Performance analysis towards a KVM-Based embedded real-time virtualization architecture

Conference Paper · January 2011

DOI: 10.1109/ICCIT.2010.5711095 · Source: IEEE Xplore

6 authors, including:

Kai Chen
South China University of Technology
93 PUBLICATIONS **2,743** CITATIONS
SEE PROFILE

Ruhui Ma
Shanghai Jiao Tong University
70 PUBLICATIONS **608** CITATIONS
SEE PROFILE

Yaozu Dong
Intel
50 PUBLICATIONS **1,054** CITATIONS
SEE PROFILE

Performance Analysis Towards a KVM-Based Embedded Real-Time Virtualization Architecture

Jun Zhang [1], Kai Chen [1], Baojing Zuo [1], Ruhui Ma [1], Yaozu Dong [2], Haibing Guan [1] *

[1] Shanghai Jiao Tong University,
Shanghai, P.R.China
{tragicjun, kchen, zuobaojing,
ruhuima, hbguan}@sjtu.edu.cn

[2] Intel China Software Center,
Shanghai, P.R.China
{eddie.dong}@intel.com

***Abstract*- In Recent years embedded world has been undergoing a shift from traditional single-core processors to processors with multiple cores. However, this shift poses a challenge of adapting legacy uniprocessor-oriented real-time operating system (RTOS) to exploit the capability of multi-core processor. In addition, some embedded systems are inevitably going towards the direction of integrating real-time with off-the-shelf time-sharing system, as the combination of the two has the potential to provide not only timely and deterministic response but also a large application base. Virtualization technology, which ensures strong isolation between virtual machines, is therefore a promising solution to above mentioned issues. However, there remains a concern regarding the responsiveness of the RTOS running on top of a virtual machine. In this paper we propose an embedded real-time virtualization architecture based on Kernel-Based Virtual Machine (KVM), in which VxWorks and Linux are combined together. We then analyze and evaluate how KVM influences the interrupt-response times of VxWorks as a guest operating system. By applying several real-time performance tuning methods on the host Linux, we will show that sub-millisecond interrupt response latency can be achieved on the guest VxWorks.**

Keywords-embedded; real-time; virtualization; KVM; VxWorks

I. INTRODUCTION

The introduction of multi-core to embedded systems in recent years has brought the availability of increased computing power to embedded devices. However, the development of microprocessor is so fast that most of legacy embedded software systems cannot adequately adapt to this new executive environment, that is to say, conventional uniprocessor-oriented software systems, such as legacy embedded RTOS, are not able to make full use of multi-core resources. For addressing this daunting issue, a great deal of efforts have been made. In particular, the most straightforward solution is modifying the kernel code of legacy RTOS [1]. But it involves excessive workload, that is, implementing new facilities then verifying if they interfere with the real-time constraints. Thus, the researchers and producers now are seeking cheaper yet viable approaches.

In addition, some embedded systems have the need of not only timely and deterministic response but also a large application base. Therefore they are inevitably going towards the direction of integrating real-time with off-the-shelf time-sharing system. Take mobile phones as an example, it would be better for them to utilize hybrid system that integrates a RTOS with a general-purpose operating system (GPOS). While the RTOS is responsible for managing time-critical tasks of the radio communication, the GPOS provides the typical set of mobile phone applications like games [2].

Virtualization technology has been widely applied in various kinds of system over the years. With a virtualization layer called the hypervisor or the virtual machine monitor (VMM), different virtual machines each running its own operating system can share the same underlying physical hardware resources. Virtualization, therefore, becomes an enabling technology for the combination of RTOS and GPOS, with each of them running on separate virtual machines. Furthermore, it paves a natural way to make full use of multi-core resources through virtual machine scaling.

In this paper, we propose an embedded real-time virtualization architecture based on KVM [3] which is one of the most popular virtualization platforms today. As depicted in Fig. 1, our architecture combines VxWorks and Linux together on a multi-core platform with the help of KVM. However, the question regarding what quality of service can be provided by this architecture exists for determining its practical applications. Our goal in this paper is to evaluate the real-time capabilities of this architecture thus providing a hint on whether it is suitable to be deployed on embedded devices like mobile phones.

The rest of the paper is organized as follows. In section 2 we give background on hardware-assisted virtualization and KVM. In section 3 we present our analysis on how KVM influences the real-time performance of the guest RTOS. In section 4 we present our experimental setups and evaluation results. Finally section 5 presents conclusion.

II. BACKGROUND

A. Hardware-Assisted Virtualization

On conventional x86 processors, the classical trap-and-emulate virtualization model cannot be applied easily [4], which gives rise to software techniques, such as binary translation utilized by VMware [5] and para-virtualization applied

* Corresponding Author

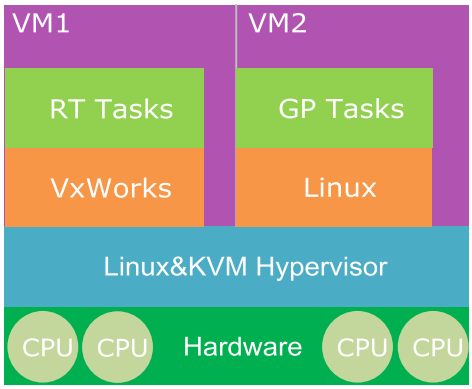

Figure 1. An embedded real-time architecture based on KVM

in Xen [6]. Although software techniques introduce viable solutions to x86 virtualization, they have various limits that prevent them from fitting all scenarios [7]. Therefore hardware extensions have been added to recent processors so that faithful virtualization can be implemented.

This section discusses hardware extensions that permit full virtualization on x86 processors. The discussion applies to both Intel's VT and AMD's SVM, but we use Intel's notions, for our experiments were conducted based on VT.

1). Processor Mode

Processor support for virtualization is provided by a form of processor operation called VMX operation. There are two kinds of VMX operation: VMX root operation and VMX non-root operation [8]. In general, the host operating system and VMM will run in root operation, while the guest operating systems along with their applications will run directly in non-root operation. Since both root and non-root operations include all x86 privilege rings, guest code can run at its intended ring, which means no de-privileging is required.

Transitions between VMX root operation and VMX non-root operation are called VMX transitions. There are two kinds of VMX transitions. Transitions into VMX non-root operation are called VM entries. Transitions from VMX non-root operation to VMX root operation are called VM exits[7, 8].

2). Control Structure

A new data structure, referred to as the virtual machine control structure(VMCS), is introduced to manage and control the execution behavior of virtual processors. While each virtual CPU should have its own VMCS, only one VMCS should be active at a time on a CPU.

Right before guest code starts executing in non-root operation, the VMM issues a VMX instruction called VMLAUNCH to cause VM entry, upon which the hardware loads guest state from the guest-state area in the VMCS, and begins guest execution. It is worth noting that during the process of VM entry, the hardware also checks some fields in the VMCS to determine whether an event injection to guest operating system was requested by the VMM. When CPU executes in non-root operation, flags in the VMCS control which events or instructions cause VM exit, such as external interrupts, exceptions, and certain I/O instructions. During the process of VM exit, the hardware first stores guest state back to the guest-state area in the VMCS and then loads host state from the host-state area in the VMCS. Moreover, information about the cause of the VM exit is recorded in the VMCS that helps the VMM determine which action should be taken next.

B. Kernel-Based Virtual Machine

The kernel-based virtual machine (KVM) is a creative virtualization solution based on Linux. It was originally developed to utilize underlying hardware-based virtualization extensions. Unlike common hypervisors that perform basic scheduling, memory management themselves. KVM finds the similarity of modules between VMM and operating system kernel, thus resorting to existing Linux kernel for all common functions. In other words, KVM turns standard Linux kernel into a hypervisor simply by adding some virtualization capabilities to it.

KVM is implemented as two components: kernel-space device driver for CPU and memory virtualization and user-space emulator for PC hardware emulation. This section describes how these two components work respectively.

1). Kernel Module

Standard Linux kernel can derive virtualization capabilities from a loadable kernel module. This kernel module, which is the key part of KVM, serves as a device driver and is responsible for providing CPU and memory virtualization features with the help of hardware extensions.

KVM kernel module provides various services like creation of virtual machines and virtual CPUs, allocation of memory to specified virtual machine, and injection of events. It exposes those services via a character device /dev/kvm which can be used by the user-space emulator through ioctl() system call.

KVM introduces a new operating mode, the guest mode, to Linux kernel. The guest mode, corresponding to Intel VT's VMX non-root operation, is where virtual machines run. KVM kernel module executes as a bridge between user mode and guest mode. On the one hand, the user-space emulator can request execution of a virtual machine by issuing ioctl() system call. When the kernel module receives this request, it will make some preparation like loading VCPU context to VMCS, injecting pending virtual event and disabling interrupt, etc. It then switches control to guest mode by issuing special instruction offered by hardware extension. From then on, the virtual machine starts executing in guest mode. On the other hand, if a VM exit caused by some special I/O instruction occurs later, the kernel module will take control to handle this exit. It first restores certain environment like enabling interrupt again and then handles this VM exits by switching back to user space for I/O emulation.

2). User-Space Emulator

KVM uses a modified version of QEMU [9] for PC platform emulation. Instead of applying dynamic translation for CPU emulation, KVM's QEMU version allows native execu-

tion by invoking services provided by KVM kernel module. It simply issues a series of ioctl() system call to create and manage virtual machines [10].

KVM turns the single-threaded execution model of QEMU into a multi-threaded model. Since every virtual machine on KVM is a process of the host Linux, a QEMU process therefore represents an instance of virtual machine. In other words, the host Linux views a virtual machine as a QEMU process. Normally, a QEMU process has two kinds of thread: main I/O thread and VCPU thread. While I/O thread is used to manage emulated devices, VCPU thread is used to run guest code. Furthermore, an AIO thread is dynamically created to handle every AIO request and signal the completion to the I/O thread.

III. REAL-TIME PERFORMANCE ANALYSIS

In this section, we firstly describe how KVM influences the real-time performance, interrupt response time (IRT) in particular, of the guest RTOS at base level and therefore the real-time capabilities of Linux as a hypervisor when using KVM. Then we apply some of the typical real-time tunings on the host Linux and analyze how they can ease certain latencies incurred by harmful workloads.

A. Base Overhead

We first define IRT as the time between physical devices raising an interrupt and the first instruction of the corresponding interrupt service routine (ISR) starting execution. In this analysis, however, we are not so interested in the guest's IRT itself, instead we care more about the latencies that are introduced by KVM.

Suppose a physical interrupt occurs at some certain time when the guest RTOS is running, there will be at least six stages to go through before this interrupt can be delivered to the guest RTOS. These stages illustrated in Fig. 2 are main sources of the latencies caused by KVM.

Apart from those six stages, Fig. 2 also highlights eight points in time that mark the beginning or the end of every stage. We describe operations related to the delivery of the interrupt at every stage below:

1) VM Exit: KVM configures the VMCS such that any external interrupt cause a VM exit. During the VM exit, information associated with this interrupt is stored by the hardware to the VMCS, like interrupt type and interrupt number.

2) After VM Exit: After VM exit, control is switched back to kernel space at point b. Before enabling interrupt again, KVM needs to fetch information of this interrupt from VMCS and then mark it as a pending interrupt that waits for being delivered to the guest.

3) Host ISR: After interrupt is re-enabled by KVM's kernel module at point c, the host Linux starts interrupt response immediately. In some cases, the host ISR may send a signal to the guest RTOS process that later makes it switch to user space for I/O device emulation. This can therefore give rise to additional overhead. But here we focus on the minimum overhead, thus leaving those scenarios for further analysis.

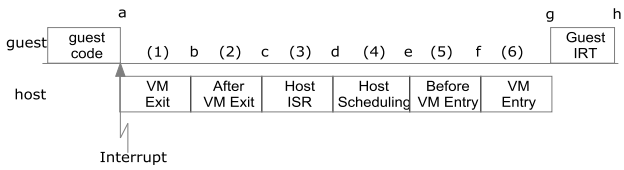

Figure 2. Six stages associated with the delivery of the interrupt to the guest OS

4) Host Scheduling: Given that the host ISR might awaken some processes, the host kernel must invoke scheduler to determine which process will be running. In the best situation, the guest RTOS process does not get preempted by other processes.

5) Before VM Entry: From point e on, control gets back to KVM's kernel module. Before next VM entry, KVM detects that there is a pending interrupt. It therefore injects a virtual interrupt to the guest by writing corresponding interrupt information to VMCS.

6) VM Entry: At the end of VM entry, based on the interrupt information stored in VMCS, the hardware emulates the process of conventional interrupt handling in the context of the guest, which means it ends up invoking the guest's handler to service the interrupt.

B. Real-Time Tunings

In last section, we analyzed how KVM incurs extra latencies to the guest's IRT at base level. Base level means that the analysis is based on an assumption that there are no other workloads running aside with the guest RTOS. However, such ideal condition obviously can never exist, because the guest GPOS is probably a heavy workload. It generally creates a number of CPU loads or interrupt loads that may lead to additional latencies. For example, at the "Host Scheduling" stage, some GPOS's CPU load is likely to preempt the guest RTOS process. We therefore view this kind of CPU load as harmful workload.

Fortunately, given that every virtual machine on KVM is a process of the host Linux, several real-time tuning knobs provided by the Linux kernel can be applied to the guest RTOS process [11, 12]. We applied two of them, prioritization [13] and CPU shielding [14, 15], and analyzed how they can ease certain latencies incurred by harmful workloads respectively.

1). Prioritization

As pre-mentioned earlier, it is possible that the GPOS's CPU loads are preferable to the guest RTOS process at the "Host Scheduling" stage, causing additional latencies to the guest's overall IRT. This situation can be eased by giving the guest RTOS process the highest real-time priority.

In another case, however, the GPOS's interrupt loads may cause physical interrupts at any time. Moreover, in the standard Linux kernel, physical interrupts can even preempt processes of the highest priority. This situation can be addressed by applying the RT patch [16] to the standard kernel, because RT patch implements threaded interrupt handlers.

There are two approaches to raising the priority of the guest RTOS process, specifically I/O thread and VCPU threads. On one hand, there is a Linux system call sched_setscheduler() that can be invoked by KVM to set priority when creating those threads. On the other, Linux shell provides a command chrt that can be used by users to change priority after start-up.

2). CPU Shielding

Suppose a physical interrupt is raised while the guest GPOS , rather than the guest RTOS, is running. It is likely that the workload is in a long interrupt-off or preemption-off region, thus leading to large latencies. This kind of scenario can be eased by dedicating one CPU to the guest RTOS process.

CPU shielding is a general approach for obtaining good real-time performance in a symmetric multi-processor (SMP) system [14]. It prevents the guest RTOS from being adversely affected by harmful workloads, like interrupt-off regions and cache pollutions, thus achieving the best real-time performance.

The Linux kernel supports both task affinity and interrupt affinity. For task affinity, we can either invoke system call sched_setaffinity() or use tools like taskset/cpuset from the shell to set CPU affinity of a process. For interrupt affinity, Linux provides a user interface via the /proc/irq/<irq_number> /smp_affinity files for setting CPU affinity of an interrupt.

IV. EXPERIMENTS

In this section we evaluate the interrupt response time of the guest RTOS running on top of KVM and the effect of applying real-time tunings, specifically prioritization and CPU shielding. We first introduce the experimental setup used for the evaluation and then present the results of our experiments.

A. Experimental Setup

Our experimental platform consisted of a Intel quad-core processor (Core2 i5 750 at 2.67GHz) and 2GB RAM, but we only used two processor cores for all tests. We furthermore got rid of system management interrupts (SMI) on the system to avoid occasional latency peaks [17]. The experimental architecture involves three kinds of operating systems: host Linux, guest RTOS and guest GPOS. The host Linux is based on CentOS 5.4, upon which we built a 2.6.33.4 kernel plus RT patch 2.6.33.4-rt20 [16]. In addition, we disabled CONFIG_ACPI_PROCESSOR to prevent response time from being interfered by processor power management service. We used VxWorks 5.5 as the guest RTOS and CentOS 5.4 as the guest GPOS respectively. As for KVM, we used KVM driver built in the host Linux kernel and qemu-kvm 0.12.50 [18] as the user-space emulator.

To ease implementation of the benchmark, we chose timer as the interrupt source, because it is straightforward to control the frequency at which timer interrupt is raised. Therefore, our benchmark measured the response time of timer interrupt on the guest VxWorks. On the other hand, we defined two load applications running on the guest Linux: simple endless loop representing computational load and bonnie 1.4 [19] representing I/O load. These two loads were important to observe worst-case performance.

For evaluating the latencies incurred by KVM, we did another experiment that used a new guest RTOS of the same Linux distribution and kernel as the host. We then measured the Process Dispatch Latency Time (PDLT) [20] on both host and guest Linux. PDLT is the time between physical devices raising an interrupt, specifically timer interrupt in this experiment, and the first instruction of the corresponding response process starting execution. However, we were not interested in the absolute value of PDLT. Instead, we focused on the difference of PDLT between host and guest which reveals the latencies incurred by KVM. For measuring PDLT, we used cyclictest from the rt-test project [21] that measures the elapsed time for a nanosleep call. Specifically, we ran cyclictest with the following parameters: -t1 -m -n –p 80 -i 10000 -l 100000 -v -q, which means a 10ms nanosleep test was performed for 100 thousand times.

B. Results: VxWorks IRT

We first compared the overall results of one without any guest GPOS workload(referred to as base) and the other with load applications running on a guest GPOS (referred to as load). Fig. 3 shows the average and maximum IRT of both cases. Obviously, in the load case, the maximum IRT is much larger than that in the base case. However, the average IRT is a little different, with the base case slightly better than the load case. It shows that the workloads on the guest GPOS have more influence on the maximum IRT of the guest RTOS.

Next we applied real-time tunings and evaluated their effects on the guest IRT.

For prioritization, we used shell command chrt to apply SCHED_FIFO scheduling policy to the guest VxWorks, specifically 98 for its I/O thread and 97 for its VCPU thread. Besides, hrtimer kernel thread should also be applied with SCHED_FIFO scheduling policy with priority 99. The result is depicted in Fig. 4. It shows that the maximum IRT decreases to about 100us in both cases.

For CPU shielding, we used cpuset to set the second processor core as a dedicated core for the guest VxWorks. It means that all threads, except for some per-core kernel threads, were prevented from running on the second processor core.

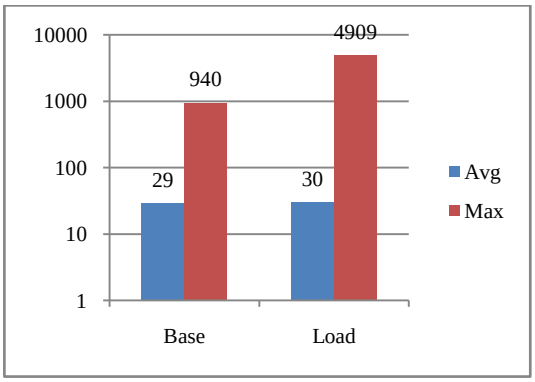

Figure 3. Latency of guest VxWorks

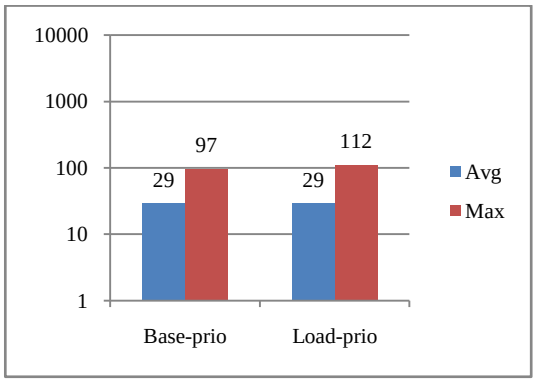

Figure 4. Latency of guest latency with prioritization tuning

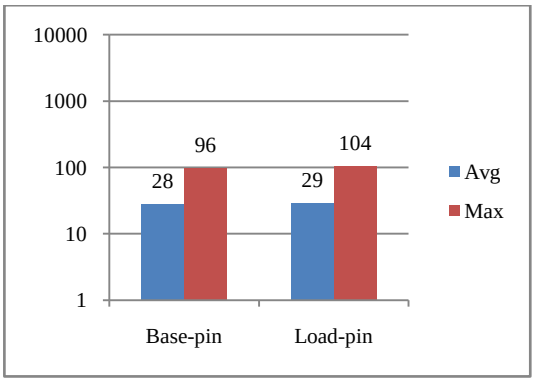

Figure 5. Latency of guest latency with CPU shielding tuning

Moreover, common interrupts, like disk and network, were affinitized to the first processor core. The result is depicted in Fig. 5. It shows that CPU shielding provides the same level of enhancement as prioritization.

The results in Fig. 4 and Fig. 5 reveal that general real-time tunings have the effect of improving the maximum IRT of the guest RTOS even under heavy-load circumstances. When it comes to average IRT, however, real-time tunings have little effect. It indicates that in average condition the guest RTOS is preferable to other workloads at the "Host Scheduling" stage.

C. Results: Linux PDLT

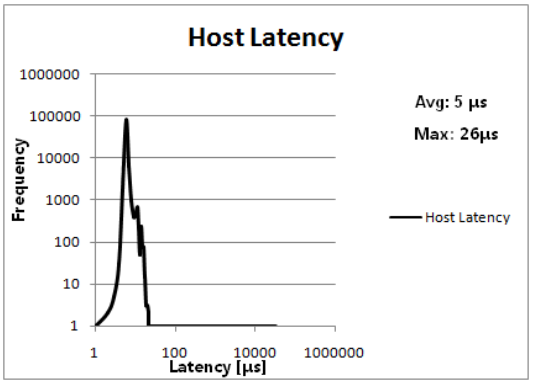

Figure 6. Latency of host Linux

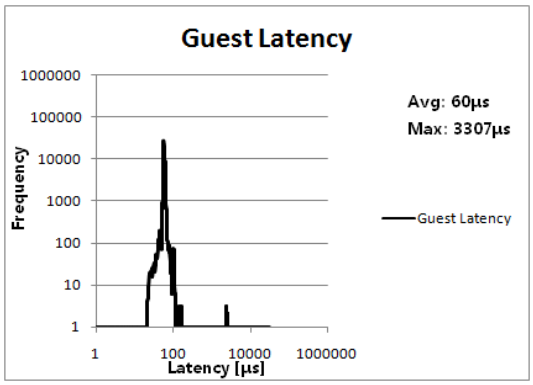

Figure 7. Latency of guest Linux

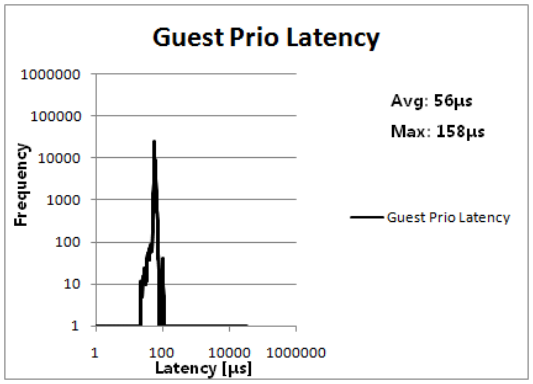

Figure 8. Latency of guest Linux with prioritization tuning

We first measured latencies natively by running cyclictest on the host Linux. Fig. 6 gives the results of the host latency in term of distribution. The X-axis shows the value of the latency and the Y-axis shows the frequency of occurrence of the corresponding latency. We see that most of the values over 100 thousand loops are below 10. It clearly indicates that, with RT patch, Linux is capable of achieving low interrupt response latency, which makes Linux applicable for certain real-time scenarios.

Next we measured the guest latency by running cyclictest on the guest Linux. The result is depicted in Fig. 7. It shows a right shift of the latency distribution as opposed to Fig. 6. Ac-

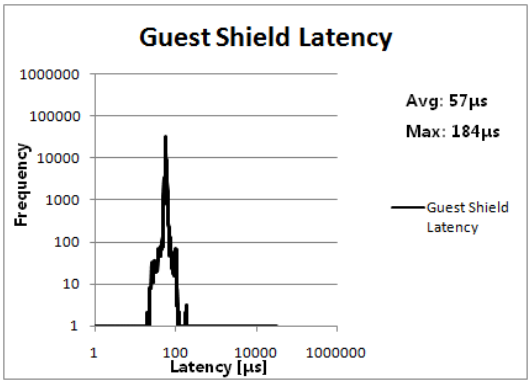

Figure 9. Latency of guest Linux with CPU shielding tuning

cordingly, the average value of the guest latency is dozens of times larger than that of the host latency. Even worse is the maximum guest latency, with a value of several milliseconds.

Learning from previous experiment for VxWorks, real-time tunings have the effect of lowering the maximum latency. Thus we also applied prioritization and CPU shielding for the guest Linux. Fig. 8 and Fig. 9 show the results for applying both real-time tunings. Much like the results for the guest VxWorks, although the average values are of little change, the maximum values are greatly decreased.

IV. CONCLUSION

Virtualization technology is a promising solution for addressing hardware (multi-core) and software (combination of RTOS and GPOS) challenges facing embedded world today. In this paper, we proposed an embedded real-time virtualization architecture that combines VxWorks and Linux together on a multi-core platform with the help of KVM. We analyzed how KVM introduces latencies to the interrupt response time of the RTOS. In the evaluation, we demonstrated that with careful system setups, like applying RT patch and real-time tunings on the host Linux, sub-millisecond response latency can be achieved, which makes this architecture applicable for certain embedded real-time scenarios.

ACKNOWLEDGMENT

This work was supported by The National Natural Science Foundation of China (Grant No.60773093, 60873209, 60970108), The Science and Technology Commission of Shanghai Municipality(09510701600), IBM SUR Funding and IBM Research-China JP Funding, Intel Research-China Funding.

REFERENCES

[1] An RTOS for an SMP Multicore Processor. http://rtcmagazine.com/. October 2006.
[2] Multi-Core with virtualization, a solution for future smart phones. http://www.alphagalileo.org. Aug 2010.
[3] A. Kivity, Y. Kamay and D. Laor, "KVM: The Linux Virtual Machine Monitor," Proceedings of the Linux Symposium, 2007.
[4] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," in Computer, vol.38, pp.39-47, May 2005.
[5] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," Proceedings of the 12th international con-ference on Architectural support for programming languages and operating systems, October 21-25, 2006, San Jose, California, USA.
[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization", In Proceedings of the 19th ACM Symposium on Operating System Principles, October 2003, pp. 164–177 (2003).
[7] G. Neiger , A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel virtualization technology: Hardware support for efficient processor virtualization," Intel Technology Journal 10, 3 (2006).
[8] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual", Vol. 3B, pp. 3-8, March 2010.
[9] F. Bellard, "QEMU, a fast and portable dynamic translator," Proceedings of the annual conference on USENIX Annual Technical Conference, pp.41-41, April 10-15, 2005, Anaheim, CA.
[10] I. Habib, "Virtualization with KVM", Linux Journal, vol. 2008 n.166, pp.8, February 2008.
[11] P. McKenney, "Real time vs. real fast: How to choose?" In Proceedings of the 11th Real-Time Linux Workshop, pp. 11-21, September 2009.
[12] H. Yoon, J. Song and J. Lee, "Real-Time Performance Analysis in Linux-Based Robotic Systems," In Proceedings of the 11th Linux Symposium, pp. 331-340, July 2009.
[13] J. Kiszka, "Towards Linux as a Real-Time Hypervisor," In Proceedings of the 11th Real-Time Linux Workshop, pp. 215-224, September 2009.
[14] S. Brosky and S. Rotolo, "Shielded processors: Guaranteeing sub-millisecond response in standard Linux," In Workshop on Parallel and Distributed Real-Time Systems, WPDRTS'03, Nice, France, April 2003.
[15] E. Piel, P. Marquet, J. Soula, J. and L. Dekeyser, "Asymmetric scheduling and load balancing for real-time on Linux SMP," In Proceedings of the Workshop on Scheduling for Parallel Computing (LNCS 3911), Springer Verlag, 2005, pp. 869–903.
[16] Real-Time Linux Wiki, 2009. http://rt.wiki.kernel.org.
[17] SMI interrupts on x86. http://www.rootninja.com. April 2009.
[18] KVM Main Page. http://www.linux-kvm.org/page/Main_page
[19] Bonnie 1.4, 2009. http://wiki.linuxquestions.org/wiki/Bonnie
[20] A. Heursch, D. Grambow, A. Hosrtkotte, and H. Rzehak, "Steps towards a fully preemptable Linux kernel," In Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming, Lagow, Poland, May 2003.
[21] RT test utils, 2009. http://git.kernel.org/?p=linux/kernel/git/tglx/rt-tests.git.