Project 5 – Regular vs Real-Time Linux latency in virtualized environment

Ho-Ren Chuang
Chris Jelesnianski
Mincheol Sung

Introduction

This project sets out to measure the latency differences between real-time-enabled Linux kernel (RedHawk) and an ordinary Linux kernel (CentOS), in a virtualized environment using the KVM hypervisor.

Description of Experiments

For this project we used:

Compal BLB3 Laptop
CPU: Intel i7 @ 2.80GHz (4 core)
RAM: 4GB
Number of Virtual CPUs: 2
Dedicated RAM to VM: 1GB

For this project, we were instructed to use *cyclictest*. *cyclictest* is a real-time Linux metric application, able to evaluate the real-time performance of a given platform. More specifically, *cyclictest*, measures the latency of response to a stimulus. We also needed to setup our test platform to investigate the four different configurations (host OS / guest OS):

- CentOS / CentOS
- RedHawk / RedHawk
- CentOS / RedHawk
- RedHawk / CentOS
- Native CentOS
- Native RedHawk

Though, to get the latency imposed when running applications in the above configurations, we also needed to run *cyclictest* on the native configurations for CentOS and RedHawk operating systems. This gives us a reference value for comparing the various VM configurations to each other as well as to native.

Once we got all configurations installed, we prepared and ran several experiments varying parameters of interest and ran them on all four combinations as well as on the native installations of CentOS and RedHawk. For each platform, we had 10 runs for each VM configuration and native stimulus. For obtaining the performance data, we ran the command listed below:

```
$  cyclictest -t -d0 -n -l 100000 -i 1000 -p 80
```

This command runs *cyclictest* with the following options:

-t indicates *cyclictest* will launch 1 thread per available processor
-d0 indicates that all threads will be launched instantaneously at the same instant
-n indicates to use nano_sleep
-l indicates 100,000 loops
-i 1000 indicates for *cyclictest's* timer to be set to expire in 1,000us
-p 80 indicates setting *cyclictest's* thread priority to 80

Initially, to get better understanding of each system configuration performance, we varied the following parameter:

-i was changed to 1,000 / 10,000 / and 100,000 .

This was done to see how *cyclictest* behaves when the timer's expiration time is varied. In our initial running of *cyclictest* on all the configurations, for the most part we noticed varying these values did not improve or worsen our results; nonetheless we show all three variations.

In addition, we were asked to investigate our results and reason about the observed latency differences between the various VM configurations. To do this, it was necessary to trace *cyclictest* throughout its runtime. Therefore, to simplify this searching and tracing process, we ran *cyclictest* with a single thread and only observed a single run, repeatedly of course, until we deciphered a detailed enough map to associate latency to the various processes that happen within *cyclictest*. The *cyclictest* experiment run was:

```
$ cyclictest -t1 -p 80 -i 1000000 -l 1 -n
```

This command runs *cyclictest* with the following options:

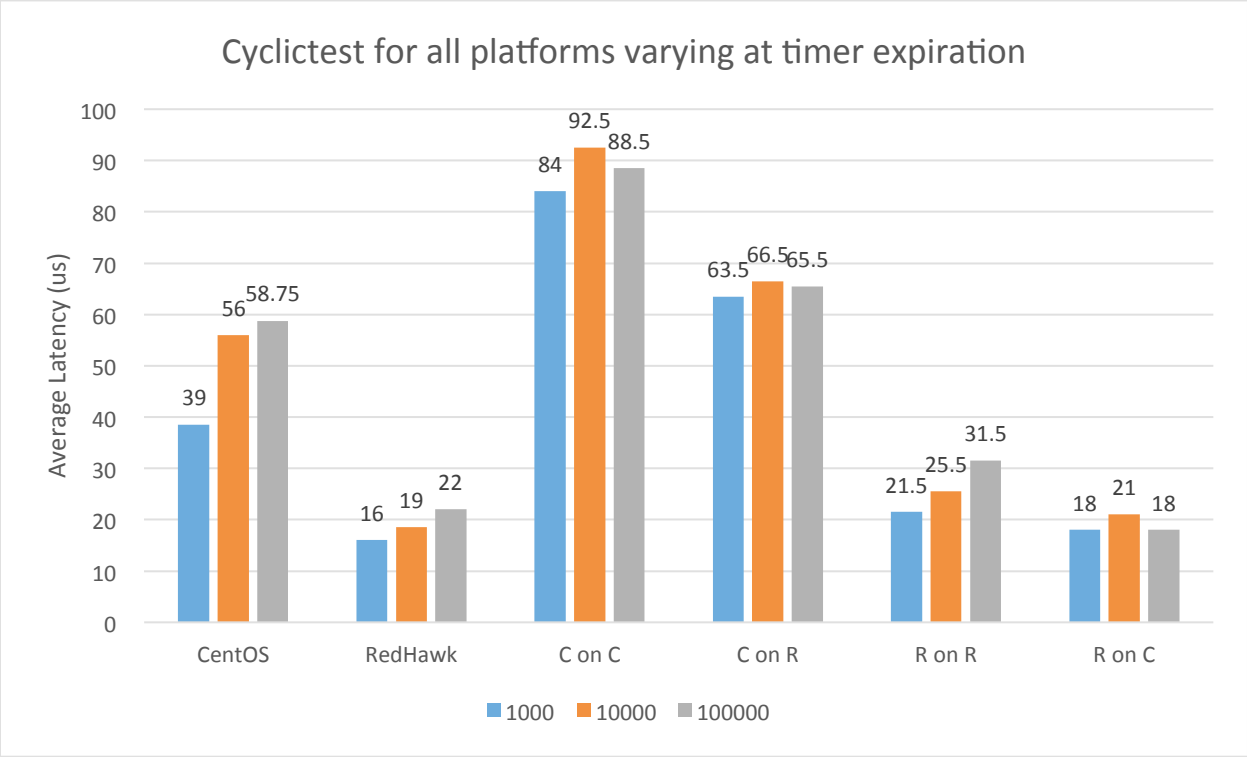
-t 1 indicates 1 thread
-p 80 indicates priority, meaning priority that *cyclictest* has when it is being scheduled.
-i 1000000 indicates how many micro-seconds *cyclictest*'s timer expire in, in this case 1 second
-l 1 indicates the the number of iterations the loop of scheduling a timer should run (essentially how many times *cyclictest should run)*
-n indicates for *cyclictest* to use nano_sleep, we specify this so that we access to a higher resolution of timing information results

By default, *cyclictest* uses option -s (which indicates to use sys_nanosleep & sys_setitimer). Using this configuration (-s not -n), Linux handles the timer interrupt handler triggered by *cyclictest by doing* posix_timer_fn().To get the best results possible, we chose to use the -n flag, instructing *cyclictest* to register hrtimer_wakeup() as its timer interrupt handler.

Measurement Data

To properly evaluate latency between the various configurations 10 runs for each timer expiration within each configuration were performed and then averaged. In addition, a large loop count was also utilized for a better value per each run. Performing the *cyclictest* on native gives us reference values for the VM host/guest configurations compared to running natively. This allows us to get the latency

imposed when running applications on one the given host/guest VM configurations. Below are our findings.



Immediately, it can be observed that native native environments (CentOS vs RedHawk) vary greatly in performance with native CentOS averaging around 50us compared to native RedHawk being less than 20us. Additionally, the overhead of running applications within a VM is apparent and visible for all four different configurations. One notable result we found is that running a real-time-enabled OS VM on an ordinary OS actually boosts overall latency performance on the VM outperforming a regular OS (CentOS) VM on a real-time-enabled host. Not surprisingly, comparing the replicated configurations (CentOS on CentOS & RedHawk on RedHawk) shows that overall real-time-enabled configuration saw less losses with virtualizing (only 28%) compared to virtualizing ordinary OS (42%).

Comparing Replicated Configurations		
Configuration		Performance compared to native
CentOS	51 us	72.91 % slower than native
CentOS on CentOS	88.33 us	
RedHawk	19 us	38.96 % slower than native
RedHawk on RedHawk	26.17 us	

Finally, looking at the results when varying *cyclictest's* timer expiration, it can be seen that depending on the timer expiration, the inherent latency of the configuration takes over and becomes the lower bound of the time needed to schedule a task when the expiration time is super small

Comparing Target VMs on the same native		
Configuration	Latency	Performance compared to CentOS VM
RedHawk on CentOS	19.00 us	78.49 % faster
CentOS on CentOS	88.33 us	

RedHawk on RedHawk	26.17 us	59.85 % faster
CentOS on RedHawk	65.17 us	

Overall, an important point is also that having a real-time-enabled host ALWAYS helps and has the best overall performance compared to a regular host.

Investigation

When we initially started this project, we first started investigating *cyclictest* by digging into its source and finding where it entered kernel space. In one methodology (which we will explain below), we simply scrutinized the code and followed its flow from setting up the timer and registering a callback function when it expires, to the time the thread is put to sleep until it is ready to wake up, to running the callback function to get back to userspace. In addition to this methodology, we also utilized ftrace and dump_stack() to assist our understanding of the call flow.

Figure 1 below is a high level view of the entire process when *cyclictest* is run from the terminal. The left side of the figure denotes actions occurring in the process context, while the right side denotes what goes on under the hood for the interrupt context. As is expected, *cyclictest* runs for a specified time before its timer is expired and returns. However unavoidable, although tiny, overhead is induced whenever interacting with kernel space. The large line in the middle (line 5) of the figure denotes the entire “kernel-side” runtime of *cyclictest* (which is approximately the time from it registers a kernel timer to when it’s rescheduled). Ideally, only line 1 exists. But due to, delay of other timer interrupts being handled (line 2), interaction with the kernel (line 3) as well as the delay waiting (line 4) to be run while in the run queue, the actual time needed to run *cyclictest* is the sum of line 1, 2, 3, and 4 (Line 5 in the figure).

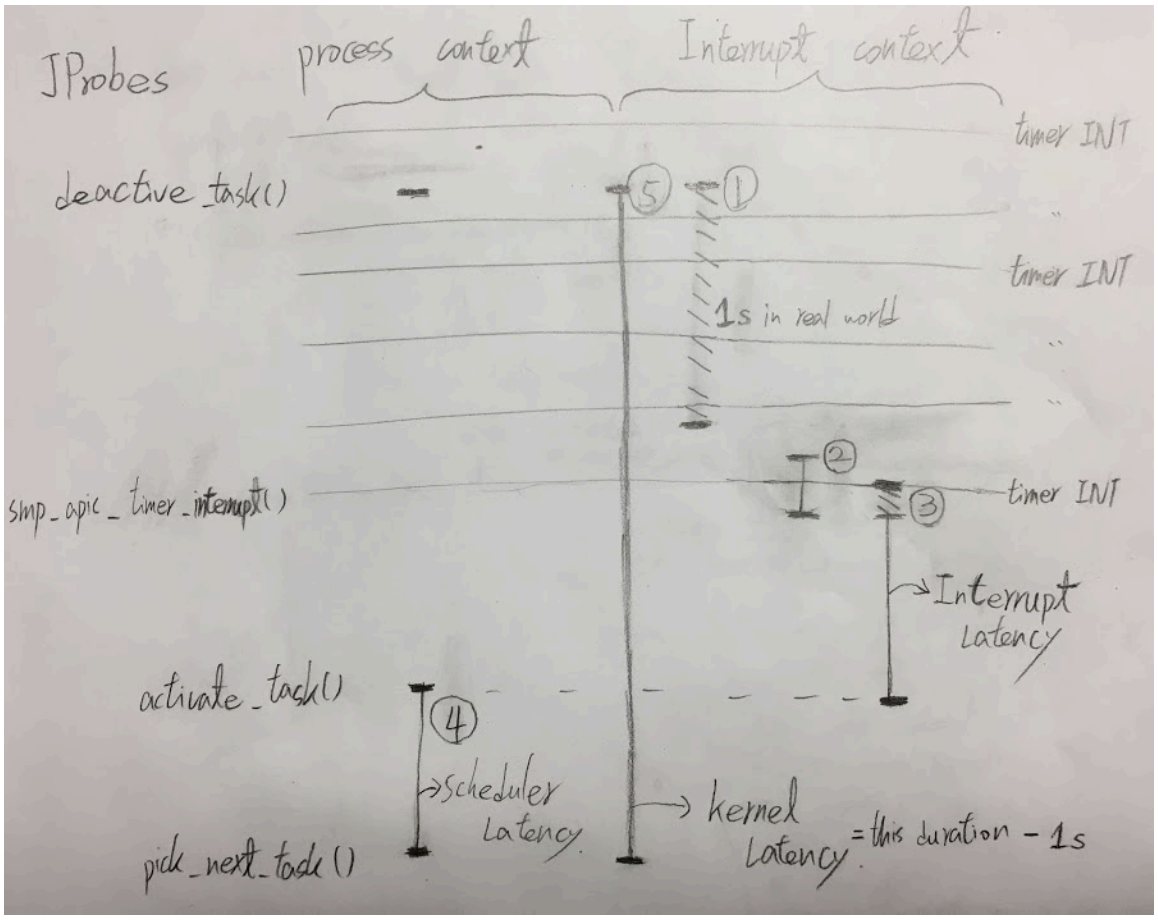


Figure 1.

(BTW: Both lines 2, 3, and 4 are exaggerated to be visible, and should not be assumed to be proportional to line 1)

To better understand the latency when interacting with the kernel, the following explains the low level detail for line 3. We found the system call which put *cyclictest* to sleep for a certain amount of time (looking at clock_nanosleep which called system call do_nanosleep). Now since we are in the kernel side of the application, we could fully understand where the latency is coming from. Being in the kernel side of execution, the cyclictest thread registers a timer in kernel and is put to sleep. The kernel timer interrupt handler takes over by periodically checking the newly registered timer. Each time the kernel timer interrupt handler goes off, the function “hrtimer_runqueue” actually checks if a given timer has expired. If it has not expired yet, the handler exits and relinquishes control of the CPU. As the kernel continues to check all of its timer in the system, eventually our *cyclictest* timer will expire. When the timer finally expires, the interrupt handler removes *cyclictest’s* timer from the list it has to check as well as change the timer state, so that the *cyclictest* can be scheduled to run via “hr_timer_wakeup” and be enqueued. This process is shown in Figure 2 below.

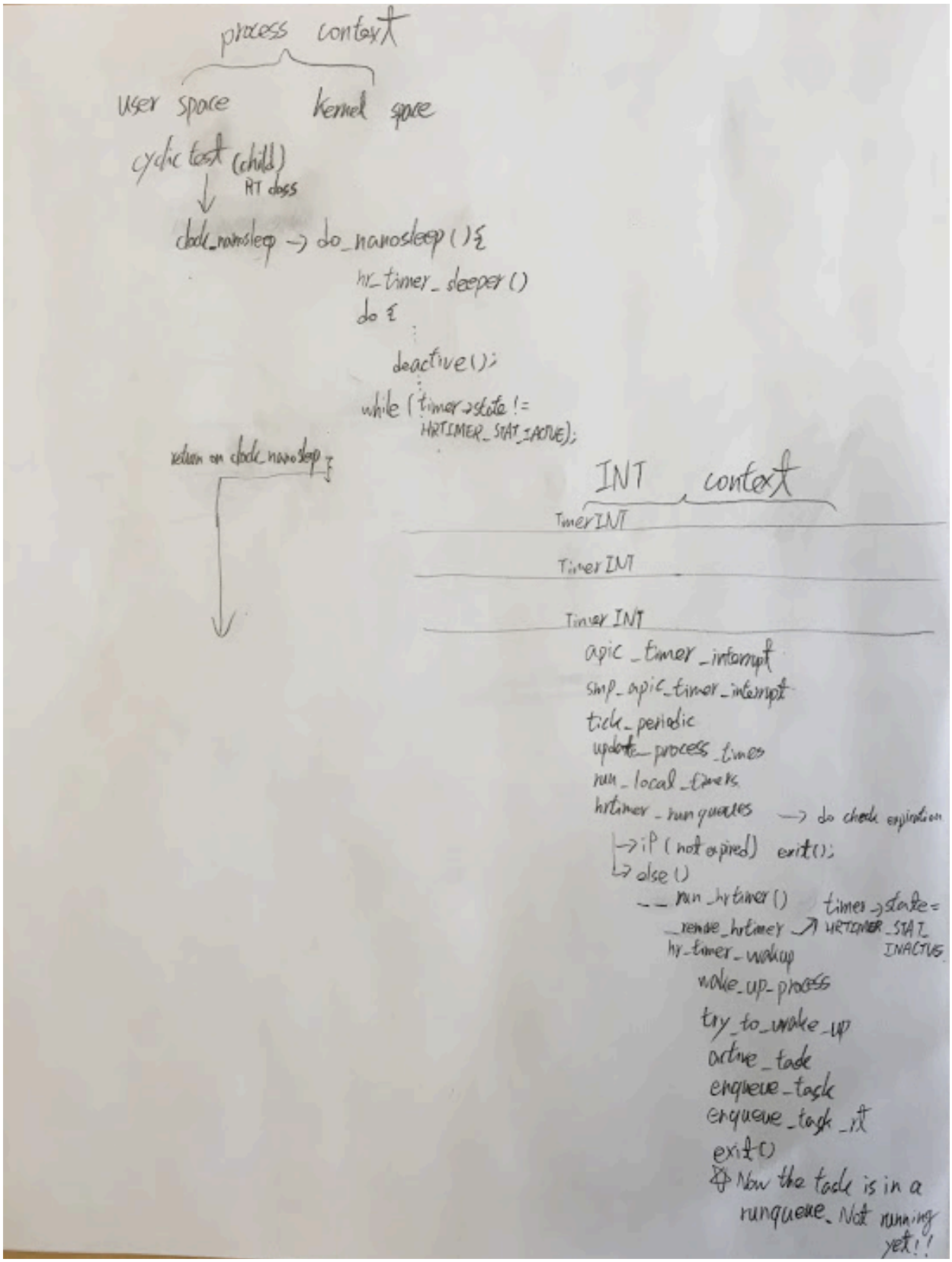
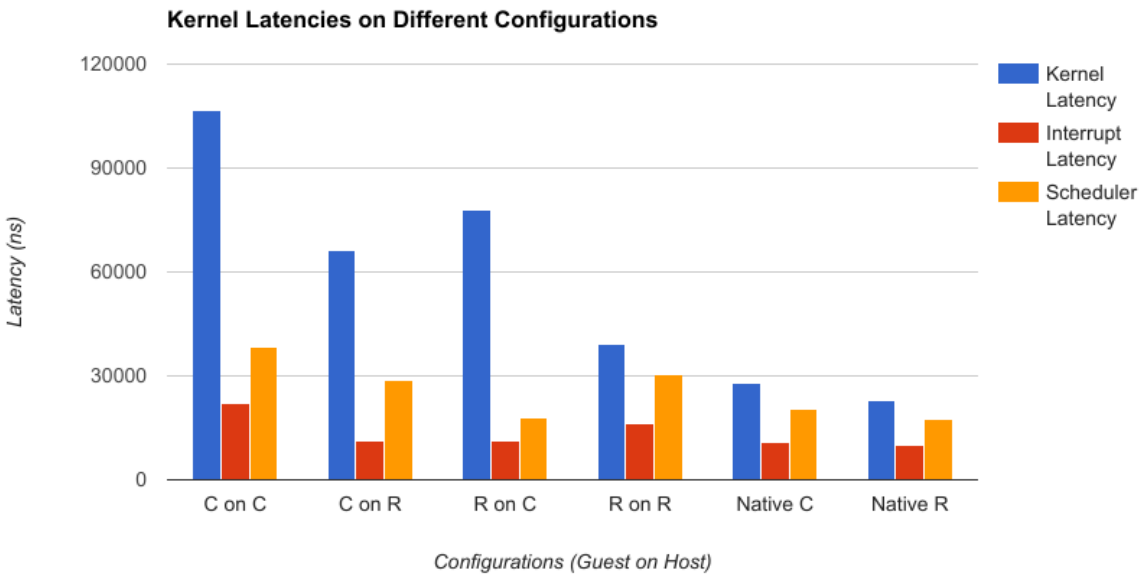

process context
user space
kernel space
cyclic test (child)
RT class
clock_nanosleep -> do_nanosleep(){
hr_timer_sleeper()
do {
deactive();
while (timer->state != HRTIMER_STATE_ACTIVE);
return on clock_nanosleep
INT context
TimerINT
TimerINT
TimerINT
apic_timer_interrupt
smp_apic_timer_interrupt
tick_periodic
update_process_times
run_local_timers
hrtimer_run_queues -> do check expiration
->if (not expired) exit();
->else()
__run_hrtimer() timer->state = HRTIMER_STATE_INACTIVE
__remove_hrtimer
hr_timer_wakeup
wake_up_process
try_to_wake_up
active_task
enqueue_task
enqueue_task_rt
exit()
Now the task is in a runqueue. Not running yet!

Figure 2.

After going through *cyclictest's* call flow, we created a module using KProbe and JProbe to hook with, what we thought, were the interesting and useful functions as shown in Figure 2 to obtain these various unavoidable overheads. Source is enclosed. As you can find in the module source code, we not only use JProbe but also use KProbe since we need to check some function when it returns. For example, on a invocation of pick_next_task(), we will not know when and which task it chooses to run until this function call returns. For that reason, JProbe is not enough so we use KretProbe instead to hook our time stamp function on a return of pick_next_task(). Moreover, we filtered unnecessary prink() messages printed by all applications except *cyclictest*. This was necessary to isolate the *cyclictest* relevant calls from all the other background processes running on the machine.

P.S. For reproducing our experimental results, please check the p5_result_and_scripts/READ_ME in our tar file for more details.

Once we ran our module on top of a running *cyclictest* instance we obtained the following results:



In this graph, "Kernel Latency" refers to line 5 (entire time in kernel) minus the assigned 1 second timer from terminal options, "Interrupt Latency" refers to line 3, and "Scheduler Latency" refers to line 4 as described in the beginning of this section. As expected, native RedHawk has the overall lowest latency and CentOS on CentOS has the overall highest (and worst) latency. Looking at the Kernel Latency, native again has the lowest latency compared to running within virtualization. No matter which OS is being virtualized, having a RedHawk host is always more performant with lower latency. When running a different virtualized OS than the host (CentOS on RedHawk, RedHawk on CentOS), again we see that having a RedHawk host is better than the CentOS host. We infer this performance happens because there is a larger benefit of the RedHawk host timing sensitivity and policys influencing the Virtual Machine's scheduling runtime.

For each of the four different configurations plus 2 native, we also performed 11 runs and performed the average. An interesting phenomenon we observed while collecting data for individual latencies

between the various functions aforementioned was that our first run always seemed to be an outlier. For this reason, we considered the first run a “warm-up” and was discarded from all averages.

Conclusion

In conclusion, we looked at the *cyclictest* README and configuration options to get a good handle of how to configure it. We ran *cyclictest* in various configurations to obtain their respective delays. As a second stage of this project, we investigated the inner workings of *cyclictest*, following its control flow within the kernel, finally this allowed us to extrapolate why and how unavoidable latency occurs on different configurations when running applications.