Message Passing Workloads in KVM

David Matlack, dmatlack@google.com

Overview

Message Passing Workloads

Loopback TCP_RR

IPI and HLT

DISCLAIMER: x86 and Intel VT-x

Halt Polling

Interrupts and questions are welcome!

Message Passing Workloads

- Usually, anything that frequently switches between running and idle.
- Event-driven workloads
  - Memcache
  - LAMP servers
  - Redis
- Multithreaded workloads using low latency **wait/signal primitives** for coordination.
  - Windows Event Objects
  - pthread_cond_wait / pthread_cond_signal
- Inter-process communication
  - TCP_RR (benchmark)

Message Passing Workloads

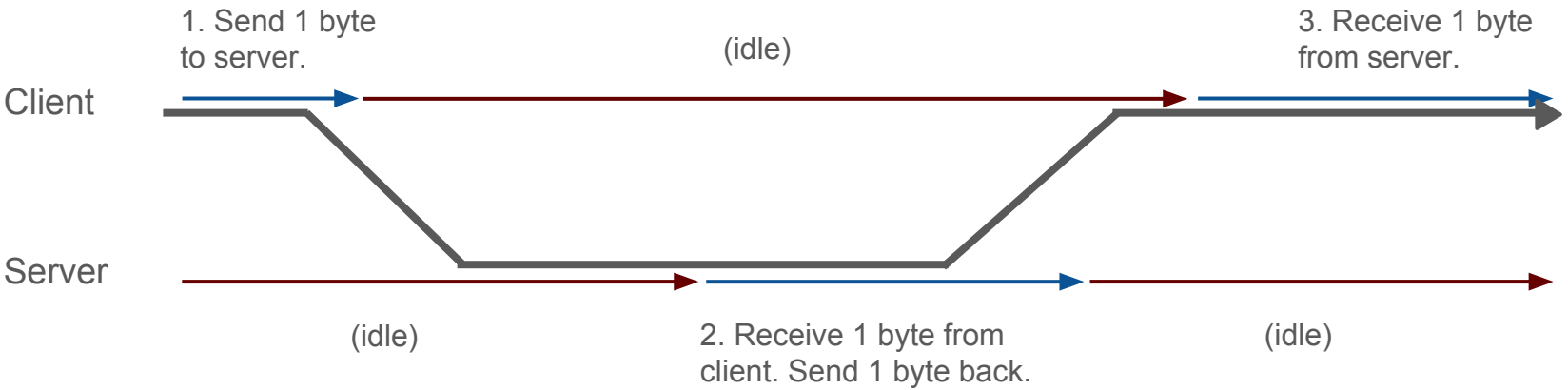
Intuition: *Workloads which don't involve IO virtualization should run at near native performance.*

Reality: Message Passing Workloads may not involve any IO but will still perform nX worse than native.

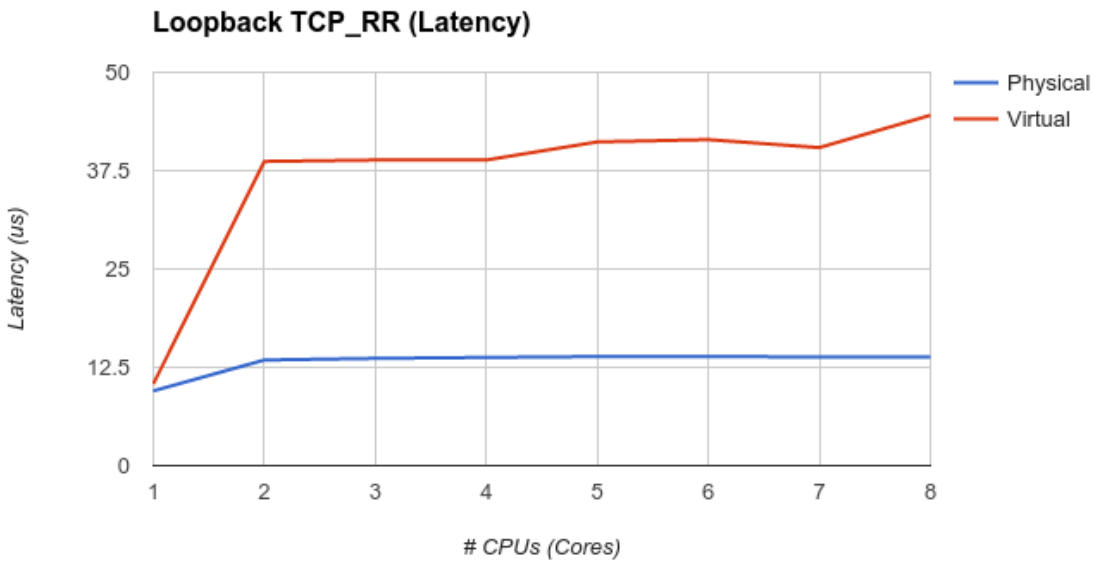
- (loopback) Memcache: 2x higher latency.
- Windows Event Objects: 3-4x higher latency.

Message Passing Workloads

- Microbenchmark: Loopback TCP_RR
  - Client and Server ping-pong 1-byte of data over an established TCP connection.
  - Loopback: No networking devices (real or virtual) involved.
  - Performance: Latency of each transaction.
- One transaction:



Loopback TCP_RR Performance



Host:
IvyBridge
3.11 Kernel

Guest:
Debian Wheezy Backports
(3.16 Kernel)

3x higher latency
25 us slower

Virtual Overheads of TCP_RR

- Message Passing on 1 CPU
  - Context Switch
- Message Passing on >1 CPU
  - Interprocessor-Interrupts
- What's going on under the hood?
- VMEXITs are a good place to start looking.
- KVM has built-in VMEXIT counters and timers.
  - `perf-kvm(1)`

Virtual Overheads of TCP_RR

	Total Number of VMEXITs		VMEXITs / Transaction	
	1 VCPU	**2 VCPU**	**1 VCPU**	**2 VCPU**
EXTERNAL_INTERRUPT	16705	12371	0.02	0.07
MSR_WRITE	2599	1704334	0.00	**9.58**
IO_INSTRUCTION	1786	762	0.00	0.00
EOI_INDUCED	613	25	0.00	0.00
EXCEPTION_NMI	289	31	0.00	0.00
CPUID	252	112	0.00	0.00
CR_ACCESS	171	272	0.00	0.00
HLT	34	354393	0.00	**1.99**
EPT_VIOLATION	2	0	0.00	0.00
PAUSE_INSTRUCTION	0	2014	0.00	0.01

- 2 HLT per Transaction
- 10 MSR_WRITE per Transaction

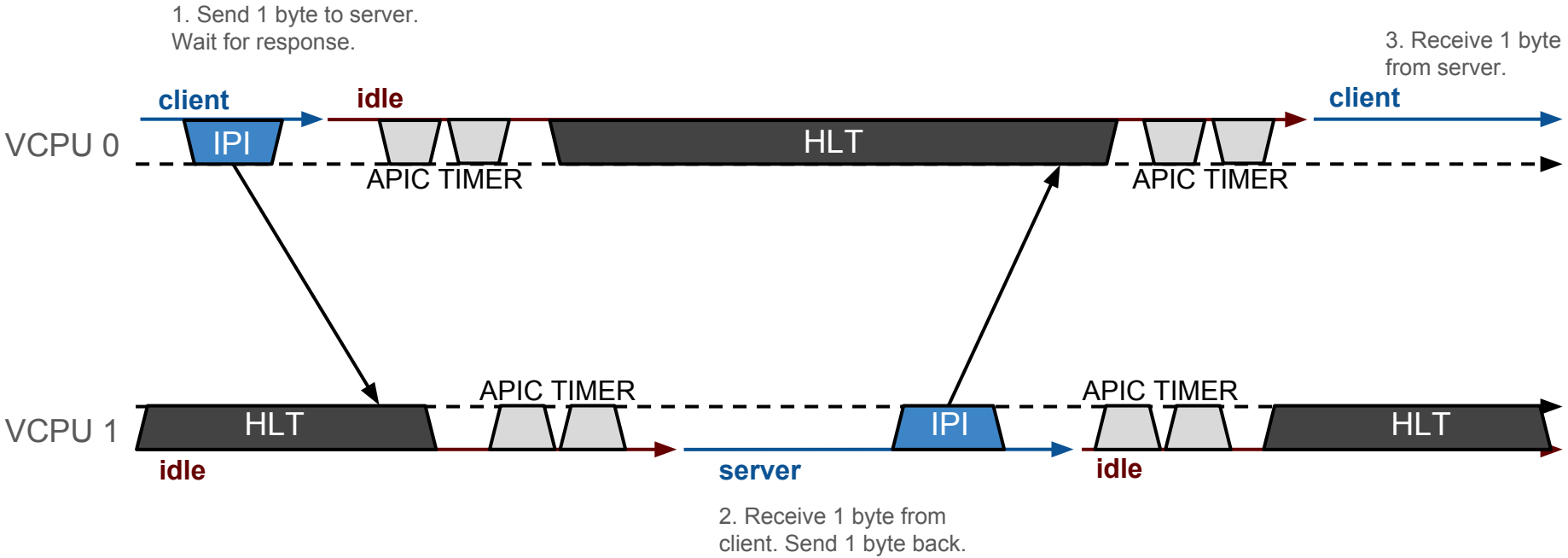
HLTs of TCP_RR

- 2 HLT
  - CPU instruction.
  - Stop executing instructions on this CPU until an interrupt arrives.
- VCPU wishes to stop executing instructions.
  - Guest OS has decided that there is nothing to do.
  - Nothing to do == idle.
- Message passing workloads switch between running and idle...

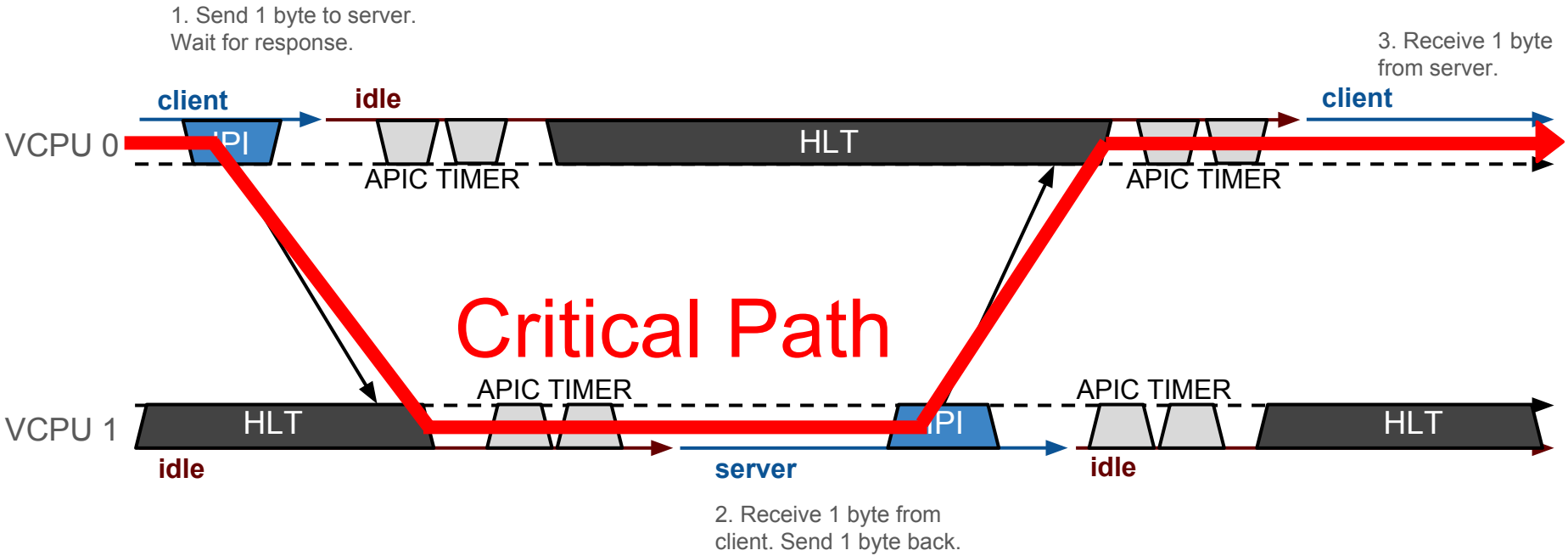
MSR_WRITEs of TCP_RR

- 10 MSR_WRITE
  - "Write to Model Specific Register" instruction executed in the guest.
- 8 APIC Timer "Initial Count" Register (MSR 838)
  - Written to start a **per-CPU timer**.
  - "Start counting down and fire an interrupt when you get to zero."
  - Artifact of NOHZ guest kernel.
- 2 APIC Interrupt Command Register (MSR 830)
  - Used to send **interprocessor-interrupts (IPI)**.
  - Used to deliver "messages" between client/server processes running on separate CPUs.

VMEXITs of TCP_RR



VMEXITs of TCP_RR

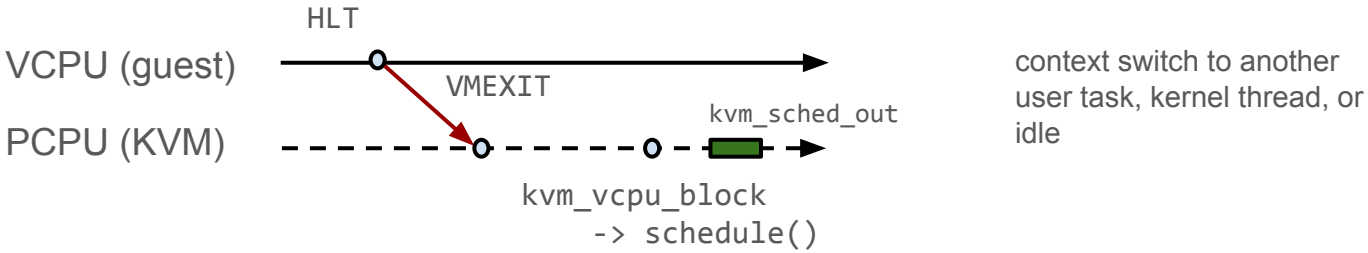


APIC Timer "Initial Count" Register

- 8 per transaction
  - 4 on the critical path
- NOHZ (tickless guest kernel)
  - "Disable" scheduler-tick upon entering idle.
  - "Enable" scheduler-tick upon leaving idle.
  - scheduler-tick == APIC Timer (could also be TSC Deadline Timer)
- Why 2 writes per transition into/out of idle?
  - `hrtimer_cancel`
  - `hrtimer_start`
- Adds 3-5 us to round-trip latency.

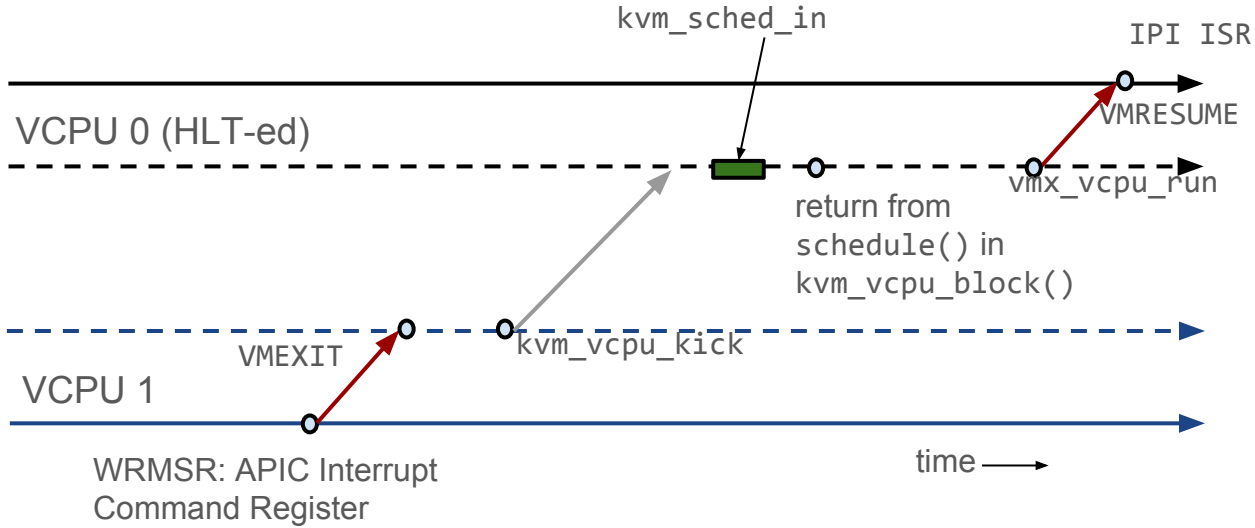
HLT

- HLT:
  - x86 Instruction.
  - CPU stops executing instructions until an interrupt arrives.
  - This part of HLT is not on the critical path!
- How it works in KVM
  - Place VCPU thread on a wait queue.
  - Yield the CPU to another thread.



IPI+HLT

- Sending an IPI to wake up a HLT-ed CPU.
  - On the critical path!



* VMEXIT and VMRESUME implemented in Hardware.

IPI+HLT

- Sending an IPI to wake up a HLT-ed CPU.
  - On the critical path!
- Same operation on bare metal is entirely implemented in hardware.
- How much overhead from virtualization?
  - Unlike APIC_TMICT, can't just time VMEXITs.
- We can compare with the same operation on physical hardware.

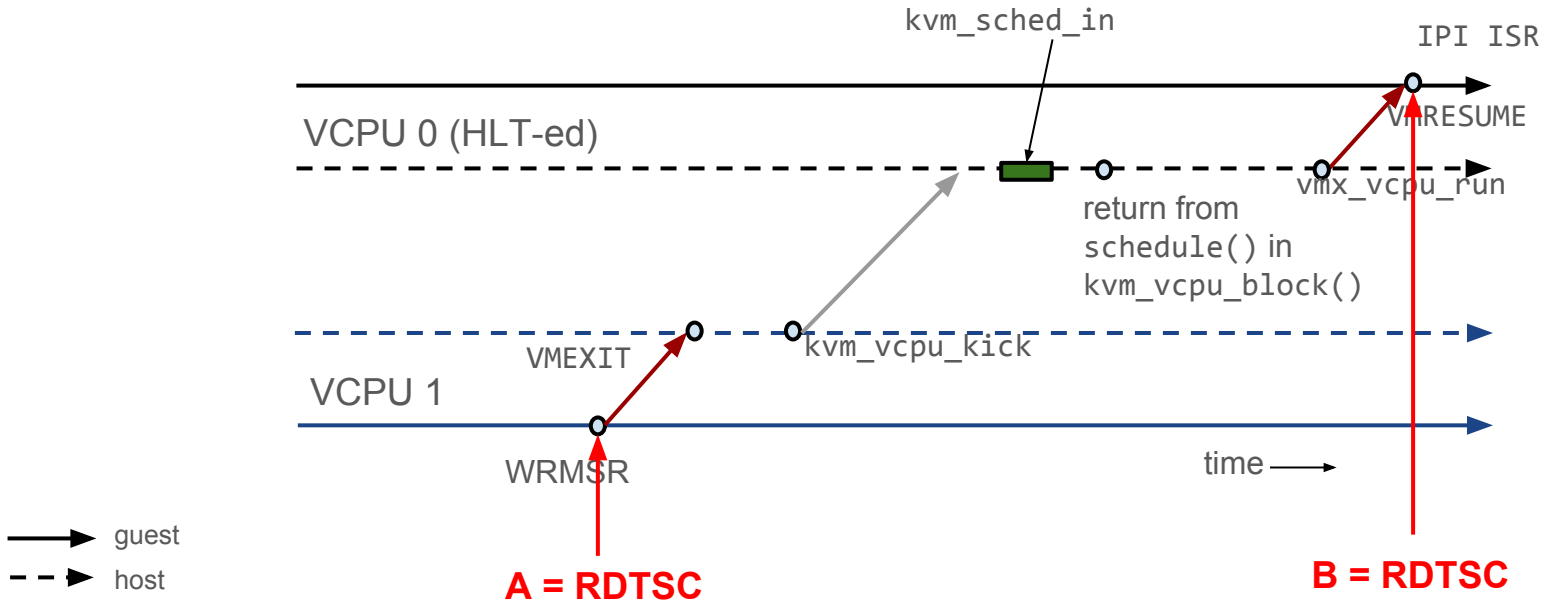
KVM versus Hardware

Ring 0 Microbenchmark (kvm-unit-tests)

1. VCPU 0: HLT.
2. ~100 us delay
3. VCPU 1: **A = RDTSC**
4. VCPU 1: Send IPI to [V]CPU 0.
5. VCPU 0: **B = RDTSC** (first instruction of IPI ISR).
6. **Latency = B - A**
7. Repeat.

Run in KVM guest and on bare-metal. Compare!

KVM versus Hardware



KVM versus Hardware

- Median: KVM is **12x slower**
- Pathological case (witnessed): KVM is **400x slower**
- Best case (witnessed): KVM is **11x slower**
- KVM: **5.7 us**; Hardware: **0.5 us**

	Cycles	
	KVM	**Hardware**
Min	13700	1200
Average	15800	1200
50%ile	14900	1200
90%ile	16000	1300
99%ile	24900	1300
Max	521000	1400

Host:
SandyBridge @ 2.6 GHz
3.11 Kernel

KVM performance is similar on IvyBridge (5.6 us) and Haswell (4.9 us).

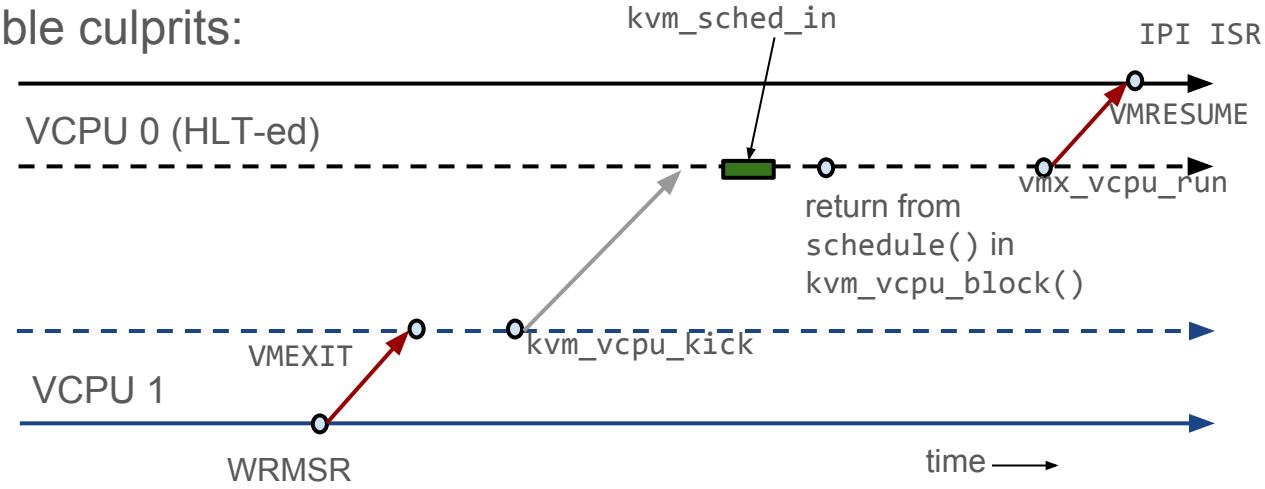
KVM versus Hardware

Notes about this benchmark:

- No guest FPU to save/restore.
- Host otherwise idle (VCPU context switches to idle on HLT).
- Host power management not the culprit.

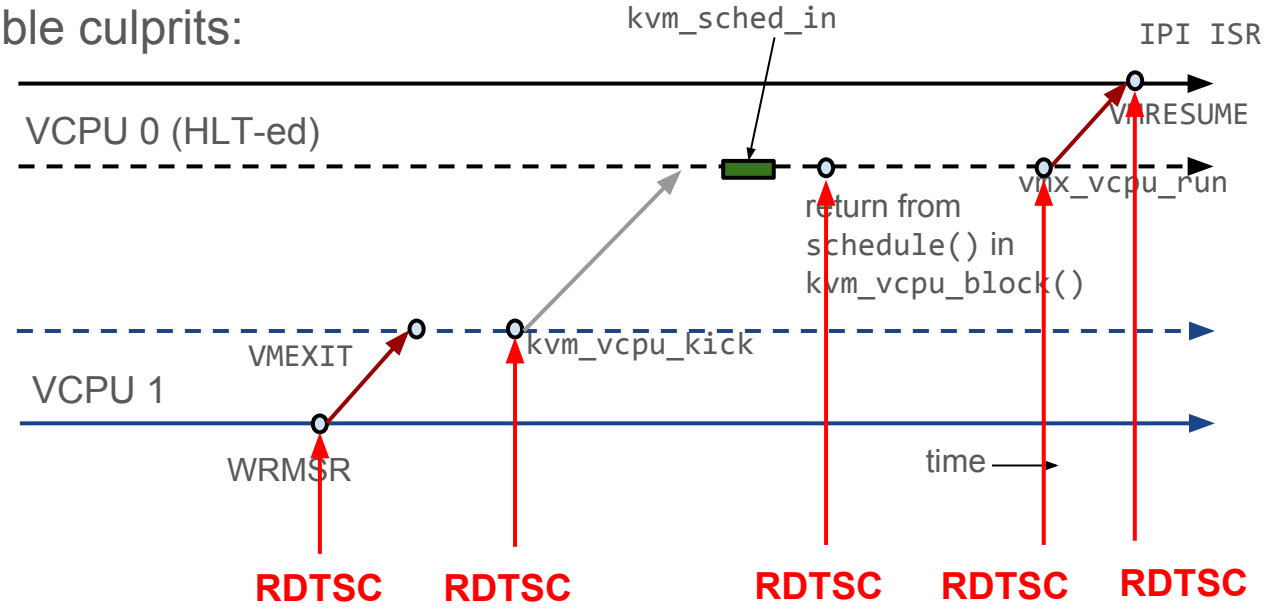
KVM HLT Internals

- So KVM is slow at delivering IPIs and/or coming out of HLT.
- But why?
- Possible culprits:

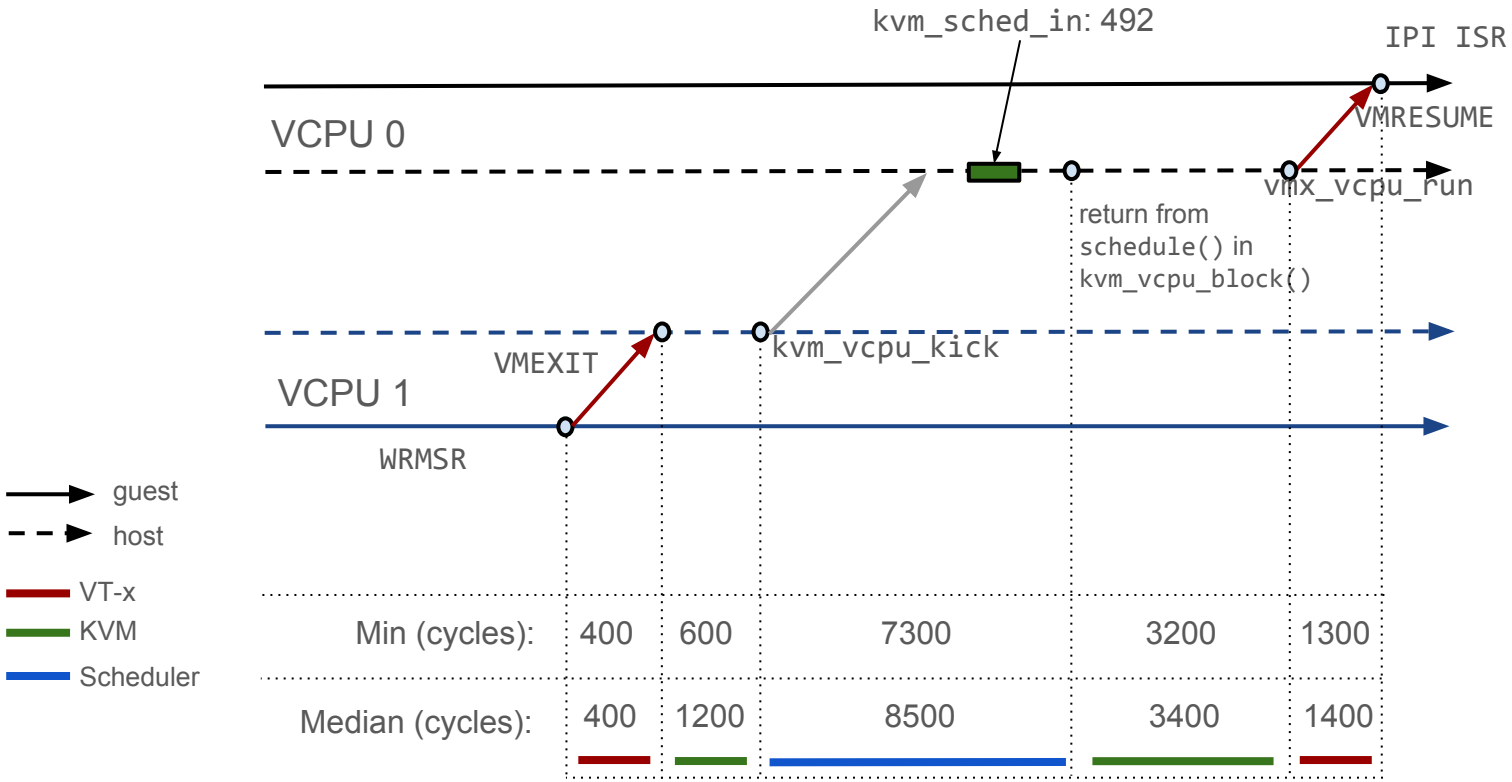


KVM HLT Internals

- So KVM is slow at delivering IPIs and/or coming out of HLT.
- But why?
- Possible culprits:



KVM HLT Internals



KVM HLT Internals

- Unsurprisingly, the scheduler takes some time to run the VCPU
  - Slow even in the uncontended, cache-hot, case.
  - Imagine if the VCPU is contending for CPU time with other threads.
- Experiment: Don't schedule on HLT.
  - Just poll for the IPI in kvm_vcpu_block.

Never schedule!

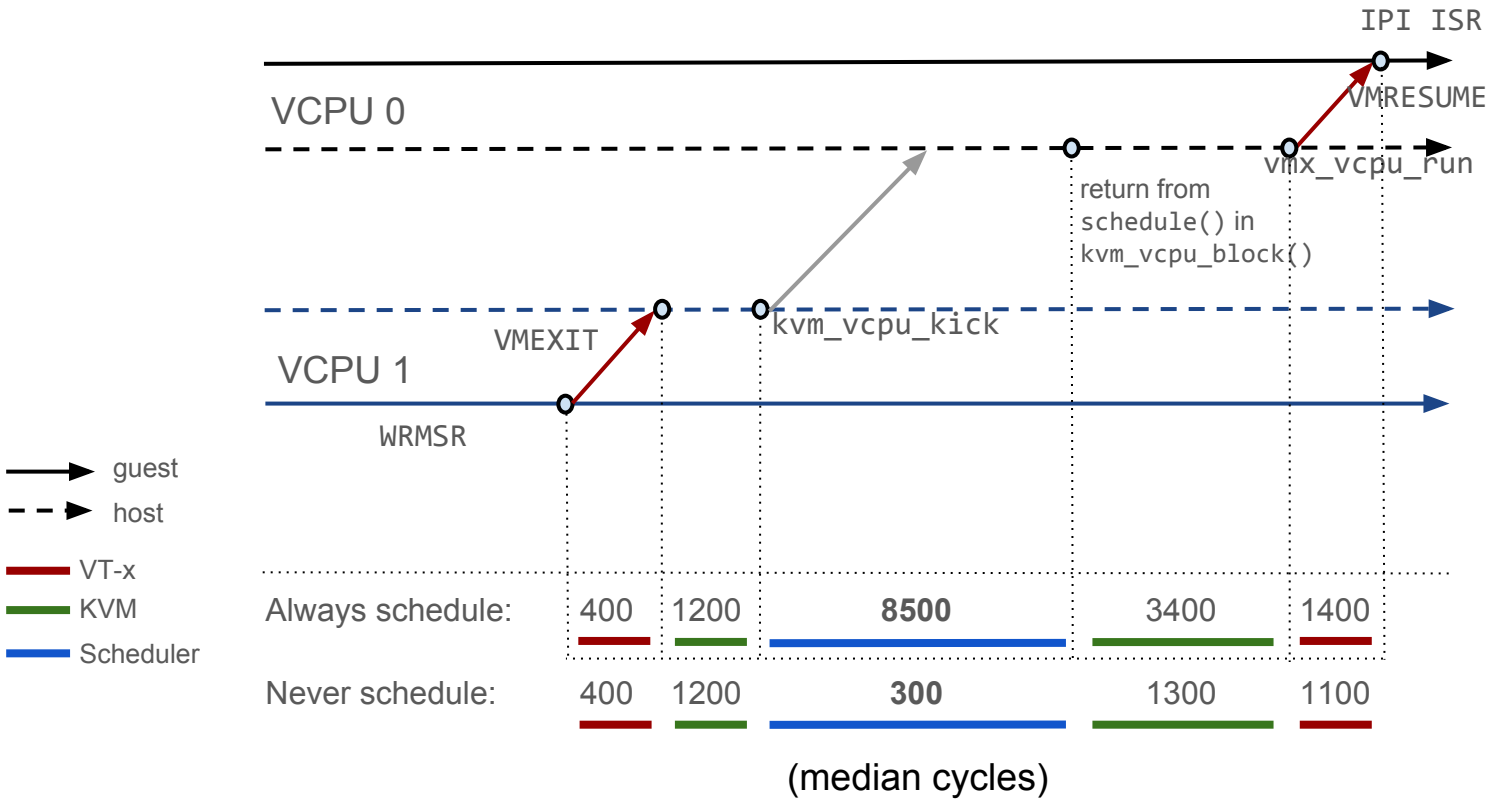
- What happens when you don't schedule on HLT?

	Cycles		
	KVM (Always schedule)	**KVM (Never schedule)**	**Hardware**
Min	13800	4000	1200
Average	15800	4400	1200
50%ile	14900	4300	1200
90%ile	16000	4500	1300
99%ile	24900	6900	1300
Max	521000	50000	1400

- KVM (Always schedule) **5.7 us**
- KVM (Never schedule) **1.7 us**
- Hardware (SandyBridge) **0.5 us**

Similar improvements on IvyBridge (5.6 us -> 1.6 us) Haswell (4.9 us -> 1.5 us).

Never schedule!



Never schedule!

- We eliminate almost all of the latency overhead by not scheduling on HLT.
- Scheduling is often the right thing to do.
  - Let other threads run or save host CPU power.
- Most of the time improves guest performance (let the IO threads run!).
- Can hurt performance.
  - See microbenchmark. See TCP_RR.

Halt-Polling

Step 1: Poll

- For up to X nanoseconds:
  - If a task is waiting to run on our CPU, go to Step 2.
  - Check if a guest interrupt arrived. If so, we are done.
  - Repeat.

Step 2: schedule()

- Schedule out until it's time to come out of HLT.

Pros:

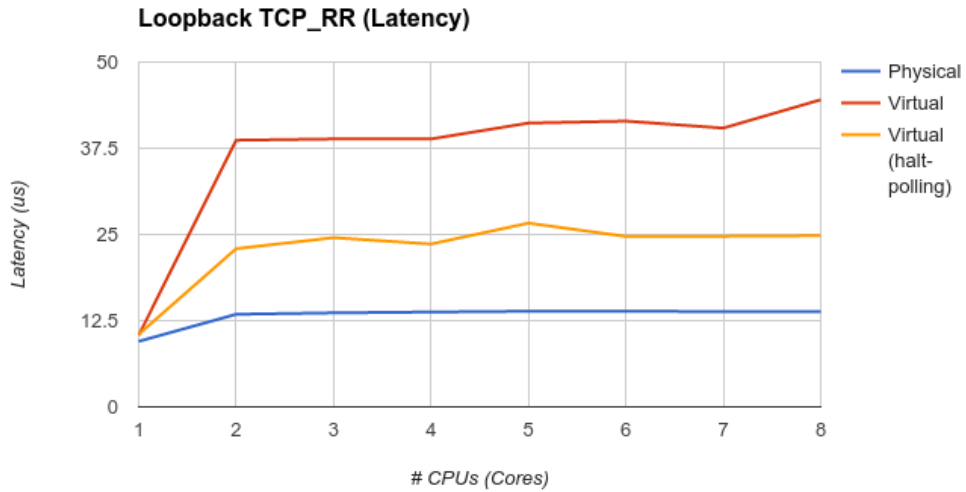
- Works on short HLTs (< X ns)
- VCPUs *continue* to not block the progress of other threads.

Cons:

- Increases CPU usage (~1% for idle VCPUs if X=200,000 ns)
  - Does not appear to negatively affect turbo of active cores.

Halt-Polling

- Memcache: 1.5x latency improvement
- Windows Event Objects: 2x latency improvement
- Reduce message passing latency by 10-15 us (including network latency).



Halt-Polling

- Merged into the 4.0 kernel
  - [PATCH] kvm: add halt_poll_ns module parameter
    - Thanks to Paolo Bonzini
  - Use the KVM module parameter **halt_poll_ns** to control how long to poll on each HLT.
- Future improvements:
  - Automatic poll toggling (remove idle CPU overhead by turning polling off).
  - Automatic halt_poll_ns
    - KVM will set (and vary) halt_poll_ns dynamically.
    - How to do this is an open question... ideas?
  - Lazy Context Switching
    - Equivalent feature, but available for any kernel component to use.

Conclusion

- Message Passing
  - Even loopback message passing requires virtualization.
  - Being idle (as a Linux guest) requires virtualization.
  - Cross-CPU communication requires virtualization.
- Halt-Polling saves **10-15 us** on message passing round-trip latency.
- Remaining round-trip latency:
  - 4 MSR writes to the APIC timer (3-5 us)
  - IPI send (~2 us)
  - HLT wakeup (even with halt-polling, still adds ~3 us!)