} = \text{Latency (time/data)}
\]
Intel DC S3700 (SSD)

The question now is: where is time being spent?

Compare time spent:
- dom0
- blkback
- qdisk

Read operations with varying block sizes / Intel SSDSC2BA40 SSD (2 x 400GB RAID0 on PERC H700)
Measurement Over Different Back Ends

• Inserted trace points using RDTSC
  • TSC is consistent across cores and domains

1. Just before issuing read()
2. On SyS_read()
3. On blkfront's do_blkif_request()
4. Just before notify_remote_via_irq()
5. On blkback's xen_blkif_be_int()
6. On blkback's xen_blkif_schedule()
7. Just before blk_finish_plug()
8. On end_block_io_op()
9. Just before notify_remove_via_irq()
10. On blkif_interrupt()
11. Just before __blk_end_request_all()
12. Just after returning from read()
Measurement Over Different Back Ends

- Initial tests showed there is a “warm up” time
- Simply inserting printk()s affect the hot path
- Used a trace buffer instead
  - In the kernel, trace_printk()
  - In user space, hacked up buffer and a signal handler to dump its contents

- Run 100 read requests
  - One immediately after the other (requests were sequential)
  - Used IO Depth = 1 (only one in flight request at a time)
- Sorted the times, removed the 10 (10%) fastest and slowest runs
- Repeated the experiment 10 times and averaged the results
Dell PowerEdge R815 (512 GB DRAM) AMD Opteron 6272 @2.1 GHz (64 Cores - 8 NUMA Nodes - 4 Sockets)
(dom0 4GB and pinned [0-3]: CentOS 6.4 + Kernel 3.11.0 x86_64 + Xen 4.3.0)
(domU 4GB and pinned [4-7]: Wheezy 7.1 + Kernel 3.11.0 x86_32)
Read operations with varying block sizes / Intel SSDSC2BA40 SSD (2 x 400GB RAID0 on PERC H700)

- sys_read start
- blkfront do_req
- blkfront kicking
- blkback kicked
- blkback scheduled
- blkback plugging
- blkback completing
- blkback kicking
- blkfront kicked
- blkfront returning
- sys_read end
- back to userland

Measurements on 3.11.0 with Persistent Grants

Time spent on device
Time spent copying data out of persistently granted memory
Measurement Over Different Back Ends

• The penalty of copying can be worth taking depending on other factors:
  • Number of dom0 vCPUs (TLB flushes)
  • Number of concurrent VMs performing IO (contention on grant tables)

• Ideally, blkfront should support both data paths

• Administrators can profile their workloads and decide what to provide
Proposals for Improving

What else can we do to minimise the overhead?
New Ideas

• How can we reduce the processing required to virtualise I/O?
New Ideas

• Persistent Grants
  • Issue: grant mapping (and unmapping) is expensive
  • Concept: back end keeps grants, front end copies I/O data to those pages
  • Status: currently implemented in 3.11.0 and already supported in qemu-qdisk

• Indirect I/O
  • Issue: blkif protocol limit requests to 11 segs of 4 KiB per I/O ring
  • Concept: use segments as indirect mapping of other segments
  • Status: currently implemented in 3.11.0
New Ideas

• Avoid TLB flushes altogether (Malcolm Crossley)
  • Issue:
    • When unmapping, we need to flush the TLB on all cores
    • But in the storage data path, the back end doesn’t really access the data
  • Concept: check whether granted pages have been accessed
  • Status: early prototypes already developed

• Only grant map pages on the fault handler (David Vrabel)
  • Issue:
    • Mapping/unmapping is expensive
    • But in the storage data path, the back end doesn’t really access the data
  • Concept: Have a fault handler for the mapping, triggered only when needed
  • Status: idea yet being conceived
New Ideas

• Split Rings
  • **Issue:**
    • blkif protocol supports 32 slots per ring for requests or responses
    • this limits the total amount of outstanding requests (unless multi-page rings are used)
    • this makes inefficient use of CPU caching (both domains writing to the same page)
  • **Concept:** use one ring for requests and another for responses
  • **Status:** early prototypes already developed

• Multi-queue Approach
  • **Issue:** Linux’s block layer does not scale well across NUMA nodes
  • **Concept:** allow device drivers to register a request queue per core
  • **Status:** scheduled for Kernel 3.12
e-mail: felipe.franciosi@citrix.com
freenode: felipef #xen-api
twitter: @franciozzy