Real-Time Virtualization

Miguel A. Arroyo, Jennifer Lam
Department of Computer Science
Columbia University
{may2114,jl3953}@columbia.edu

CSEE6118: Advanced Operating Systems - Final Report

I. INTRODUCTION

As Moore's Law continues to hold, the Internet of Things (IoT) movement has been on the rise. Smart-homes, smart-wearables, drones, and autonomous automobiles are just a few examples. These systems, which interact with the physical world, are known as cyber-physical systems (CPS). CPS exhibit certain properties that deviate significantly from those in traditional computing. The most important differences are that these systems must comply with stringent timing and fault tolerance requirements in order to properly interact with the physical world. Because of these performance requirements, CPS applications typically run on bare-metal.

However, running applications bare-metal may no longer be required. Virtualization has been increasingly popular over the last several decades, mostly in server and consumer computing. With microcontrollers (MCUs) and single-board computers (SBCs) becoming more and more powerful, virtualization has gained traction in the embedded world. For example, ARM–the leading designer of embedded CPUs–has added hardware virtualization support to their higher end processors. Virtualization has a wide range of applications and uses; specifically when used in embedded systems, it can greatly help with security and fault-containment. In this paper, we focus on this use case.

With regards to the real-time performance that many embedded systems require however, virtualization does have its limits. Virtualization implies a hierarchical two-level scheduling, which makes scheduling inherently less predictable and more complex. Up until now, this it has primarily been used to improve server workloads, most of which are not concerned with real-time latency, so little work has been done in the area of real-time virtualization. However, with more embedded devices providing hardware support for virtualization and growing interest in this area, we felt that this would be an interesting problem.

Given modern embedded system advancements, we seek to examine whether current virtualization technologies are capable of supporting CPS applications with real-time constraints. We systematically conduct a series of experiments using open-source embedded projects to demonstrate that CPS real-time workloads on embedded processors may still not be ready for virtualization.

II. WORKLOADS

As a case study, we looked at an open-source Engine Control Unit (ECU), rusEFI, and an embedded benchmark suite, Mibench. These workloads are concerned with Worst Case Execution Time (WCET) because of their hard real-time constraints. Hard real-time systems are systems where missing a deadline leads to a total system failure (as compared to soft real-time systems where the only consequence of missing a deadline is that the usefulness of your results degrade). While hard real-time workloads are not particularly taxing in the average case, they demand that the system sufficiently service any sudden bursts of quick computation.

A. rusEFI

rusEFI [1] is an open-source effort to create an engine control unit for everything from small engines to actual cars. It is implemented with small 100+ MHz microcontrollers and operates in an event driven (interrupt) manner.

rusEFI runs on a trigger position sensor event. Once per revolution, engine parameters are computed and scheduled to occur within the revolution period. These scheduled events are implemented using timer interrupts that expire at the scheduled time and transfer controll to their respective callback functions. The rate at which these events are scheduled is proportional to the RPM of the engine. Typical engines usually max out at around 9000 RPM, therefore these systems require millisecond resolution.

1) Modifications: In order to have rusEFI work under our experimentation environment, some modifications were made to the underlying library OS, ChibiOS [2]. ChibiOS is a compact, fast real-time operating system microkernel that supports a wide range of platforms. As of this writing, ChibiOS has support to run the kernel as a Win32 process, but no support for a POSIX environment. Thus, One of our main efforts was then to port ChibiOS to work in a POSIX environment for both x86 and ARM. As a result of porting ChibiOS, minimal changes were then made in order to get rusEFI to work properly.

B. MiBench

MiBench [3] is an embedded benchmark suite in the same vein as the SPEC benchmarks. It is divided into six suites, each suite targeting a specific area of the embedded

market. We focus on the Automotive and Industrial Control, Consumer Devices, and Networking categories. All programs are available as standard C source code.

1) Automotive and Industrial Control: The Automotive and Industrial Control category focuses on basic math computations, bit manipulation, data I/O and simple data organization. The *basicmath* test performs simple mathematical calculations that often dont have dedicated hardware support in embedded processors. The *bitcount* algorithm tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers. The *qsort* test sorts a large array of strings into ascending order using the well known quick sort algorithm. The *susan* is an image recognition package developed for recognizing corners and edges in brain MRI's.

2) Consumer Devices: The Consumer Devices category represents consumer devices, such as scanners, printers, etc, that have become popular. Only one of the benchmark tests, *jpeg encode/decode*, is run. *jpeg* is a standard lossy compression format that is quite common.

3) Network: The Network benchmark represents a workload typical of embedded processors in network devices, such as switches and routers. The algorithms tested are: (1) finding the shortest path in a large graph of nodes (Dijkstra's algorithm), and (2) traveling a Patricia trie tree.

C. KVM ARM

With ARM's recent push to penetrate the server market, they have made their architecture virtualizable and as such, we have seen the technology propagate down to the embedded space as well as to Linux. The KVM project introduced support for ARM virtualization with the Linux kernel starting from version 3.9.

The ARM hardware extensions differ quite a bit from their x86 counterparts. It is important to understand how KVM ARM uses these extensions in its implementation in order to understand latency sources. We refer our readers to Dall and Nieh's KVM/ARM paper for further details [4].

III. Experimental Setup

A. Hardware

To run our experiments we use two hardware platforms: the Raspberry Pi 2 and Cubieboard 2. Both were chosen, because (1) they are very popular SBCs with a wide support community, and (2) they are representative of the CPS systems that we are interested in. While they have many similarities, the key distinction between them that makes them particularly ideal for testing real-time workloads is that they handle interrupts differently. The Raspberry Pi 2 (RPi) uses a 900MHz quad-core ARM Cortex-A7 CPU in a BCM2836 SoC, which does not provide a GIC (Generic Interrupt Controller). Thus, the RPi does not provide a vGIC (Virtual GIC) interface. On the other hand, the Cubieboard2 uses a Dual-Core ARM Cortex-A7 1GHz CPU in an Allwinner A20 SoC, which provides full hardware virtualization support. This support includes includes a hardware GIC. Because of this difference, Raspberry Pi interrupts under KVM use an emulated GIC mechanism whereas Cubieboard's use the hardware GIC. Real-time workloads are typically interrupt-driven, so the difference in interrupt-handling mechanisms is important; being able to pinpoint any discrepancy in latency due to interrupt-handling can be informative.

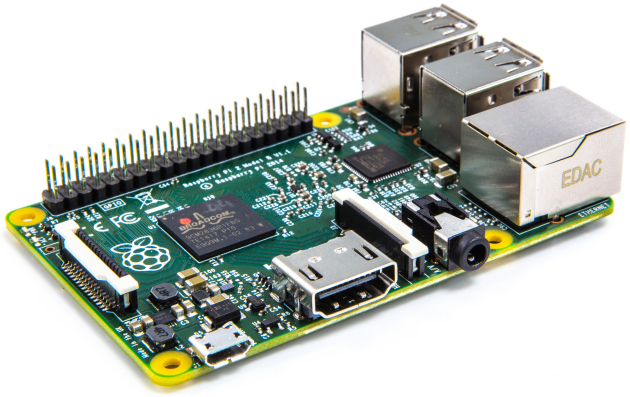

Fig. 1: **Raspberry Pi 2** Specifications for the Raspberry Pi 2: 900MHz quad-core ARM Cortex-A7 CPU–BCM2836 SoC.

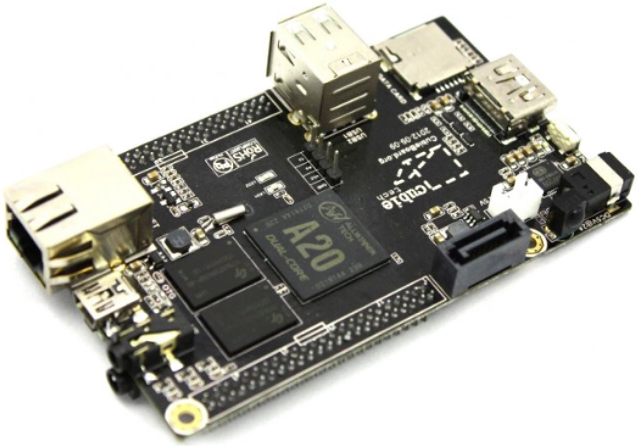

Fig. 2: **Cubieboard 2** Specifications for CB2: Dual-Core ARM Cortex-A7 1GHz CPU in an Allwinner A20 SoC.

B. Choosing a Kernel

The mainline Linux kernel has different levels of scheduler preemptibility. Before settling on the PREEMPT_RT Patch we first evaluated the PREEMPT scheduler available in mainline. The comparison was evaluated using the Raspberry Pi 2 with Linux installed bare-metal. The results of this comparison are shown in Figure 4 and Table I. The maximum latency in the mainline scheduler led us to using the PREEMPT_RT Patch which improves the latency threefold.

Real-time applications have operational deadlines for an application's response to certain events. Due to these constraints, RTOS are typically used, as the vanilla Linux kernel was not designed to take these constraints into account. For our experimental setup we used Linux with the

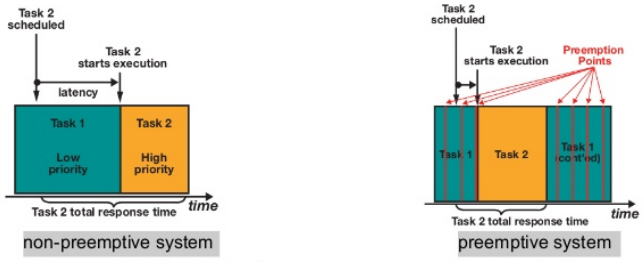

Fig. 3: Linux Kernel Preemption[5]

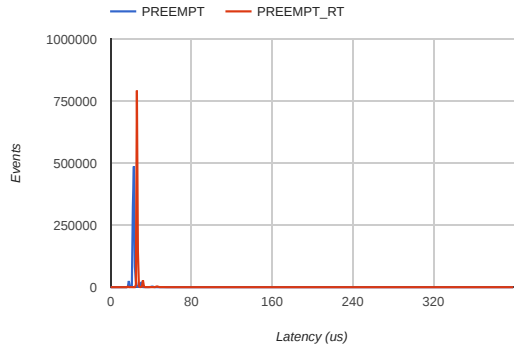

Fig. 4: **Mainline vs RT Patch.** Cyclictest run on both the mainline and the RT patched kernel. Although these curves imply that the latencies between the mainline and patched kernels are similar, closer examination at the actual numbers show that the max latency on patched is much better. See Table I.

PREEMPT_RT Patch. The PREEMPT_RT patch converts Linux into a fully preemptible kernel, allowing for more deterministic scheduling, which limits the overall latency. Figure 3 gives an overview of a non-preemptive kernel vs a preemptive one.

The kernel version used for the Raspberry Pi 2 is v3.18.8-rt5-v7+. This is the Raspberry Pi Foundation's latest kernel with the PREEMPT_RT patch applied. For the Cubieboard 2, we used the mainline kernel v4.4.6-rt11-sunxi with the PREEMPT_RT patch applied. Unfortunately, we were not able to use a higher kernel version for the Raspberry Pi 2 due to lack of support. However, from our measurements the latencies for both devices running bare-metal is consistent, ruling this out as a source of error.

C. Optimizations

1) isolcpus: In order to eliminate multi-core latency we used `isolcpus` to isolate certain cores from kernel scheduling. This isolation effectively minimizes the latency introduced from core to core communications. We used two configurations for the Raspberry Pi 2:

1) 1 core host OS and 1 dedicated to KVM.
2) 1 core host OS and 2 dedicated to KVM.
3) 3 core host OS and 1 dedicated to KVM.

On the Cubieboard 2 we configured 1 core for the host and 1 for KVM. The reason we test on multiple guest-host core configurations is to isolate the effect of the number of cores on guest and host performances.

2) taskset: In order to further reduce latency, we used `taskset` to set the KVM/QEMU processes to the dedicated cores. Finally, we prioritized these processes at the highest possible real-time priority.

TABLE I: Mainline vs RT Patch - Maximum Latency

	RPi 2
Mainline	185
Patch	61

D. Tools

1) Cyclictest: Cyclictest measures the amount of time that passes between a timer expiration and thread execution. It does this by taking two timestamps, one prior to waiting for a specific time interval, the other after the timer finishes. It then compares the theoretical wakeup time with the actual wakeup time which corresponds to the latency for that timer wakeup. Cyclictest has many options for adjusting how measurements are made, which are useful in identifying latencies within the kernel. [6]

2) ftrace: The RT kernel patch contains a subsystem known as `ftrace` that is useful for specifically identifying issues that occur in the kernel. Although primarily designed for function tracing, `ftrace` has been developed as a framework with helpful tracing utilities. Primarily, we use latency tracing to determine what events occur within the kernel scheduler. To better visualize our data, we use KernelShark [7], a GUI frontend for Ftrace.

IV. Results and Discussion

To evaluate our hypothesis of whether real-time workloads are virtualizable, we first measure the latencies natively achievable on the bare-metal host. Then, we compare them to latencies measured on the guest–under idle and on various workloads.

A. Effect of Multiple CPU Cores on Performance

In order to isolate the effect of multiple CPU cores on performance, we ran cyclictest on the RPi2 with the different configurations listed in the `Optimizations` section and compared the latencies for each. We also ran cyclictest on the Cubieboard2 for comparison. These measurements are performed on an Idle kernel with no additional workload. See figure 5 for the Raspberry Pi 2 and its various configurations, figure 6 for the Cubieboard 2, and table II for the corresponding maximum latencies for all four experiments.

We interpret our results to mean that the difference in max latencies is due to a performance difference in QEMU. The latencies for bare-metal on both platforms are actually very similar; the virtualized instances for both are where differences appear. For both platforms (and their different configurations), the maximum latencies are very large (see TableII). We note the difference in latency between the

various configurations of the Raspberry Pi 2. Between configurations 1 (guest cores : host cores = 1:1) and 3 (1:3), the maximum latency is roughly halfed. This implies that assigning more cores to the host actually improves performance in the guest. In addition, between configurations 1 (1:1) and 2 (2:1), the maximum latency is slightly decreased. Thus, assigning more cores to the guest also improves performance. A likely reason for these improvements is that assigning more cores to the host or the guest is equivalent to providing more resources for QEMU.

B. Effect of Virtual vs. Hardware GIC on Performance

We also note the difference in latencies between configuration 1 (guest cores : host cores = 1:1) of the Raspberry Pi 2 and Cubieboard 2. These differences result from hardware variations between the two platforms. As we mentioned earlier, the Raspberry Pi 2's BCM2835 has no VGIC implemented. Instead a simple register-based interface is used to route the global top-level interrupt to core 0; therefore, core 0 receives and services all interrupts. Because the Cubieboard 2 Guest cores service their own interrupts, this might explain why the CB2 has a higher latency than RPi2, config. 1; despite having a hardware GIC, the CB2 has higher latency, because the guest's interrupts are serviced by the guest's core.

C. Benchmarks: Mibench Tests

For the MiBench benchmarks we focus our attention to the Cubieboard 2 platform, and use this as our baseline since it has full hardware virtualization support. Figures 8, 9, 10 show the latency under various of the MiBench test suites. For all the tests in the benchmark, we point out that the curves reperesenting the host are narrow, pronounced spikes whereas the curves representing the guest are wide, less-definitive bumps; we call the width of these curves jitter. The lower the jitter means that events have a clear, predictable latency when run on the host, but their latencies have wider variance–and are thus less predictable–on the guest. In fact, some of the guest curves have two local maxima, increasing the unpredictability. In addition, the guest curves are always shifted rightwards of the host curves; running on the guest takes more time. These graphs tell us that running real-time tasks on a virtualized operating system may not be immediately feasible, because latencies are both longer and less predictable than the host.

D. rusEFI Workload

Figure 7 shows the results when running the rusEFI workload. While the averages remain the same, the histogram curves are wider. The results here are better than those from MiBench, but we still see increased jitter.

E. Maximum Latency

In order to determine other potential sources of latency we began looking at potential bottlenecks, such as CPU and File I/O. First, we tested for CPU-bound latencies by running `sysbench`'s primes computation (max-prime=2000) on both the bare-metal and virtual instances. We found only a negligible difference in the metrics reported by `sysbench`; for the average case, our `cyclictest` latency graphs looked similar to those from MiBench. The maximum latency was only slightly better. We then set out to test File I/O using `sysbench` and, again, noticed similar average case latencies. However, the maximum latencies for sysbench were much higher (approximately 16000). This suggests that File I/O is a potential large contributor to the latency.

TABLE II: Maximum Latencies

Shows the maximum latencies for an idle workload corresponding the Raspberry Pi 2 (all three of its configurations) and the Cubieboard 2. Data obtained by running cyclictest with 1,000,000 loops for all four.

RPi 2 (1)	RPi 2 (2)	RPi 2 (3)	CB 2
7214	5277	3097	9895

The fact that the average latency is consistently larger by the same factor suggests that virtualization may impose a constant overhead with respect to the kernel's preemption quanta. This is potentially be due to the virtualization of timers, which the kernel uses to schedule tasks. We can also conclude that the core configurations play a major factor in the performance of the virtualized instance; a host with more cores ends up providing better latency for the guest. From our analysis, we conclude that worst-case timer driven guest scheduling latencies below 1 ms and average latencies below 160 μs can be achieved with careful optimizations. For better performance, a lighter weight virtualized OS or lighter weight hypervisor should be used.

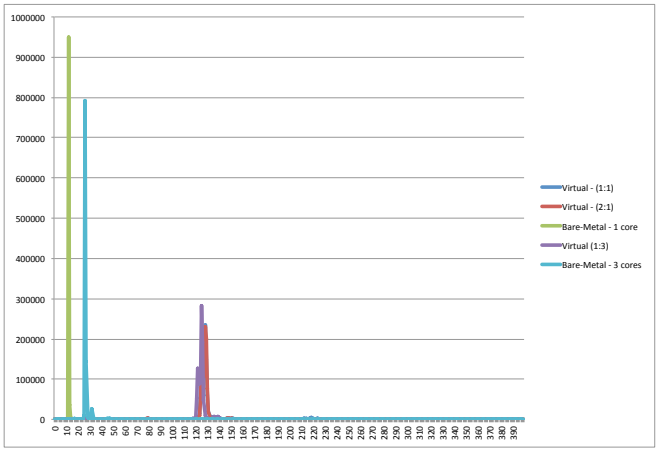

Fig. 5: **Raspberry Pi 2 - Idle Latency.** Shows cyclictest running 1,000,000 loops on the virtualized OS for all three configurations of guest to host cores ratios. For comparison, we included cyclictest running the same test on the host as well (one and three cores). Experiments ran under idle workload.

V. Related Work

Previous work, such as that of Jan Kiszka [8] on x86, showed that worst case timer-driven guest latencies below

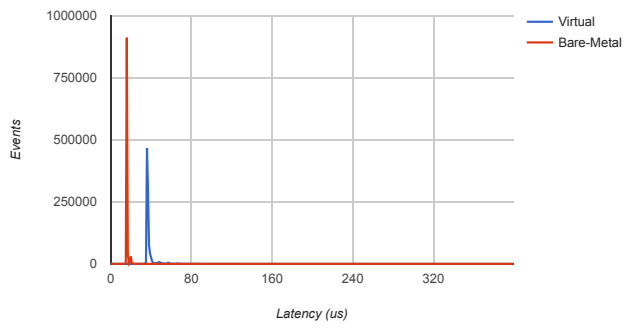

Fig. 6: **Cubieboard 2 - Idle Latency.** Points obtained from cyclictest running on the guest and host operating systems. Experiments ran under idle workload.

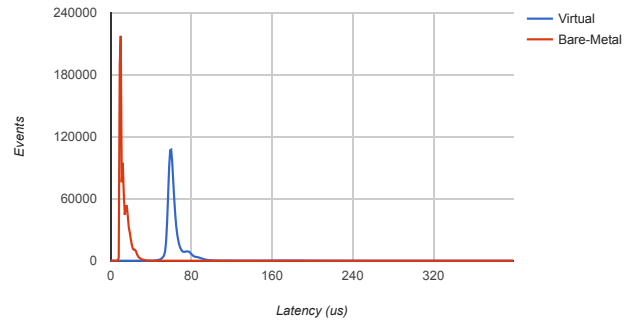

Fig. 7: **Cubieboard 2 - rusEFI Latency.** Data points obtained from cyclictest running in the backaground on the guest and host operating systems while the rusEFI workload ran in the foreground.

1ms and average latencies below 100 μs can be achieved on a PREEMPT_RT kernel. After our semester's work on ARM, we conclude similar findings. Depending on how stringent a real-time application's deadline requirements are, the application may feasibly run on a virtualization platform. However, as our data suggests, requiring hard-realtime accuracy in the μs range imposes too large of an overhead.

Since his original findings, Kiszka has since continued with this line of research. His most recent work is a talk in collaboration with Rik van Riel at the KVM Forum 2015. [9], [10] Their work focuses on the feasibility of Real-time on Linux running on powerful x86 server processors. They demonstrate that real-time virtualization can be done with certain limitations and specific sets of optimizations. One particular optimization that they suggest, which we did not try, was partitioning the virtualized guest. Similar to how `isolcpus` and `taskset` were used to isolate cores between the host and the guest, partitioning the guest in a similar manner may also further isolate cores. By doing this, Kiszka and Rik van Riel were able to cut down latency significantly.

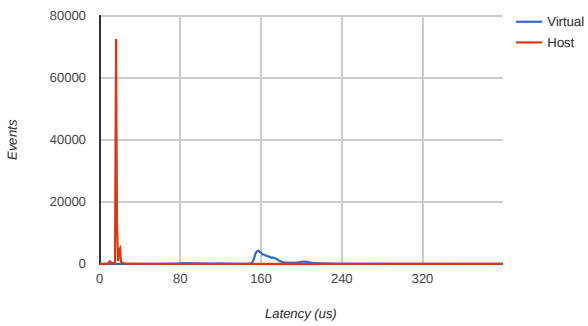

Fig. 8: **Cubieboard 2 - MiBench:BasicMath Latency.** Data obtained from cyclictest running in the background for guest and host as Mibench's basicmath test ran in the foreground. basicmath is a CPU bound workload that runs an extremely long list of basic arithmetic computations that do not have special hardware support.

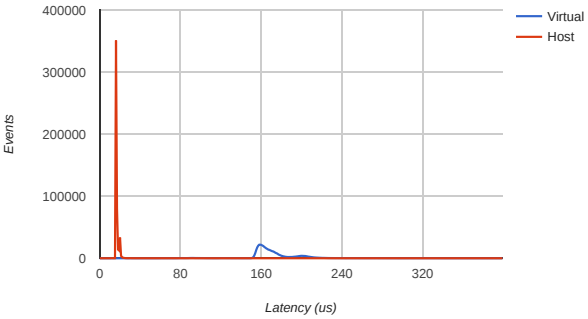

Fig. 9: **Cubieboard 2 - MiBench:Bitcnts Latency.** Data obtained from cyclictest running in the background for guest and host, Mibench's bitcount test ran in the foreground. bitcount tests the ability of the CPU to manipulate bits.

Gu and Zhao [11] provide a survey of embedded system virtualization. It demonstrates that people have had success in running real-time workloads under virtualization. Unfortunately, most related work is proprietary, or does not target ARM processors, as such, this makes it very difficult to replicate and evaluate. With our project we seek to make our work transparent to others by using open-source technologies.

VI. Future Work

A significant amount of effort was devoted to porting ChibiOS to the POSIX platform on x86 and ARM. As a result, we would like to contribute our code back to the community and work on getting it accepted to mainline. We would also like to expand rusEFI and have it follow in the same vein as the JPapabench project [12], which serves as a benchmark utility borrowing from the code and algorithms of Paparazzi UAV project [13]. The lack of openly available

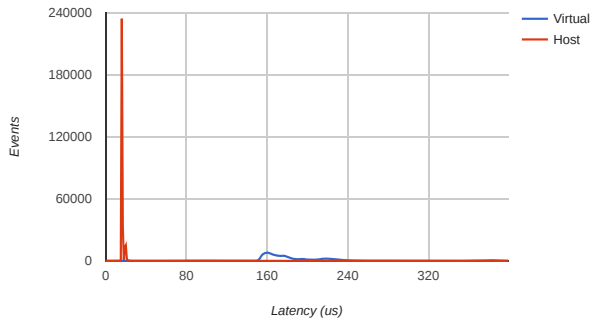

Fig. 10: **Cubieboard 2 - MiBench:QSort Latency.** Data obtained from cyclictest running the background, Mibench's qsort test ran in the foreground. qsort sorts a large array of strings in ascending order using the Quicksort algorithm.

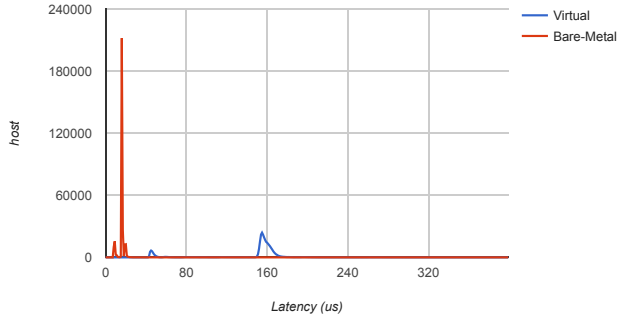

Fig. 11: **Cubieboard 2 - Sysbench CPU Latency.** Data points obtained by running cyclictest for 1,000,000 loops in the background as sysbench's prime numbers computation ran in the foreground. The maximum prime for sysbench was set to 2000.

real-time workloads that can be used for academic research is troublesome, and we would like to contribute to solving this problem by making our work publicly available. Finally, we would like to implement Jan and Kizka's partitioning as another test and measure its effect on performance.

REFERENCES

[1] rusefi wiki. [Online]. Available: http://rusefi.com/wiki/

[2] Chibios homepage. [Online]. Available: http://www.chibios.org/dokuwiki/doku.php

[3] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001, pp. 3–14.

[4] C. Dall and J. Nieh, "Kvm/arm: the design and implementation of the linux arm hypervisor," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 333–348, 2014.

[5] J. Huang. Making linux do hard real-time. [Online]. Available: http://www.slideshare.net/jserv/realtime-linux

[6] F. Rowand. Using cyclictest. [Online]. Available: http://events.linuxfoundation.org/sites/events/files/slides/cyclictest.pdf

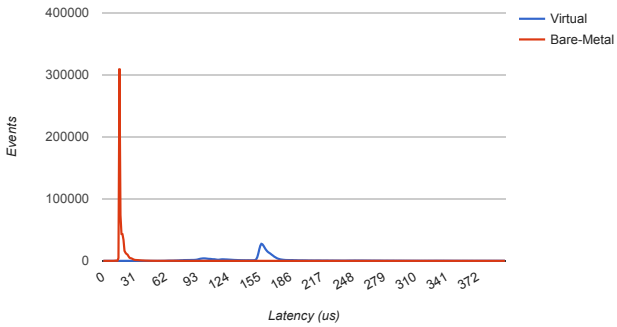

Fig. 12: **Cubieboard 2 - Sysbench File I/O Latency.**

TABLE III: **CB2: Sysbench - CPU**. Data points obtained by running cyclictest for 1,000,000 loops in the background as sysbench's file-io test ran in the foreground. The memory used for the test was 500M across the the guest and the host.

	Bare-Metal	**Virtualized**
Total Time (s)	12.0141	12.4151
Min (ms)	1.18	1.18
Avg (ms)	1.20	1.24
Max (ms)	10.89	22.94
95th Percentile (ms)	1.22	1.38

[7] S. Rostedt. Using kernelshark to analyze the real-time scheduler. [Online]. Available: https://lwn.net/articles/425583/

[8] J. Kiszka, "Towards linux as a real-time hypervisor," in *Eleventh Real-Time Linux Workshop*. Citeseer, p. 205.

[9] R. van Riel. Real-time kvm from the ground up. [Online]. Available: http://www.linux-kvm.org/images/2/24/01x02-rik_van_riel-kvm_realtime.pdf

[10] J. Kiszka. Real-time kvm for the masses. [Online]. Available: http://www.linux-kvm.org/images/0/0d/01x03-jan_kiszska-kvm_rt_for_masses.pdf

[11] Z. Gu and Q. Zhao, "A state-of-the-art survey on real-time issues in embedded systems virtualization," 2012.

[12] M. Malohlava. jpapabench. [Online]. Available: http://d3s.mff.cuni.cz/ malohlava/projects/jpapabench/

[13] Paparazzi uav wiki. [Online]. Available: https://wiki.paparazziuav.org/

[14] Virtualization extensions. [Online]. Available: https://www.arm.com/products/processors/technologies/virtualization-extensions.php

TABLE IV: **CB2: Sysbench - File I/O**.

	Bare-Metal	**Virtualized**
Total Events	69700	36100
Min (ms)	0.04	0.05
Avg (ms)	0.19	0.80
Max (ms)	58.32	17.25
95th Percentile (ms)	0.28	1.77
Read (MB/s)	3.63	1.88