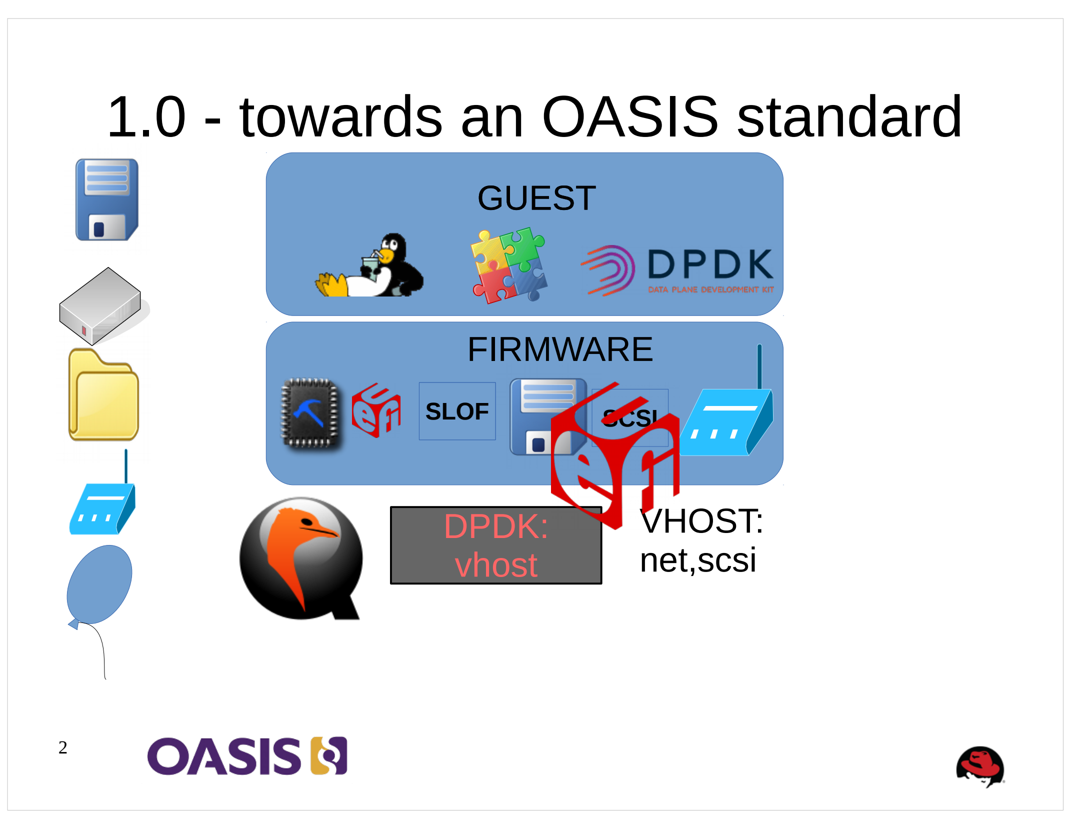
Quo Vadis Virtio?

2016

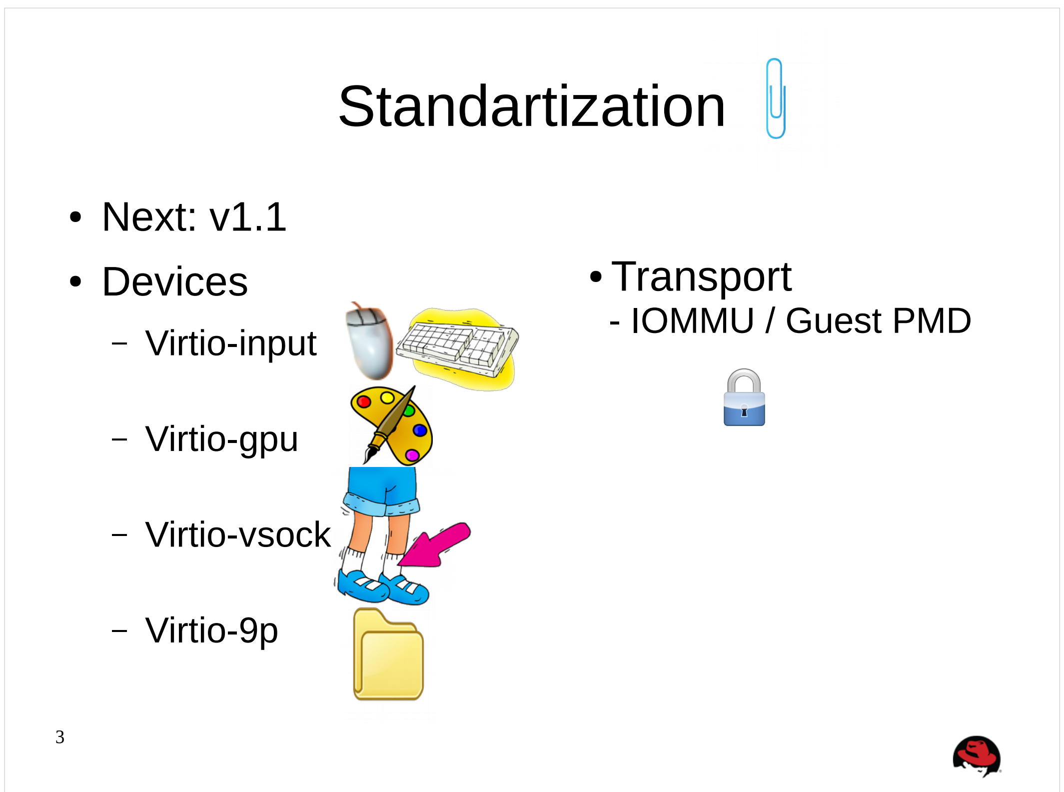
Michael S. Tsirkin
Red Hat

Uses material from https://lwn.net/Kernel/LDD3/
Gcompris, tuxpaint, childplay
Distributed under the Creative commons license, except logos which are C/TM respective owners.

Hello!
I'm Michael Tsirkin from Red Hat
I''m the chair of the virtio T.C. As part of my job I'm working on the virtio family of pravirtualized interfaces; This talk is going to be about future directions for virtio. This is not going to be a status update. Rather, I chose to focus on where we are heading going forward – this and next year.

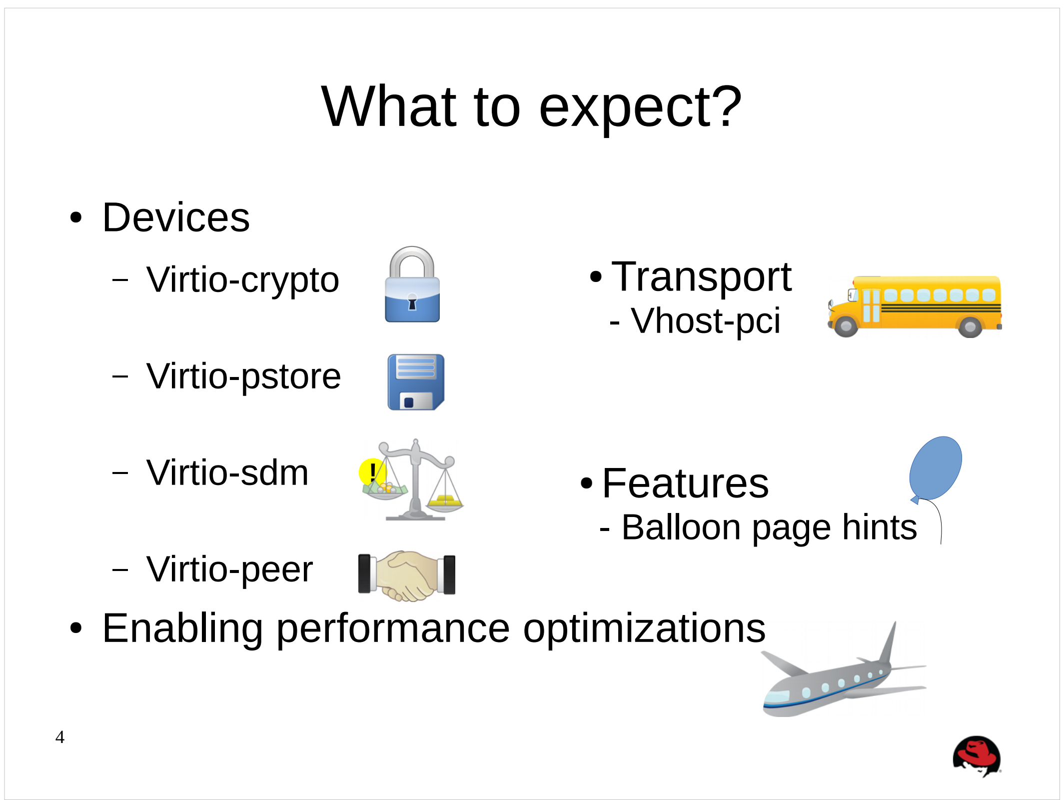


Virtio 1.0 is headed towards becoming an OASIS standard. It's current status is a Committee Specification – which means that it was approved after a public review.
To achieve an OASIS standard status, three statements of use are required. I'm happy to report that we have more than 3 users at all levels of the stack: Within guest, you can use windows or linux kernel level drivers, or userspace dpdk drivers with virtio 1 support. There are virtio 1 firmware drivers in ROMs used by the bios, uefi and slof for network, block and scsi virtio devices. On the host side - besides QEMU - virtio 1 is supported by the dpdk vhost backend, as well as vhost net and scsi in Linux.

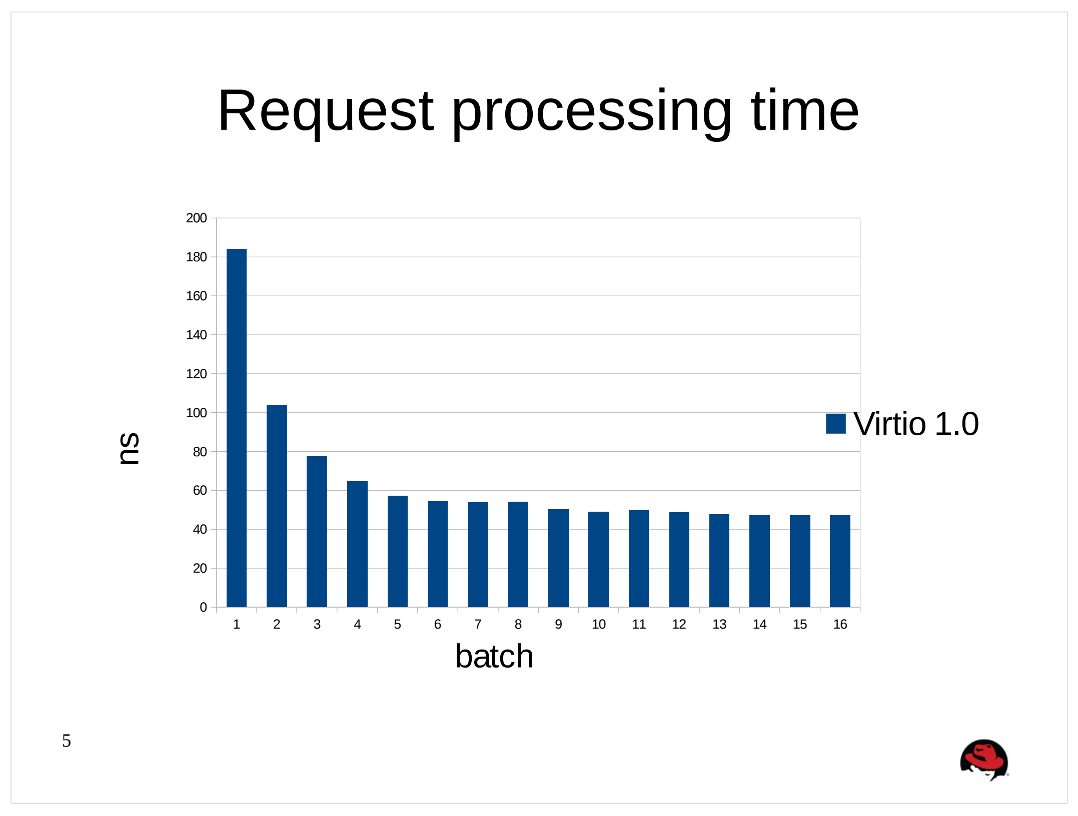


The next virtio spec revision will be virtio 1.1. There, I expect us to move forward with standartization of new interfaces - implementations for which are already upstream: including relatively new devices like input, gpu and socket devices, and possibly the virtio-9p filesystem – if we can find someone to write it up :).

We are also going to standardize recent upstream features such as the virtual IOMMU support which enables secure use of guest userspace drivers with virtio devices.

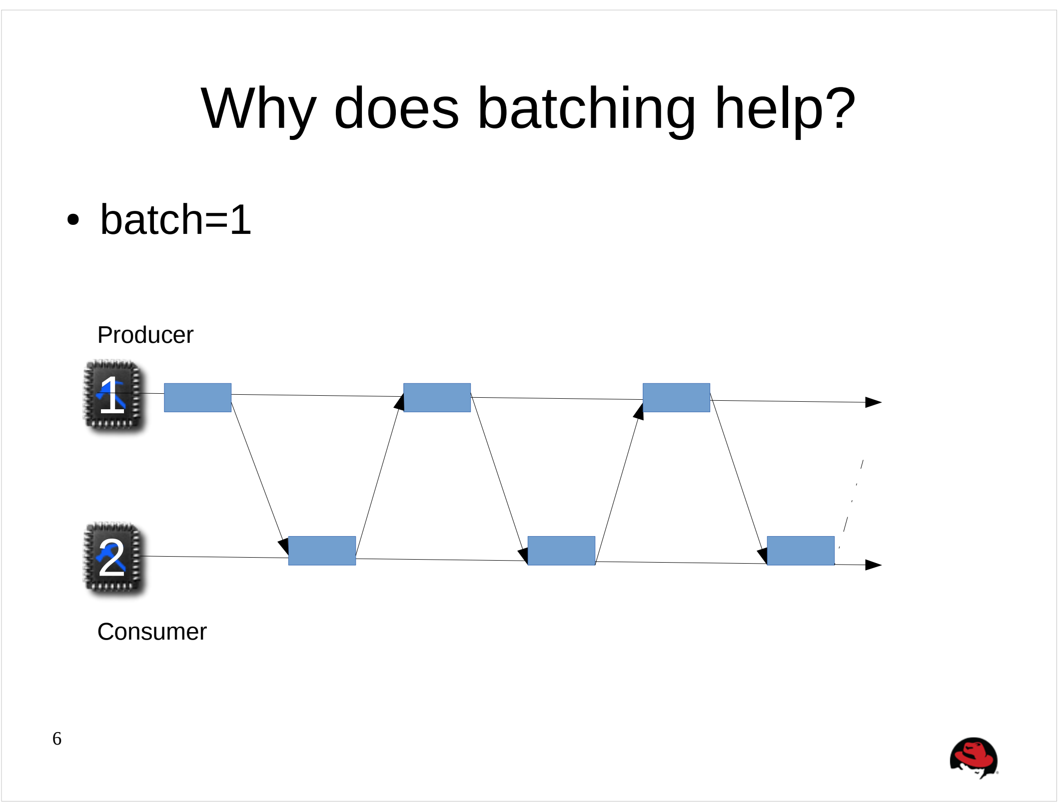


I expect a slew of new devices: virtual crypto and persistent storage devices, a signal distribution module device, possibly a shared memory device. Existing devices will gain new features, for example the balloon will report guest page usage hints to optimize guest memory placement on the host. We might also see completely new transports, in particular vhost pci devices can be used to allow one VM to access memory of another VM using virtio. Finally, we will see changes enabling new performance optimizations – some of which I am going to discuss next.

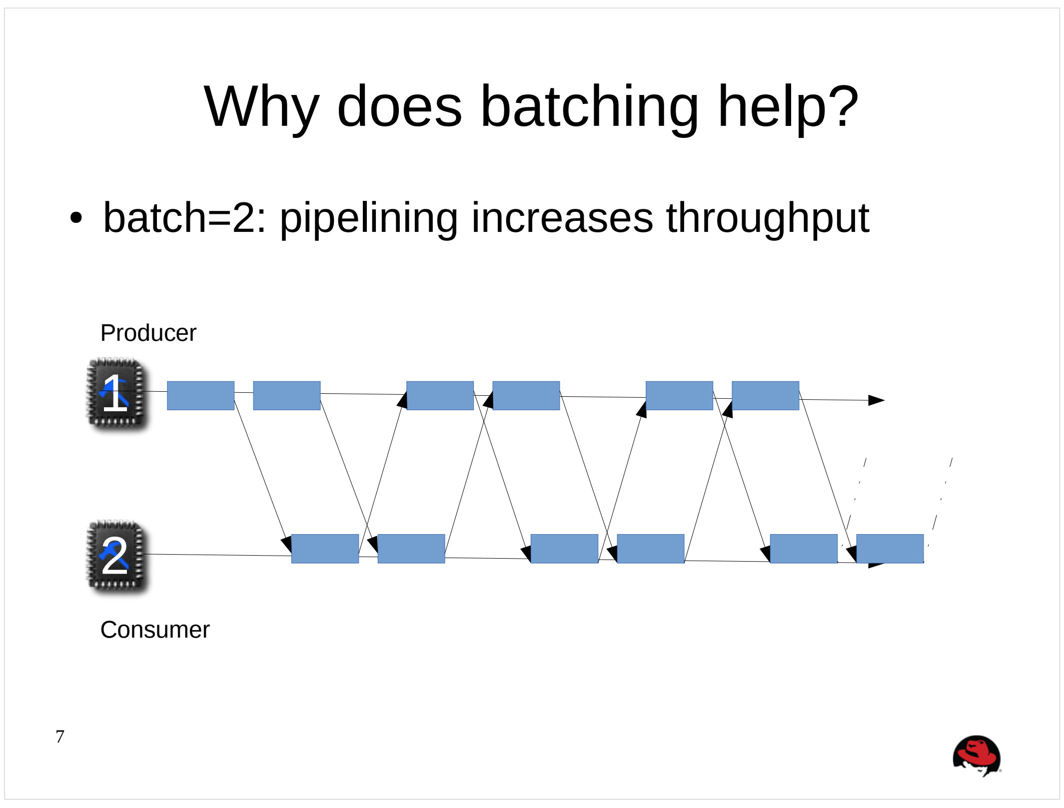


How much overhead does passing messages between CPUs using virtio incur? Here's a chart showing per-request processing time depending on batch size – that is the maximum number of outstanding request buffers in the queue.

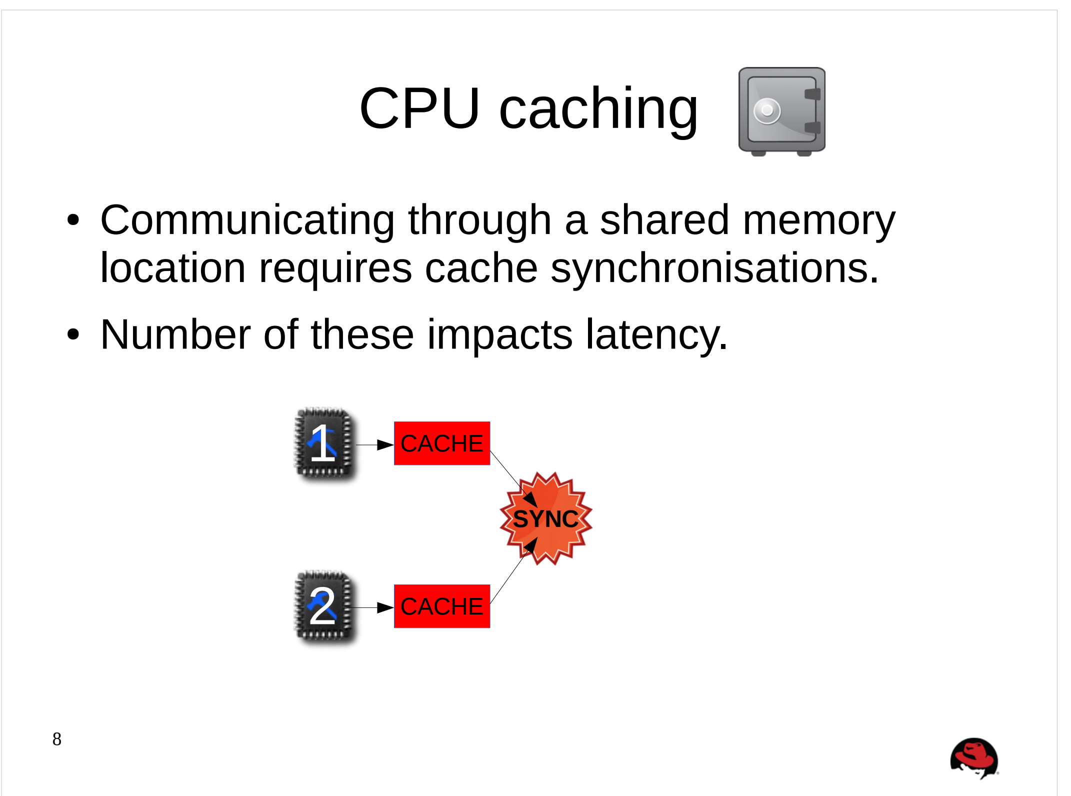
Why does increasing batching help improve the time? This is not about signaling overhead because this test used polling exclusively. There are two main reasons:



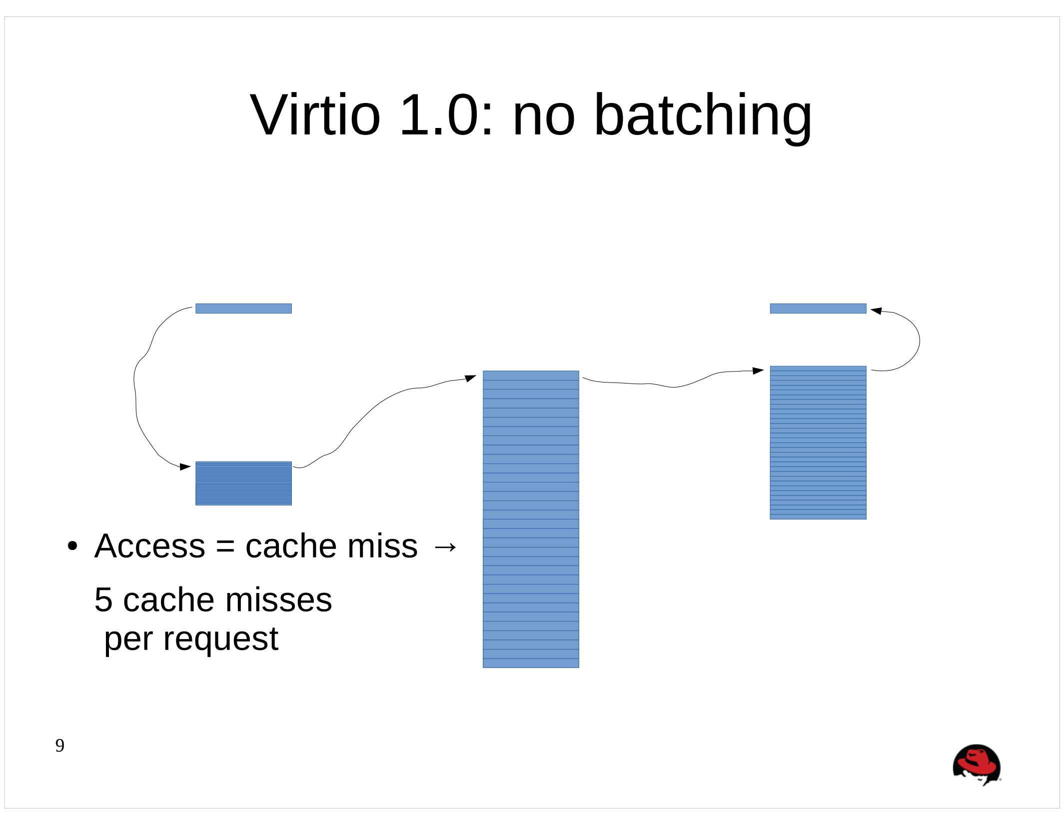
The first has to do with pipelining. If we limit ourselves to a single outstanding buffer, CPUs running both host and guest will spend part of their time idly waiting for more work in case of host/consumer, or for the previous request to be processed in case of guest/producer.



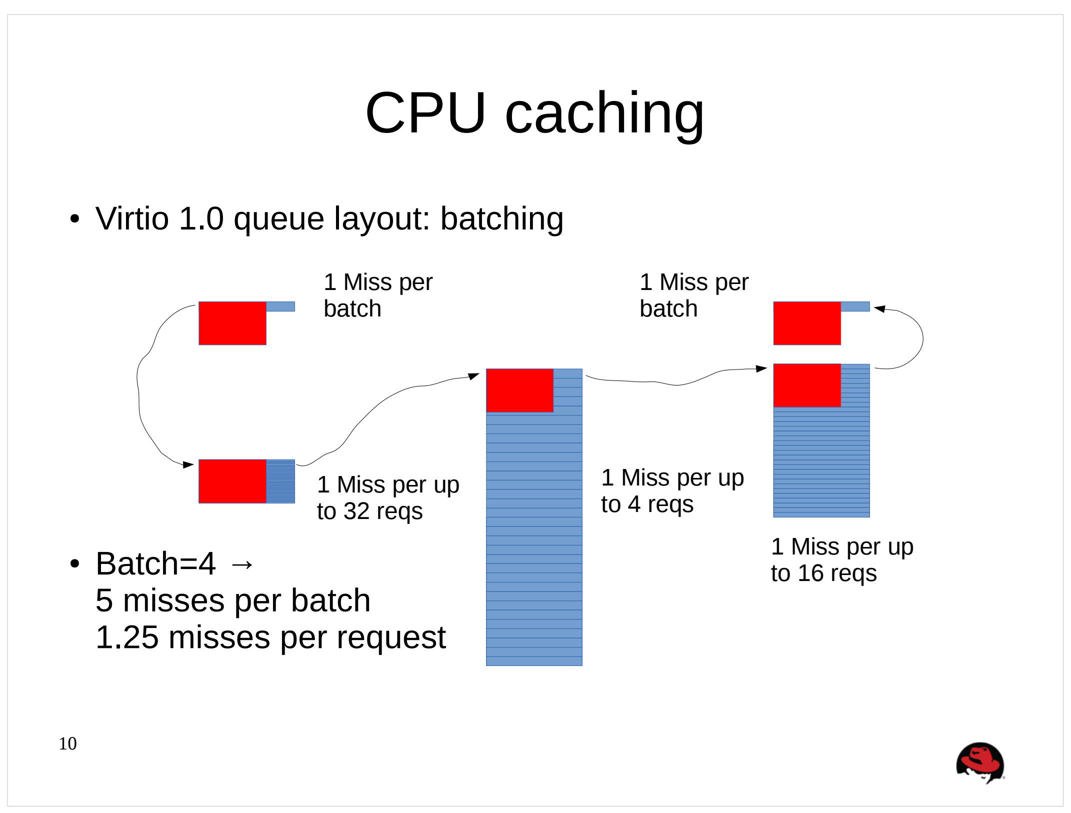
If guest is allowed to submit multiple buffers asynchronously, processing time per buffer decreases as buffers are processed in previously idle time.



Another reason is related to how CPUs communicate using shared memory. When a CPU accesses memory that was previously accessed by another CPU, this might cause a cache synchronisation: which could be for example a cache miss or invalidation depending on the CPU. The number of these cache misses directly impacts the communication latency.

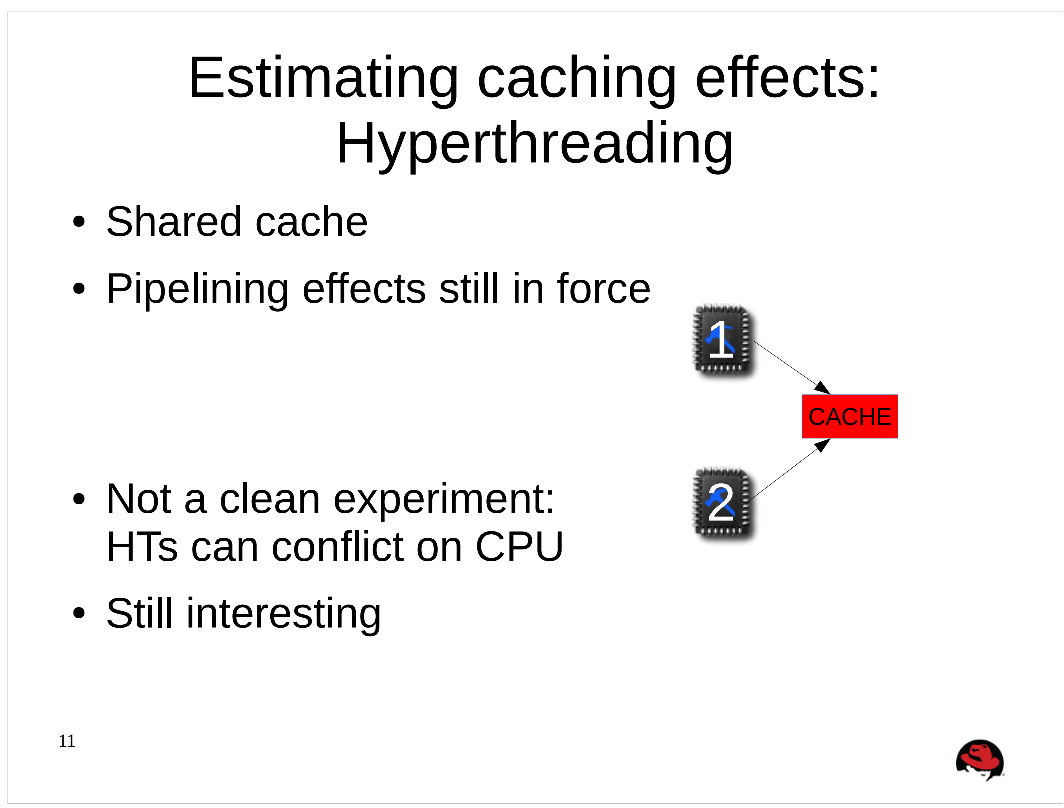


Let's count these misses when using virtio 1.0. The queue consists of 5 parts. As a buffer moves between host and guest, each of these parts is written and read at least once, implying at least 5 cache misses per buffer if no batching is allowed.

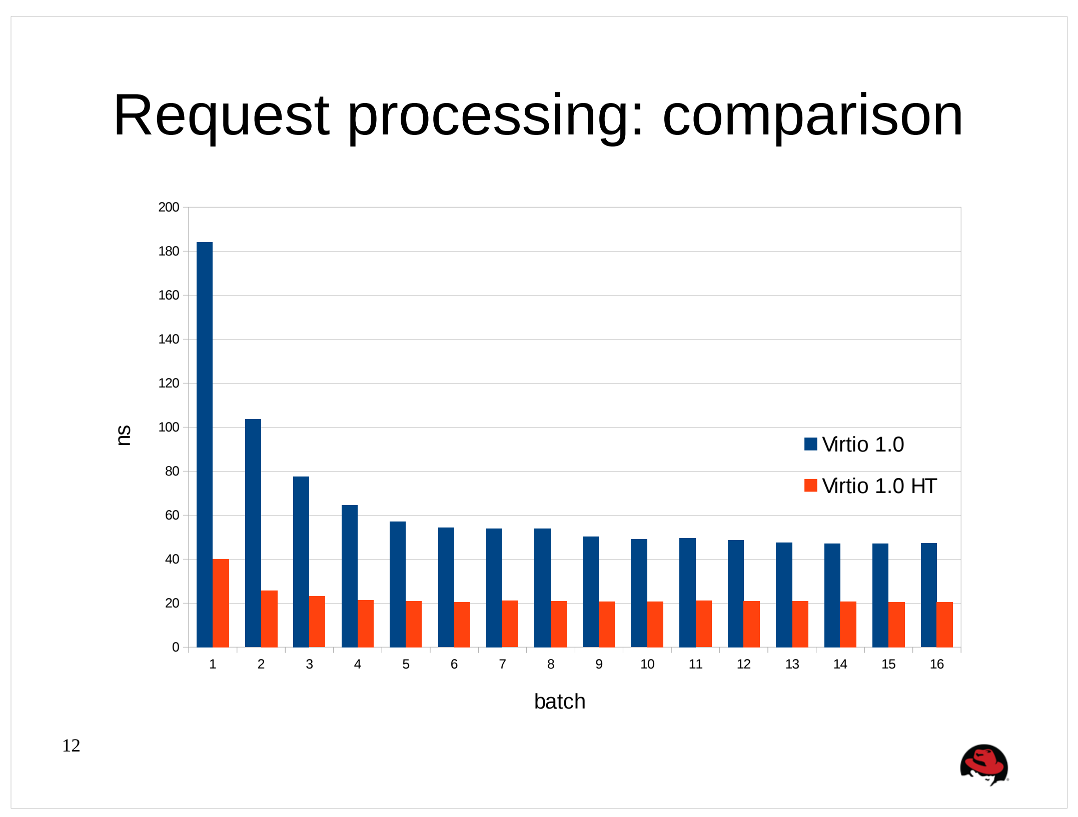


With batching the per buffer overhead is lower, as multiple buffers fit in a single cache line. How many – depends on the cache line and batch size.
With batch size set to 4, a single 64 byte cache line holds 4 entries of each kind. This gives us 5 misses per 4 buffers.

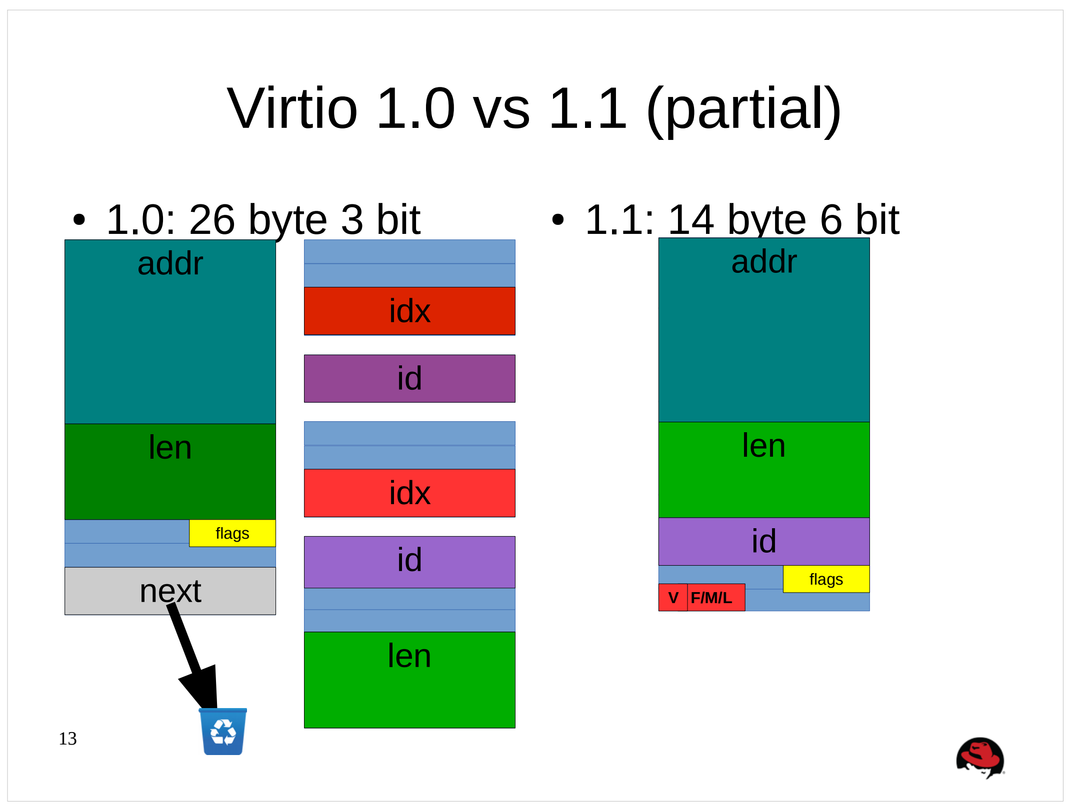
Is there some way to measure how much overhead does each of the two effects contribute?



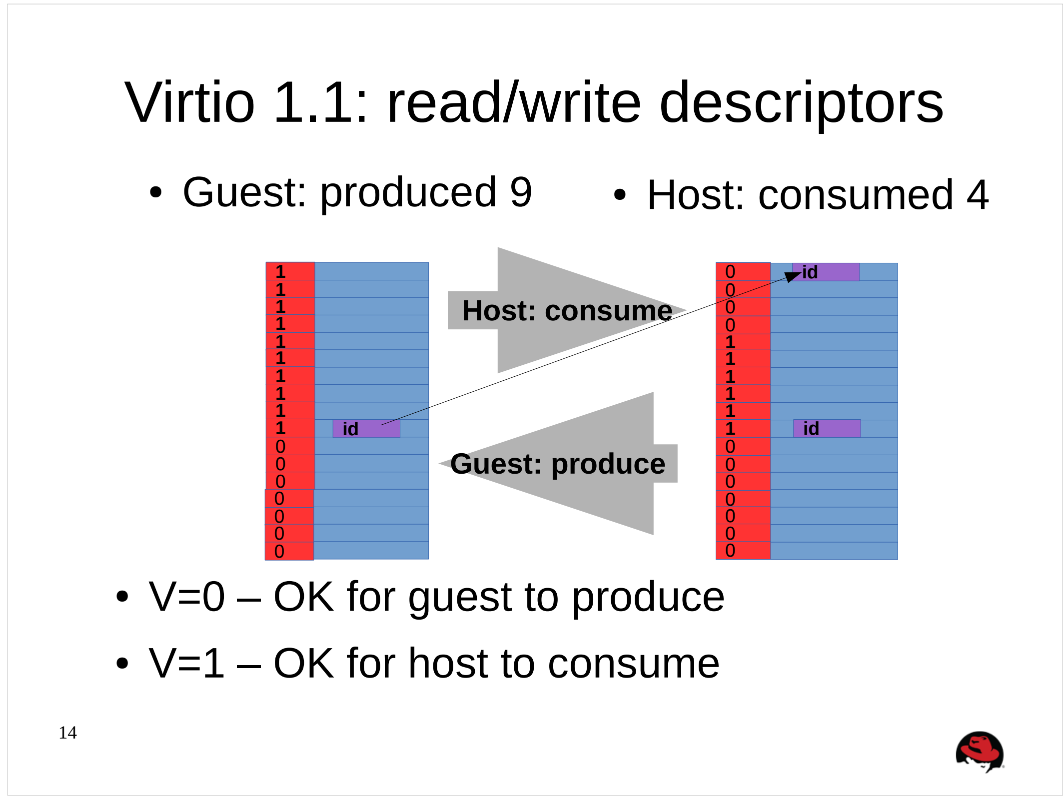
One way to answer this question is by running the test on a hyper-threaded system.
A Hyper-thread is in fact a partition within a physical CPU. Hyper threads share a cache, so using hyper-threading we can eliminate most of the cache misses.
Unfortunately this is not a clean experiment, since hyper-threads might compete for internal CPU resources. But I think it's still interesting.



Most time in this benchmark is spent on cache synchronization - especially without batching. The issue is these 5 queue parts causing cache misses. Can we use something faster in virtio 1.1?



Instead of using 5 data structures, we pack everything into a single 16 byte descriptor. Address, length and flags fields are unchanged. Length is reused to report the used length from host to guest. Id field in the available and used rings allows using buffers out of order, identifying both the request and the descriptor location. We put it in the descriptor itself, storing the descriptors in a ring structure, so the next descriptor pointer is no longer required. What about available and used index? These serve 2 purposes. One is to mark a buffer valid for host or guest use. 1 bit is enough for this. Another is to support adding a batch of multiple descriptors atomically. We label descriptors as first, middle or last in the batch, to the total of 3 instead of 32 bits.



Let's see how is this compact layout is used.
To make buffers available to host, guest writes them out in descriptors in the ring and then sets the valid bit to 1, which implies that it's now ok for host to consume them. Host can process them in any order. Each descriptor has an id field. After processing, host writes the processed descriptor id in the ring, and then flips the valid bit to 0, specifying to guest that an entry has been used. The guest can now reuse this entry for a new buffer.
As you can see this is in fact much easier to implement than virtio 1.0.

Host: pseudo code (in-order)

```
while(!desc[idx].v)      ←miss?
    relax();
process(&desc[idx]);
desc[idx].v = 0;          ←miss?
Idx = (idx + 1) % size;
```

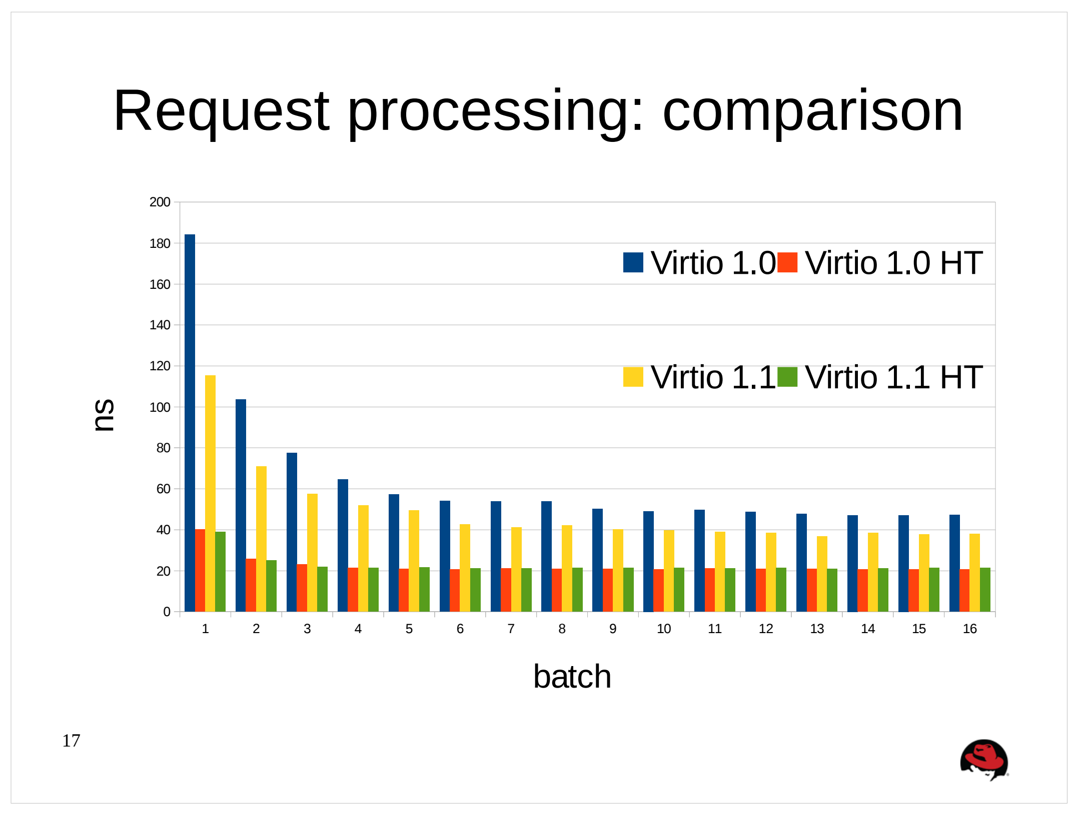
- Write access can trigger miss

Here's a host implementation in pseudo code, this one assumes processing entries in order by host. As you can see, descriptor processing involves reading an entry and later writing it out. In the best case, read will cause a cache miss while write will hit the cache. In the worst case, both accesses cause a cache miss. Guest side looks very similar.

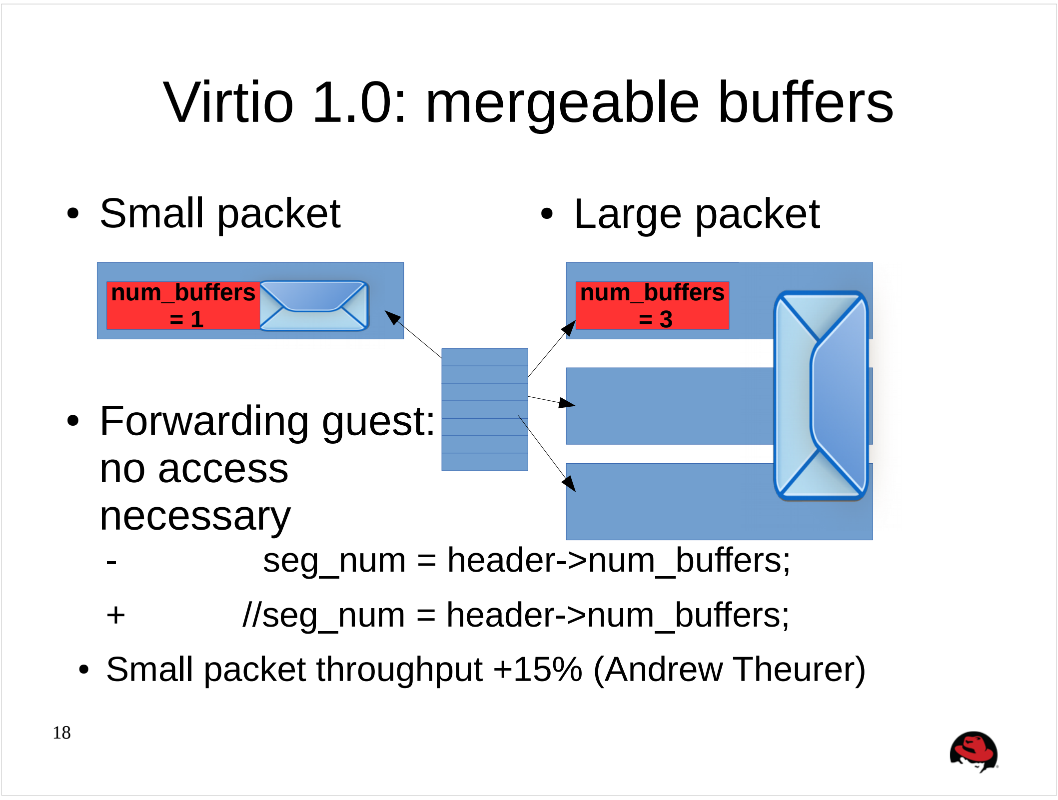
CPU caching

- Both host and guest incur misses on access
- No batching: 2 to 4 misses per descriptor
- Batch=4:
  2 to 4 misses per batch
  4 descriptors per cache line →
  0.5 to 1 misses per descriptor
- Better than virtio 1.0 even in the worst case

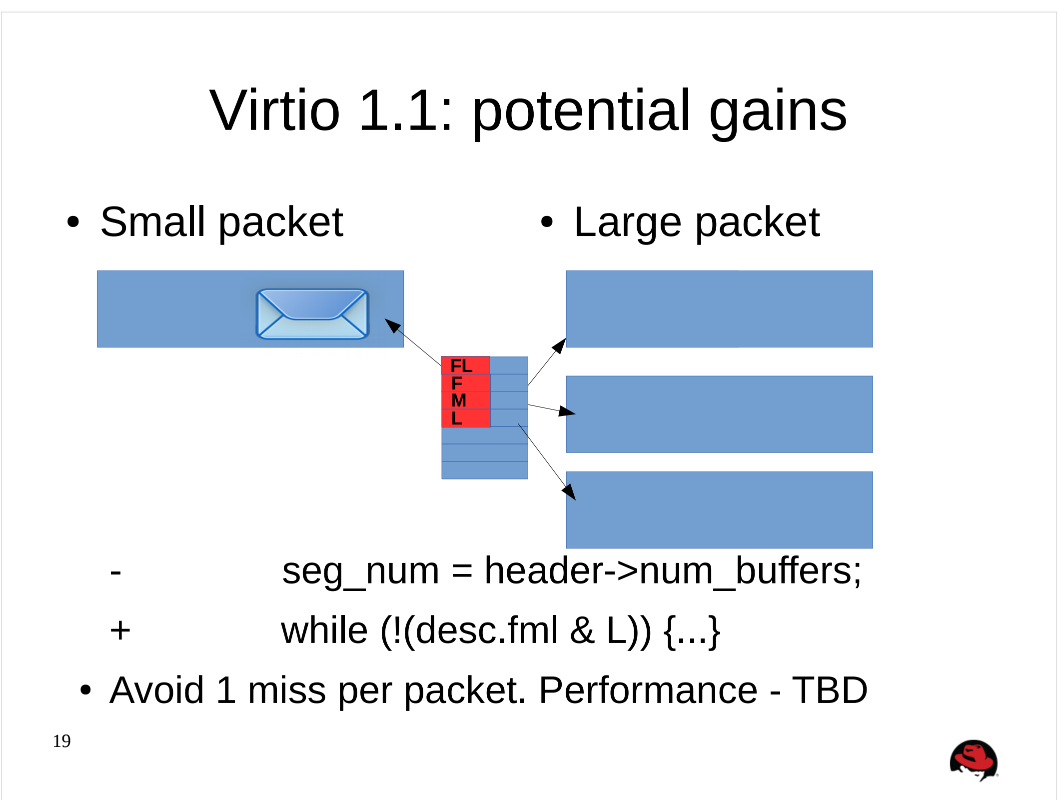
If we do the math, we get 2 to 4 misses per buffer with no batching, and half to one miss when batching up to 4 buffers.
This seems better than virtio 1.0 even in the worst case.
Let's look at some numbers.



As you see this performs pretty well: we see that without batching, we managed to reduce the cache overhead almost by a factor of 2. With batching, the effect is more modest but still very noticeable. I also tested the hyper-threaded performance of the new layout to measure the pipelining and processing overhead. As you can see it performs more or less the same as the old one, so better cache behaviour is the main reason that the new layout is faster.



Extending the ring layout actually allows us to address another performance problem. Consider a networking device. Some workloads exchange predominantly small packets, but still need to be able to handle large packets once in a while. A single large packet then spans mutliple receive buffers. When this happens, a field called “number of buffers” is set by device to a number > 1. Guest must read this field to find out how many buffers are used. Unfortunately this incurs a cache miss overhead per packet. It was reported that simply commenting out a single line of code reading this value increases the throughput by about 15%.



With the new layout, host labels descriptors for a large packet as first, middle and last, so we no longer need the number of buffers to find out where does the packet end – guest can simply look for the last label in the descriptor. I expect to observe the throughput improvement once the new ring layout is integrated in virtio net.

Parallel ring processing?

- Virtio 1.0: workers contend on idx cache line
- Virtio 1.1: can host or guest parallelize?
- If order does not matter
  (e.g. network TX completion):

- Each worker polls and handles its own descriptors

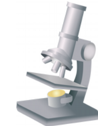
Sometimes the work request order does not matter. For example, this is the case for used transmit packets for a network device: all we need to do is free up the memory so it gets reused. As each descriptor is now completed independently, with the new layout it might be possible to partition the ring such that multiple worker threads process different parts of the ring, each handling a separate cache line.

IO kick / interrupt mitigation

- event index mechanism
  - Similar to avail/used idx
  - Miss when enabling interrupts/IO
- flags mechanism
  - keep interrupts/IO enabled under light load
- first/middle/last to get interrupt per batch
  - Linux: batching using skb->xmit_more

IO and interrupts are expensive, so virtio ring includes two mechanisms for IO and interrupt mitigation: event index and flags.
I did not examine them for cache efficiency yet – both seem to have advantages and disadvantages, this is to be done.
With 1.1 layout there's another option: use first/middle/last labels to deliver 1 interrupt per batch. This is possible if the requests are supplied in batches, as is the case for Linux networking which batches packets on transmit using the xmit_more flag.

Research



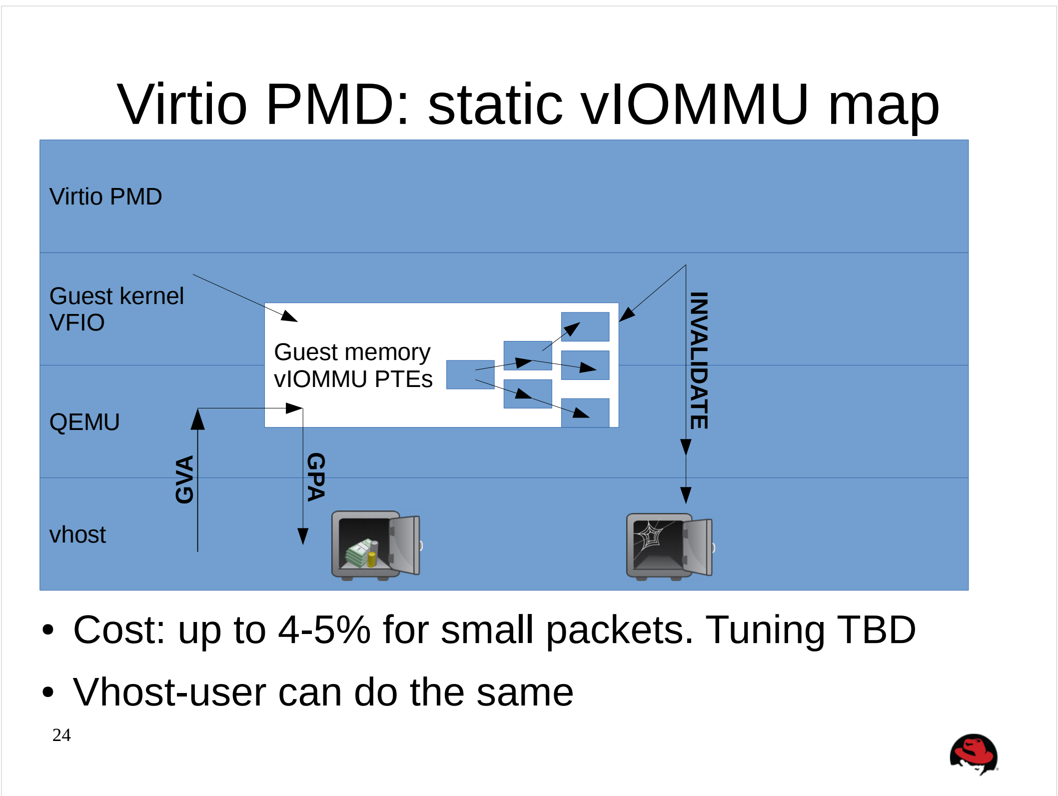
- Rings are RW
  - security issue?
  - Virtio-peer proposal?
- Test on different CPUs
  - AMD (MOESI)
  - ARM
  - Power
- Integrate in existing virtio implementations

What's left to do to benefit from these performance gains? We need to determine whether making descriptors writeable has security implications on any systems.
Testing was done on intel processors so far. We should test more processors and compare the 1.0 and 1.1 performance.
And of course we need to implement it.

VIRTIO_F_IOMMU_PLATFORM

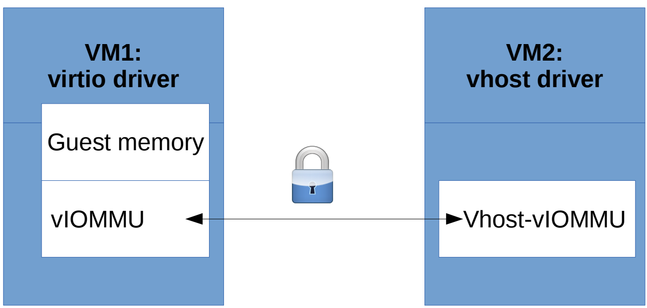
- Legacy: virtio bypasses the vIOMMU if any
  - Host can access anywhere in Guest memory
  - Good for performance, bad for security
- New: Host obeys the platform vIOMMU rules
- Guest will program the IOMMU for the device
- Legacy guests enabling IOMMU will fail
  - Luckily not the default on KVM/x86
- Allows safe userspace drivers within guest

One other recent development is support for virtual IOMMUs. Traditionally virtio ignored a vIOMMU even if guest attempted configuring it. This was done for performance, but it prevents using userspace virtio drivers with VFIO.
In virtio 1.1 there will be a new option to fix this and obey the virtual IOMMU rules.
Latest Linux drivers already learned to handle this option correctly.
If set, older guests that enable the vIOMMU will break – luckily this is off by default at least on x86.



vIOMMU support was recently implemented in vhost. When set, vhost treats all addresses as virtual, and sends each address to QEMU. QEMU will look it up in vIOMMU page tables and respond with a guest physical address. Doing this for each transaction would be too expensive, so vhost implements a translation cache. Whenever guest changes any of the vIOMMU page tables, it sends a cache invalidation notification to QEMU, which is in turn forwarded to vhost.
Current implementation still shows a small performance hit, so we need to work some more on tuning.

- Vhost-pci: VM2 can access all of VM1 memory



- Vhost-vIOMMU can limit VM2 access to VM1

25

Besides guest userspace drivers, we are likely to see new uses for the vIOMMU support. In particular, I mentioned the new vhost-pci proposal where one VM can allow a second VM access to its memory through a virtio device. In this setup, it might be a good idea to implement a vhost IOMMU, to limit the damage that a bug in the second VM can cause to the first VM.

Wild ideas

- Apic programming: about 20% of exits
  - Virtio-apic might help coalesce with host polling?
- Idle – kvm already doing some polling
  - Virtio-idle and combine with vhost polling?
- Kgt – write-protect kernel memory in EPT
  - Extend virtio-ballon page hints?

Here are some wilder ideas – I hope that listing them will help spur more innovation.
We have a proposal for a signal module ditribution device – should we extend this to a full-featured virtio apic? Maybe - some networking workloads spend about 20% of their exits on APIC programming. Today, whenever a VCPU tries to go idle, KVM does some polling. So, would a polling virtio-idle driver make sense? A kernel guard technology project is a hypervisor with hypercalls to control the EPT – such as write-protecting guest kernel text memory. As we are extending the balloon device with guest page hints – maybe this can provide similar functionality.

Implementation projects

- Indirect descriptors – extra indirection
  - when not to use?
- Vhost polling
  - Scalability with overcommit – better integration with the scheduler?
- Error recovery
  - Host errors: restart backend transparently
  - Guest errors: guest to reset device

There are also lots of projects which do not require changing the host/guest interfaces. Here are some:
- linux always uses indirect descriptors to pack maximum descriptors per ring – should we avoid this if the descriptor list is very short?
- vhost polling is effective in reducing latency but comes at a cost of high host cpu utilization. Can we get hints from the linux scheduler to do better when the host is busy?
- we need to be better about recovering from errors. We can recover from host errors by restarting the backend transparently, and from guest errors by making guest detect them and reset the device.

Contributing

- Implementation
  - virtualization@lists.linux-foundation.org
  - qemu-devel@nongnu.org
  - … if in doubt – copy more
- Spec (must copy on interface changes)
  - virtio-dev@lists.oasis-open.org
- Driving spec changes
  - Report: virtio-comment@lists.oasis-open.org
  - https://issues.oasis-open.org/browse/VIRTIO

How does one contribute? Implementation patches just should be posted to the correct mailing lists. Two main ones are virtualization and qemu-devel for Linux and qemu patches respectively. If one is unsure – it's generally better to copy widely.
If your patch affects the host/guest interface, you should copy the virtio-dev mailing list.
If you are proposing a spec change, you should copy the virtio-comment mailing list. Virtio TC members can then create an issue and vote on accepting the proposed resolution in the next spec revision.

Summary

- Virtio 1.1 is shaping up to be a big release
  - Performance
  - Security
  - Features
- Join the fun

  - Spec is open: 9 active contributors / 7 companies

  - Implementations are open > 60 active contributors in the last year

To summarize: virtio 1.1 is shaping up to a be a big release, with performance and security enhancements and a slew of new features.

Contribution is open to everyone – so join us!