KVM and Big VMs

KVM Forum 2012

Andrew Theurer
IBM Linux Technology Center

Topics

- Motivation
- Current state
- NUMA
- Locking
- IO
- Example workload: OLTP

Motivation

- Why Big VMs?
  - Virtualization not just about consolidating under-utilized servers
  - There are workloads which are “big” on bare-metal
    - Users would like to move those to “the cloud”
  - Perhaps some day all enterprise servers will have hypervisor built in
  - KVM should be able to do anything bare-metal can do well

How Well Do KVM VMs scale today?

- Quite well!
- In April, we published an SAP benchmark with 80 vCPU VM [1]
  - This is #1 among the virtualized x86 results:
    - IBM x3850X5 with KVM: `10700 users`
    - Cisco UCS B230 M2 with KVM: `5100 users`
    - Fujitsu RX300 S6 with VMware: `4600 users`
- We also demonstrated #1 disk I/O result [2]:
    - 1.6 million disk I/O ops/sec for host (4k random read/write)

Is there more we can do?

- Yes, of course
  - NUMA
  - Locking
  - Virtual IO

NUMA

- The use of a NUMA topology within a VM is important for two reasons:
  - Promoting a CPU-Memory locality
    - This requires help from the host to place vCPUs and memory properly
  - Maintaining “data partitioning”
    - This can at times be far more important than CPU-Memory locality!
    - The OS likes to partition resources based on NUMA topology
- You can specify a NUMA topology for a VM today, but this is not done automatically

NUMA and Data Partitioning

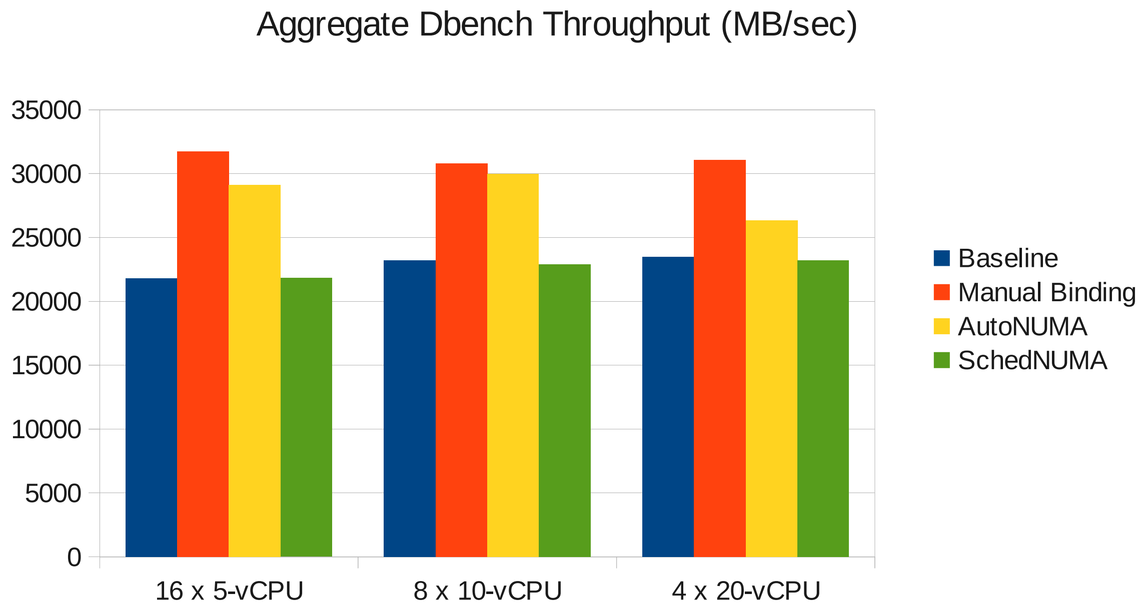
- The number of NUMA nodes inside the VM directly effects the VM's performance
  - Kernel compile on 80-vcpu VM on 80-cpu host:
  - 1 NUMA node: `292 seconds`
  - 4 NUMA nodes (same as host): `189 seconds` `(54% better performance)`
- This is because many locks are per-node, and more nodes = finer grain locks, less lock contention
  - 42% reduction in total lock-wait time (as seen by /proc/lock_stat)
  - 97% reduction in zone->lru_lock wait time

NUMA and CPU-memory Locality

- Current Linux host scheduler does not do enough to keep vCPUs and memory [for same VM] node-local
- This is further complicated with very large VMs, where vCPUs and memory cannot be contained in a single host NUMA node [like previous example with 80-vCPU VM]
- The optimal host will recognize smaller VMs and place them wholly in a host NUMA node
- Optimal host will also recognize larger VMs and partition vCPU threads and VM memory, such that vCPUs and memory belonging to vNode X will be placed together in host Node Y.
  - This can be difficult, as there is no explicit way to indicate to the host what Qemu [vCPU] threads and what Qemu memory belong “together”.
  - Host must figure out which threads access which memory and locate vCPUs/memory accordingly

NUMA and CPU-memory Locality

- AutoNUMA & SchedNUMA
  - Two different solutions to this problem, both have some similar concepts, but not exactly the same solution.
  - Some basic testing for both:
    - Host with 4 NUMA nodes, 40-cores / 80-threads
    - Three different configurations tested: 16 x 5-vcpu, 8 x 10-vcpu, and 4 x 20-vcpu VMs
      - vCPUs = host CPU threads, no CPU over-commit
      - Dbench run in tmpfs (no I/O)

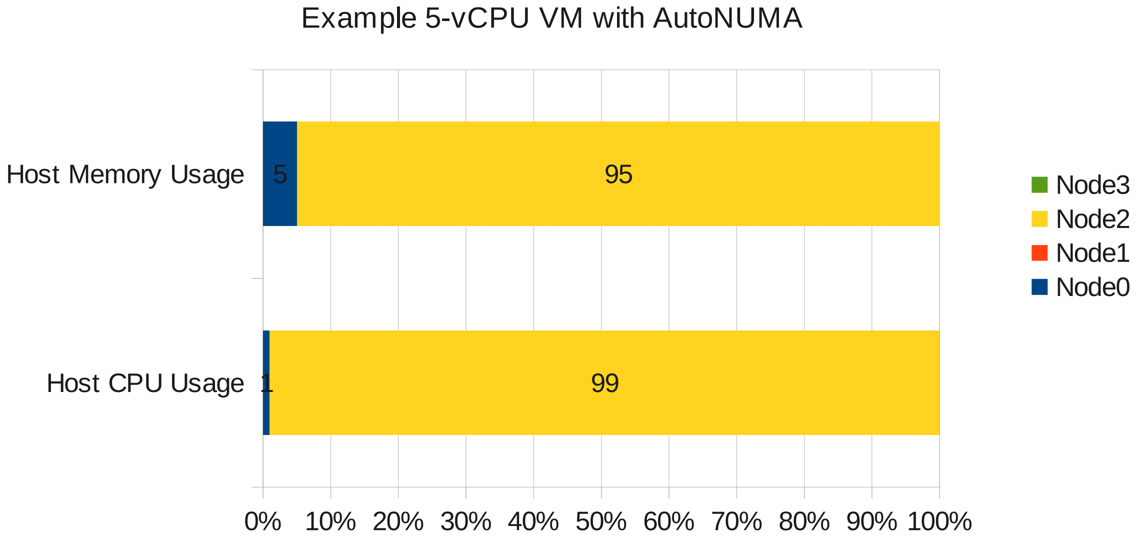


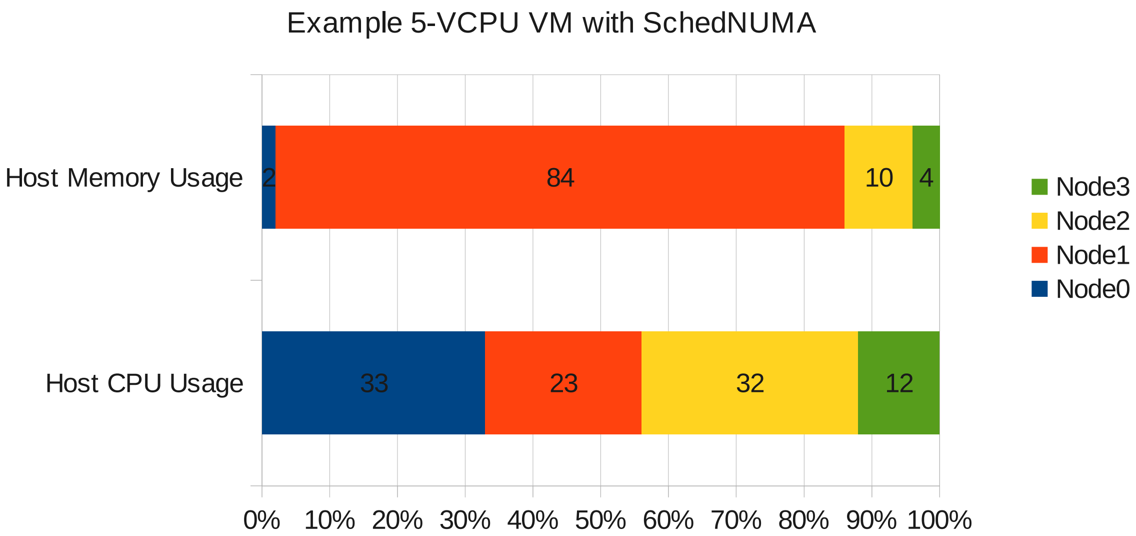
NUMA and CPU-memory Locality

- AutoNUMA analysis:
  - Very good at grouping vCPU threads and memory on 5, 10, and 20-vCPU VMs
  - Some host overhead in handling page faults (host perf callgraph)
    - `82% raw_spin_lock()`
      - `tdp_page_fault()`
      - `kvm_mmu_page_fault()`
      - `handle_ept_violation()`
  - Can be mitigated somewhat by lower page scanning rate
  - Would benefit from native THP migration

- SchedNUMA analysis
  - Typically majority of VM memory ends up in single host NUMA node, but vCPU is spread out

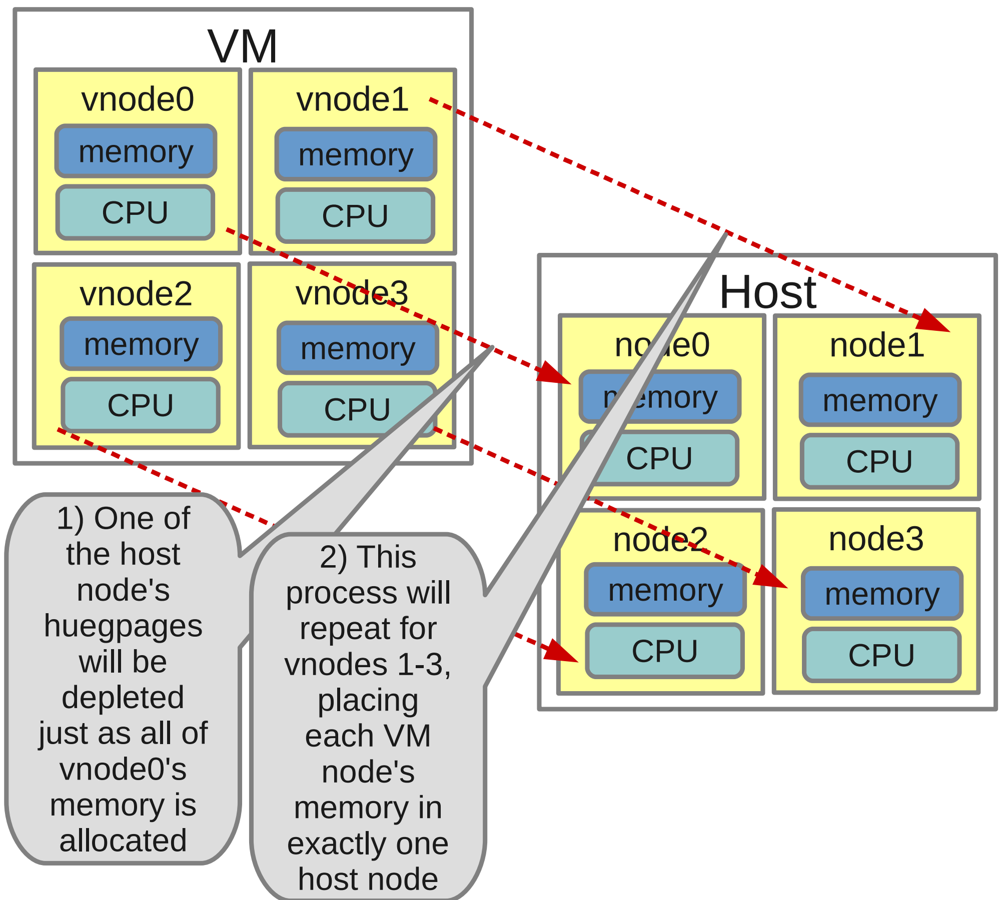
VM Resource usage





NUMA and CPU-memory Locality

- VCPU & memory placement for really big VMs
  - Doing this placement manually is tricky today
    - You can specify where vCPUs get to run
    - But you cannot specify multiple locations for VM memory
      - You can specify a -single- location for memory, but we need more than one location for a large, multi-node VM.
      - There is a trick to get around this:



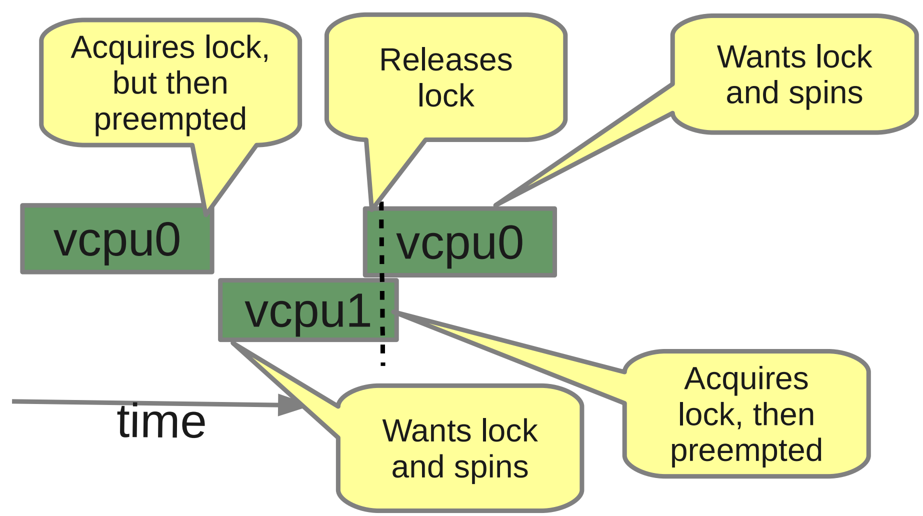
- For the example to the left: reserve the number of huge-pages equal to VM memory (and no more) – but make sure the reservation is spread equally among 4 host nodes
- In the VM XML configuration, configure for huge-pages
- When VM is started either:
  - Monitor /sys/devices/system/node/node[0-3]/meminfo to determine which node-order hugepages were allocated
  - Use pagemapscan [3] to tell you where the VM memory is located on the host
- Once you know where the VM's vnode memory is, you can then pin vCPUs to match

NUMA and CPU-memory Locality

- VCPU & memory placement for really big VMs
  - Does placing vCPU and VM memory properly help?
    - 80-vCPU VM Kernel compile times reduced another 5%
    - 80-vCPU VM SPECjbb2005: 45% performance improvement
  - Ideally, we should never have to do this manually
    - AutoNUMA, schedNUMA, etc, should do this for us....

Locking

- The expected behavior of spin_lock() can change when the virtual CPU has different characteristics of a physical CPU.
- When vCPUs do not have simultaneous execution, spin time can be significantly increased. This well known problem, lock holder preemption, has been addressed with different solutions to date
  - Para-virtual
    - Accurate, efficient, but requires OS changes (and not just one's favorite OS)
  - HW detection of spin
    - Good HW support (PLE, PF)
    - More challenging for larger VMs
    - Possibly false positives



Dealing with Lock-holder Preemption

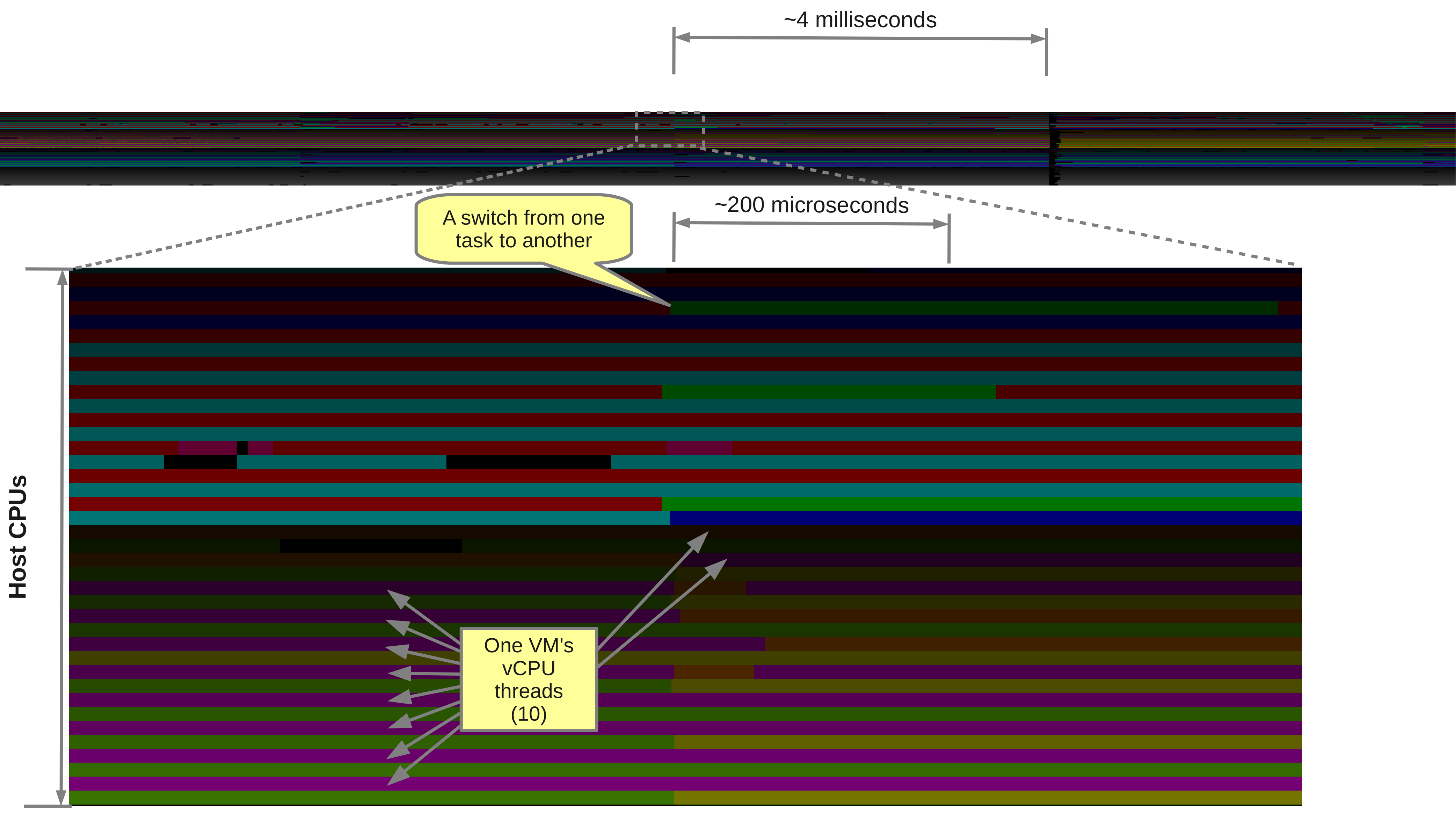
- We concentrate on HW based solution to PLE
  - If at all possible, it is desirable to not implement changes in the guest OS
- Current HW approach:
  - vCPUs which spin are detected by HW and cause vm_exit
  - While in host, vCPU yields time to another runnable-but-not-running sibling vCPU
    - who knows, maybe *that* vCPU is the one holding a lock (or maybe not!)
    - This is different from just a “yield()” in that we are specifying who we want to give to
  - This process will hopefully get the preempted lock-holding vCPUs running again
- Today the HW approach works extremely well for VMs ~10 vCPU or less
- However, for larger VMs, the current approach does not work as well

Dealing with Lock-holder Preemption

- Issues with current HW approach:
  - The more vCPUs in a VM, the more candidate vCPUs to yield to
  - PLE handler with yield_to() is not a cheap operation
    - vCPU-to-task lookup
    - Double run-queue lock
    - *This can use over 50% of all CPU!*
  - The more vCPUs in a VM, the more likely there will be lock contention
    - More vCPUs spinning
    - More exits & double run-queue locks
  - Possible that HW may detect spin too aggressively
    - VM might exit when there is no lock holder preemption
    - And still pay the expense of a yield_to()

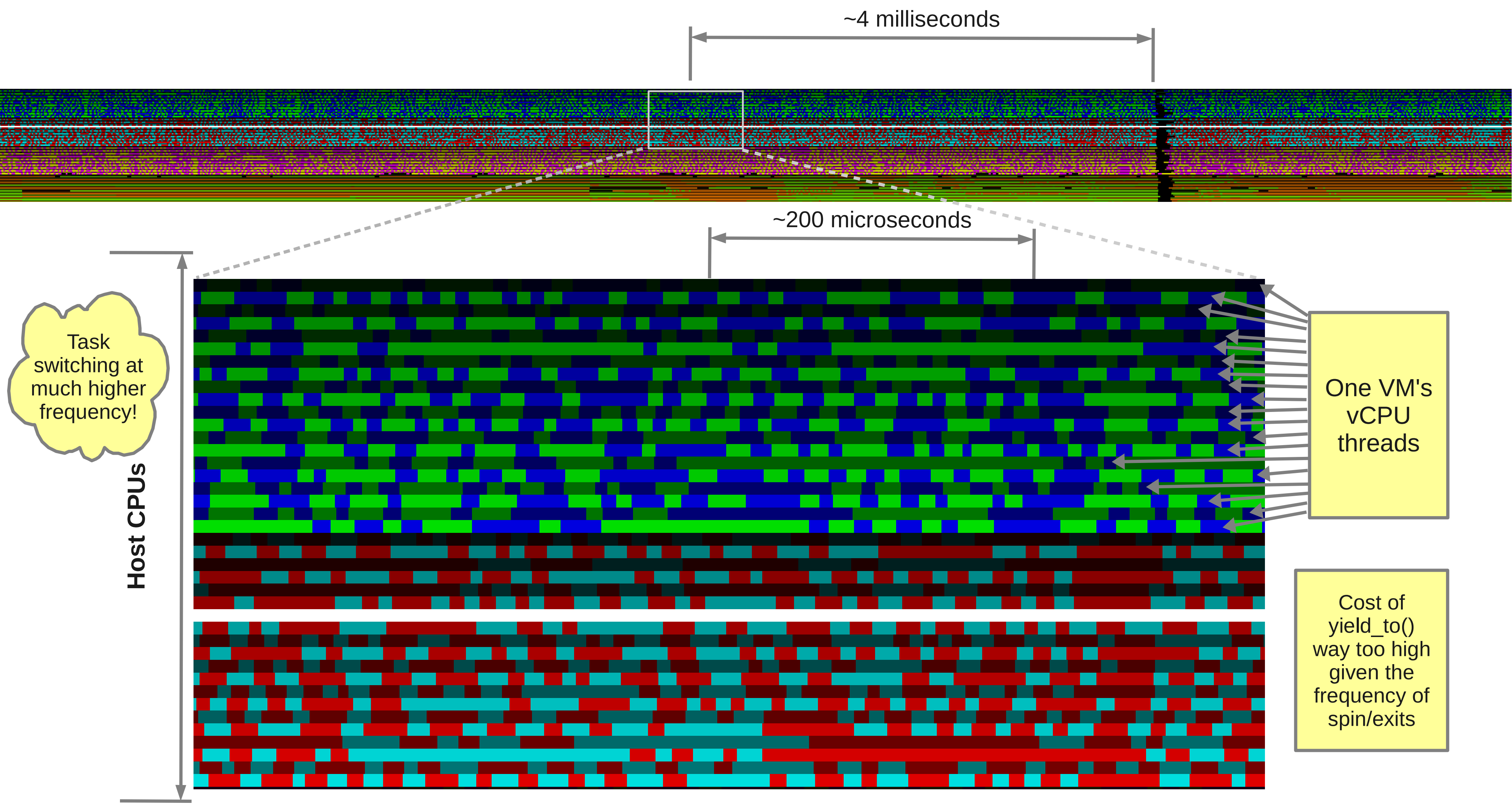
Example of PLE/yield_to() Working Well

below is a bitmap of 'perf sched map' with PLE enabled
VMs have 10 vCPUs and 2x CPU over-commit
Each VM a unique color (with different brightness per vCPU)



Example of PLE/yield_to() not Working Well

below is a bitmap of 'perf sched map' with PLE enabled
VMs now have 20 vCPUs 2x CPU over-commit
Each VM a unique color (with different brightness per vCPU)



Dealing with Lock-holder Preemption

- Observations from 10 to 20 vCPU VMs:
  - The detection of spin/exits in VM goes up massively
  - Once this happens, lock contention in host goes up massively from double-runqueue lock
  - As vCPU count increases, the number of candidate vCPUs to yield_to() also goes up
    - For example: 20-vcpu VM @2.0x CPU over-commit may have, on average, 10 vcpus which are preempted
    - How does one decide which vCPU to yield to?
    - Of the 10, there may be only 1 holding a lock
    - Other 9 may not need to run immediately
    - Result is a lot of unwanted yield_to() -and a lot of overhead to do so

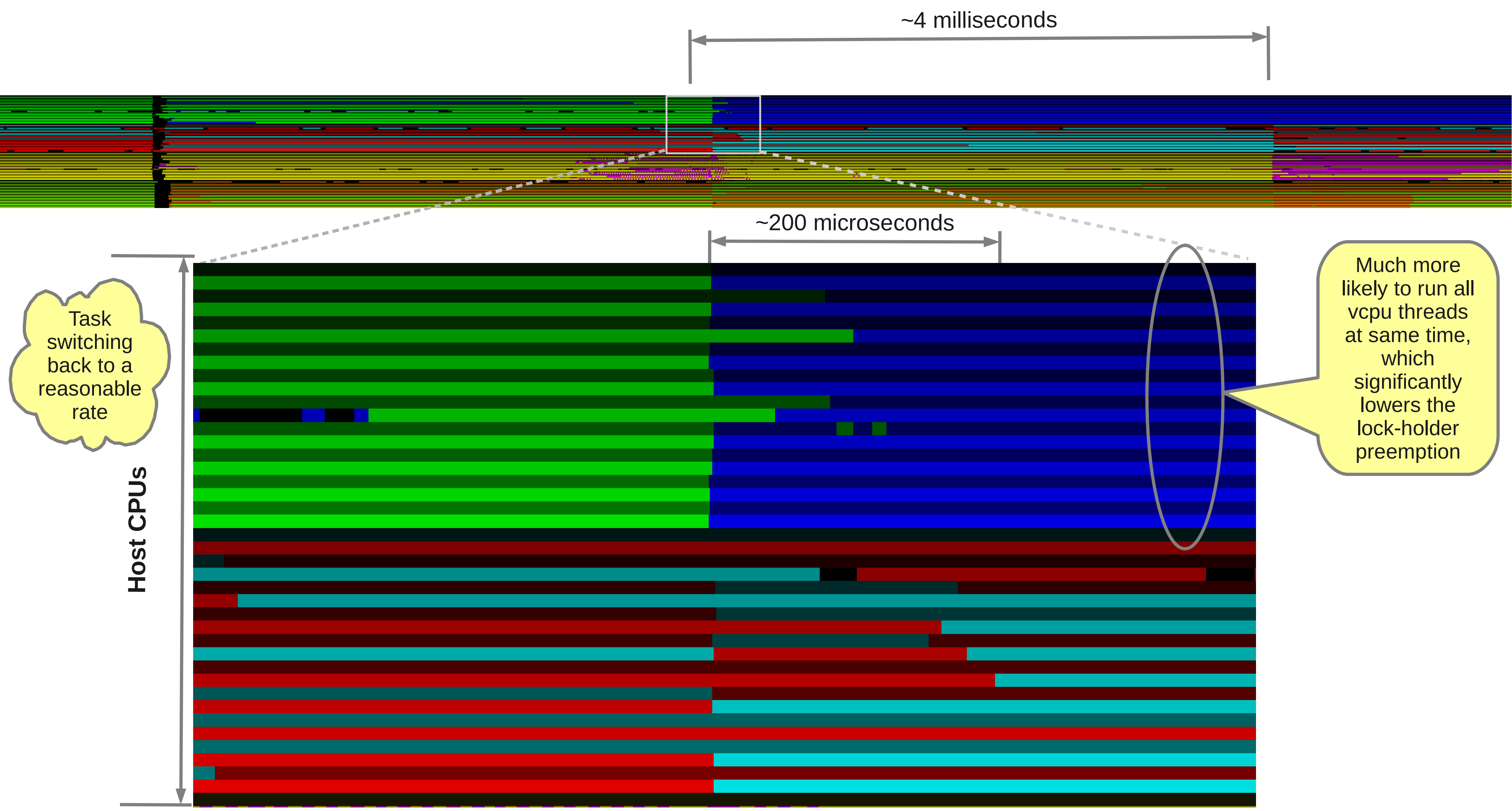
Dealing with Lock-holder Preemption

- Some potential fixes [to improve yield_to()]
  - Lower cost to determine candidate vcpus
    - Check if target vcpu to yield_to() is running before double runqueue lock
  - Better predict the candidate vcpu to yield_to()
    - Heuristics on vcpu spin activity

  Both of these help, but do not approach maximum performance potential
- Alternative fix – if yield_to() is not helping, then encourage vCPUs from same VM to run together
  - Spinning vcpus have two possible reactions
    - yield_to(), *but change this to only one per jiffie*
      - Any more often is not considered productive
      - This works for smaller VMs and is essentially same behavior as current code
    - Other exits must simply yield()
      - With the assumption that when they do get to run again, the lock-holding vcpu will also be running
      - It is important that spinning vcpus yield to *other VMs* and not their sibling vcpus! Must encourage ***all*** same-VM vcpus to run/not-run at the same time

Example of Throttled yield_to()

below is a bitmap of 'perf sched map'
VMs with 20 vCPUs and 2x CPU over-commit
Each VM a unique color (with different brightness per vCPU)



Dealing with Lock-holder Preemption

- Results
  - 8 x 20-vcpu VMs, running dbench workload in tmpfs (no disk IO):
    - 3.6 with PLE off: `394 MB/sec`
    - 3.6 with PLE on: `8175 MB/sec`
    - 3.6 with PLE off & gang-scheduling: `32001 MB/sec`
    - 3.6 with PLE on & throttled yield_to() `30614 MB/sec`
- Throttled yield_to() works best when yielding to tasks which are not vcpus from same VM
  - If this is not met, throttled yield_to() will still offer better performance, but not the significant jump we are looking for
  - There is no policy currently in the scheduler to enforce non-shared runqueue
    - These tests used restricted vcpu placement such that no vcpus from same VM were on same runqueue
    - One could possibly create a scheduler policy to *always* ensure same-VM vcpu threads do not share a runqueue
    - Or... maybe one could use PLE to correct this situation *only when it's necessary*
      - On detection of high frequency of yields from same vcpu, check for sibling vcpus on same runqueue, and swap tasks from neighbor runqueue to remedy

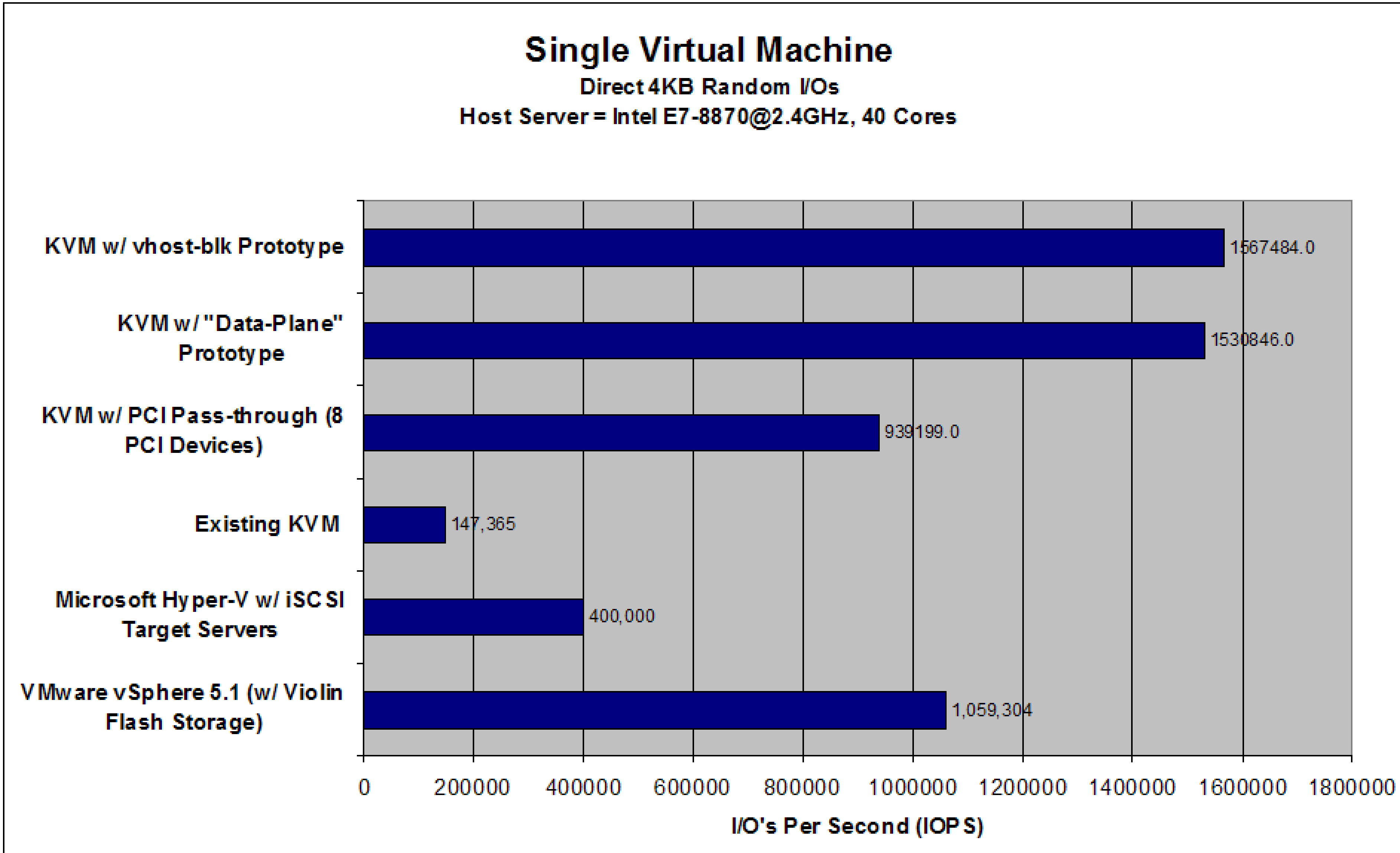
Dealing with Lock-holder Preemption

- One other problem: detecting spin & false positives
  - HW may detect a spin but there is actually no lock-holder preemption
  - Why? Some locks simply have a longer spin because the lock is held longer
  - With PLE, we can adjust the sensitivity (ple_window), but what's the right setting?
  - Even when there really is no CPU over-commit, the exit handler is still very high overhead
    - Simply discovering that there are no candidate vcpus to yield to is expensive
    - We might be able to quit early if the host was certain there was no over-commit
      - But that can be tricky, as each vcpu could be subjected to different levels of over-commit. The detection itself could get too costly
  - However, implementing the throttled yield_to() reduces exit handler cost significantly
    - Example: Time to boot 80-vcpu VM: (no CPU over-commit here!)
      - 3.6 with PLE on: `369 seconds`
      - 3.6 with PLE off: `25 seconds`
      - 3.6 PLE & throttled yield(): `28 seconds`
  - *Using a throttled yield_to() may eliminate the need to tune ple_window*

I/O Scalability

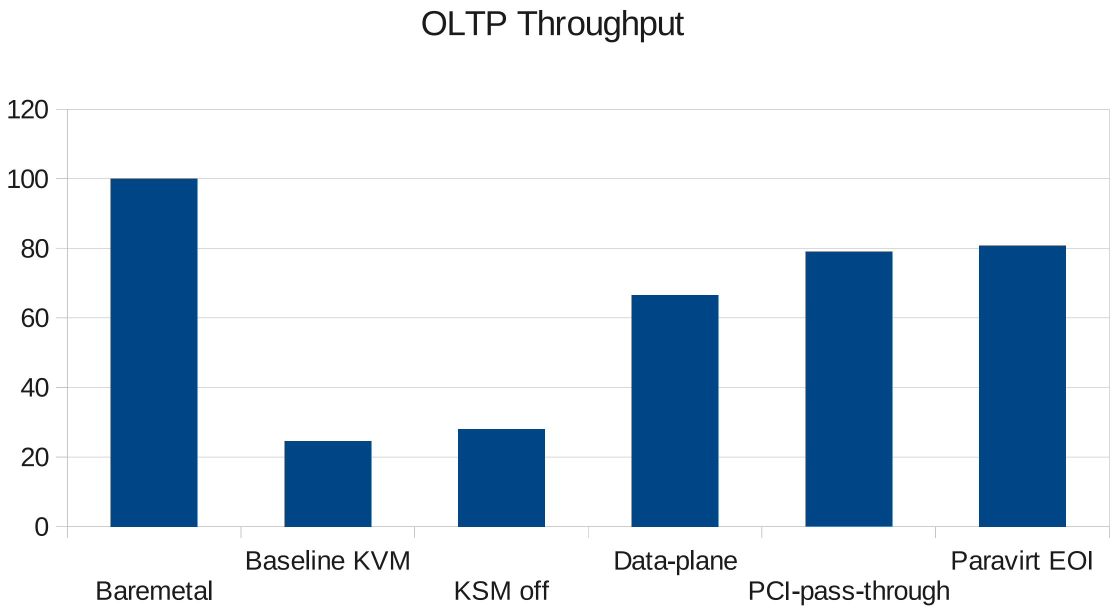
- Disk I/O via PCI-pass-through is quite good
  - Demonstrated 1.6 millions IOPs this year (4k random read/write)
  - However, VM scalability it is actually limited by maximum PCI device limit today
    - Currently limited to 8 devices
      - We demonstrated 860,000 IOPs per VM
      - More can be done with higher performing PCI devices
- We believe improving virtio is much more relevant to users
  - Current virtio-blk is currently limited to about 150,000 IOPs
    - This is due to the Big Qemu Lock
  - There are multiple alternatives
    - Data-plane
      - In-Qemu, could potentially support Qemu disk formats (qcow, etc)
    - Vhost-blk
      - In-kernel
    - Vhost-scsi
      - *Coupled with tcm_vhost driver introduces some really interesting options*
  - *We expect all of these solutions to scale well. We are now focusing on effciency*

I/O Scalability



OLTP Workload

- *OLTP = Online Transaction Processing*
  - *Big Database (will not fit in memory)*
  - *Lots of IO (hundreds of thousands of IOPS)*
- *Our test-bed:*
  - *8 Intel Westmere-EP cores (16 threads)*
  - *144 GB memory*
  - *42 SSD*
- *Software:*
  - *RHEL VM & IBM DB2*



OLTP Analysis

- *Performance & Scalability challenges:*
  - *PLE*
    - *No over-commit, but was disabled due to some exits and kvm_vcpu_on_spin() overhead*
  - *KSM*
    - *KSM is engaged at a certain memory threshold*
    - *We allocated all but 3G of host memory for this VM which triggered KSM*
    - *KSM can break down a significant number of transparent huge-pages, degrading performance up to 10%*
  - *Virtio-blk*
    - *Big Qemu lock causes disk I/O bottleneck which limits OLTP transaction rate to less than ½*
    - *Data-plane showed significant improvement, but still below PCI-pass-through*
    - *Vhost-blk and vhost-scsi will be tested eventually*
  - *PCI-pass-through*
    - *KVM overhead from high interrupt rate -efficient I/O dependent on coalescing interrupts to lower rate*
    - *Lowering rate may be in direct conflict of a low latency database transaction log device*
  - *EOI*
    - *Pv-EOI reduced total vm_exits by 20%*

OLTP Analysis (continued)

- *Performance & Scalability challenges:*
  - *NUMA memory & vCPU placement*
    - *CPU-memory locality critical for this workload*
    - *Manual placement was used – plan to test autoNUMA/schedNUMA*
  - *KVMclock*
    - *We have observed that gettimeofday() can be as much at 10x slower with kvmclock vs tsc*
    - *Our tests did not include user-space gettimeofday() for kvmclock (we have been just using tsc) – but we will be testing this soon*
    - *There are quite a number of users of the clock, like delay accounting, cgroups, the database application, etc.*
  - *Timer Interrupts*
    - *Dropping timer interrupts to 100Hz in host and guest can improve performance 3%*
    - *We'd expect maybe 1% improvement from bare-metal*
  - *In-guest IPI*
    - *These are much more expensive than host IPI*
    - *Guest IPI → vm_exit → host IPI → virtual IRQ injection*
    - *For a database with high transaction rates, this is major influence on performance*
      - *Lots of signalling between threads to indicate a transaction is “logged”*
      - *7.13% CPU in reschedule_interrupt() for KVM test*
      - *0.09% CPU in reschedule_interrupt() for bare-metal test*

Questions?

Thanks!

Thank you to my team members for their help: Barry Ardnt, Karl Rister, Khoa Huynh, Mark Peloquin, Steve Dobbelstein, Steve Pratt, and Tom Lendacky

A special thank you to all to the KVM & Qemu developers for their help and to Red Hat for their support

[1] SAP SD 2-tier results, http://www.sap.com/solutions/benchmark/sd2tier.epx
[2] "Achieving Unprecedented Virtualization Results for Maximum IOPS per Host", ftp://public.dhe.ibm.com/linux/pdfs/2012RHEL_KVM_Hypervisor_Performance_Brief_19_v4.pdf
[3] pagemapscan.c, https://docs.google.com/open?id=0B6tfUNlZ-14wTEYzM1FjVUo4QW8