JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 29, 1021-1035 (2013)

DOI:10.1688/JISE.2013.29.5.13

Performance Tuning Towards a KVM-based Embedded Real-Time Virtualization System*

RUHUI MA[1], FANFU ZHOU[1], ERZHOU ZHU[2,+] AND HAIBING GUAN[1]
[1]*Department of Computer Science and Engineering*
Shanghai Key Laboratory of Scalable Computing and Systems
Shanghai Jiao Tong University
Shanghai, 200240 P.R. China
[2]*School of Computer Science and Technology*
Anhui University
Hefei, 230601 P.R. China

Virtualization is a fundamental component in cloud computing because it provides numerous guest VM transparent services, such as live migration, high availability, rapid checkpoint, *etc.* Utilizing virtualization technology to combine real-time operating system (RTOS) and off-the-shelf time-sharing general purpose operating system (GPOS) is attracting much more interest recently. Such combination has the potential to provide a large application base, and to guarantee timely deterministic response to real-time applications, yet there remain some issues, such as responsiveness of RTOS running on top of a virtual machine (VM), system performance and CPU resource utilization rate, *etc.* 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. Furthermore, we also find out that prioritization tuning results in waste of CPU resources when RTOS is not executing real-time tasks, so we design a dynamic scheduling mechanism – co-scheduling to improve system performance. Experimental results with SPEC2000 and bonnie 1.4 load, show that this new architecture tuned by CPU shielding, prioritization and co-scheduling, can achieve better real-time responsiveness and system performance.

Keywords: KVM, muti-core, real-time, co-scheduling, virtualization

1. INTRODUCTION

In recent years, the introduction of multi-core to embedded systems has brought the availability of increased computing power to embedded devices, such as dual-core ATOM microprocessor developed by Intel [1]. However, the development of microprocessor is so fast that most of legacy embedded software systems cannot adequately adapt to this new executive environment. To address this daunting issue, lots of efforts have been made [24]. In terms of hardware, the most straightforward solution is to modify the kernel code of legacy RTOS [2]. But it involves excessive workload and is also too expensive. In terms of software, each traditional embedded system serves only one special

Received May 2, 2011; revised February 5, March 22 & April 11, 2012; accepted May 20, 2012.
Communicated by Tei-Wei Kuo.
* This work was supported by National Natural Science Foundation of China (No. 61202374, 61272101), International Cooperation Program of China (No. 2011DFA10850), and China Postdoctoral Science Foundation (2012M511096).
+ Corresponding author: ezzhusjtu@gmail.com.

application, and even some of them are uncontrolled by OS. Unlike the simple one mentioned, the sophisticated one provides with not only the timely and deterministic responsiveness but also some general-purpose services. To satisfy these requirements, there are two solutions: on the one hand, it is to extend the function of RTOS, but RTOS cannot provide enough power and mainstream API for programmer to further exploit various applications; on the other, it is to extend the GPOS and add some real time feature into it. But re-developing the legacy RTOS must bring too many works. Therefore RTOS and GPOS are inevitably going towards the direction of integrating real-time with off-the-shelf time-sharing system. A famous use case is in industrial control area, where an RTOS is used to take over the time-critical tasks and a GPOS is used aside it to run some monitor applications, *etc.* Take mobile phones as an example, it would be better for them to utilize a hybrid system that integrates an RTOS with a 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 [3]. And with the increasing importance of real time in enterprise server computing scenarios, for example in an online telephone backend server, the overlap with virtualization becomes larger as well. Though this combination can satisfy more current requirements, the isolation between GPOS and RTOS, how to reasonably assign physical resources and the lower responsiveness and system performance, *etc*., are still not addressed adequately.

To address the issues mentioned above, virtualization technology has been employed. The advantage of virtualization technology is that different virtual machines each running its own operating system can share the same underlying physical hardware resources. In addition, virtualization provides isolation between RTOS and GPOS. Finally, with virtualization, uniprocessor-friendly RTOS executed on multi-core platform needs not modifying to efficiently utilize multi-core resources. Although several advantages can be achieved through adding VMM, the real-time responsiveness of RTOS is also affected by VMM and other GPOSes. For instance, the famous VMM – XEN is not designed to serve real time embedded system [5].

In this paper, we propose an embedded real-time virtualization architecture based on KVM [4]. As depicted in Fig. 1, our architecture combines VxWorks and Linux together on a multi-core platform with KVM. In this architecture, responsiveness and system throughput are the key points. In addition, how the KVM and GPOS affect the RTOS is

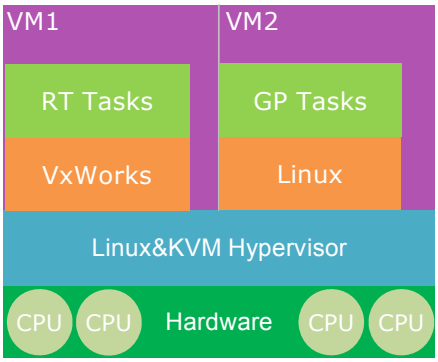

Fig. 1. An embedded real-time architecture based on KVM.

analyzed in detail. To satisfy the soft real-time requirements, several tuning methods, such as CPU shielding, prioritization, are applied on this architecture to minimize the maximum value of latency. Meanwhile, we present co-scheduling algorithm not only to promise the real-time responsiveness but also to improve system throughput through efficiently utilizing multi-core resources. There are several contributions in this paper:

- An embedded real-time virtualization platform based on Kernel-Based Virtual Machine (KVM), is presented in this paper.
- Several tunings are used on this new platform to ensure real-time responsiveness.
- A novel scheduling algorithm – co-scheduling is presented in this paper.

The rest of the paper is organized as follows. Section 2 introduces related work about scheduling strategy and multi-core technique applied on real-time environment. In section 3 we present our analysis on how KVM influences the real-time performance of the guest RTOS, and introduce several real-time performance tuning methods. In section 4 we propose co-scheduling policy. In section 5 we present our experimental setups and evaluation results. Finally, section 6 presents conclusions.

2. RELATED WORKS

As real-time embedded system has become powerful enough to take in multi-core technique, some embedded software manufacturers have built several VMMs to better utilize multi-core resources. Since the execution of real-time tasks can be affected by general-purpose applications, recent research papers focus on employing multi-core technique to address it through assigning different cores for various tasks. Currently, RTS Hypervisor [6] designed by Real-time System Corporation, VirtualLogix Corporation's VLX [7] and Intel's WindRiver [8] adopt this method. Though this static assignment method can improve RTOS's real-time responsiveness, extra CPU slices are wasted by RTOS because soft real-time applications are periodic in nature, leading to terrible system performance.

As we know, an embedded virtualization system provides multi-guests with security isolation and resource sharing, but it is difficult to support real-time tasks better, especially scheduling policy. Seehwan Yoo *et al.* [5] designs a VMM – MobiVMM used to extend mobile phone function. It adopts priority-based preemptive scheduling and pseudo-polling mechanism and gives RTOS the highest priority. Though this method can provide better real-time responsiveness, redesigning this new VMM must spend lots of workloads. In addition, MobiVMM promises only one RTOS. Robert Kaiser classifies guest VMs into three parts: non real-time, time-driven, and event-driven [9]. He discusses the requirements of each type of VMs for VMM and also proposes a specialized scheduler for each type of VMs. When running, a system will select different schedulers through some priority policies. But this method is only analyzed in theoretical, and is not in practice. Even in this paper, there is no experimental result to testify it. Jan Kiszka designs a priority strategy for real-time VM based on KVM, which can lower responsiveness latency of real-time applications [10]. However, there is only one RTOS in this architecture, that is, non-real-time VMs are not considered in this architecture. Like Jan

Kiszka, Jiang Wei *et al.* [11] also improve KVM to achieve better real-time performance. He adjusts Completely Fair Scheduler (CFS) scheduling strategy to address Lock-Holder Preemption (LHP) issue when VMM manages guest VMs. But this idea has been utilized by chip manufacturers. Hitoshi Mitake [18] develops a new technique for avoiding the LHP problem. The approach can ensure both the real-time responsiveness of RTOS and the high throughput of GPOS that supports shared memory multi-processors. Gernot Heiser [19] provides a number of examples of present or likely use cases of virtualization in embedded systems, and explains the motivation and benefits, as well as some of the differences to server-style virtualization.

In a word, these ideas mentioned focus on improving the priority of real-time applications to occupy CPU resources all the time, avoiding interruption from non-real-time applications. However, some non-real-time tasks still exist in RTOS, and likewise, some tasks with higher priority may be in GPOS, such as audio/video task. In addition, VMM considers the VM as a black box, and it is not aware of the detailed requirement of each task in VM. This leads to priority inversion. So in this paper, we propose co-scheduling policy to cope with this issue.

3. REAL-TIME PERFORMANCE ANALYSIS

In this section, we firstly describe how KVM influences the real-time performance, and 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.

3.1 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:

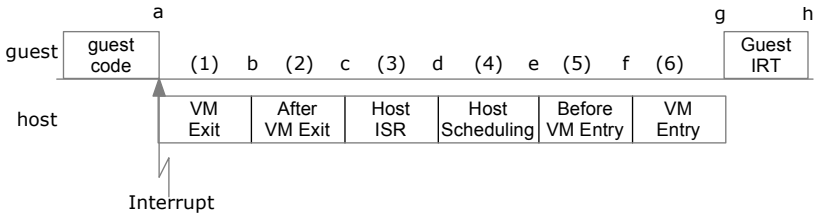

Fig. 2. Six stages associated with the delivery of the interrupt to the guest.

(1) VM Exit: KVM configures the VMCS such that any external interrupt causes 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.
(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.

3.2 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 [15, 16]. We apply two of them, CPU shielding [17, 18] and prioritization [10], and analyze how they can ease certain latencies incurred by harmful workloads respectively [14]. Finally, to improve system performance and promise real-time responsiveness, co-scheduling is introduced in detail [13].

3.2.1 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 dedicat-

ing 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 [17]. It prevents the guest RTOS from being adversely affected by harmful workloads, like interrupt-off regions and cache pollution, 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/hirq_numberi/smp_affinity* files for setting CPU affinity of an interrupt.

3.2.2 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 [21] 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.

4. CO-SCHEDULING MECHANISM

In this section, we will describe co-scheduling mechanism. Obviously, boosting the RT guest OS affects the whole CPU throughput badly when there are non-real-time applications in RTOS. The execution time of RT task is usually short or periodic [19]. So there are times that the guests don't need CPU to do any RT related tasks. We should check whether the guest is doing RT tasks. Based on this information we can dynamically determine whether the guest should be boosted or not. But as introduced before, KVM guest appears to the underlying Linux as a normal ordinary process. In no way can we obtain the inner states of the guest without any change to the guest. A straightforward idea is to modify the guest to inform the underlying Linux host about its states. When the host receives the information, it determines what to do with it dynamically. Such an approach is already used for resolving priority inversion problem [10] and efficiency issue related to guest CPUs is waiting on contended spinlocks [20].

4.1 Paravirt_ops

Assisted by open source community, Xen presented its para interface to achieve high system performance. But this interface should provide virtualization for some de-

vices impacting system performance, such as interrupt controller, page table, and timer, *etc.* To normalize these para-virtualization interfaces, Rusty Russell from IBM designed paravirtualization interface – *paravirt_ops*. It has been developed on x86 as virtualization support via API, not ABI. It allows each hypervisor to override operations which are important for hypervisors at API level. In this paper, we utilize *paravirt_ops* to build co-scheduling module, where a hook function is inserted. And the architecture is depicted as follows: In this function, two function pointers direct two hypercalls to inform VMM the scheduling information of RTOS.

4.2 Co-scheduling Mechanism

The key idea of co-scheduling mechanism is very simple: If the guest is going to enter a time-critical execution, it informs the host about the policy and priority which it needs. Upon receiving the boost request, the host boosts the guest. In our implementation, a VCPU is scheduled according to the last scheduling policy and priority reported by the guest. When the guest finishes the interrupt handling, it informs the host again, and the host deboosts it. Co-scheduling mechanism is depicted in Fig. 3. Then we will introduce this mechanism implemented in VMM and in guest OS respectively.

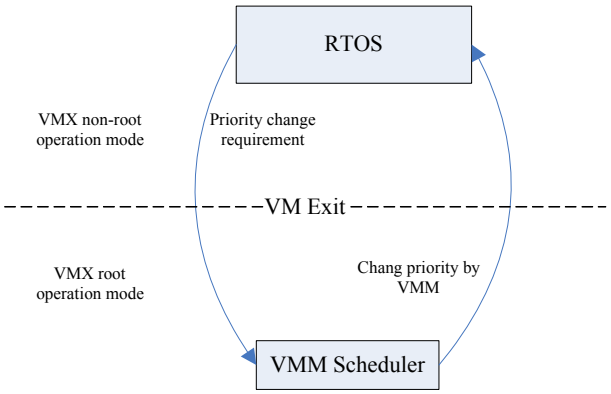

Fig. 3. Co-scheduling architecture overview.

From Fig. 3, we can see that VT-x technique divides CPU operations into VMX root operation mode and VMX non-root operation mode. Indeed, VMM and Guest OS run on VMX root operation mode and VMX non-root operation mode respectively. Though root operation mode is equal to general CPU operation mode, in non-root operation mode, implementing some instructions requires context switch from non-root operation mode to root operation mode. This is called VM Exit. In VMX architecture, VMCALL instruction is inserted to do context switch from non-root operation mode to root operation mode.

In this paper, we utilize VMCALL to implement VMM hypercall mechanism. It leads that guest OS's operation can be called like synchronous communication in system call. Here we do not adopt function call since guest OS and VMM run on different privilege level. In addition, system call changes control power from ring 3 to ring 0 in guest

OS, and hypercall changes control power from ring 0 in non-root operation mode to ring 0 in root operation mode. Like system call, hypercall also follows Trap-and-Emulate mechanism. Then we will introduce this mechanism implemented in guest OS and in VMM respectively.

In guest OS side, process priority change occurs in two situations. On the one hand, function *schedule()* is called when a new process is building. On the other, process calls set *scheduler()* to alter scheduling policy and priority. Consequently, we must monitor scheduling policy change in these two situations. When real-time process or a process enhanced to real-time-priority level has been scheduled, guest OS will inform VMM this change through *boost_vcpu_priority()* presented by us. That is, a virtual interrupt is inserted into VMM to boost process priority through VMM. *boost_vcpu_priority()* function is depicted as following:

```
static DEF_PER_CPU(int, policy) = SCHED_NORMAL;
void boost_vcpu_priority (int cpu, int policy)
{
    If (policy == per_cpu(policy, cpu)) return;
        Per_cpu(policy, cpu) = policy;

        if (policy == SCHED_FIFO || policy==SCHED_RR) {
           kvm_hypercall1(KVM_HC_BOOST_VCPU_PRIORITY,
                                              cpu_physical_id(cpu));
        }
}
```

In *boost_vcpu_priority()* function, *per_cpu(policy; cpu)* is utilized to record the last scheduling policy. *kvm_hypercall* implements VM Exit through VMCALL mentioned above. When each interrupt exits, system checks whether there is an important task worth to be scheduled. If it doesn't exist, system will call *deboost_priority()* to do deboost priority. Like boost priority, this operation still calls VMCALL to do context switch.

In VMM side, VMCALL leads to VM Exit, and requirement enters into VMM. Indeed, *kvm_emulate_hypercall* function in *arch/x86/kvm/x86.c* is used to cope with this VM Exit. Here, We add two hypercalls *hc_boost(int cpu, int policy, int priority)* and *hc_deboost(int cpu)* to fulfill the requirements from guest OS. *hc_boost(int cpu, int policy, int priority)* function is depicted as following:

```
int hc_boost_vcpu (struct kvm_vcpu *vcpu, int policy,
                   int priority)
{
                   int err=0;
                   struct sched_param param;
                   raw_spin_lock(&vcpu->sched_lock);
                   param.sched_priority = vcpu->max_rt_prio;
                   vcpu->curr_policy = policy;
if (!vcpu->prio_boost) {
                              err = sched_setscheduler(vcpu->task,
                                                  policy, &param);
                   }
                   raw_spin_unlock(&vcpu->sched_lock);
                   return err;
}
```

Actually, the essence of co-scheduling is to dynamically adjust corresponding VCPU priority according to scheduling information from guest OS. That is, in RTOS, if an important task should be processed, VMM will change its scheduling policy as real-time scheduling policy (boosting its priority). When this task is finished, guest OS will again inform VMM to do deboost operation. As pointed in [10], enabling paravirtual scheduling won't have any negative impacts on worst-case and average latency. In this way, by using co-scheduling, the VCPU on which RTOS resides on is deboosted when RTOS is not executing interrupt handler. Therefore more physical CPU resource is available for other computing tasks and thus CPU utilization is improved. Compared to static boosting priority method – prioritization tuning in section 3.2.2, co-scheduling will lead better CPU utilization.

5. EXPERIMENT

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 CPU shielding, prioritization and co-scheduling. We first introduce the experimental setup used for the evaluation and then present the results of our experiments.

5.1 Experiment Setup

Our experimental platform consisted of an Intel quad-core processor (Core2 i5 750 at 2.67GHz) and 2GB RAM, but we only used two processor cores for all tests. 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 with RT patch 2.6.33.4-rt20 [21]. In addition, we disabled *CONFIG_ ACPI_PROCESSOR* to prevent response time from being interfered by processor power management service (The reason will be analyzed in detail). 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 [22] 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 [23] 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) 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 [25]

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. Finally, we tested co-scheduling from two sides: interrupt response latency in real-time VM and CPU resource utilization rate with computational and I/O loads.

5.2 System Management Interrupt

When CPU gets a system management interrupt (SMI), it goes into a special mode called System Management Mode (SMM) and jumps to a hard-wired location in a special SMM address space (which is probably in BIOS ROM). SMM is a special purpose operating mode provided for handling system-wide functions like system hardware control. It is intended to be used only by system firmware instead of applications software or general-purpose system software. The only way to enter SMM is by means of SMI. When SMM is invoked, the processor saves the processor's context and switches to a separate operating environment contained in system management RAM. The processor executes SMI handler code to perform some specific operations. When it finishes its operations, a resume instruction is executed to switch back to the previous processor's context, and execute the interrupted program [14].

SMIs are harmful to real-time system. They can last for hundreds of microseconds or even more than 1 millisecond in our test, which is unacceptable for many real-time applications. Furthermore, SMIs are the highest priority in the system and you cannot interrupt them because they don't have a vector in the CPU. And finally, they cannot be disabled globally [26]. There is no better solution for this problem yet. The best solution is to contact the hardware manufacture to disable SMI or minimize its affection on latency. In our experiment result, we find out that the USB legacy devices are the main factor that causes SMIs. It's a good idea to disable the USB legacy option in BIOS and *CONFIG_ACPI_PROCESSOR*, and use PS/2 mouse and keyboard instead of USB.

5.3 Results: Linux PDLT

We first measured latencies natively by running cyclictest on the host Linux. Fig. 4 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. 5. It shows a right shift of the latency distribution as opposed to Fig. 4. Accordingly, 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. Figs. 6 and 7 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.

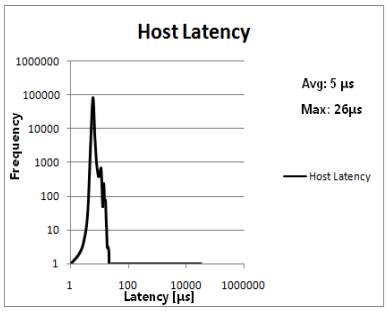

Fig. 4. Latency of host Linux.

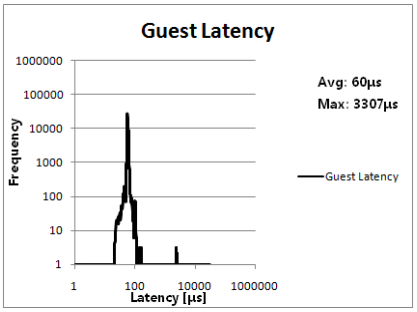

Fig. 5. Latency of guest Linux.

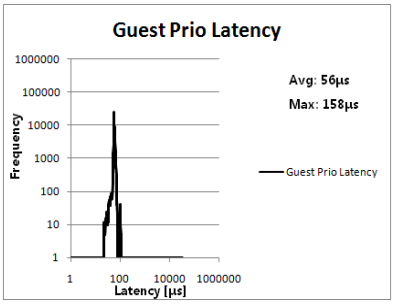

Fig. 6. Latency of guest Linux with prioritization tuning.

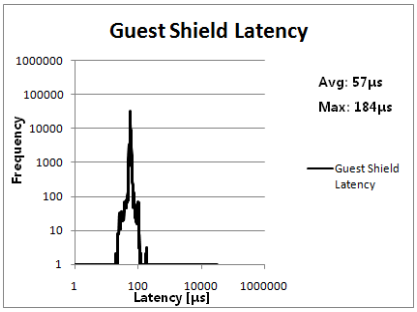

Fig. 7. Latency of guest Linux with CPU shielding tuning.

5.4 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. 8 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 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. Moreover, common interrupts, like disk and network, were affinitized to the first processor core. The result is depicted in Fig. 9, where "Base-pin" represents VxWorks with CPU shielding tuning, while "Load-pin" represents Vxworks running load mentioned. It shows that CPU shielding provides the same level of enhancement as prioritization. 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. 10. It shows that the maximum IRT decreases to about 100us in both cases.

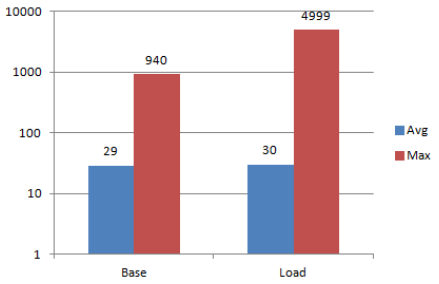

Fig. 8. Latency of guest VxWorks.

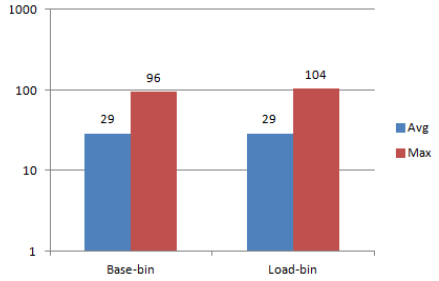

Fig. 9. Latency of VxWorks latency with CPU shielding tuning.

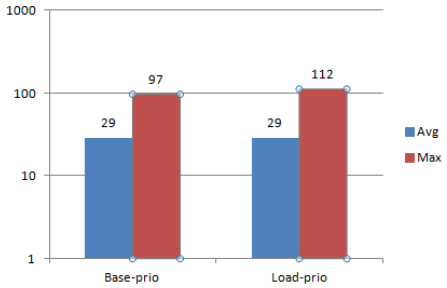

Fig. 10. Latency of VxWorks latency with prioritization tuning

The results in Figs. 9 and 10 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. In addition, we can see that more cycles of CPU will be lost with prioritization tuning in RTOS, because RT task is periodic and occupies CPU all the time even if it is finished.

5.5 Results: Co-scheduling

To promise interrupt response time and improve cpu resource utilization rate, co-scheduling is employed when system running with prioritization tuning. Compared to static prioritization tuning, co-scheduling doesn't influence real-time responsiveness too much, but achieving better system throughput. Here, with co-scheduling, we test interrupt response time of RTOS and CPU resource utilization rate. We select several computing-intensive programs – 164.gzip, 176.gcc, 197.parser, 255.vortex, 256.bzip2 and 300.twolf – from SPEC2000 [21], and run cyclictest along with CPU load in the RTOS. The system performance with and without co-scheduling is showed in Fig. 11.

Here, we measure how fast it compete a single task. With CPU load and I/O load, Fig. 11 shows that executing general-purpose applications by this embedded virtualization system with co-scheduling is much faster (even more than two time) than that with prioritization tuning. This dynamic scheduling method makes CPU busy all the time,

rather than idle intermittently. In addition, we not only achieve performance improvement but also get some acceptable results about real-time responsiveness showed in Fig. 12. The average latency of real-time responsiveness is not changed, but the worst-case latency is increased from 112 to 130. That is because real-time tasks re-occupying CPU resource also needs spending some time, which is occupied by general-purpose applications. Another reason is that two extra VM-Exit operations cause higher response time. Though the max latency is increased a little, this can be accepted in some soft real-time environment, like smartphone.

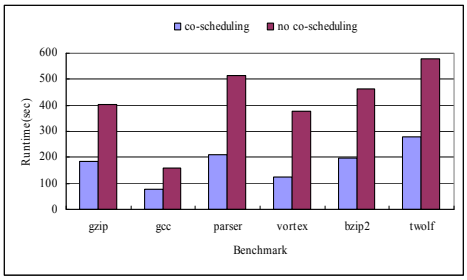

Fig. 11. The performance of embedded virtualization system with co-scheduling.

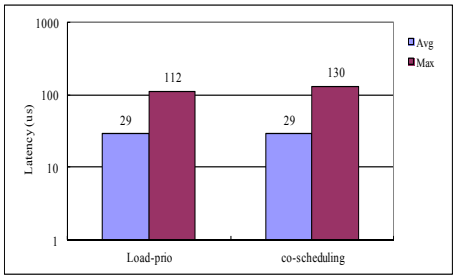

Fig. 12. Latency of VxWorks with and without co-scheduling tuning.

6. CONCLUSIONS

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 addition, we proposed co-scheduling that not only promises real-time responsiveness but also brings system improvement. 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.

REFERENCES

1. http://www.intel.com/pressroom/archive/releases/2010/20100304comp.htm.
2. http://rtcmagazine.com/.
3. http://www.alphagalileo.org, August, 2010.
4. A. Kivity, Y. Kamay, and D. Laor, "KVM: The Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, 2007, pp. 225-230.
5. S. Yoo, Y. Liu, C. Hong, C. Yoo, and Y. G. Zhang, "MobiVMM: a VirtualMachine monitor for mobile phones," in *Proceedings of the 1st Workshop on Virtualization in Mobile Computing*, 2008, pp. 1-5.
6. http://www.real_time_systems.com.
7. http://www.osware.com.

8. http://www.windriver.com/products/platforms/real_time_core/.
9. R. Kaiser, "Alternatives for scheduling virtual machines in real-time embedded systems," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, 2008, pp. 5-10.
10. J. Kiszka, "Towards Linux as a real-time hypervisor," in *Proceedings of the 11th Real-Time Linux Workshop*, 2009, pp. 205-215.
11. W. Jiang, Y. S. Zhou, Y. Cui, W. Feng, Y. Chen, Y. C. Shi, and Q. B. Wu, "CFS optimizations to KVM threads on multi-core environment," in *Proceedings of the 15th International Conference on Parallel and Distributed Systems*, 2009, pp. 348-354.
12. J. Zhang, K. Chen, B. Zuo, R. Ma, Y. Dong, and H. Guan, "Performance analysis towards a KVM-based embedded real-time virtualization architecture," in *Proceedings of the 5th International Conference on Computer Sciences and Convergence Information Technology*, 2010, pp. 421-426.
13. B. Zuo, K. Chen, A. Liang, H. Guan, J. Zhang, R. Ma, and H. Yang, "Performance tuning towards a KVM-based low latency virtualization system," in *Proceedings of the 2nd International Conference on Information Engineering and Computer Science*, 2010, pp. 1-4.
14. Intel Corporation, "Intelr 64 and IA-32 architectures software developer's manual," Vol. 3B, 2010, pp. 3-8.
15. P. McKenney, "Real time vs. real fast: How to choose," in *Proceedings of the 11th Real-Time Linux Workshop*, 2009, pp. 11-21.
16. H. Yoon, J. Song, and J. Lee, "Real-time performance analysis in Linux-based robotic systems," in *Proceedings of the 11th Linux Symposium*, 2009, pp. 331-340.
17. S. Brosky and S. Rotolo, "Shielded processors: Guaranteeing sub-millisecond response in standard Linux," in *Proceedings of the 17th International Symposium on Parallel and Distributed Real-Time Systems*, 2003, pp. 120.1.
18. G. Heiser, "Virtualizing embedded systems: why bother?" in *Proceedings of the 48th Design Automation Conference*, 2011, pp. 901-905.
19. G. Heiser, "The role of virtualization in embedded systems," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, 2008, pp. 11-16.
20. http://www.xen.org/files/xensummitboston08/LHP.pdf.
21. Real-Time Linux Wiki, 2010, http://rt.wiki.kernel.org.
22. http://www.linux-kvm.org/page/Main_page.
23. http://wiki.linuxquestions.org/wiki/Bonnie.
24. H. Mitake, Y. Kinebuchi, A. Courbot, and T. Nakajima, "Coexisting real-time OS and general purpose OS on an embedded virtualization layer for a multicore processor," in *Proceedings of ACM Symposium on Applied Computing*, 2011, pp. 629-630.
25. http://git.kernel.org/?p=linux/kernel/git/tglx/rt-tests.git.
26. https://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application.

Ruhui Ma (马汝辉) received his Ph.D. degree in computer science from Shanghai Jiao Tong University, China, in 2011. He received the B.S. and M.S. degree at School of Information and Engineering from Jiangnan University in 2006 and 2008, China, respectively. He is currently an Assistant Professor with the Faculty of Computer Science, Shanghai Jiao Tong University (Shanghai, China). His main research interests include in virtual machines, computer architecture and compiling.

Fanfu Zhou (周凡夫) is currently a Ph.D. candidate at Shanghai Jiao Tong University, China. He received the B.S. in Computer Technology from Shanghai Jiao Tong University, M.S. degree in Computer Science and Technology from Huazhong University of Science and Technology. His main research interests include virtual machines, cloud computing.

Erzhou Zhu (朱二周) is currently a Lector at Anhui University, Hefei, China. He received the Ph.D. degree in Computer System and Architecture of Shanghai Jiaotong University (Shanghai, China) in 2012, received the M.S. degree and B.S. degree in Computer Science and Technology of Anhui University (Anhui, China) in 2004 and 2008 respectively. His research interests include virtual machine, binary translation and computer architecture.

Haibing Guan (管海兵) received his Ph.D. degree in Computer Science from the TongJi University (China), in 1999. He is currently a Professor with the Faculty of Computer Science, Shanghai Jiao Tong University (Shanghai, China). His current research interests include, but are not limited to, computer architecture, compiling, virtualization andhardware/software co-design.