Real-Time Scheduling Evaluation

Bojun Li, Yitong Ding, and Ziwei Zhao

{bojun, ding.yitong, ziweizhao}@wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA

Dec 2, 2015

ABSTRACT

With the waves of new technologies, real-time system plays a vital role in the industry and commerce today. More and more scientists and companies, like Google, Tesla, Boeing and Apple, focus on researching real-time systems which are divided into soft real-time systems and hard real-time systems. Among these companies, Google even has announced its unmanned vehicles with real-time systems. In this paper, we are going to evaluate the real time system based on Raspberry Pi to find whether the performances of real situation match that of the theory. Moreover, we will analyze the reasons behind the performances and make several conclusions.

Keywords: Real-Time System; Timing APIs; Raspberry Pi; Evaluation; Scheduling Latency; WCET.

1 Introduction

Real-time systems have many unique characteristics and properties: periodic tasks, aperiodic tasks, deadline, priority, response time, etc. All these factors contribute to the useful and powerful real-time systems. We have learnt the knowledges about real-time systems in the class. In seeking to deepen our understanding, we would like to make experiments based on Raspberry Pi to verify our questions and supposes.

Raspberry Pi: Raspberry Pi is a portable device with strong functions: run operating systems, play games, watch movies, connect to Internet, etc. There is a fast microprocessor, BCM 2835, in the center of the circuit board. With the HDMI cable, we even can show all informations on screen. In addition, Raspbian is the embedded linux system running on Raspberry Pi. It provides both GUI and terminal windows to users.

2 Timing APIs

This section describes several important timing APIs. In addition, we will analyze the advantages and disadvantages of using these APIs. In the end, we realize timing function rdtsc which is not available with Raspberry Pi.

2.1 Overview of Timing APIs

Linux operating system provides some useful and powerful timing APIs, such as time(), clock_gettime(), gettimeofday() and clock(). Some of them have microsecond and even nanosecond resolution, which means we may use them to measure short time intervals. Here are some APIs:

clock_gettime(): clock_gettime() provides time with nanosecond precision. It returns a structure with two long integers, tv_sec for seconds and tv_nsec for nanoseconds. However, it takes a bit time to return results, because it will reach the system kernel, do some system calls and do mathematical computations to fit the structure it shows. Moreover, the speed of the processor in Raspberry Pi is 700Mhz that we cannot get the precise nanosecond, since the CPU is not as fast as 1000Mhz. Therefore, clock_gettime() cannot provide the precise time we need.

gettimeofday(): gettimeofday() returns the time of wall clock with microsecond resolution. We may get two structures - timbal and timezone with gettimeofday(). tv_sec for seconds and tv_msec for microseconds. Nevertheless, we face the same trouble as clock_gettime() - the precision is not satisfied with the requirement for sometimes we need to measure very short intervals.

rdtsc(): rdtsc() is one of the most critical timing functions, because it will return the time from system timers and system counters. Unfortunately, the operating system, Raspbian, in Pi is an embedded linux system and has been cut off some functions. Thus, there is no rdtsc() in Raspberry Pi. But we emulate rdtsc() by ourselves in the following section.

time() and **clock()** provide time with precision in seconds that we do not consider these functions to do the measurement.

2.2 Evaluation of Timing APIs

In seeking to find the proper timing functions, we made an experiment to show the time cost of each functions.

We use code below to measure the time cost by each timing functions.

```
start = gettimeofday();
for(int i = 0; i < 10000000; i++){
        timing API();
```

}

end = gettimeofday();

interval = end - start;

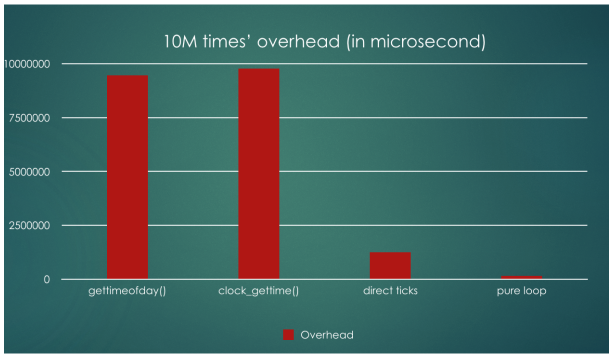


Figure 1: Time cost comparison

Figure 1 illustrates the time cost of different timing functions. Each function runs 10,000,000 times with a for loop. We record the start time and the end time. Test of 10,000,000 calls to gettimeofday() done in 0.946s. clock_gettime() takes approximate same time to do the job - 0.977s. The rdtsc, which is direct ticks in Figure 1, reduces the time significantly to 0.148s. We find that getting time with rdtsc() is much faster. Since rdtsc() reads time from system clock directly, there is no system calls or mathematical calculation.

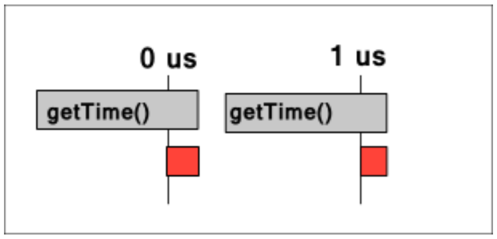


Figure 2: Ideal situation of clock_gettime()

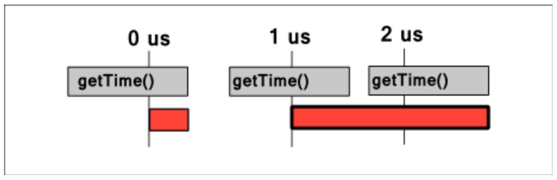


Figure 3: Real situation of clock_gettime()

Figure 2 shows the ideal situation of clock_gettime(). In this way, clock_gettime() just takes a little time that even can be ignored. However, in the real situation, showed in Figure 3, the function would take much longer time, 2us, to get results from hardware each call that it takes at least 4us to measure a time interval. That is because system will enter kernel mode and do math to fit the structure when calling clock_gettime(). Therefore, the precision of gettimeofday() and clock_gettie() cannot satisfy the requirement of real-time system.

2.3 Emulating Rdtsc in Raspberry Pi

As we observed in previous experiments, the ready timing APIs cannot guarantee the precision we need. We find rdtsc is another way to get high resolution time. But rdtsc is not available in Raspbian, operating system of Pi. Then we decide to realize rdtsc in Pi by ourselves.

There are two ways to emulate rdtsc in Raspberry Pi, getting direct ticks from system clock and extending base kernel with loadable kernel module. We try both of them and succeed with the first method but fail in the second.

First of all, getting ticks from system timer needs to know the memory address of the system clock register. We look it up in BCM2835 data sheet and find the base physical address is 0x20003000.

Offset	Name	Description
+0x00	CS	Control
+0x04	CLO	Lower 32 bits
+0x08	CHI	Upper 32 bits
+0x0C	C0	IRQ line 0
+0x10	C1	IRQ line 1
+0x14	C2	IRQ line 2
+0x18	C3	IRQ line 3

Table 1: system timer address

Table 1 shows the base address and the offset of each useful timer. Since we need to measure the time interval but not the exact time, the lower clock is the most important one. There are six steps to read the time out:

1 Switch the user to root

2 Open /dev/mem

3 Using mmap() to map the system clock into memory

4 Set a volatile unsigned long int pointer: timer_api

5 timer_api points to 0x20003004

6 Using *timer_api whenever you need to get the time

An example of using rdtsc:

*starttime = *timer_api;*

do something();

*endtime = *timer_api;*

interval = endtime - starttime;

Another way is to load the kernel module to extend the base kernel of Raspberry Pi to add support for rdtsc. We may use loadable kernel module dynamically to add or remove functions.

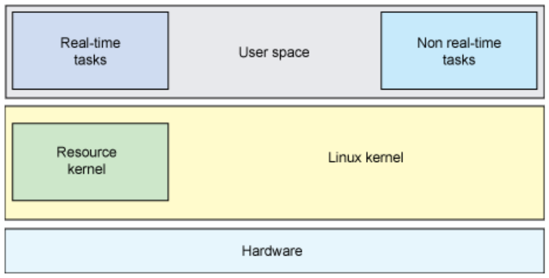


Figure 4 Loadable Kernel Module

Figure 4 shows how loadable kernel works: user inserts the extended kernel into the base kernel. This kernel could provide service for both real-time tasks and non real-time tasks. There are three steps to do it:

1 Write the kernel code

2 Use makefile to compile it

3 Use command insmod to insert the kernel into the base one

However, it returns many of errors when we insert it. We will explore this method in the future.

3 Scheduling Latency Measurement

We chose *cyclictest* to measure the scheduling latency of Raspberry pi and Raspberry pi 2. We will first review the definition of scheduling latency and describe the *cyclictest* bench benchmark. Furthermore, we will discuss the differences of scheduling latency in idle system, CPU-busy system and I/O-busy system. Finally, we will discuss the multi-core Raspberry Pi 2.

3.1 Scheduling Latency Review

In real-time systems, the scheduling latency is time until the highest priority task is executed[1]. Scheduling latency indicates a system's ability to schedule real-time tasks.

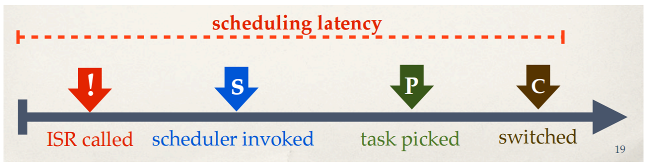


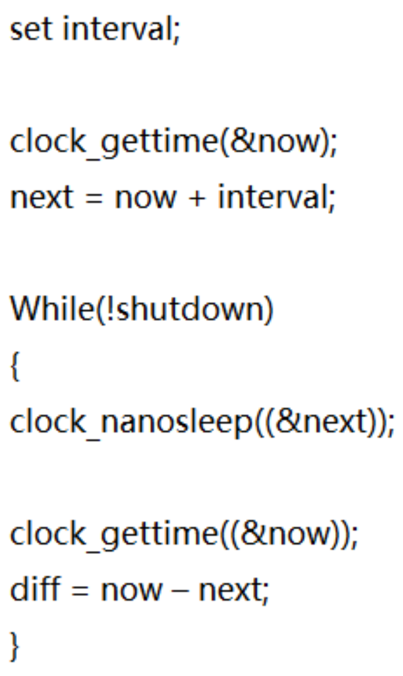
Figure 5: Scheduling Latency

For pages other than the first page, start at the top of the page, and continue in double-column format. The two columns on the last page should be as close to equal length as possible.

Figure 5 shows an example of scheduling latency. To measure the scheduling latency, we need to know the time difference between the release and the execution of a task.

3.2 Cyclictest Implementation

Cyclictest is high resolution test program written by User:Tglx[2]. The theory of *cyclictest* is to create sample jobs which are released at specific times(0 1000 2000 3000 ...), record the time when these tasks are executed, then calculate the scheduling latency by simple subtraction. Figure 6 shows pseudocode for the test loop algorithm[3].


```
set interval;

clock_gettime(&now);
next = now + interval;

While(!shutdown)
{
clock_nanosleep((&next));

clock_gettime((&now));
diff = now – next;
}
```

Figure 6: Algorithm of *cyclictest*

To measure the scheduling latency, *cyclictest* was run with this command line:

```
sudo ./cyclictest -p 98 -i 1000 -l 50000
```

Cyclictest will generate 50000 samples tasks with interval 1000 microseconds and priority 98.

In our experiments, we used this command line:

```
sudo ./cyclictest -p 98 -m -n -c 0 -h 200 -q -l 50000>test.txt
```

So *cyclictest* will lock system memory, choose nanosleep and monotonic clock. *Cyclictest* will also create a histogram of samples with scheduling latency of 0~200 microseconds. In Secs. 3.3-3.5, we will discuss our results from *cyclictest* under different system.

3.3 Idle System

As our first step, we evaluated the scheduling latency of an idle system. An idle system is a system without background tasks. To emulate an idle system, we run *cyclictest* on a Raspberry Pi Model B, with only a keyboard connected. All other devices (mouse, HDMI, Ethernet) are disconnected to eliminate additional I/O interrupts.

Fig. 6 shows the CDF chart of scheduling latency for the idle system.

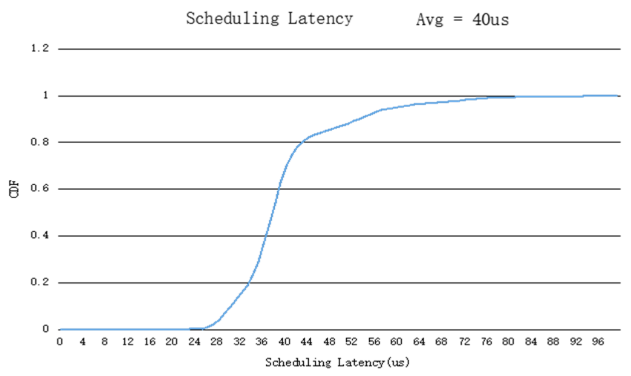


Figure 6: Latency for Idle System

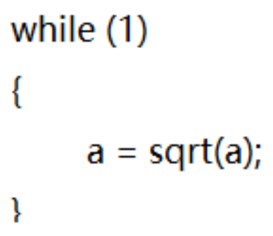
As shown in Fig. 6, most of the samples have scheduling latency of 30-50 us, the average scheduling latency for the idle system is 40 microseconds.

Idle system represents the best case scenario of latency. In the absence of background workloads, idle system has fewer interrupts and contentions, therefore has the best scheduling latency.

3.4 CPU-Busy System

To generate CPU-bound and cache-bound background workloads, we introduce the *sqrt* and *cachebench* benchmark.

sqrt: sqrt is a CPU workload generator, it will create an infinite loop of square root calculation. Its algorithm is shown in Fig 7.



```
while (1)
{
        a = sqrt(a);
}
```

Figure 7: Algorithm of *sqrt*

cachebench: cachebench will generate cache-level workloads that causes the system cache-busy.

In our experiment, we run *sqrt* and *cachebench* alongside *cyclictest* to simulate a CPU-busy system. The result are shown in Fig 8.

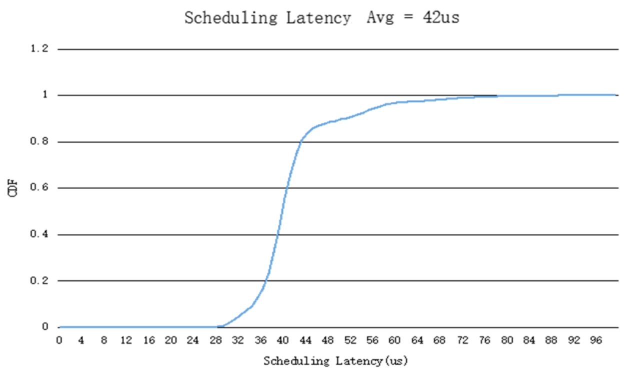


Figure 8: CPU-busy system

The comparison between idle system and CPU-busy system is shown in Figure 9.

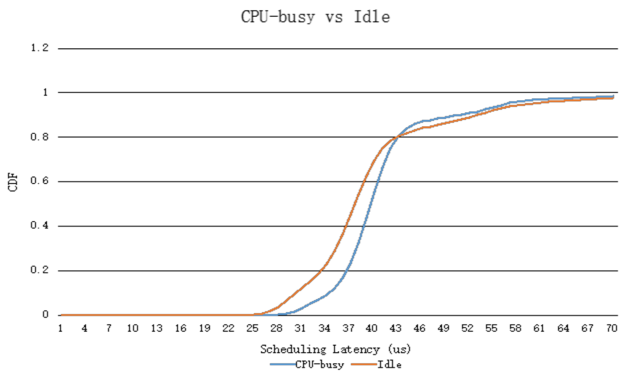


Figure 9: Idle vs CPU-busy

For the CPU-busy system, the average scheduling latency increased to 42 microseconds. Theoretically, a CPU-busy system is expected to have worsened scheduling latency[4] due to the increase in cache misses and context switches.

3.5 I/O-Busy System

I/O-busy system represents the interrupt-busy scenario, we chose *hackbench* and *netperf* to emulate an I/O busy system.

Hackbench: hackbench is a standard stress test tool for Linux system. It creates socket exchanges that frequently cause system calls.

Netperf: netperf is a benchmark that can be used to measure the performance of many different types of networking[5]. It send data package through the network to generate network interrupts.

In this part of experiment, we combined hackbench with netperf to simulate an interrupt-intensive scenario. Meanwhile, an Ethernet cable is connected to the Raspberry Pi.

Fig. 10 and Fig. 11 show the scheduling latency of the I/O-busy system.

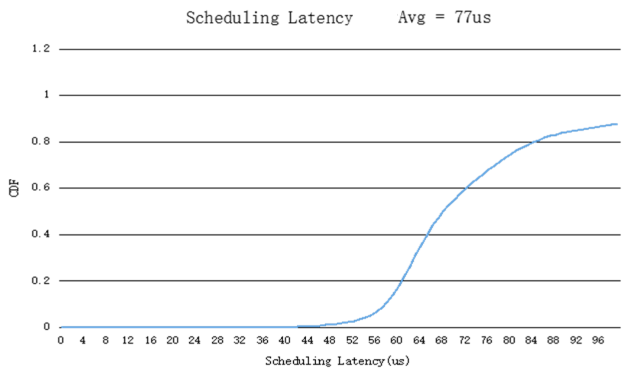


Figure 10: I/O-busy system

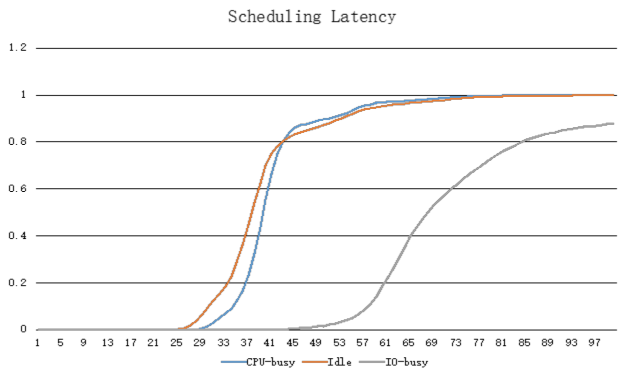


Figure 11: Idle, CPU-busy and I/O-busy.

As shown in figure 11, the scheduling latency of I/O-busy system is severely affected, which confirms that interrupt is a major source of latency.

3.6 Raspberry Pi 2

As an extension of our experiment, we implemented *cyclictest* on a Raspberry Pi 2 machine.

Compared to Raspberry Pi model B, Raspberry Pi 2 is equipped with a quad-core ARM Cortex-A7 CPU with 1GB RAM.[6]

To pin *cyclictest* on a specific processor, we used the command line:

Taskset -c 0 ./cyclictest

So cyclictest will run on processor 0.

Fig. 12 shows the scheduling latency of different processors

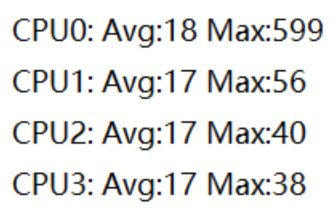
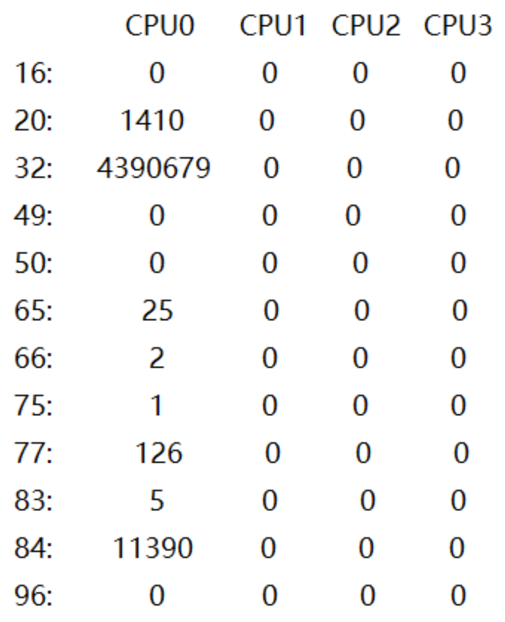

CPU0: Avg:18 Max:599
CPU1: Avg:17 Max:56
CPU2: Avg:17 Max:40
CPU3: Avg:17 Max:38

Figure 12: scheduling latency of different processors

As can be seen, CPU0 clearly has higher latency than the other 3. To explain this, we checked /proc/interrupts, as shown in Fig. 13



	CPU0	CPU1	CPU2	CPU3
16:	0	0	0	0
20:	1410	0	0	0
32:	4390679	0	0	0
49:	0	0	0	0
50:	0	0	0	0
65:	25	0	0	0
66:	2	0	0	0
75:	1	0	0	0
77:	126	0	0	0
83:	5	0	0	0
84:	11390	0	0	0
96:	0	0	0	0

Figure 13: /proc/interrupts

Most of the system interrupts happened on CPU0. Moreover, using ps -aF, we found out that *-bash* was running on CPU0 as a background task, which could explain the high latency of CPU0.

4 Worst Case Execution Time Measurement

In this section, we present our experimental evaluation of worst case responds time(WCRT) under different system environment. We have two main objectives: (1) to evaluate by how much different kind of system resource bound may affect the responds time of the real-time benchmark; (2) to find out if there is any interference may affect the responds time of the real-time benchmark;

Since there is no way to capture the release time of the task, the responds time we measured here are calculated by the end time subtract start time of the task. We know this assumption may not be very accurate, but the schedule latency is been discussed in the previous section of this paper.

4.1 Experiment Setup

Platform:

We perform our experiments on a first generation of Raspberry Pi, which includes an 700 MHz ARM1176JZF-S processor, VideoCore IV GPU, and RAM. It has a Level 1 cache of 16 KB and a Level 2 cache of 128 KB [2]. The operation system running on the Raspberry Pi is Raspbian, which is the Foundation's official supported operating system, with 32-bit Linux 3.18.11 Kernel with preempt feature. The runtime reservation is been set to 0, instead of 5% by default, to allowed the real-time process to fully utilize the CPU resource. The CPU governor is been set to ‘performance’ mode, instead of ‘power save’ mode, to pin the CPU frequency at 700MHz, for uniform the CPU resource.

Benchmark:

We used the sqrt() [3]benchmark, which simply calculate the square root of constantly increasing integer i.

```
for (i = 0; i < 2000; i++)
    temp = sqrt((double)i*i);
```

It is a single path program involve loop and floating point calculation. The iteration times is been set to achieve 500 us execution time, which in previous experiment is 5000 us. From the previous result, we realized the longer the execution time is, the more interference the task may encounter. Shorter execution time diminish the chance of task been affected by interference. The period of task is been set to 100ms, which in previous experiment is 10ms. We realized that such short period may cause CPU utilization ratio boots up too high, which, of course, cause uncertainty and fluctuation of the responds time of the benchmark. The priority of the benchmark is been set to 98, instead of the highest priority, 99, because there are a few management threads which need to run with higher priority than the benchmark. Those real-time management threads shouldn’t be a problem, since they would appear every time the benchmark runs, which even can be seen as a part of the benchmark.

System environment:

For idle system, we disconnect the Ethernet cable, HDMI cable and mouse cable to minimize the unpredictable interference that may cause by the I/O interrupt or CPU usage generated by these peripheral hardware. As we observed in the previous experiments, the interference generated by them can affect the performance of the benchmark significantly, especially the Ethernet cable, which constantly transport data with internet, even if there is no program running in the system by the user. We attribute these background data flow to the system’s build-in threads. In fact, we can observe those threads by using top function. Once the Ethernet cable been unplugged, those threads will never be active any more, hence the impact on benchmark is gone. Also, the GUI (start_x) is, of course, been turned off, to avoid non-essential CPU & I/O usage.

For CPU busy system, we used the same sqrt() function, but with infinite loop, whom we can make sure to take the same CPU resource as the benchmark do. It can boost the CPU utilization to 100%. We used to use the game – Minecraft, to generate CPU busy. However, the environment it generated was neither uniform nor pure, as at different time, the game may be busy on different calculations, and may also take some I/O resource, which is unstable and unrepeatable compare with the later sqrt() function.

For cache busy system, we used the benchmark – Cachebench, which is designed for test the memory architecture of a Linux system. The benchmark was set to ‘Read/Modify/ Write’ mode to fully unutilized the cache. It will mess up the data in the cache and trash it with meaningless data over and over again. In this way, we can simulate a system environment that is cache bounded, more importantly, uniformly and repeatable.

For I/O busy system, we used the benchmark – Netperf. Netperf is a benchmark that can be used to measure the performance of many different types of networking. What it does is that it will send data through the Ethernet port periodically, hence generate uniform I/O usage, which cause I/O interrupts.

We perform the same experiment as above under for different system environment, each collect 100 execution data for three times, to minimize the uncertainty of the effect cause by keep CPU busy for too long time.

4.2 Interference Listing

During the test experiment, we observe the CPU and memory utilization for each thread running on the system by top function. And here are the thread that runs repeatedly, which may cause interference on the real-time benchmark:

Init: In Unix-based computer operating systems, init (short for initialization) is the first process started during booting of the computer system. Init is a daemon process that continues running until the system is shut down. There is no way to remove it.

Ifplugd: ‘ifplugd’ is a daemon which will automatically configure ethernet device when a cable is plugged in and automatically unconfigure it if the cable is pulled. It appears quite often and definitely cause I/O interrupt, since it must check the cable port.

ntpd: The ‘ntpd’ program is an operating system daemon which sets and maintains the system time of day in synchronism with Internet standard time servers. Since Raspberry Pi does not have an on-chip off-power clock, it use this function to get time from internet repeatedly. And internet usage means I/O interrupt.

RCU: ‘RCU’ is a synchronization mechanism that is optimized for read-mostly situations. The basic idea behind RCU is to split updates into "removal" and "reclamation" phases. It appears when the cachebench is running. As it manage the read operation, I/O interrupt is definitely involved.

kworker: ‘kworker’ is a placeholder process for kernel worker threads, which perform most of the actual processing for the kernel, especially in cases where there are interrupts, timers, I/O, etc. It is not something that can be safely removed from the system in any way

ksoftirqd: ‘ksoftirqd’ is a per-cpu kernel thread that runs when the machine is under heavy soft-interrupt load. Soft interrupts are normally serviced on return from a hard interrupt, but it's possible for soft interrupts to be triggered more quickly than they can be serviced. If a soft interrupt is triggered for a second time while soft interrupts are being handled, the ksoftirq daemon is triggered to handle the soft interrupts in process context. If ksoftirqd is taking more than a tiny percentage of CPU time, this indicates the machine is under heavy soft interrupt load.

As the running time of these threads are all under 2ms, it is no likely for us to count how many times these threads are run from the top function. And there may be more threads been run during the test but not caught by us.

4.3 Idle system

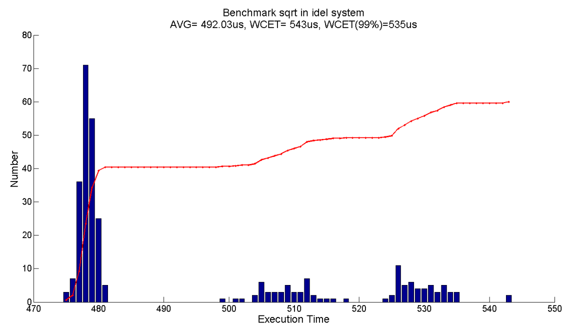


Figure 14: Benchmark sqrt in Idle System

Figure 14 shows PDF and CDF plot of the responds time of sqrt() benchmark in idle system environment. As we can see, there are clearly three discrete section in the PDF plot: one around 477 us; one around 510 us; one around 530 us. Obviously, the 477 section have more existence than the other the sections. We think, the results lies in 477 section are generated by the tasks that are really executed in the idle system, e.g. without any interference, since, as we can see later in the results of other three system environment, this section all appears in those result without any position change, which mean that is the real excitation time of the benchmark.

Those result lies in the other two section are somewhat affected by the interference mentioned in the previous section. By the amount of time, those interference may be grouped in to 35 us group and 55 us group, which cause this kind of discrete distribution of the experiment result. Most likely, those interference are cause by the I/O interrupt triggered by the daemons, because the I/O interrupt are the only tasks that have higher priority than the real-time tasks, which may have priority larger than 100, and may affect the continues execution of the benchmark.

Anyway, the idle system test provide a control-group result for the rest experiments.

4.4 CPU Busy System

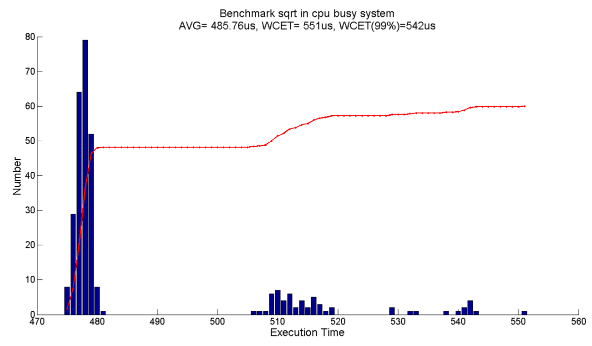


Figure 15: Benchmark sqrt in busy system

Figure 15 shows PDF and CDF plot of the responds time of sqrt() benchmark in CPU busy system environment. As we can see, the discrete distribution appears again, which caused by the same reason as idle system circumstance. Intuitively, we thought the average responds time will increase as for the heavy CPU workload. However, instead of increasing, the average responds time is reduced by a small amount. At the beginning, we thought this might be caused by experimental error. After a few more retest, result kept the same. Boldly, we attribute this counterintuitive result to the fact that high CPU restrain the number of threads, who trigger the I/O interrupt, been run. That is why the result lies in the longer-time section decreases. The increase of WCRT, compare with idle situation, might cause by those interference threads runs slower due to the high CPU utilization, hence need more time to release the I/O interrupt.

4.5 Cache Busy System

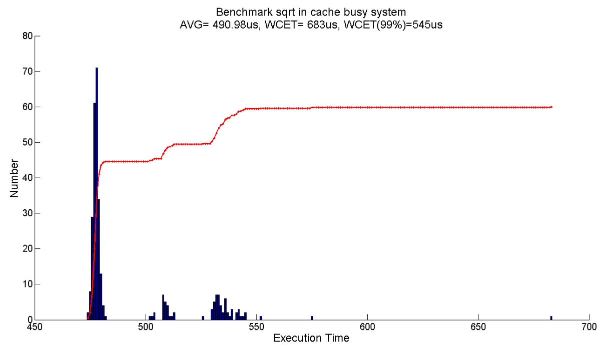


Figure 16: Benchmark sqrt in cache busy system

Figure 16 shows PDF and CDF plot of the responds time of sqrt() benchmark in cache busy system environment. As we can see, the result is pretty much the same as it for CPU busy circumstance. This is because the cachebench cannot generate a cache busy system without heavy CPU utilization. Besides, the sqrt() benchmark does not use much of cache. That is why the CPU busy and cache busy circumstances share almost the same result.

4.6 I/O Busy System

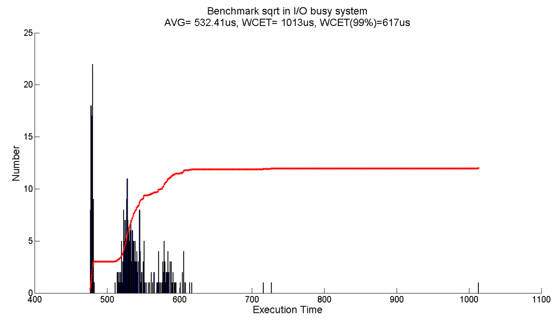


Figure 17: Benchmark sqrt in I/O busy system

Figure 17 shows PDF and CDF plot of the responds time of sqrt() benchmark in I/O busy system environment. As we can see, the average responds time increase by more than 10%, and WCRT almost doubles. The 470 us section is compressed into less than 50% for the first time. This result confirms the theory we point out in the previous section that the I/O interrupt contributes most of the interferences.

5 Conclusion

We confirm that scheduling latency can be affected by CPU-bound and I/O-bound background tasks, and idle system has the best scheduling latency performance.

As been discussed in the previous sections, the most significant interferences to the real-time benchmark are caused by I/O interrupts, who have higher priority than the real-time task in the system and sometimes may cost the quite amount of execution time to cause WCRT increases significantly.

To get a better performance from the real-time tasks, the critical point is to eliminate the effect caused by I/O relative tasks. There are two ways:

1 Reduce the priority of the I/O interrupts so that they can no longer preempt the real-time tasks. This can be done by install RT-patch on the Kernel of the system so that the real-time tasks will always have the highest priority than any other tasks. However, there are few peer works, who tried to do so on the Raspberry Pi's Kernel, but end with system crash. Apparently, more work need to be done to realize this kind of implantation on the Raspberry Pi. There is a drawback on this solution, which is it might hurt the performance of the real-time wireless communication system, whose major job is transport data from one I/O port to another. In this kind of application, I/O interrupt should always have higher priority than any other tasks since the timing of income data is unpredictable, so that any delay may cause data missing.

2 Arrange I/O related tasks as periodic tasks so that even if it have higher priority than the real-time tasks, the inferences are constant and predictable. But, again, this will definitely affect the performance of the real-time tasks.

Between these two solutions, trade-off should be made base on the specific application.

6 References

1. DOI= http://doi.acm.org/10.1145/90417.90738.
1. [1]Scheduling Latency: http://www.cse.wustl.edu/~lu/cse520s/slides/projects.pdf
2. [2]Cyclictest: https://rt.wiki.kernel.org/index.php/Cyclictest
3. [3]Using and Understanding the Real-Time Cyclictest Benchmark: https://www.youtube.com/watch?v=f_u4r6ehZKY
4. [4]A Comparison of Scheduling Latency in Linux, PREEMPT RT, and LITMUSRT: https://www.mpi-sws.org/~bbb/papers/pdf/ospert13.pdf
5. [5]Netperf: http://www.netperf.org/netperf/
6. [6]Raspberry Pi 2: https://www.raspberrypi.org/products/raspberry-pi-2-model-b/