Google

Vcpu priority boosting for latency sensitive workloads

Vineeth Pillai (Google)
Joel Fernandes (Google)

Google

Agenda

- The problem
- The root cause
- The solution
- Some performance numbers
- Future work

The Problem

- Guest virtual machine experiences considerable latencies when the host is under load (sometimes 100s of ms of latency).
  - Overcommitted physical cpus(multiple VMs, host processes, …)
- Android apps runs in a virtual machine in chromeOS and on low-end devices under considerable load, we observe
  - Audio glitches
  - Stuttering and unsmooth Video playback

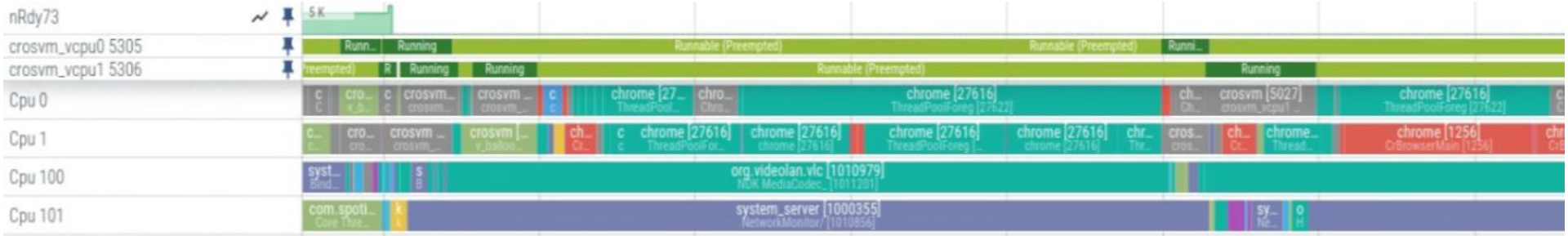
The root cause

- Double scheduling: Host schedules vcpu threads and the guest schedules the tasks running inside the guest
- But both schedulers are unaware of the other
  - Hosts schedules vcpu threads without knowing what's being run on the vcpu
  - Guest schedules tasks without knowing where the vcpu is running physically
- vCPUs are regular CFS tasks in the host and does not get to run in a timely fashion when the host is experiencing load
- Host scheduler tries to be fair and doesn’t know about the priority requirements
- This can cause issues with latencies, power consumption, resource utilization etc.

An example

Traces collected (guest and host) when video lag was happening

(android's VLC player)

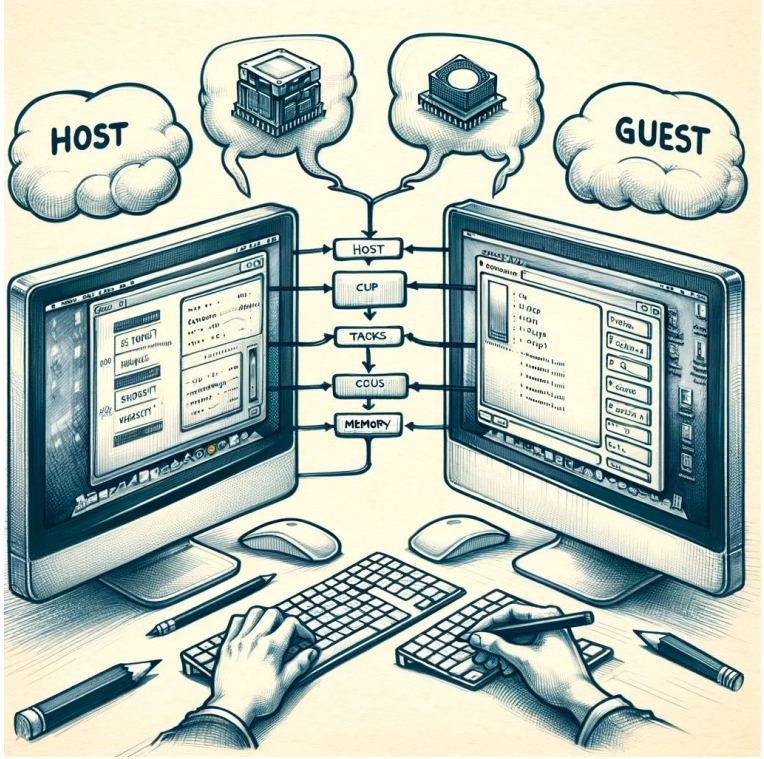


A (possible) Solution for minimizing latencies

- CFS is not strictly priority aware, so let's do RT!
- Boosting priority of vcpus to RT helps
- But vcpus running as RT through the life time of a VM has negative impact to the whole system
- The issue is manifested - vcpus may hijack the host!
  - We have the same problem now: vcpus running a normal workload may preempt a latency sensitive workload in the host.

Another (possible) solution

- Let the host and guest talk ;-)
- Share scheduling information between host and guest
- A cooperative scheduling framework by sharing the information between host and guest so that both could make educated scheduling decision



Dynamic vcpu priority management

- One specific use case for cooperative guest/host scheduling
- Aimed at reducing latencies for latency sensitive tasks in the guest.
- Guest shares the priority requirements and host can boost/unboost as needed

Dynamic vcpu priority management

- Host proactively boosts the vcpu thread when it knows that the guest will be running a latency sensitive work load, example: interrupt injection
- Guest shares its need for a priority boost when running latency sensitive workloads
  - Host boosts the vcpu thread and shares the boost status with guest
- Guest can request an unboost when it no longer runs latency sensitive workloads
- Host implements throttling and forceful unboosting for buggy/rogue guest kernels

Latency sensitive guest contexts

- NMIs, interrupts, softirqs
- Tasks with priority higher than SCHED_OTHER
- Preemption disabled in guest

Priority Inversion - Need to boost spinlocks too!

- May happen if not all Vcpus are boosted
- CFS VCPU preempted after taking a spinlock
- RT VCPU blocks on the spin lock acquired by the above CFS VCPU
- Even worse if the the RT VCPU preempts the CFS VCPU holding the lock on the same physical CPU.

Implementation (Proof of Concept)

- x86 (Intel and AMD) only but easily portable to other architectures
- Guest uses MSR to communicate the GPA of shared memory location
  - This also acts as guest intent to use the functionality
- Boosts the vcpu to RT
  - SCHED_RR and priority 8 by default and is tunable
- Unboost to SCHED_OTHER with nice 0
  - TODO: Could be made tunable. POC it is fixed.

https://github.com/vineethrp/linux/commits/kvm-hypercall-6.5.y

Implementation (Proof of Concept)

Enabling the feature:

- Host uses CPUID to advertise the feature to guest
  - VMM can enable/disable the advertisement (PoC always advertises)
- Knobs to enable/disable this feature globally and per-vm.

https://github.com/vineethrp/linux/commits/kvm-hypercall-6.5.y

Implementation Details: Synchronous boost/unboost

- Uses hypercall mechanism.
- Guest shall request a boost via hypercall if needed:
  - Currently not used. Only lazy (async) boosting is used in the POC.
- Guest requests an unboost via hypercall once it completes the latency sensitive workload:
  - Switching to SCHED_OTHER (normal tasks) in scheduler
  - Return to user mode SCHED_OTHER after servicing interrupt, softirq, preempt enable etc.
  - IRQ disabling/enabling doesn't do boosting currently.

What's hypercall overhead like?

- Overhead of Hypercall seen to be 10-20 micro seconds, or so.
  - Caveat: Unless VM is nested virt.
- Seems very reasonable, as the latencies this fixes are 3 orders of magnitude higher (10s or 100s of ms) !

Implementation Details - Asynchronous boosting

- Guest shares its intent (boost, preemption state) via shared memory and continues execution as long as it can.
- On the very next VMEXIT, host takes action.
- Avoids an extra VMEXIT caused by hypercall.

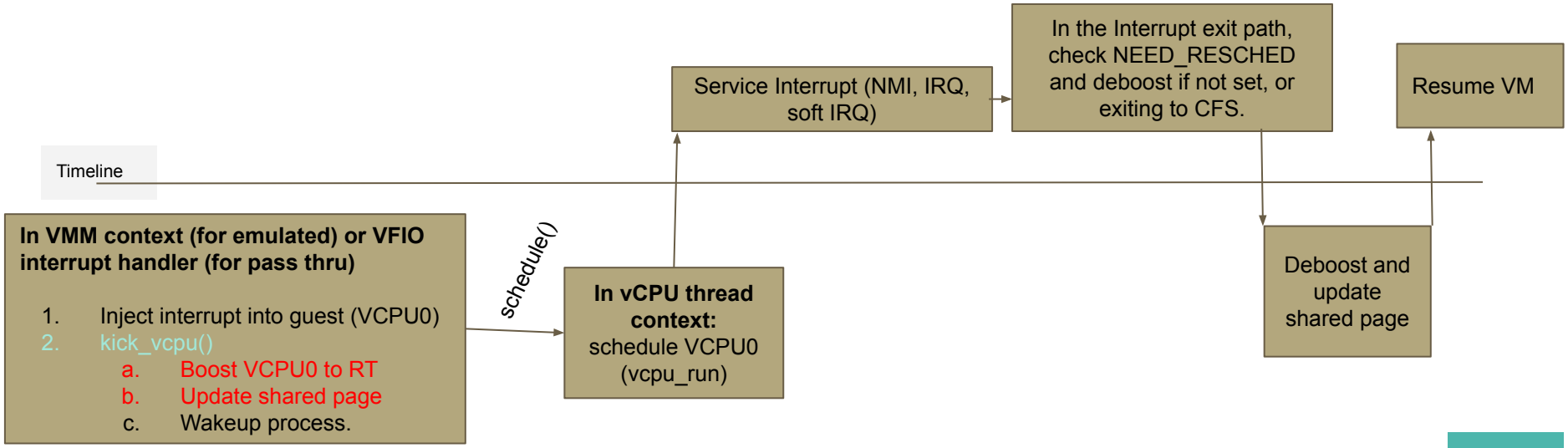
Implementation Details: Proactive boosting

- Host boosts the vcpu threads on situations where it has clear information that guest is running a latency sensitive workload
  - Interrupt Injection
    - Before injecting an interrupt into the guest, host boosts the priority of the vcpu
  - Guest VCPU halt/blocking
    - When the guest vcpu halts, the wakeup would be due to an event which needs to be handled immediately(interrupts, IPIs etc).
    - Priority is boosted as the guest vcpu task is switched out.

Care to be taken during interrupt handling

Scenario: Emulated device interrupt and VFIO pass thru devices. Host needs to inject an interrupt in guest on VCPU0

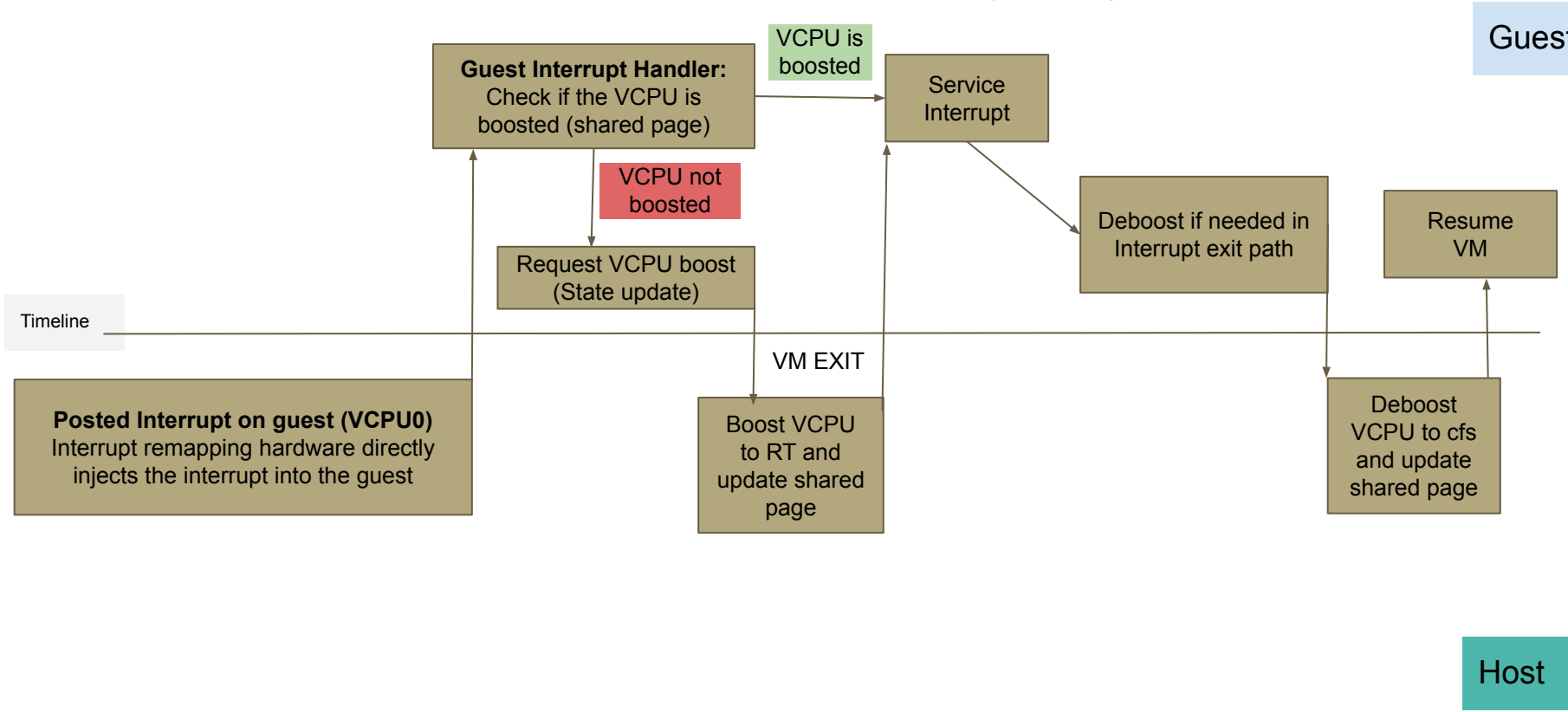
Guest



Host

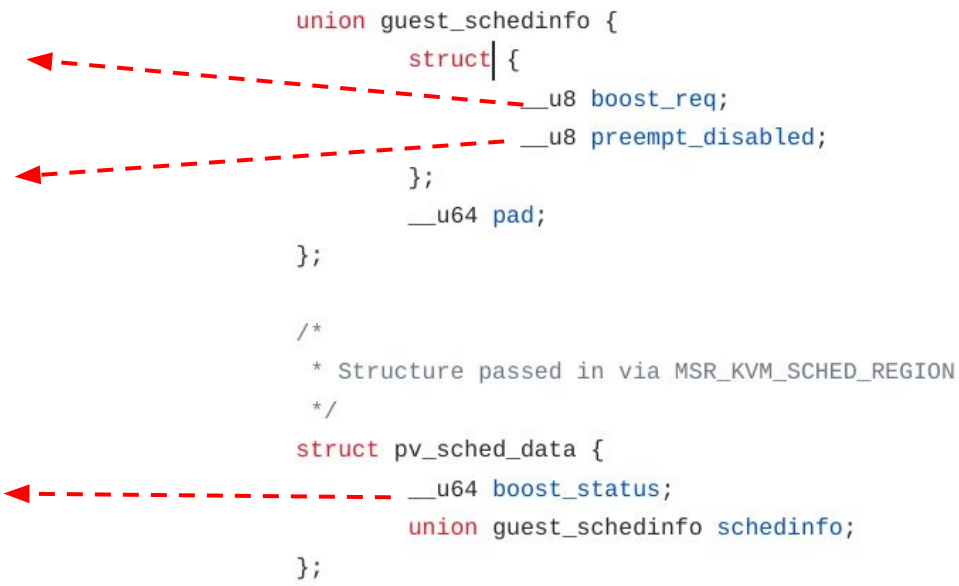
Care to be taken during POSTED interrupt handling

Scenario: Guest is ON CPU. Posted interrupt is received directly in the guest on VCPU0



Implementation Details: Shared memory communication

- Guest lets the host know about its need for boosting/un boosting
- Guest lets the host know if preemption is disabled or not
- Host sees the above on the very next VMEXIT and takes action accordingly.
- Host updates the shared memory with the boost status. Guest uses this to avoid unnecessary memory updates and hypercalls



```
union guest_schedinfo {
        struct {
                __u8 boost_req;
                __u8 preempt_disabled;
        };
        __u64 pad;
};

/*
 * Structure passed in via MSR_KVM_SCHED_REGION
 */
struct pv_sched_data {
        __u64 boost_status;
        union guest_schedinfo schedinfo;
};
```

Performance

- Synthetic Test (micro benchmarks)
  - Cyclictest (guest and host) when host is idle and busy
  - Busy host simulated by stress-ng
  - Intervals 500 us and 1000 us.
- Simulating real world workloads
  - Oboetester glitch test.

Synthetic tests - Idle Host (cyclictest - SCHED_RR)

VM: Average latency in micro seconds

Interval(us)	vanilla	vcpu_boost	static_rt
500	9	9	10
1000	34	35	35

Host: Average latency in micro seconds

Interval(us)	vanilla	vcpu_boost	static_rt
500	5	3	3
1000	3	3	3

VM: Max latency in micro seconds

Interval(us)	vanilla	vcpu_boost	static_rt
500	1577	1433	140
1000	6649	765	204

Host: Max latency in micro seconds

Interval(us)	vanilla	vcpu_boost	static_rt
500	1577	1526	15969
1000	697	174	2444

Legend:
Vanilla: Host kernel: 6.1 guest kernel: 5.10
Vcpu_boost: Host and guest kernels with the patches
Static_rt: vanilla 6.1, with vcpu threads boosted to RT

Synthetic tests - Busy Host (cyclictest - SCHED_RR)

VM: Average latency in micro seconds

Interval(us)	vanilla	vcpu_boost	static_rt
500	887	21	25
1000	6335	45	38

Host: Average latency in micro seconds

Interval(us)	vanilla	vcpu_boost	static_rt
500	6	6	7
1000	11	11	14

VM: Max latency in micro seconds

Interval(us)	vanilla	vcpu_boost	static_rt
500	216835	13978	1728
1000	199575	70651	1537

Host: Max latency in micro seconds

Interval(us)	vanilla	vcpu_boost	static_rt
500	2075	2114	2447
1000	1886	1285	27104

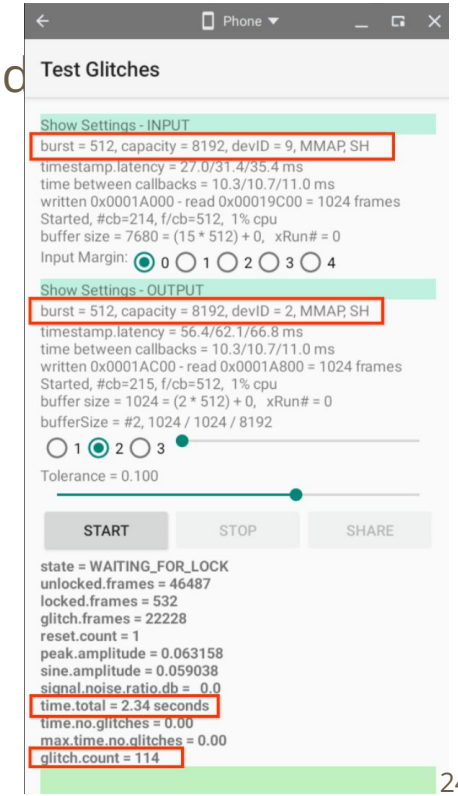
Legend:
Vanilla: Host kernel: 6.1 guest kernel: 5.10
Vcpu_boost: Host and guest kernels with the patches
Static_rt: vanilla 6.1, with vcpu threads boosted to RT

Simulating real world workloads: Audio glitches

Oboetester app in android vm on chromeos used to test aud

Host kernel : 6.1

Guest Kernel: 5.10

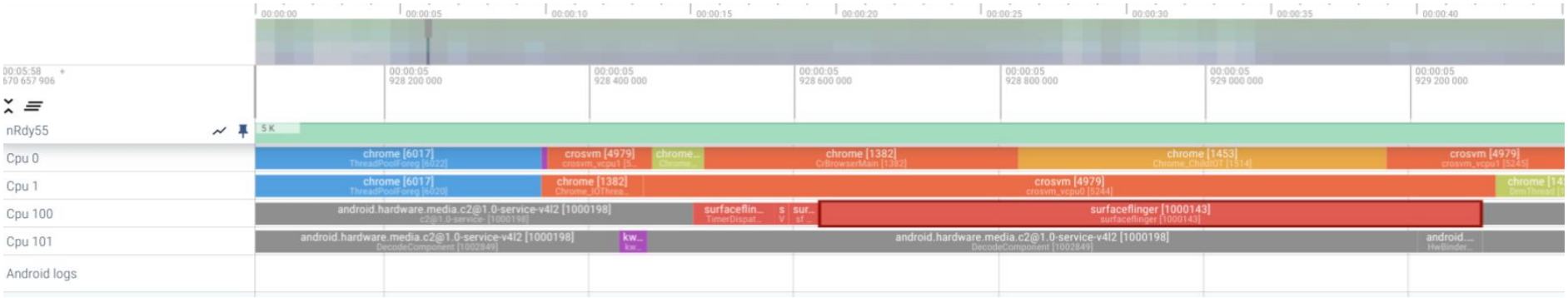


Buffer size	Noload		Speedometer	
	Vanilla 6.1	vcpu_boost	Vanilla 6.1	vcpu_boost
96 (2ms)	20	4	1365	67
256 (5ms)	3	1	524	23
512 (10ms)	0	0	25	24

Video lag example with the fixes applied

Traces collected (guest and host) during video playback

(android's VLC player)



Other use cases for cooperative scheduling

- Host can share the physical cpu load and pressure to guest and also the vcpu physical placement.
  - Guest can effectively decide the most effective vcpus for the tasks it needs to run
  - Guest can also request an effective vcpu placement if it has the above information. Host can accept or deny the request based on its requirements.
- Guest can share the vcpu load with the host and may include task level information(priority, load etc)
  - Host can make an educated decision on where to place the guest vcpus appropriately
  - Host can also suggest task placement hints to the guest.

Google

Future Work

- Generic implementation which would make it easy for porting to other architectures
- Use paravirt ops
- Optimizations in the critical path for speed and cache efficiency.
- Standardize the memory sharing framework so as to extend it for other use cases.
- Priority management for irq disable/enable code paths.
- Optimizing proactive boosting during interrupt injection.

thank you