sustainability

MDPI

Article

Design and Analysis of Multiple OS Implementation on a Single ARM-Based Embedded Platform

Byoungwook Kim [1] and Min Choi [2,*]

[1] Creative Informatics & Computing Institute, Korea University, 145 Anam-ro, Seongbuk-gu, Seoul 02841, Korea; byoungwook.kim@inc.korea.ac.kr

[2] Department of Information and Communication Engineering, Chungbuk National University, 1 Chungdae-ro, Seowon-gu, Cheongju, Chungbuk 28644, Korea

* Correspondence: mchoi@cbnu.ac.kr; Tel.: +82-10-9969-5699

Academic Editor: James J. Park

Received: 5 April 2017; Accepted: 15 April 2017; Published: 25 April 2017

Abstract: Recently, with the development of embedded system hardware technology, there is a need to support various kinds of operating system (OS) operation in embedded systems. In mobile processors, ARM started to provide the virtualization extension support technology which was intended for processors in PC processors. Virtualization technology has the advantage of using hardware resources effectively. If the real-time operating system (RTOS) is operated on a hypervisor, there is a problem that RTOS performance is degraded due to overhead. Thus, we need to compare the performance between a single execution of the RTOS and simultaneous execution of multiple OS (RTOS + Linux). Therefore, in this paper, we measure the performance when the RTOS operates independently on the NVidia Jetson TK-1 embedded board supporting virtualization technology. Then, we measure the performance when the RTOS and Linux are operating simultaneously on top of a hypervisor. For this purpose, we implemented and ported such a RTOS, especially FreeRTOS and uC/OS, onto two embedded boards, such as the Arndale board (SAMSUNG, Seoul, South Korea) and the NVidia TK1 board (NVIDIA, Santa Clara, CA, USA).

Keywords: embedded system; real-time operating system; performance evaluation; multiple operating system

1. Introduction

Recently, embedded systems can run multiple operating systems through virtualization technology. Through this hypervisor, we implemented an embedded system that simultaneously runs RTOS and Linux on a single platform. The RTOS performs hardware control tasks requiring a real-time property. At the same time, Linux runs a variety of user programs. Therefore, a highly-reliable high-performance embedded system can be realized. Multiple operating systems on a single platform using a hypervisor provides isolation between the operating systems, but the requirement for an embedded hypervisor are distinct from hypervisors targeting servers and applications [1–3]. While desktop and enterprise environments use hypervisors to consolidate hardware and isolate computing environments from one another, in an embedded system, the various components typically function collectively to provide the device's functionality [4,5]. An implementation of an embedded hypervisor must deal with a number of issues, which include the highly-integrated nature of embedded systems, the requirement for isolated functional blocks within the system to communicate rapidly, the need for real-time/deterministic performance, the resource-constrained target environment, and the wide range of security and reliability requirements [6]. When a hypervisor does not provide a real-time property, the context switching overhead causes a delay, resulting in performance degradation in the guest operating system. This paper compares and analyzes the performance between Linux and a RTOS

on a single embedded platform. In order to conduct the experiment, we make use of the Rhealstone benchmark, which was developed by Boger et al.

The rest of this paper is organized as follows: Section 2 describes background. Sections 3 and 4 provide system design and experimental results. Finally, we conclude and summarize our work in Section 5.

2. Materials and Methods

A hypervisor is a middleware that can run multiple guest operating systems on a single type of hardware. The hypervisor monitors and manages the resources used by the guest OS [7,8]. In this environment, each OS is isolated as if it were running alone. The ARM Cortex-A15 processor's Virtualization Extension features include:

(1) Introduction of Privileged Level HYP Mode

In the ARM Virtual Extension, a hypervisor can have higher privileges than the guest OS through the Hyp mode. It is not necessary for the guest OS to recognize the existence of the privileged mode, Hyp mode, which is newly introduced in ARM Cortex-15. The guest OS can still operate in supervisor mode and user mode.

On the ARMv7 Cortex-A15 processor, the OS kernel uses supervisor mode. On the ARMv7 Cortex-A15 processor, the user application uses the user mode. In addition, the virtualization hypervisor uses Hyp mode. The hypervisor mode has access to special registers that the user mode or supervisor cannot access. For example, the hypervisor mode has its own Stack Pointer (SP), Saved Processor Status Register (SPSR), and Exception Link Register (ELR). Additionally, registers, such as physical counters, are accessible only in hypervisor mode.

(2) Interrupt Virtualization: GICv2

Previously, if an interrupt occurred during the operation of the guest OS, the virtual machine would have to generate a trap to switch to hypervisor mode first and then process the interrupt. On the other hand, the ARM Cortex-A15 processor introduces the concept of virtual interrupt to eliminate this inconvenience as shown in Figure 1. A virtual interrupt allows the virtual machine to receive an interrupt signal directly or to clear an interrupt that has occurred [9,10]. The virtual machine can process its own interrupt at the guest OS level without the hypervisor's intervention (however, hardware and device drivers must support the virtual CPU interface for this purpose). In the ARM Cortex-A15, the generic interrupt controller can collect/analyze and arbitrate a large number of interrupt sources: (1) interrupt masking; (2) interrupt priority; (3) interrupt distribution for target processor; (4) interrupt status tracking; and (5) interrupt generation for software.

In the ARM Cortex-A15, the interrupt can be (1) passed to the currently-running guest OS; (2) passed to the other guest OS; (3) passed to the hypervisor; or (4) passed to the OS or RTOS which runs in the secure TrustZone. In the ARM Cortex-A15, the hypervisor takes priority in physically interrupting. However, if the interrupt is an interrupt to be passed to the guest OS, the hypervisor provides the ability to map the virtual interrupt to the guest OS. In addition, a hypervisor can create a virtual interrupt even if no physical interrupt has occurred [11,12].

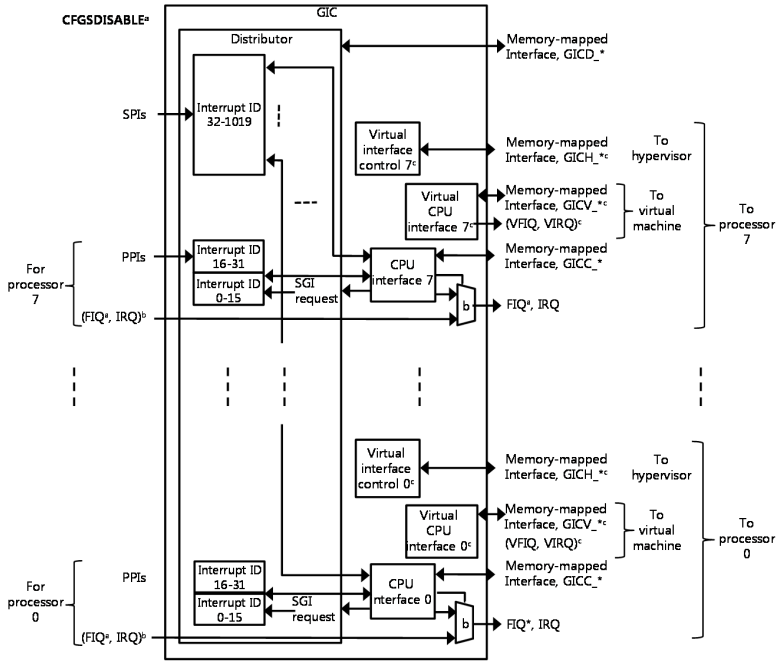


Figure 1. Generic interrupt controller architecture.

(3) Timer Virtualization Support: GTimer

System counter: Gtimer is a shareable "always on" counter. Gtimer has fast read access capability for reliable time distribution. It has a 64-bit count value and counts from a value of zero at initialization. The operating system reads the CNTFRQ register, which is the system control register at system boot time. The operating system initializes the system counter frequency through the CNTFRQ register and uses the system counter.

Virtual counter: Virtual counters allow virtualization to measure time on each virtual machine during operation. All processors with generic timers always provide virtual counters, even if virtualization extension technology is not applied (at this time, the virtual counter represents virtual time). If the virtualization extension technology is not applied, the virtual time is the same as the physical time. The virtual counter has the same value as the physical counter. If virtualization extension technology is applied, the virtual counter has a value obtained by subtracting the virtual offset of 64 bits from the physical counter. The CNTVOFF register is an RW register. It is accessible in the non-secure hypervisor mode [13]. If the SCR.NS value is set to 1, it is accessible even in the secure monitor mode. The current virtual counter value is stored in the CNTVCT register as 64 bits. If one wants to access this register, they need special privileges to access the register [14,15] (you can access this register on secure PL1 mode, non-secure PL1 mode, and non-secure PL2 mode).

Timer function that raises an event after a certain period of time: The generic timer provides a timer function, which varies slightly depending on the processor manufacturer. Timers provide the ability to send signals to the system. If the processor is connected to the GIC, the signal is transferred through the PPI. When virtualization extension technology (VE) is applied, the processor provides one non-secure PL1 physical timer, one secure PL1 physical timer, one non-secure secure PL2 physical timer, and one virtual timer. Each timer provides three registers: a 64 bit compare value register for a 64-bit unsigned up-counter, a 32-bit timer value register for a 32-bit signed down-counter, and a 32-bit control register, which is available as both an up-counter and down-counter.

The RTOS kernel requires some modifications to support ARM Cortex-A15 virtualization on the hypervisor. To this end, the system structure and source tree of uC/OS-II and FreeRTOS are briefly shown.

Figure 2 is a conceptual diagram that changes from the traditional structure to the one in which the hypervisor is used. The ARMv7 Cortex-A15 processor is implemented by virtualization extensions (VE), a hardware-supported virtualization to support virtualization technology. It also supports interrupt virtualization technology (GICv2). The GIC is a technology that reduces the overhead by handling the interrupts directly by the guest OS without intervention of the hypervisor when the interrupt occurs. Therefore, the guest OS needs to operate in the supervisor mode and the user mode in the same manner as when it is executed by itself. There are various RTOSs in the market. However, in this study, uC/OS-II and FreeRTOS are used, for which the source code is free [16].

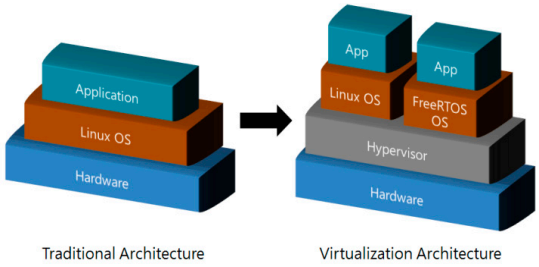
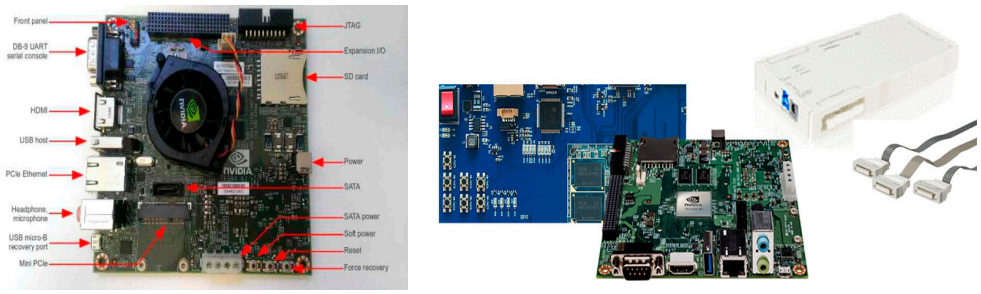


Figure 2. System architecture.

3. Design and Implementation

In this study, we used NVidia's Jetson TK-1 development board with an ARMv7 Cortex-A15 processor, as shown in Figure 3. We installed the QPlus Hypervisor as a hypervisor in Jetson TK1. Linux (Ubuntu 14.04) and FreeRTOS were used as guest OS. We make use of the embedded hardware debugger Lauterbach Trace 32, which is a JTAG debugger. The debugger enables us to debug line by line on the embedded system, dump a segment of memory, view the content and value of a certain register, and so on.



(**a**) (**b**)

Figure 3. Working platform and debugger. Note: (**a**) TK1 board view from the top; (**b**) TK1 and Arndale board with embedded debugger, the Trace32.

We evaluated the situation of FreeRTOS operating alone and with Linux and FreeRTOS operating together with QPlus Hypervisor. In this study, we measured the task transition delay time and interrupt latency of FreeRTOS. When running on the Qplus hypervisor, Linux was executed with a background process and no other running processes.

Figure 4 shows how the uC/OS porting implementation onto the ARM-A15 evaluation board, especially Arndale and TK1, progress. The OS kernel starts with startup.S, the ARM assembly start-up code. Then it calls the main and OSInit functions in the left top of the Figure 4. Then, it initializes and sets several structures and registers related to generic timer, general interrupt controller, UART, memory, task control blocks, and so on.

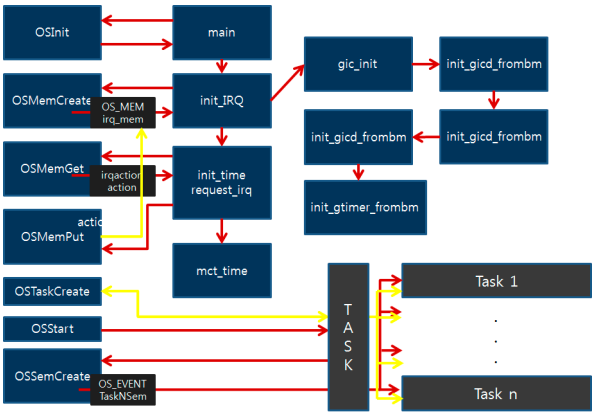


Figure 4. uC/OS kernel architecture.

We make use of the co-processor interface, CP15, to configure the generic timer. The generic timer uses virtual timer control, interrupt source PPI4, IRQ number 27. We set the IRQ number to 30 for the physical timer. The following register settings in Figure 5 are required for the GTimer timer interrupt to be recognized as a GIC register. First, we set the ICENABLER and ISENABLER registers for GICD's interrupt clear and set. Then, we set the ITARGETSR register to set the CPU to recognize the interrupt. In addition, we initialize the timer by setting the bits of the offset corresponding to IRQ 27 in the IPRIORITYR register to determine the priority of the interrupt [17–19].

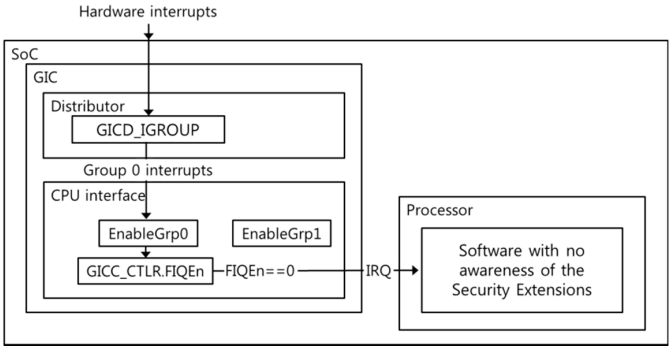


Figure 5. Interrupt processing on hardware initialization.

On power-up, or after a reset, a GIC implementation that supports interrupt grouping is configured with all interrupts assigned to Group 0, and the FIQ exception request disabled. This means that Group 0 interrupts are signaled using the IRQ interrupt request. Figure 6 shows this configuration.

CNTKCTL	c14	0	c1	0	-[a]	32-bit	Timer PL1 Control Register, see the *ARM Architecture Reference Manual*
CNTP_TVAL			c2	0	UNK	32-bit	PL1 Physical TimerValue Register, see the *ARM Architecture Reference Manual*
CNTP_CTL				1	-[b]	32-bit	PL1 Physical Timer Control Register, see the *ARM Architecture Reference Manual*
CNTV_TVAL			c3	0	UNK	32-bit	Virtual TimerValue Register, see the *ARM Architecture Reference Manual*
CNTV_CTL				1	b	32-bit	Virtual Timer Control Register, see the *ARM Architecture Reference Manual*

Figure 6. Co-processor control register for the use of generic timer.

When the timer is initialized, the timer periodically generates an interrupt. At this time, uC/OS-II checks which interrupt is generated. The OS can use the information in the GIC register to determine the IRQ number. The OS executes the interrupt service routine according to the IRQ number. In the event of a timer interrupt, the Timer Tick function is called. This function calls the uC/OS-II OS kernel function OSTimeTick().

The GIC maintains a state machine for each supported interrupt on each CPU interface. Figure 7 shows an instance of this state machine and the possible state transitions.

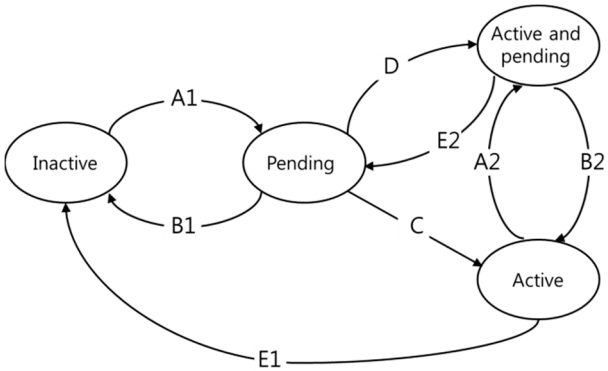


Figure 7. State transition diagram for interrupt processing.

The study builds on the ARM A15 embedded board to run two or more operating systems simultaneously through the hypervisor. In particular, we made Ubuntu Linux and RTOS (uC/OS and FreeRTOS) run simultaneously on the embedded board. Due to the nature of the hypervisor, two operating systems share the same UART console. Therefore, as shown in Figure 8, the RTOS outputs a 'hello arndale' message and a 'Goodbye arndale' message to the screen. At the same time, on Ubuntu Linux, the root user's Linux prompt is displayed on the screen.

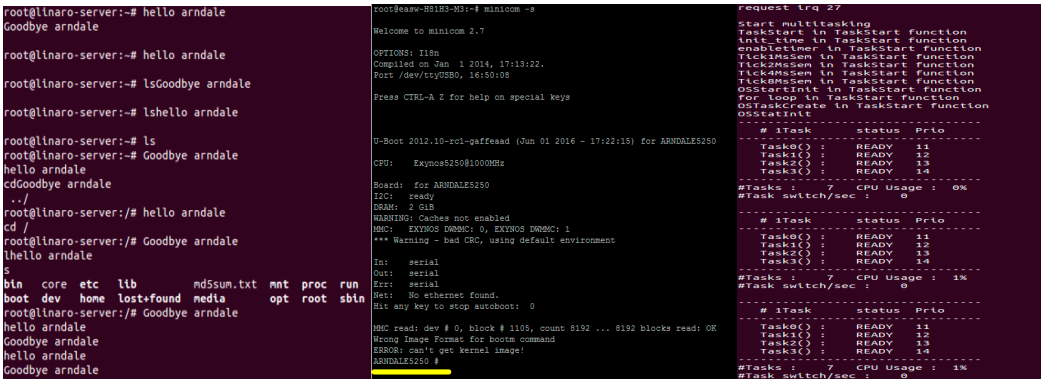


Figure 8. Simultaneous execution of multiple OS on embedded system.

This study was performed with a Tektronix oscilloscope using the GPIO ports of the Jetson TK-1 development board to verify the operating characteristics of the hypervisor and guest OS. We also performed some of the Rhealstone benchmark sets [8] for performance comparisons. The Rhealstone benchmark was proposed in 1989. It does not measure the quality of the complete solution, but is, instead, a measurement targeted toward a multitasking solution. In the task transition delay time measurement, we repeatedly output HIGH-LOW to each GPIO port for two tasks. Each task outputs high to its GPIO port when it is running. Therefore, after task switching, the corresponding GPIO port remains low. Task 1 and Task 2 use their own GPIO ports. Therefore, Figure 9 shows the status change of each GPIO port simultaneously through the two-channel input to the oscilloscope.

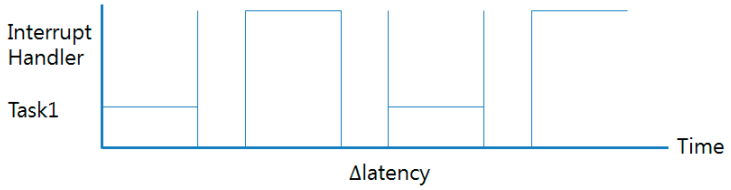


Figure 9. Task switching time.

Task switching time is the average time the system takes to switch between two independent and active, not suspended or sleeping, tasks of equal priority. Task switching is synchronous and non-preemptive and is an important measure of any multitasking system. This metric is influenced by the host CPU's architecture, instruction set, and features, and is designed to assess the compactness of task control data structures and the efficiency with which the executive manipulates the data structures in saving and restoring contexts. Task switching time, additionally, measures the executive's list management capabilities. Figure 10 shows the related data structures maintained by the OS kernel during the task switching.

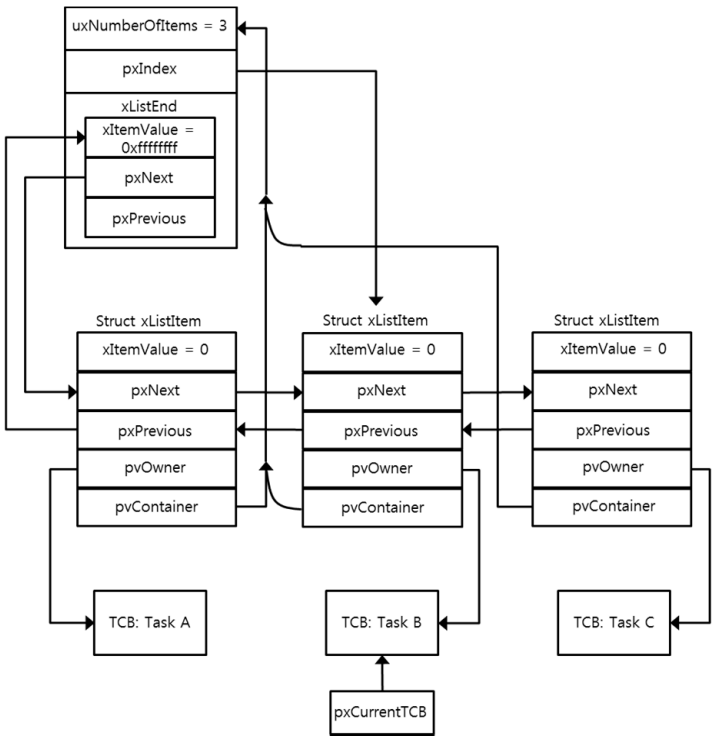


Figure 10. Related data structures for switching tasks.

The task-Switching benchmarking sets up two tasks with equal priority. The tasks switch back and forth between the processor. To first determine the time it takes to perform the for loop "work", the benchmark just measures the loops performing no work and not task switching.

As shown in Figure 11, we measured the interrupt latency in the following way. Two tasks output HIGH (Task 1) and LOW (Task 2) status for one GPIO port, respectively. We start the measurement when an interrupt occurs. It outputs a signal to a specific GPIO when the execution flow enters the interrupt handler. Thus, we can check whether an interrupt handler is from Task 1 or Task 2 by the waveform of the oscilloscope.

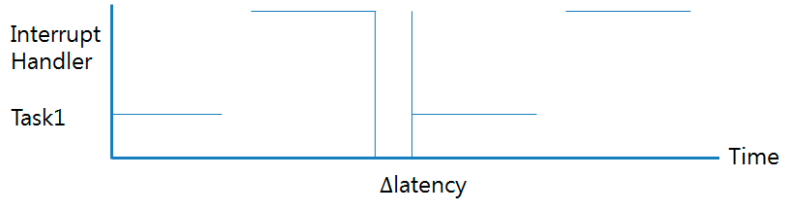


Figure 11. Interrupt processing delay.

4. Experimental Results

In this study, we implemented two real-time operating systems that operate through an ARM A-15 processor with virtualization support: uC/OS and FreeRTOS. We run two ported RTOSs with each Linux operating system. These RTOSs generate a digital clock in the form of a wave every second. We use the generated clock signal to output to the digital timer board, as shown in Figure 12.

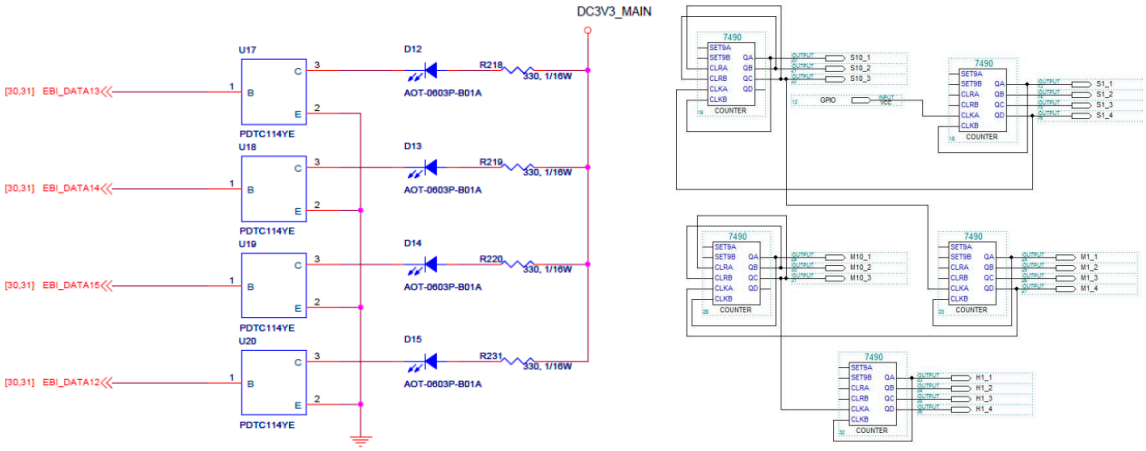


Figure 12. Digital timer board schematic.

Figure 13 represents the waveform simulation results generated by SPICE tools using the above digital timer board schematic of Figure 12. The digital timer board was implemented using digital ICs (7447, 7490, etc.). We connected the digital clock output from the evaluation board to two timer boards as shown in Figure 14.

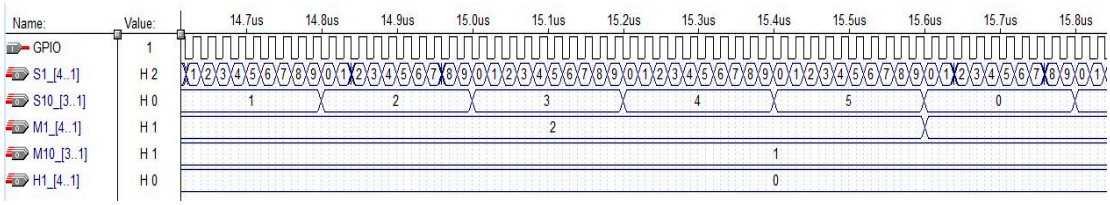


Figure 13. Digital timer circuit simulation.

Figure 14 shows the environmental setup for this experiment. We make use of an oscilloscope to measure the digital high/low states when multiple OS and hypervisors are working on the evaluation board.

Figure 14. Digital timer circuit simulation environment.

The server provides the operating system image in Tftp on the evaluation board. We communicate via the Minicom program with the evaluation board and the UART. The evaluation board provides a clock with a digital timer board. At this time, we measure the clock timing with an oscilloscope to measure the clock waveform, as shown in Figure 15.

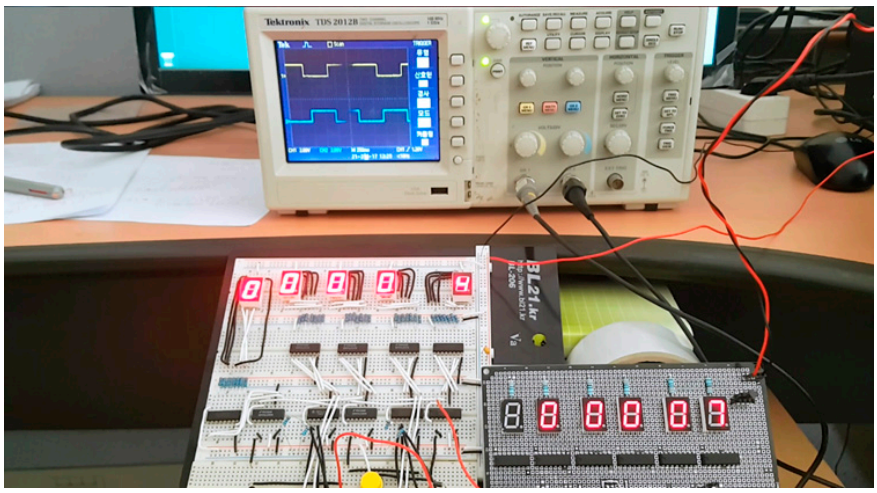


Figure 15. Digital timer circuit simulation.

In Figure 15, the left digital timer board and the right digital timer board may look different. However, both boards are only show a difference between implementing the same circuit on a breadboard and implementing it on a universal board. The left board in Figure 15 receives the clock from the Linux operating system. The right board in Figure 15 receives the clock from the RTOS operating system.

Especially, we obtained the following four results from this experiment. Task switching latency measurement results are as follows: Figures 16 and 17 show the first experimental results, which are captured from the oscilloscope. Of the two waveforms shown in the figures, the upper waveform corresponds to Task 1, and the lower waveform corresponds to Task 2. We set the time axis to 5.00 µs when FreeRTOS operates alone. Since one scale on the oscilloscope is 1.00 µs, the task transition delay time was measured to be approximately 3.50 µs on average.

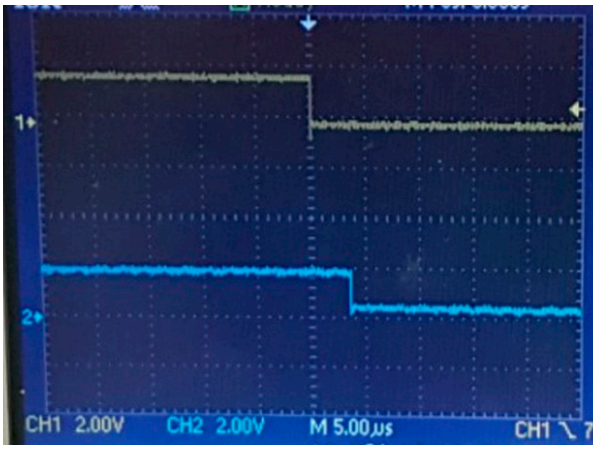


Figure 16. Task switching time on single RTOS execution.

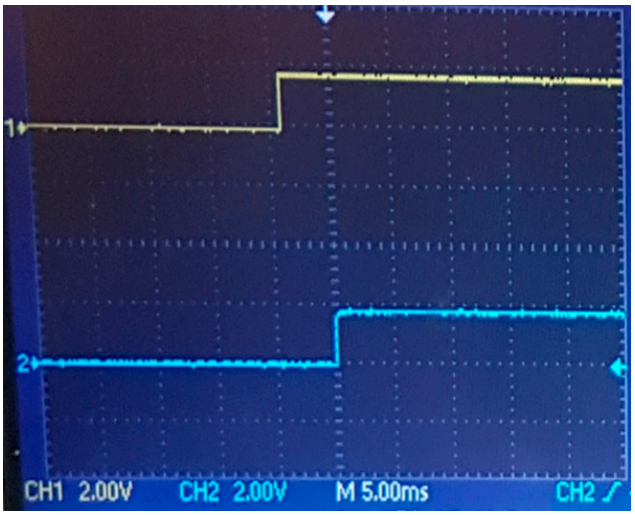


Figure 17. Task switching time on multiple OS (RTOS + Linux) execution.

When Linux and FreeRTOS work together through QPlus Hypervisor, the result is as follows. The time axis was set to 5.00 ms. Since one scale on an oscilloscope is 1.00 ms, the task transition delay time was measured to be about 4.00 ms on average. There are three processes, which have all the same priorities. However, the process "T" starts with the highest priority at initialization time. After initialization, the process will reduce the priority by adding 1 to the tskIDLE_PRIORITY value of itself in order to yield the CPU to the process "F" and process "S". Figure 18 shows how we controlled the GPIO ports in the FreeRTOS application tasks, which are used to evaluate the task switching time and interrupt delay.

```
for( ;; )
{
        XGpioPs_WritePin(&emio_pmod2,                    EMIO_54,
        0x1);vTaskPrioritySet(xHandleThird, tskIDLE_PRIORITY + 1);
        XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0);
}
```

Figure 18. GPIO control code example.

The taskYIELD()function yields the CPU to other processes, which is in the ready queue. This code does not actually create tasks, but simply determines the execution time of the portions of the code that are not part of the task-switching measurement. The execution of this code segment is recorded with the oscilloscope and the second portion of the code runs. The second portion utilizes two tasks. This time the two tasks perform task switching for the desired number of iterations.

The result of measuring the interrupt latency is as follows: Among the two waveforms shown in the figure, the upper waveforms are alternating between Task 1 and Task 2 and output HIGH and LOW states. The waveform below, in Figures 19 and 20, are generated from the waveform for the interrupt handler. We calculated the time from when the bottom waveform changes from HIGH to LOW, and when the top waveform changes from LOW to HIGH (or HIGH to LOW). When operating using FreeRTOS alone, we set the time axis to 2.50 µs. Since the interval is 0.50 µs, the interrupt latency is measured to be approximately 3.00 µs on average.

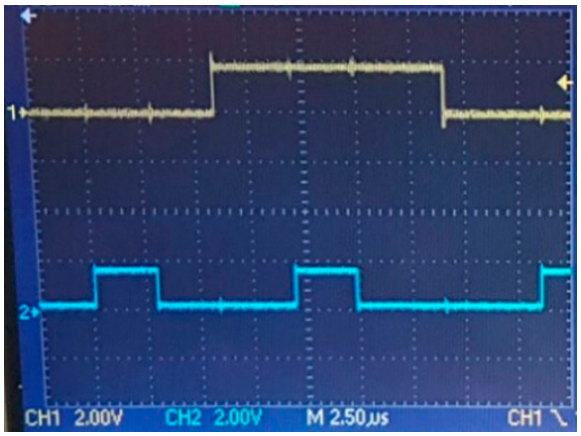


Figure 19. Interrupt latency on a single RTOS execution.

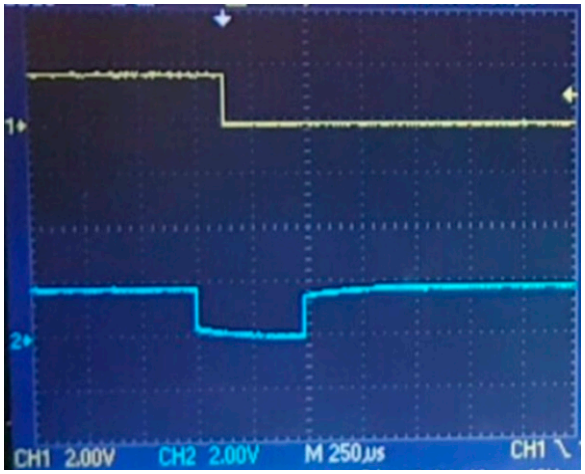


Figure 20. Interrupt latency on multiple OS (RTOS + Linux) execution.

When Linux and FreeRTOS work together through the QPlus Hypervisor, we set the time axis of the oscilloscope to 250 µs. Since the scale on an oscilloscope is 50 µs, the interrupt latency was measured to be approximately 150 µs on average. Based on the measured results, FreeRTOS operating with Linux through the QPlus Hypervisor showed that the task switching delay time is about 1140 times and the interrupt latency is about 50 times slower than when it is performed alone.

Based on the experimental results, we can deduce the following. There is no significant difference in the task transition delay time and interrupt latency time when FreeRTOS operates alone. However,

the FreeRTOS operating on the QPlus Hypervisor shows a large difference in task transition delay time. Additionally, the interrupt latency is slowed down.

Figure 21 show our experimental results in which there are two digital timer boards and they are supplied a clock signal from two different digital timer boards, respectively. They start counting the digital timers at the same time, but the timings are skewed occasionally due to the operating system delays [19].

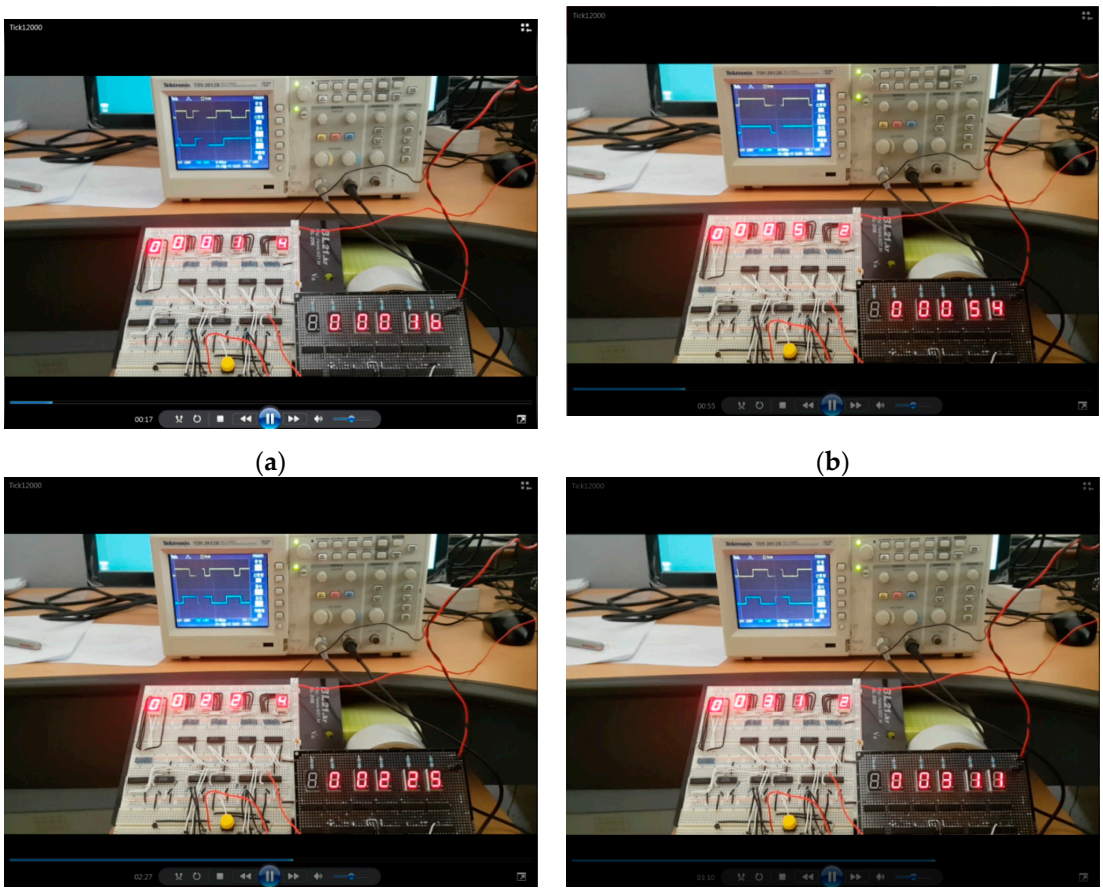


Figure 21. Digital timer board execution by clock generated from RTOS and Linux, respectively. Note: (**a**) at time 00:17; (**b**) at time 00:55; (**c**) at time 02:27; (**d**) at time 03:10.

Importantly, the Cortex-A15 processor supports interrupt virtualization (GICv2). This reduces the hypervisor's interrupt processing overhead by virtualizing interrupts that occur in each guest OS. Due to these advantages, it can be seen that the interrupt latency is measured more quickly in the guest operating system and FreeRTOS operation than the hypervisor shown above. The following problems exist when guest OSs run through hypervisor that do not guarantee a real-time property. Due to the scheduling of the hypervisor, the RTOS does not match the time actually operated and the time on the system. Therefore, the system causes a slight performance degradation, as shown in Table 1.

Table 1. Experimental results.

	Single RTOS(FreeRTOS) Execution	**Linux and FreeRTOS Simultaneously**
Task switching delay	3.5 µs	4.0 ms
Interrupt delay	3.0 µs	150 µs

5. Conclusions

There is a problem that RTOS performance is degraded due to overhead. Thus, we need to compare the performance between a single execution of the RTOS and a simultaneous execution of multiple OSs (RTOS + Linux). Therefore, in this paper, we measured the performance when the RTOS operates independently on the NVidia Jetson TK-1 embedded board supporting virtualization technology. Then, we measured the performance when RTOS and Linux are operated simultaneously on top of a hypervisor.

To this end, we implemented and ported such a RTOS, especially FreeRTOS and uC/OS, onto two embedded boards, such as an Arndale board and NVidia TK1 board. There are such problems when multiple guest OSs are running on hypervisor that do not guarantee a real-time property. Due to the scheduling of the hypervisor, the RTOS does not match the time actually operated and the time on the system. Therefore, the system causes a slight performance degradation in a different order of measurement time.

Acknowledgments: This work was jointly supported by a Project no. K-17-L01-C01 of Korea Institute of Science and Technology Information, and supported by a Basic Researcher Project No. NRF-2016R1C1B1012189 of National Research Foundation. This work was also jointly supported by a Startup Growth Technology Development Project of Small and Medium Business Administration in Korea.

Author Contributions: For the research reported, ByoungWook Kim conceived of and designed the overall architecture and conducted the experiment. Min Choi is the corresponding author, and performed the overall design and the source code implementation for uC/OS II and FreeRTOS porting.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kihong, L.; Dongwoo, L.; Young, I.E. A Study on Trend for Hardware-assisted Virtualization Technology. In Proceedings of the Korean Information Science Society, Jeju, Korea, 15–16 November 2013; pp. 1316–1318.
2. Young, G.J.; Byung, J.L.; Hee, Y.Y. The comparative analysis of Hypervisor according to Virtualization Method. In Proceedings of the Korean Society of Computer Information Conference, Cheongju, Korea, 22–24 January 2015; pp. 251–253.
3. Travis, L. Exploring the Design of the Cortex-A15 Processor. Available online: https://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf (accessed on 15 April 2017).
4. In-Kyu, H. K-Hypervisor: Design and Implementation of Hypervisor Based ARM Cortex-A15. Available online: http://www.dbpia.co.kr/Journal/ArticleDetail/NODE02444451?TotalCount=0&Seq=4&isIdentifyAuthor=1&Collection=0&isFullText=0&specificParam=0&SearchMethod=0&Page=1&PageSize=20 (accessed on 15 April 2017).
5. Prashant, V.; Gernot, H. Hardware-supported virtualization on ARM. In Proceedings of the Second Asia-Pacific Workshop on Systems, Shanghai, China, 11–12 July 2011.
6. FreeRTOS Project. Available online: http://www.freertos.org (accessed on 15 April 2017).
7. Rabindra, P.K.; Kent, P. Rhealstone: A Real-Time Benchmarking Proposal. Available online: http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1989/8902/8902a/8902a.htm (accessed on 15 April 2017).
8. ARM Generic Interrupt Controller Architecture Specification. Available online: http://www.cl.cam.ac.uk/research/srg/han/ACSP35/zynq/arm_gic_architecture_specification.pdf (accessed on 15 April 2017).
9. Simon, D. OS Porting & Analysis for Dual Core ARM Cortex-A9 Based Systems. Available online: http://www.imperas.com/documents/ARM_TechCon_2012_Imperas_Paper_OS_Porting_and_Analysis_for_Cortex_A9MP_Based_Systems.pdf (accessed on 15 April 2017).
10. Introduction to Basic RTOS Features Using SAM4L-EK FreeRTOS Port. Available online: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42247-Introduction-to-Basic%20RTOS-Features-using-SAM4L-EK-FreeRTOS-Port_Training-Manual.pdf (accessed on 15 April 2017).

11. Sung-Min, L.; Sang-Bum, S.; Bokdeuk, J.; Sangdok, M.; Brian Myungjune, J.; Jung-Hyun, Y.; Jae-Min, R.; Dong-Hyuk, L. Fine-grained I/O access control of the mobile devices based on the Xen architecture. In Proceedings of the 15th Annual International Conference on Mobile Computing and Networking, Beijing, China, 20–25 September 2009.
12. Gernot, H.; Ben, L. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. Available online: https://ts.data61.csiro.au/publications/papers/Heiser_Leslie_10.slides.pdf (accessed on 15 April 2017).
13. Gernot, H. Hypervisors for consumer electronics. In Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference, Las Vegas, NV, USA, 11–13 January 2009.
14. ARM Cortex-A15. Available online: https://en.wikipedia.org/wiki/ARM_Cortex-A15 (accessed on 15 April 2017).
15. Francois, A.; Michel, G. A Practical Look at Micro-Kernels and Virtual Machine Monitors. In Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference, Las Vegas, NV, USA, 10–13 January 2009.
16. François, A.; Michel, G.; Gilles, M.; Gregory, M. Shared device driver model for virtualized mobile handsets. In Proceedings of the First Workshop on Virtualization in Mobile Computing (MobiVirt'08), Breckenridge, CO, USA, 17 June 2008.
17. Keir, F.; Steven, H; Rolf, N.; Ian, P.; Andrew, W.; Mark, W. Safe Hardware Access with the xen Virtual Machine Monitor. Available online: http://www.cl.cam.ac.uk/research/srg/netos/papers/2004-oasis-ngio.pdf (accessed on 15 April 2017).
18. Anderson, T.E.; Bershad, B.N.; Lazowska, E.D.; Levy, H.M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. Available online: http://flint.cs.yale.edu/cs422/doc/sched-act.pdf (accessed on 15 April 2017).
19. Video Clip for the Digital Timer Board in Experimental Results Section. Available online: https://youtu.be/4wMuyhlW97o (accessed on 15 April 2017).