# Regular vs real-time Linux latency in virtualized environment

Abhishek Chauhan (zxcve@vt.edu)

Kanika Sabharwal (kanikas3@vt.edu)

Sujay Yadawadkar (sujayr91@vt.edu)

**Virginia Tech**

## 1. Introduction and Motivation:

Real time systems are intended to serve real time applications within a finite fixed time frame. However due to several contributing factors there will be delay in responding to external stimulus which are costly for real time applications. Identifying the bottlenecks and fixing the loop holes to set a deterministic bound to the delay is one of the challenging and most important part of real time system development. In this project, we aim at studying the **latency** of linux systems in both real time as well as non-real time setting. We utilize standard benchmarking tools that come with linux in order to measure the latency as well as finely profile the kernel code to measure the performance bottlenecks. We also run these performance measurements in a virtualized setting in order to understand the hindrance put up by virtualization. The primary linux versions which we work with in this project are,

- **CentOS** : Regular Linux Kernel.
- **RedHawk**: Real-Time enabled Linux Kernel.

## 2. Approach:

### 2.1 Latency:

Latency in a general context is the time between a request and actual time when the request is serviced. Putting it in context of a real time linux kernel, it is time when the interrupt occurs and the start of the user level code corresponding to this interrupt. Although latency exist in almost every system, in the context of real time the goal is to have a deterministic latency. Several factors lead to increased latency, few of them are,

1. Several other interrupts waiting to be serviced.
2. Virtualization may cause physical core to be pre-empted

In this project we measure the latency between occurrence of an interrupt and the execution of process/thread corresponding to this interrupt. We use Cyclictest, which comes built in with CentOS and Redhawk to measure the latency.

## 2.2 Cyclictest:

Cyclic test is a preinstalled tool in RedHawk and CentOS systems. It is used to determine the latency of a realtime systems and provides several configurations including multithreaded options to measure the latency.

The principle on which cyclic test works is as follows. Cyclic test sets a timer to expire in future and goes to sleep waiting for a signal from timer expiry. As soon as it receives the signal it measures the time difference between current time and the actual timer expiry time. The difference is considered to be latency. Below is the time difference captured by cyclic test.

T1 : Settimerexpire(T)

T2: Put Cyclic test process to sleep and wait for signal.

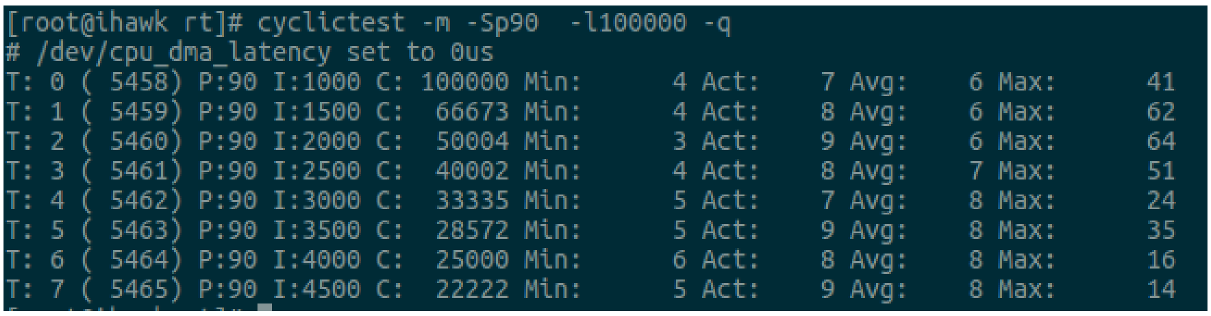
T3= T: Timer expires.

T4: Send signal

T5: Cyclic test wakes up and measures, Tdiff= T5- T, and treats this to be the latency.

The above latency is calculated for several iterations and cyclic test gives out an average latency.

Figure 1 below shows a simple cyclic test run.

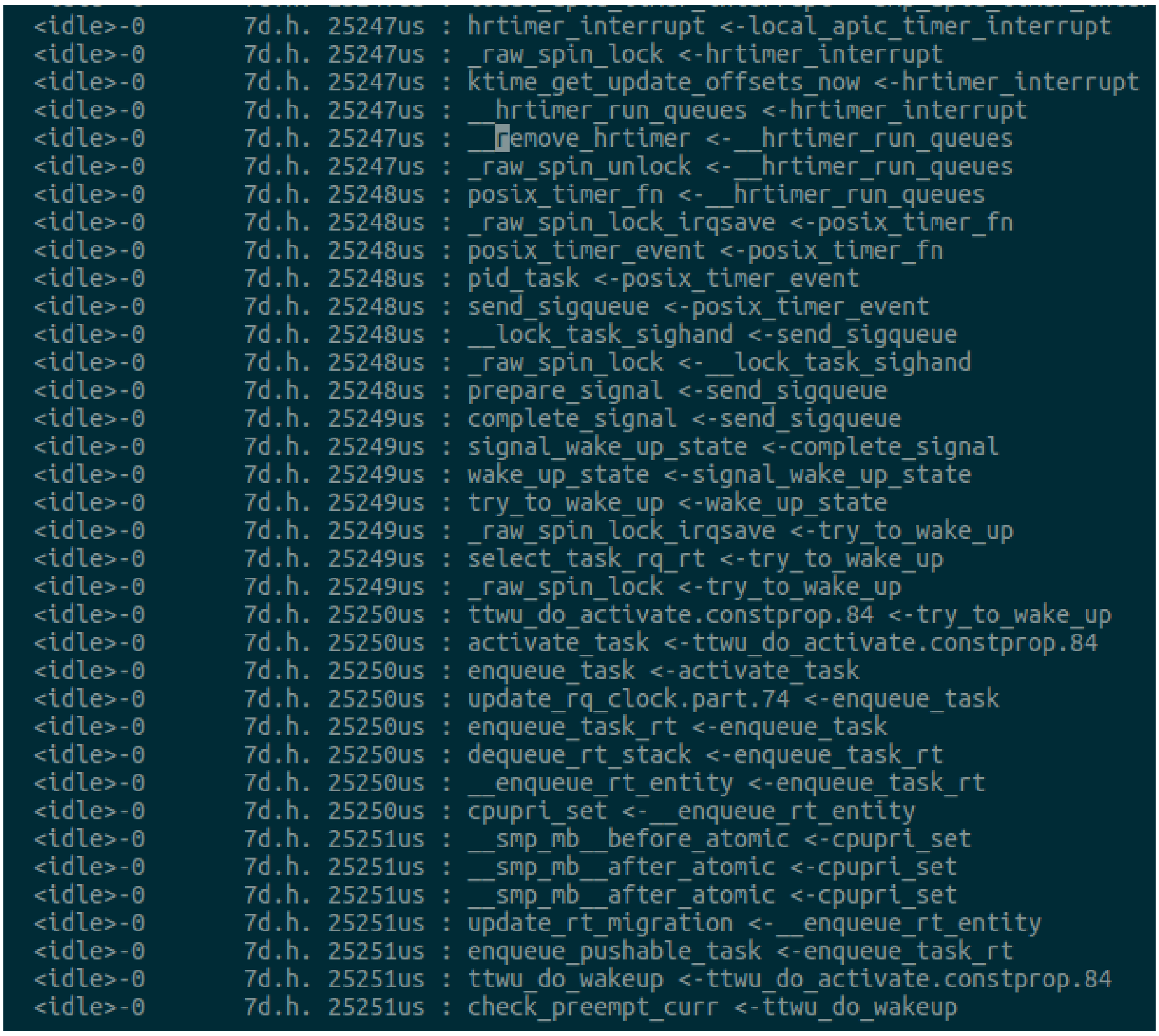


```
[root@ihawk rt]# cyclictest -m -Sp90  -l100000 -q
# /dev/cpu_dma_latency set to 0us
T: 0 ( 5458) P:90 I:1000 C: 100000 Min:      4 Act:    7 Avg:    6 Max:      41
T: 1 ( 5459) P:90 I:1500 C:  66673 Min:      4 Act:    8 Avg:    6 Max:      62
T: 2 ( 5460) P:90 I:2000 C:  50004 Min:      3 Act:    9 Avg:    6 Max:      64
T: 3 ( 5461) P:90 I:2500 C:  40002 Min:      4 Act:    8 Avg:    7 Max:      51
T: 4 ( 5462) P:90 I:3000 C:  33335 Min:      5 Act:    7 Avg:    8 Max:      24
T: 5 ( 5463) P:90 I:3500 C:  28572 Min:      5 Act:    9 Avg:    8 Max:      35
T: 6 ( 5464) P:90 I:4000 C:  25000 Min:      6 Act:    8 Avg:    8 Max:      16
T: 7 ( 5465) P:90 I:4500 C:  22222 Min:      5 Act:    9 Avg:    8 Max:      14
```

## 2.3 Tracing:

Our primary goal of this project was to understand the loop holes in interrupt service routine which leads to latency. For this we need to understand the flow of execution of from the timer interrupt till service of timer expiry and control back to cyclic test. Linux kernel has a subsystem called ftrace to get the trace of function execution. We utilize ftrace to get the trace of execution.

Figure 2 below shows a sample trace.



```
  <idle>-0       7d.h. 25247us : hrtimer_interrupt <-local_apic_timer_interrupt
  <idle>-0       7d.h. 25247us : _raw_spin_lock <-hrtimer_interrupt
  <idle>-0       7d.h. 25247us : ktime_get_update_offsets_now <-hrtimer_interrupt
  <idle>-0       7d.h. 25247us : __hrtimer_run_queues <-hrtimer_interrupt
  <idle>-0       7d.h. 25247us : __remove_hrtimer <-__hrtimer_run_queues
  <idle>-0       7d.h. 25247us : _raw_spin_unlock <-__hrtimer_run_queues
  <idle>-0       7d.h. 25248us : posix_timer_fn <-__hrtimer_run_queues
  <idle>-0       7d.h. 25248us : _raw_spin_lock_irqsave <-posix_timer_fn
  <idle>-0       7d.h. 25248us : posix_timer_event <-posix_timer_fn
  <idle>-0       7d.h. 25248us : pid_task <-posix_timer_event
  <idle>-0       7d.h. 25248us : send_sigqueue <-posix_timer_event
  <idle>-0       7d.h. 25248us : __lock_task_sighand <-send_sigqueue
  <idle>-0       7d.h. 25248us : _raw_spin_lock <-__lock_task_sighand
  <idle>-0       7d.h. 25248us : prepare_signal <-send_sigqueue
  <idle>-0       7d.h. 25249us : complete_signal <-send_sigqueue
  <idle>-0       7d.h. 25249us : signal_wake_up_state <-complete_signal
  <idle>-0       7d.h. 25249us : wake_up_state <-signal_wake_up_state
  <idle>-0       7d.h. 25249us : try_to_wake_up <-wake_up_state
  <idle>-0       7d.h. 25249us : _raw_spin_lock_irqsave <-try_to_wake_up
  <idle>-0       7d.h. 25249us : select_task_rq_rt <-try_to_wake_up
  <idle>-0       7d.h. 25249us : _raw_spin_lock <-try_to_wake_up
  <idle>-0       7d.h. 25250us : ttwu_do_activate.constprop.84 <-try_to_wake_up
  <idle>-0       7d.h. 25250us : activate_task <-ttwu_do_activate.constprop.84
  <idle>-0       7d.h. 25250us : enqueue_task <-activate_task
  <idle>-0       7d.h. 25250us : update_rq_clock.part.74 <-enqueue_task
  <idle>-0       7d.h. 25250us : enqueue_task_rt <-enqueue_task
  <idle>-0       7d.h. 25250us : dequeue_rt_stack <-enqueue_task_rt
  <idle>-0       7d.h. 25250us : __enqueue_rt_entity <-enqueue_task_rt
  <idle>-0       7d.h. 25250us : cpupri_set <-__enqueue_rt_entity
  <idle>-0       7d.h. 25251us : __smp_mb__before_atomic <-cpupri_set
  <idle>-0       7d.h. 25251us : __smp_mb__after_atomic <-cpupri_set
  <idle>-0       7d.h. 25251us : __smp_mb__after_atomic <-cpupri_set
  <idle>-0       7d.h. 25251us : update_rt_migration <-__enqueue_rt_entity
  <idle>-0       7d.h. 25251us : enqueue_pushable_task <-enqueue_task_rt
  <idle>-0       7d.h. 25251us : ttwu_do_wakeup <-ttwu_do_activate.constprop.84
  <idle>-0       7d.h. 25251us : check_preempt_curr <-ttwu_do_wakeup
```

### 2.4 Identifying probe points:

An important part of this project was with identifying the probe points so that we can understand the loop holes which contributes to latency. Our initial approach was to identify points belonging to interrupt context as well as the one which execute in the process context. Since we are working with timer interrupt, the points which were of primary interest to us were,

- Start of interrupt execution
- Checking of Timer Expiry
- Call for Task wakeup due to signal handling
- Context switch complete.

After dwelling in to the details of ftrace as shown in figure 2, we came up with the below points for probing.

**hrtimer_start_range_ns ->** This corresponds to the function when the htimer is set by the cyclictest task. I am probing this for initializing of each iteration of the cyclic test. This helps me to control the state of each iteration.

**__remove_hrtimer ->** This corresponds to very initial top half timer interrupt handling. We decided to take some latency hit from do_irq to this point as we have more control here to remove uninteresting prints.

**ttwu_do_wakeup ->** This corresponds to last portion of top half timer interrupt handling. The call to ttwu_do_wakeup happens in hardirq as it is called when signal handling is done in top half. We decided to take some latency hit from irq_exit to this point as we have more control here to remove uninteresting prints.

**finish_task_switch ->** This corresponds to the moment when the cyclictest task is scheduled in after context switch. This is in process context. We decided not to probe anything in between irq_exit to this function as I did not have control on filtering of the prints.

Although the above probe points look less, I decided to remove many other prints to avoid overhead on cyclictest and to have more controlled prints. We found that hrtimer performs handling of expired timers in top half, contrary to normal timers which use softirq to run expired timers. Hence, we did not probe anything in softirq.

### 2.5 Module for Time Stamping using Jprobes:

Once we identified the probe points the next step was to get the time stamp at these points during the cyclic test execution. Inserting printk statements requires rebuilding of kernel. In order to circumvent this, we built a module which utilizes kernel jprobes [1].

### 2.6 Filtering the probe points:

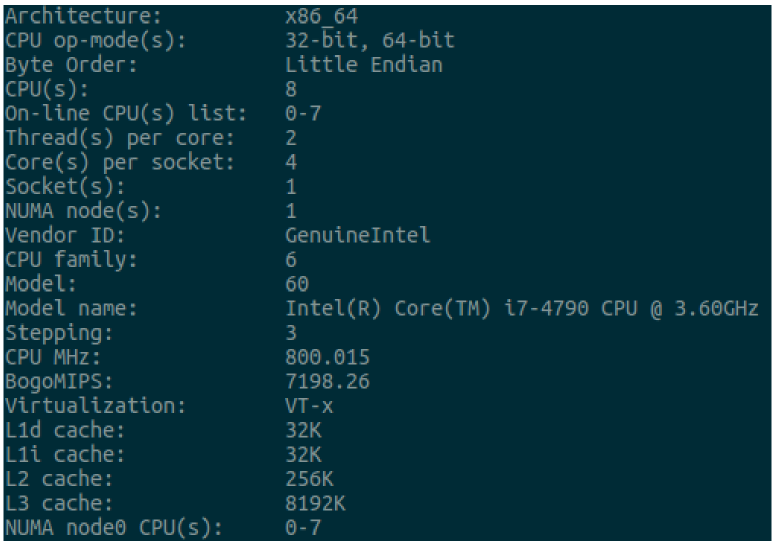
A challenging part of this project was to filter out the probe points with respect to cyclic test process, identifying whether the probe is triggered from interrupt context or process context. Below is the technique used to filter each of the below probe points.

We earlier used in_irq() and in_softirq() to decide the current context type. We removed it avoid overhead in probes and hardcoded the context based on the prints we get earlier. We tried to have a bare minimal overhead which first checks if the task we are dealing with is cyclictest or not and then checks if this is a parent process of cyclictest or the threads of cyclictest. We ignore parent process as we only need to capture the trace of the threads under execution.

## 3. Experiments:

The system we used for carrying out experiments has 16 GB of DDR3 RAM, 1 TB HDD and below cpu specification.

Figure 3 below shows cpu specification.



```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 60
Model name:            Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
Stepping:              3
CPU MHz:               800.015
BogoMIPS:              7198.26
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```

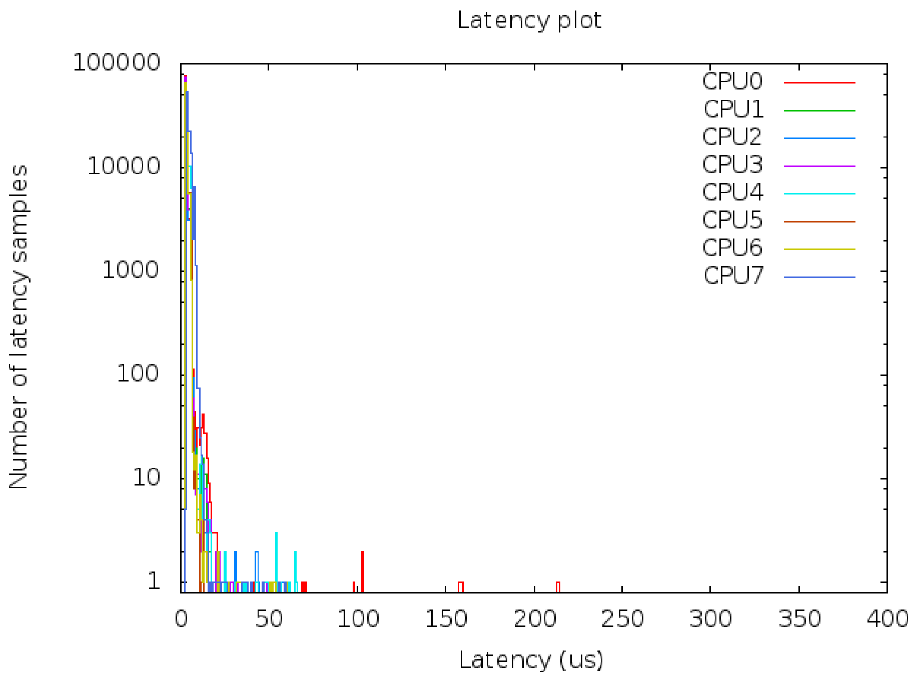
We did experiments in both native as well as virtualized setting. For each environment we get the following performance metrics. We run cyclictest in memory locked mode, for 100000 iterations with **RT priority 90**.

**Performance Metrics:**

- Histogram Plots**:**
  A histogram of latency for each of the virtualized & native environment setting.
- Average Time between probe points:

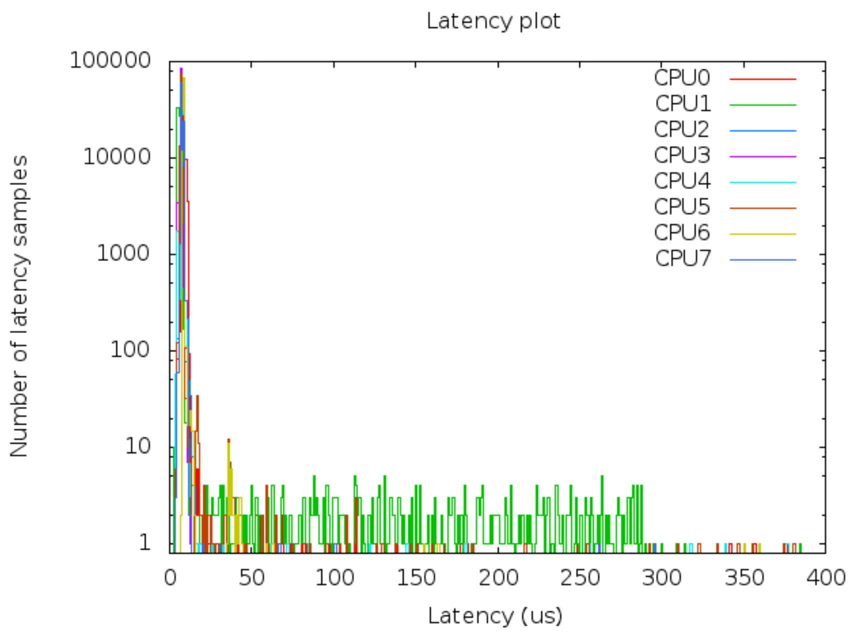
**Results:**

1. CentOS Native:



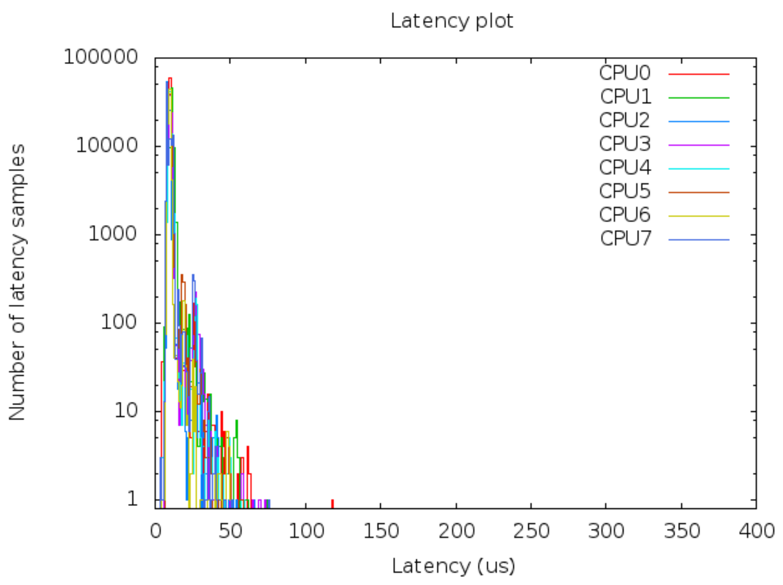
Inference: The latency in centos native has few samples with higher latency and the average latency is almost equal to that of redhawk native average latency. We have explained about it in the evaluation section.

2. CentOS Host - RedHawk Guest



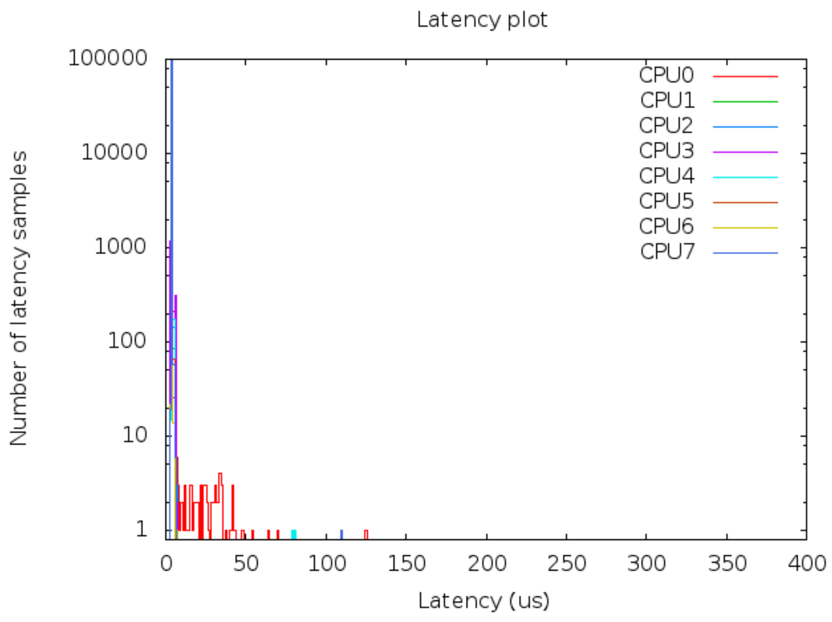
Inference: Processor Shielding is enabled in Real Time RedHawk where we bind a CPU for high priority tasks and shield it from interrupts and daemons. However, since this is a virtual CPU, CentOS treats the RedHawk VM as a non-real time normal application and can pull away this shielded CPU[6] without any constraints thereby increasing the latency on that CPU. Thus, we observe the presence of jitters in the above graph.

3. CentOS Host-CentOS Guest



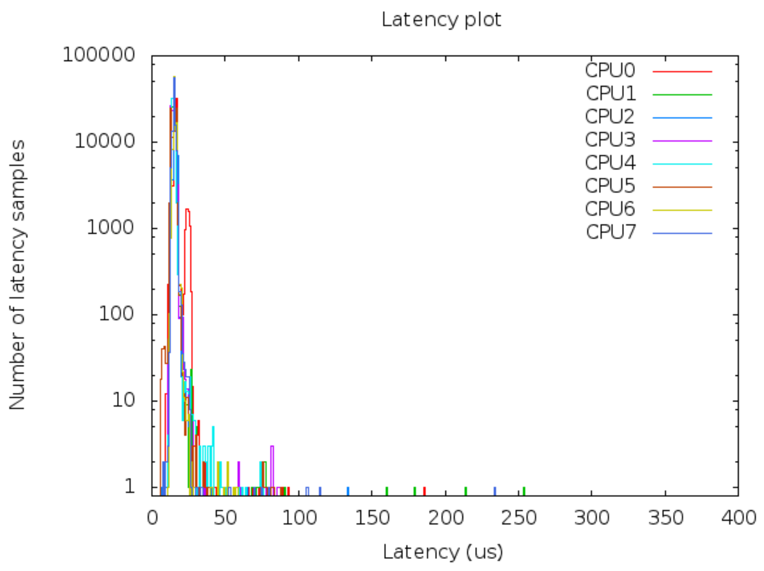
Inference: We observe that the latency performance of CentOS guest on CentOS host is worse than that of CentOS native. This is expected since the real time application on CentOS guest is treated as a normal priority VM application (Since VM is running with non-real time priority) on CentOS host and there is an overhead of virtualization. Thus, we observe more number of samples with higher latency.

4. RedHawk Native:



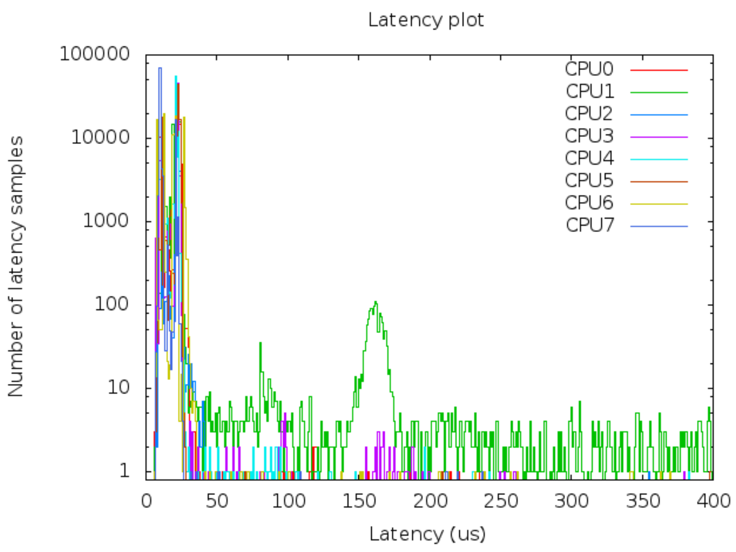
Inference: We observe very less average latency for RedHawk native experiment. When compared with centos native, the latency is comparable. We do observe presence of jitter for CPU 0, but considering the high number of samples present in a low latency range we get a lower average. Thus, on an overall basis it meets the expectations of a Real Time Operating System.

5. RedHawk Host-CentOS Guest



Inference: Here we are running CentOS guest on RedHawk host. As centos guest, does not have optimized scheduler for real time OS and it is running in virtualized settings, we observe higher latency. This is in accordance with our results where we observe higher number of samples with a greater latency when compared with RedHawk host.

6. RedHawk Host-RedHawk Guest



Inference: Here we are running RedHawk guest on RedHawk Host. The RedHawk host treats RedHawk guest as a non-real time application thereby abstaining it from real time scheduling benefits. Although RedHawk guest has enabled processor shielding[6] as a real time feature, the shielded core allocated to the guest can be evicted by the RedHawk host to meet his own real time requirements. Hence, we can observe even a higher latency than for CentOS guest on RedHawk host. This could be a reason behind the presence of jitter in the above graph.

## 4. Evaluation:

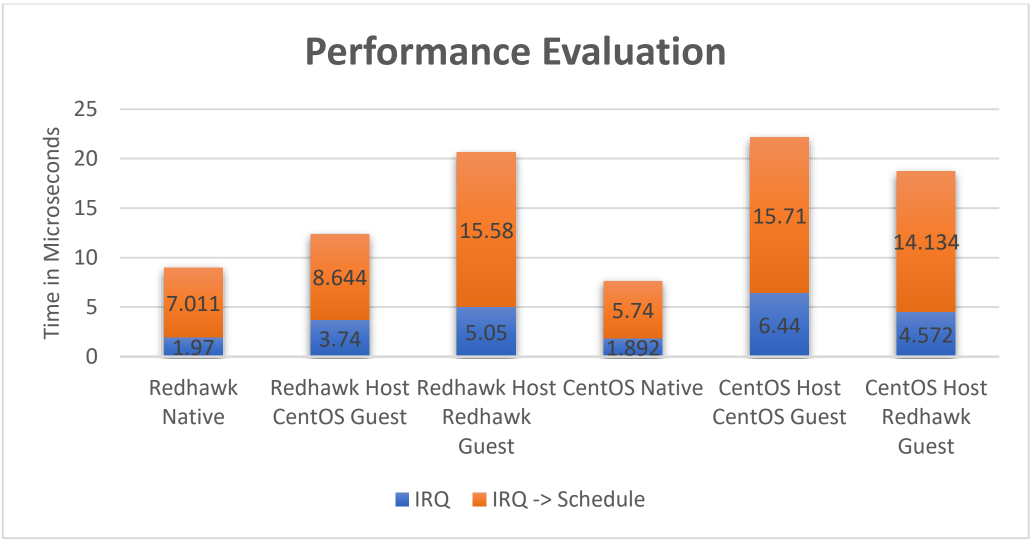


*Figure 4: Comparative Analysis of the latency of different Operating System Configurations with RT priority = 90*

- **RedHawk Native v/s CentOS Native:** From the above chart we observe that latency performance of CentOS and RedHawk native is comparable. We also experimented without jprobes and with both real time and non-real time priority for cyclictest. Below are the tabulated results of the same for 100,000 iterations of cyclictest.

| | CentOS native | RedHawk native |
|---|---|---|
| Non-real time priority | 51 us | 25 us |
| Real-time priority | 4 us | 4 us |

The discrepancy in evaluation chart (Fig 4) can be explained from the above table. We observe that for non-real time processes, RedHawk gives a much better performance than CentOS. This is due to the optimisations present in the RedHawk OS. For real time processes, we observe that since the CPU is not significantly loaded, hence the process almost always finds a vacant CPU to schedule. Using jprobes for real time processes, give similar results. We did not incorporate jprobes with non-real time processes because we prefer to evaluate it with real time processes.

- **CentOS host-CentOS guest v/s RedHawk host-CentOS guest:** We observe that latency of CentOS guest on CentOS host is almost double to that of CentOS guest on RedHawk host. There is disparity in the interrupt handling of the two configurations. Using jprobes we find that the time to wake up the real time task after expiry of hrtimer is double. This may be due to fast wakeup services [2] enabled in RedHawk, which is not present in CentOS host. In the time interval between wakeup and scheduling of task there is also disparity present where CentOS host gives latency twice to that of RedHawk host. This might be due to scheduling enhancement features enabled in RedHawk host, which are not enabled in CentOS host.

- **RedHawk host-RedHawk guest v/s CentOS host-RedHawk guest:** The performance of the two configurations is comparable since the bottleneck in both configurations is the RedHawk guest and in both configurations it is being treated as a non-real time application. Thus, the guest might not be able to gain advantage from the real time features and hence has greater latency in both configurations.

- **CentOS host-CentOS guest v/s CentOS host-RedHawk guest:** We observe that CentOS guest has more latency than RedHawk guest. The wakeup is slower in CentOS since it is a non-real time system and does not have the fast wakeup service enabled. The time interval after wakeup is comparable for the two guests. However, it is slightly better for RedHawk guest since it has faster scheduling for real time process.

## 5. Conclusion:

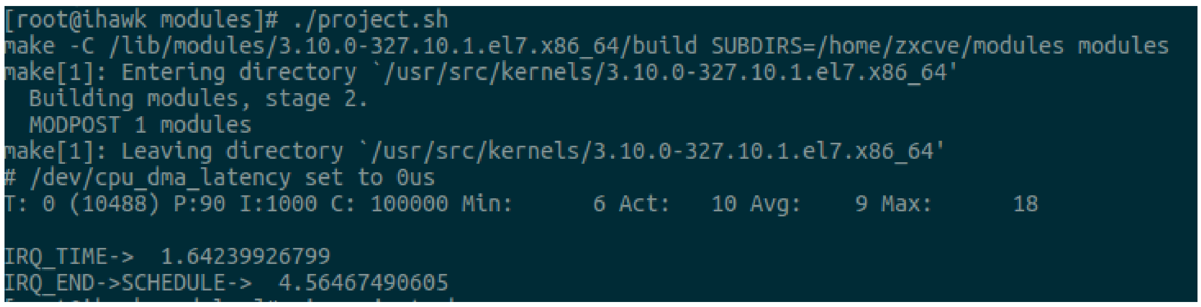
- Overall, RedHawk native and CentOS native performs best with Real time processes.
- CentOS host with CentOS guest performs worst of all as it does not have any real time features enabled.
- We see that with virtualization performance drops due to slower wakeup of process and higher scheduling time.
- Among all virtualized experiments we observe that RedHawk host-CentOS guest performs best. This is since the guest is benefiting from the enhancements of a real time enabled host.
- We conclude that using RedHawk guest is not advisable for obtaining low latency performance. This is because features like CPU shielding expects the CPU to be reserved but it gets evicted by the host there by degrading the performance.

## 6. Reference:

[1]http://lxr.free-electrons.com/source/Documentation/kprobes.txt
[2]https://wiki.centos.org/HowTos/KVM
[3]http://redhawk.ccur.com/docs/root/1Linux/1RedHawk/7.2/0898003-7.2.pdf
[4]http://people.redhat.com/williams/latency-howto/rt-latency-howto.txt
[5]http://events.linuxfoundation.org/sites/events/files/slides/cyclictest.pdf
[6] http://news.ccur.com/isdmanuals/2RedHawk%20Linux/0898004-300_RedHawk_Linux_1.X_Users_Guide.pdf
[7] http://www.osadl.org/Create-a-latency-plot-from-cyclictest-hi.bash-script-for-latency-plot.0.html

## 7. Appendix:

Run "./project.sh", this command will perform make for modules and install the modules followed by running the python user space program to parse the dmesg.



```
[root@ihawk modules]# ./project.sh
nake -C /lib/modules/3.10.0-327.10.1.el7.x86_64/build SUBDIRS=/home/zxcve/modules modules
nake[1]: Entering directory `/usr/src/kernels/3.10.0-327.10.1.el7.x86_64'
  Building modules, stage 2.
  MODPOST 1 modules
nake[1]: Leaving directory `/usr/src/kernels/3.10.0-327.10.1.el7.x86_64'
# /dev/cpu_dma_latency set to 0us
T: 0 (10488) P:90 I:1000 C: 100000 Min:      6 Act:    10 Avg:    9 Max:      18

IRQ_TIME->  1.64239926799
IRQ_END->SCHEDULE->  4.56467490605
```

To get the histogram execute "./hist.sh". By default, the core is set to 1. If you want to change it you have to change $core variable in hist.sh[7].