IPSJ Transactions on Advanced Computing Systems Vol. 2 No. 3 116–130 (Sep. 2009)

Regular Paper

Using a Virtual Machine Monitor to Slow Down CPU Speed for Embedded Time-Sensitive Software Testing

Tetsuya Yoshida,[†1] Hiroshi Yamada[†1,†2] and Kenji Kono[†1,†2]

The use of time-sensitive software has been popular for embedded systems like mobile phones and portable video players. Embedded software is usually developed in parallel with real hardware devices due to a tight time-to-market constraint, and therefore it is quite difficult to verify the sensory responsiveness of time-sensitive applications such as GUIs and multimedia players. To verify the responsiveness, it is useful for developers to observe the software's behavior in a test environment in which the software runs in real time rather than in simulation time. To provide such a test environment, we need a mechanism that slows down the CPU speed of test machines because test machines are usually equipped with high-end desktop CPUs. A CPU slowdown mechanism needs to provide various CPU speeds, keep a constant CPU speed in the short term, and be sensitive toward hardware interrupts. Although there are a couple of ways of slowing down CPU speed, they do not satisfy all the above requirements. This paper describes *FoxyLargo* that smoothly slows down CPU speed with a virtual machine monitor (VMM). FoxyLargo carefully schedules a virtual machine (VM) to provide an illusion that the VM is running slowly from the viewpoint of time-sensitive applications. For this purpose, FoxyLargo combines three techniques: 1) fine-grained, 2) interrupt-sensitive, and 3) clock-tick based VM scheduling. We applied our techniques to Xen VMM, and conducted three experiments. The experimental results show that FoxyLargo adequately meet all the above requirements. Also, we successfully reproduced the decoding behavior of an MPEG player. This result demonstrates that FoxyLargo can reproduce the behavior of real applications.

1. Introduction

Modern mobile devices such as mobile phones, audio/video players, and game players are all equipped with *time-sensitive* applications. For example, mobile phones such as the iPhone have sophisticated graphical user interfaces (GUIs), whose responsiveness determines the usefulness of the device. Mobile audio or video players are naturally time-sensitive; if an audio or a video player is not scheduled in a timely fashion, the quality of the audio or video deteriorates. Since mobile devices are equipped with low-power, slow processors, to meet their timing constraints, time-sensitive applications must be implemented carefully. We use the term *time-sensitiveness* to denote interactive applications as well as soft real-time applications.

To meet a tight time-to-market constraint, embedded software is usually developed in parallel with real hardware devices, and is cross-developed on ordinary PCs. This process of development complicates the debugging of embedded software. In particular, it is quite difficult to verify the *sensory* responsiveness of time-sensitive applications. Even if time-sensitive applications run smoothly on full-fledged desktop PCs, we cannot conclude that they would also do so on low-power, slow processors embedded in mobile devices; deadlines may be missed on slow processors. The sensory quality of time-sensitive software is best tested on the hardware running at the same speed as the target one.

To bridge the gap between the development and real hardware environments, simulation- or emulation-based approaches are employed. For example, *virtual platforms* [3),19)] enable us to test embedded software at various levels of accuracy from cycle- to functionality-level. The developers carefully choose an appropriate level of simulation and/or emulation, and design each virtual component of an embedded system [10)]. We present a technique that smoothly slows down CPU speed in order to test the sensory responsiveness of time-sensitive applications. With this technique, developers can use full-fledged desktop PCs to develop and test time-sensitive applications; they can test the responsiveness with their applications executed at the speed of target CPUs because the speed of the CPU used in the test can be arbitrarily degraded. Strictly speaking, we also have to address other factors such as the difference in cache, bus, and peripheral devices to reproduce the behavior of embedded systems more precisely. However, as the first step toward the complete emulation of embedded systems, this paper focuses on slowing down CPU cycles.

Slowing down CPU speed is not easy, and it poses several challenging issues.

†1 Keio University
†2 CREST(JST)

First, we have to be able to slow down CPU speed to arbitrary levels. The lineup of embedded CPUs is superfluous and their clock cycles vary from 1 MHz to 1 GHz [9]. Therefore, we need a mechanism that slows down high-end CPUs for desktops to arbitrary clock cycles. Although dynamic voltage scaling (DVS)[17] provides a hardware-level mechanism of slowing down CPU speed, DVS is limited to simply setting several levels of CPU speed. For example, SpeedStep [11], DVS from Intel, allows eight levels of CPU speed to be used.

Second, applications should consume CPU cycles continuously and run at a constant speed. Imagine that we are slowing down CPU speed with a traditional virtual machine monitor (VMM)[5]. Since a VMM controls the resources, including CPU cycles, allocated to each virtual machine (VM), the VMM, if it gives each VM only a small portion of the entire CPU cycles, provides the illusion that the VMs are running slowly. If a time slice given to each VM is coarse-grained (for instance, 30 msec), a time-sensitive application in a VM runs at the full speed for 30 msec and then suddenly stops for a long time (30 msec or more). That is, time does not flow continuously from the viewpoint of the time-sensitive application; a video would not be played smoothly if an MPEG player is executed. Since time-sensitive applications require finer time resolutions, traditional VMMs do not achieve a continuous time flow.

Third, we have to carefully take hardware interrupts into account. The speed of handling hardware interrupts is slowed down as the effect of slowing down the CPU speed. However, the delivery of hardware interrupts should not be delayed since time-sensitive applications are sensitive to hardware events such as keyboard inputs and mouse clicks. The term *interrupt sensitiveness* means that the mechanism of slowing down CPU speed should be sensitive to hardware interrupts.

We present *FoxyLargo*, a mechanism of slowing down CPU speed by using a VMM. FoxyLargo adequately meets the three requirements previously listed. Since FoxyLargo is tailored to software testing, we assume that there is only one VM running on it. FoxyLargo combines three techniques to slow down the CPU speed: 1) fine-grained VM scheduling, 2) interrupt-sensitive VM scheduling, and 3) clock-tick based VM scheduling. By combining these techniques, FoxyLargo provides an illusion that the VM is running slowly from the viewpoint of time-sensitive applications. We incorporated our techniques into a Xen virtual machine monitor [2], and demonstrated that FoxyLargo smoothly slows down the speed of the CPU. FoxyLargo reproduced behaviors very similar to those produced by DVS-based slowdown. In particular, FoxyLargo reproduced the behavior of decoding MPEG video frames.

The rest of the paper is organized as follows. The issues related to slowing down CPU speed are described in Section 2. In Section 3 we present our system design, and Section 4 describes the implementation. The experimental results are shown in Section 5 and Section 6 describes work related to ours. We conclude the paper in Section 7.

2. Issues on CPU Slowdown Mechanisms

Our goal is to verify the sensory responsiveness of time-sensitive applications such as GUIs and audio/video players. To verify the sensory responsiveness, these applications should be tested in real time rather than in simulation time. Thus, they are best tested on real hardware devices. However, the developers must test them *before* the devices become available. In this paper, we explore the possibility of producing an illusion that a high-end CPU is running at a slower speed, at least, from the viewpoints of time-sensitive (i.e., GUI and multimedia) applications. To clarify the requirements for CPU slowdown mechanisms, we briefly describe the existing mechanisms in this section.

2.1 Dynamic Voltage Scaling

Dynamic voltage scaling (DVS) is a mechanism directly provided by the underlying CPU to change its voltage and corresponding frequency during runtime. DVS technology is widely used in several CPU vendors [1],[11]. For instance, Intel CPUs support DVS called *SpeedStep*. In SpeedStep, the users can control the CPU speed by changing the value of a special-purpose register, which specifies the rate of clock signals. By adjusting CPU cycles, DVS allows us to produce several CPU frequencies.

Although DVS technology can precisely produce a target CPU frequency, the range of CPU frequencies DVS can produce is strictly fixed. Thus, we cannot freely use various CPU frequencies by DVS. In fact, Intel SpeedStep provides only eight choices, and this limitation is significantly critical in terms of develop-

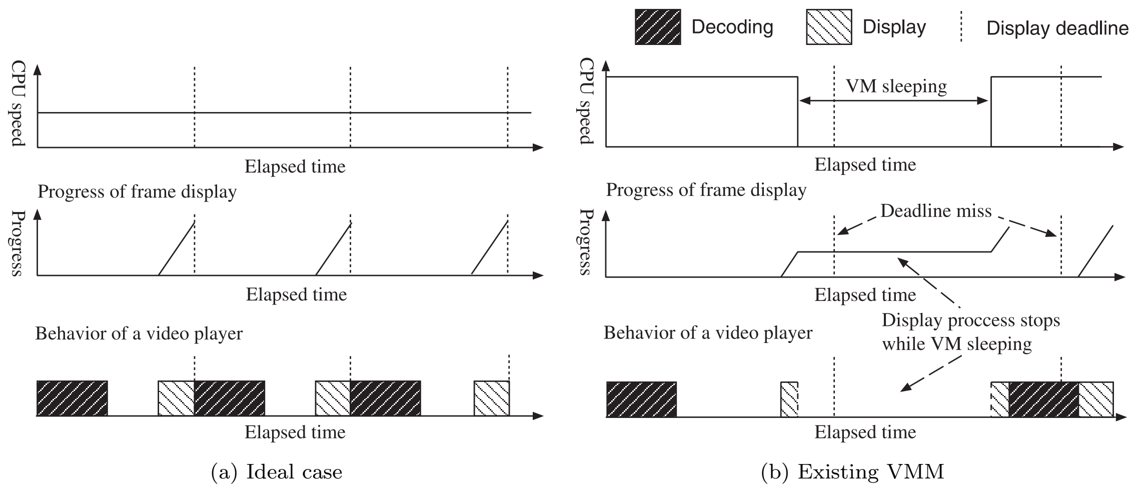


Fig. 1 Comparison of periodic task processing. The display process of a video frame is given a deadline to display the frame on time. However, a video player running on an existing VMM could cause the deadline misses of frame display because of the coarse-grained VM scheduling.

ment in embedded time-sensitive applications. For example, the ARM processor, which is one of the most widespread embedded CPUs, has more than 20 types of frequencies, and the frequencies of them are from 1 MHz to 1 GHz.[9]. To test embedded time-sensitive applications, a CPU slowdown mechanism should be able to produce many more levels of CPU frequencies than DVS.

2.2 Existing Virtual Machine Monitors

In virtual machine environments, a virtual machine monitor (VMM) is a primary software layer that is used to manage physical resources and assign them to running virtual machines (VMs). By controlling CPU cycles assigned to a VM, the VMM can emulate a slow CPU environment on it. For example, a traditional VMM such as Xen [2] assigns CPU times to each VM on a proportional-sharing basis. If we configure CPU usage of a VM to 50%, the VM runs at 50% of the speed of the underlying CPU. By adjusting the CPU usage of a VM, an existing VMM can emulate various CPU frequencies.

Although existing VMMs allow us to slow down CPU speed, they are not suitable for our purpose for the following reasons. First, existing VMMs apportion CPU cycles in a coarse-grained fashion; a VM is given a long time slice (for instance, 30 msec). Thus, a time-sensitive application in a VM runs at the full speed of the underlying CPU for a given time slice and suddenly stops for a longer time, waiting for the next time slice. This behavior of a VM makes it almost impossible to verify the responsiveness of time-sensitive applications, because periodic tasks in time-sensitive applications should be scheduled to meet their timing constraints.

For example, a video player typically decodes and displays 30 frames per second. So, it should be scheduled every 33 msec to display each frame in time. Suppose that we are trying to verify the behavior of this video player in a VM on Xen and set the CPU share of the VM to 50% to emulate 50% speed of the real CPU (**Fig. 1**). The minimum interval of VM scheduling in Xen is 30 msec. Figure 1 (a) shows the behavior of the video player running on a real machine. All display processes meet their deadlines. In contrast, Fig. 1 (b) shows the behavior of the video player running on Xen. Since the VM is suspended in the middle of displaying the first video frame, the display process misses the deadline. The second frame is not displayed in time either. Like this, video players running on an existing VMM potentially cause deadline misses. We therefore cannot distinguish deadline misses caused by a bug in the video player from those caused by inappropriate VM scheduling.

Next, traditional VMMs cannot verify the interactiveness of GUIs. VMs should be notified of GUI events such as key inputs and mouse clicks as soon as possible. However, in a traditional VMM, hardware interrupts are delivered to the target VMs only when they are running. If a target VM is suspended, the VMM delays the delivery until it wakes up. Thus, the delivery of a hardware interrupt may be delayed for a considerable amount of time. If we set the CPU share of a VM on Xen to one-fifth, the maximum delay of interrupt delivery is 120 msec (the length of four time slices). This delay is critical to verifying the interactiveness of GUIs.

3. FoxyLargo

In this paper, we present *FoxyLargo*, which smoothly slows down CPU speed. To slow down CPU speed for testing embedded time-sensitive applications, we explore the possibility of using the virtual machine technology. As described above, because existing VMMs manage underlying resources in a coarse-grained fashion, time-sensitive applications running on VMs behave very differently from those running on the real CPU. To overcome this drawback, FoxyLargo manages a VM much more precisely than conventional VMMs, and thus produces a target CPU frequency so that we can verify the sensory responsiveness of time-sensitive applications. In this section, we elaborate on our approach to overcome the drawbacks of existing VMMs.

There is one thing to be noted. We assume that FoxyLargo provides only one VM on top of it. FoxyLargo forges a slow CPU for a single VM devoted to software testing. This assumption is reasonable since FoxyLargo is tailored to software testing.

3.1 Fine-Grained VM Scheduling

Our goal is to test embedded time-sensitive applications, which process their computational tasks once per a period of time. We refer to the length of this period as T. In most of our target applications, T is longer than 10 msec. For example, a mobile game displays 60 frames per second; T is about 16 msec. A video player performs 30 frames per second at best; T is about 33 msec. Soft modems wake every 16 msec to execute modem functions [12].

To provide an illusion that time-sensitive applications are running at the targeted CPU speed, FoxyLargo carefully lowers CPU throughput of the VM so that it becomes the throughput of the target CPU, if averaged over period T. We consider the throughput over T because time-sensitive applications are periodic and execute their own tasks every T. By doing so, the amount of work that a time-sensitive application can do during period T becomes almost equivalent to the amount that would be possible on the real slow CPU. As a result, FoxyLargo can reproduce the behavior of time-sensitive applications. Suppose that a video player whose T is 33 msec is executed at the speed one-third of the real CPU. If the video player is assigned CPU time in a way that the CPU throughput over 33 msec becomes one-third to the real CPU, the video player behaves similarly on the real slow CPU because it is assigned CPU time every 33 ms.

To lower the CPU throughput over T, FoxyLargo schedules the VM every t_{period}, which is much shorter than T. Based on the targeted CPU frequency, FoxyLargo adjusts the length of the time slice assigned to the VM. The length of the time slice, refereed to as t_{vm}, is defined as follows. S_{target} is the targeted CPU speed and S_{real} is the speed of the underlying real CPU.

$$t_{vm} = t_{period} \cdot \frac{S_{target}}{S_{real}}$$

Suppose that FoxyLargo reduces the CPU speed of a VM to 50% of the the underlying CPU. **Figure 2** shows the behavior of a video player running on the VM, and this video player decodes and displays a video frame every 33 msec.

In each period, the CPU throughput on FoxyLargo almost equals that on the target real machine shown in Fig. 1(a). Therefore, the behavior of the video player on FoxyLargo is similar to the behavior on the target machine.

3.2 Interrupt-Sensitive VM Scheduling

In terms of time-sensitive applications, we need to take into account hardware interrupts from peripheral devices. As described in Section 2.2, time-sensitive applications are sensitive to hardware events such as keyboard inputs and mouse clicks. If hardware interrupts are not managed properly, FoxyLargo causes the de-

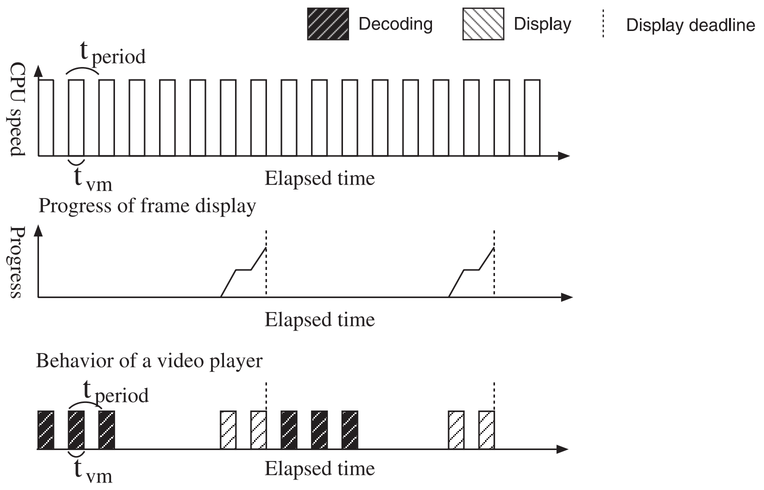


Fig. 2 Periodic task processing on FoxyLargo. FoxyLargo schedules the VM for a short period to enable the video player to process periodic tasks on time.

lays of interrupt notifications to the destination VM. At worst, when a hardware event occurs just after the VM has exhausted its own time slice, the notification delay becomes $t_{period} - t_{vm}$.

To minimize the delay of the notification, FoxyLargo sends a hardware interrupt to the destination VM just after it has occurred. If the VM has been suspended when an interrupt occurs, FoxyLargo immediately resumes the VM, and forces it to handle the hardware event.

FoxyLargo runs the resumed VM for a given time shorter than the VM scheduling period. We refer to the time given to the resumed VM as t_{resume}. The value of t_{resume} is a parameter, and we set it up as 100 μsec for our prototype to enable time-sensitive applications to receive a hardware event during the time.

Strictly speaking, with the above method the CPU frequency forged by FoxyLargo becomes inconstant, because the suspended VM incrementally gains CPU time when a hardware event occurs. Hence, FoxyLargo might produce the CPU throughput different from that of the targeted CPU, and we cannot correctly observe the responsiveness of time-sensitive applications. To mitigate this side-effect, when a hardware interrupt occurs, FoxyLargo borrows the CPU time from the next time slice and assigns it to the VM. By adjusting the length of the next time slice, FoxyLargo keeps producing the CPU throughput similar to that of the targeted CPU.

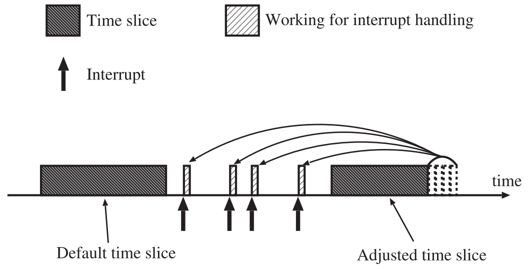


Fig. 3 Interrupt-sensitive VM scheduling. To handle hardware interrupts immediately, the destination VM is scheduled when an interrupt occurs. The length of the next time slice is reduced to compensate for the time of handling interrupts.

If hardware events occur n times while the VM is suspended, the next time slice becomes $t_{vm} - n \cdot t_{resume}$ (**Fig. 3**).

3.3 Clock-Tick Based VM Scheduling

To reproduce the behavior of embedded time-sensitive software, we must pay careful attention to the time management of the guest OS running on top of FoxyLargo. If the guest OS does not perform the time management correctly, the guest OS's process scheduler behaves differently from that running on the real embedded system. As a result, FoxyLargo cannot reproduce the applications' behavior since interactive applications running on it are scheduled differently from those running on the real system.

In general, general purpose OSes, which are commonly used as development environments for embedded applications, keep the system time and measure the length of time slices by counting periodic clock-ticks. Since FoxyLargo immediately notifies the VM of the clock-tick interrupts (due to the interrupt-sensitive VM scheduling), the guest OS can count clock-ticks at the correct timing.

However, counting clock-ticks on time is insufficient for the guest OS to perform the time management correctly. Unless we do not take clock-ticks into account to define t_{period} (the time period of the VM scheduling), the guest OS's process scheduler behaves differently from that of real embedded systems. We explain

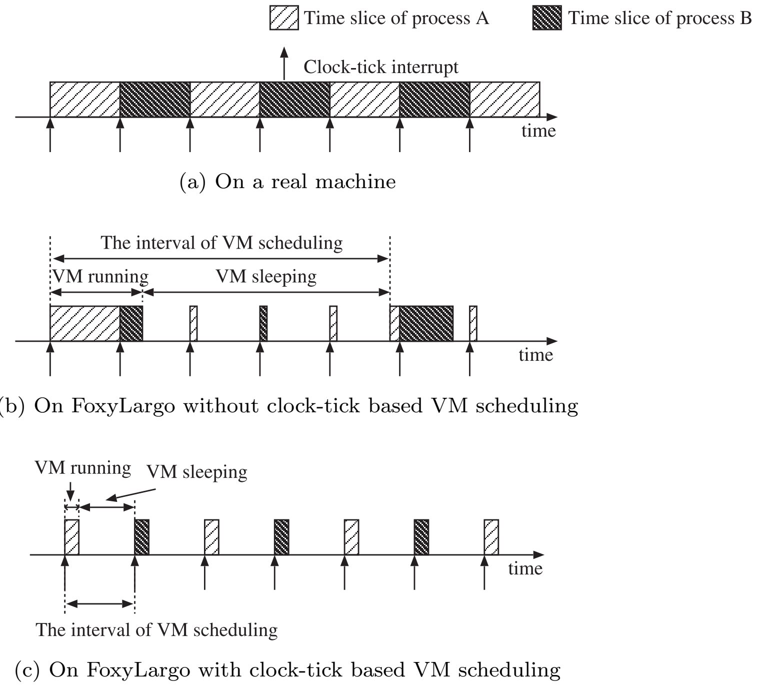

Time slice of process A
Time slice of process B
Clock-tick interrupt
time
(a) On a real machine
The interval of VM scheduling
VM running
VM sleeping
time
(b) On FoxyLargo without clock-tick based VM scheduling
VM running
VM sleeping
time
The interval of VM scheduling
(c) On FoxyLargo with clock-tick based VM scheduling

Fig. 4 Comparison of process scheduling in a guest OS. Interrupt-sensitive scheduling has an adverse effect on the scheduling in a guest OS, if a VM is not given the entire time slice in accordance with clock ticks.

how differently the scheduler behaves if we vary the VM scheduling period with **Fig. 4**. In Fig. 4 (a), the OS runs directly on a real machine. The processes run for the same time in this case.

On the other hand, the run times of the processes are different in Fig. 4 (b). In this case, the OS runs on FoxyLargo where the period of the VM scheduling is longer than that of clock-ticks. The third clock-tick interrupt occurs when the VM is not scheduled. In order to notify the VM of the interrupt immediately, FoxyLargo wakes up the VM. Then, the process scheduler schedules process A. Since FoxyLargo suspends the VM when the VM finishes the interrupt handling, process A is also suspended. At the fourth clock-tick interrupt, the process scheduler dose not schedule process A but B although process A's time slice remains. It is because the scheduler recognizes the time flow by only counting the clock-tick interrupts; the scheduler considers that process A runs out of its time slice when it counts the fourth interrupt. In contrast, process A is running between the first and second interrupts without being suspended, since these interrupts occur while the VM is running. As mentioned above, the length of processes' run time between each clock-tick varies if the period of the VM scheduling is longer than that of clock-ticks.

In response to the issue, FoxyLargo assigns CPU time to the VM every clock-tick interrupt; i.e., t_{period} is equivalent to that of clock-ticks. By doing so, processes run for the same time between each clock-tick as shown in Fig. 4 (c).

In addition, as we suggest in Section 3.1, FoxyLargo needs to schedule the VM in a shorter interval than that of the periodic tasks of time sensitive software. Scheduling every clock-tick interrupt meets this requirement. For example, the clock-tick interval is 1 msec in Linux 2.6, whereas the display period of video players is ordinarily 33 msec.

4. Implementation

We have implemented FoxyLargo in Xen[2] VMM version 3.0.4-1. Xen is an open source VMM for the Intel x86 architecture. Xen provides a paravirtualized[21] processor interface, which reduces the virtualization overhead at the expense of porting guest OSes. We carefully avoided making use of this feature of Xen when implementing our mechanism. Thus, our mechanism can be applied to a more conventional VMM such as VMware[16),20)].

For ease of implementation, our current prototype does not fully support interrupt-sensitiveness. Our prototype supports the sensitiveness only for clock-ticks and NICs. This limitation is *not* caused by the essential difficulty in implementing interrupt-sensitiveness. In fact, we are currently working on other interrupts to support the sensitiveness. To implement our CPU slowdown mechanism, we modified the Xen VM scheduler, Xen VMM's interrupt handlers, and Xen network backend driver. This modification consists of about 450 lines of code.

5. Experiments

To demonstrate that our approach works well, we conducted three experiments. All machines used in the experiments are equipped with a 2.8 GHz Pentium 4 PC with 512 MB of RAM and an SCSI hard disk drive. To compare our approach with the existing techniques described in Section 2, we conducted the experiments in the following three configurations.

In the first configuration, we ran Linux 2.6.22 on a machine and used SpeedStep to slow down the CPU speed of it. We refer to the configuration as *Native Linux*. We assume that CPU speeds are ideally slowed down in this configuration, and regard the experimental results of this configuration as ideal cases (i.e., identical to the results on the real slow CPUs). In the second configuration, the original Xen 3.0.4-1 was used to slow down the CPU speed. One guest domain and the driver domain ran on Xen with Linux 2.6.16 executing in each domain. We assigned 256 MB of RAM to the guest domain, 200 MB to the driver domain. We refer to this configuration as *Original VMM*. In the last configuration, FoxyLargo was used to slow down the CPU speed. Since FoxyLargo is based on Xen, the configurations of the driver and guest domains are the same as those of Original VMM. If the behavior of time-sensitive applications in FoxyLargo is similar to that in Native Linux, we can say that FoxyLargo successfully produces desired CPU speeds and we can verify the sensory responsiveness of time-sensitive applications with it.

Note that Native Linux used Linux 2.6.22 whose version differs from Linux 2.6.16 used in Original VMM and FoxyLargo. This is because the latest version of Linux supported by Xen 3.0.4-1 is 2.6.16. In the beginning, we tried to use Linux 2.6.16 in Native Linux. But the device driver of SpeedStep for Linux 2.6.16 supports 3 choices (62.5%, 75%, 87.5% speed of the underlying CPU). To check the accuracy of the CPU speeds produced by our approach in as many cases as possible, we chose Linux 2.6.22 whose driver supports all the choices provided by SpeedStep (the multiples of 12.5% speed of the underlying CPU).

5.1 Microbenchmark

5.1.1 Effectiveness of Clock-Tick Based VM Scheduling

To confirm that FoxyLargo produces various CPU speeds, we ran a CPU intensive benchmark in the three configurations, producing 12.5%, 25%, 37.5%, 50%, 62.5%, 75% speed of the underlying CPU. The benchmark executes a loop in which the system time is recorded every 1000 repetitions. We observed the progress of the benchmark in each configuration. We set the clock-tick interval to 10 msec.

Figure 5 shows a portion of the experimental results where the CPU speeds are 12.5%, 25%, 37.5% of the underlying CPU. The x-axis represents the elapsed time (msec), and the y-axis represents the number of loop executions in million unit. From this figure, we can see that FoxyLargo consumes CPU cycles constantly. In all the CPU speeds except for 12.5%, the gradients of the graphs of FoxyLargo are significantly similar to those of Native Linux. Later in this section, we discuss the reason FoxyLargo appears to fail to produce the 12.5% speed. The gradients of the graphs of Original VMM are not linear and different terribly from those of Native Linux, because the time periods of domain switching in Original VMM is much longer than those of FoxyLargo. In particular, when we produced a very slow CPU speed (12.5% speed of the underlying CPU), the influence of the long periods appeared very clearly in the result; the progress of the benchmark stopped for up to about 150 msec.

Using the same benchmark, we demonstrate that FoxyLargo can produce CPU speeds unavailable in SpeedStep. We used FoxyLargo to produce the CPU speeds of 6.25%, 18.75%, 31.25%, 43.75%, 56.25% and 68.75% of the underlying CPU. To confirm that FoxyLargo produces the target CPU speeds correctly, we compare the progress of loop execution; that is, the number of loops executed per millisecond. We used the method of linear least squares to calculate the progress. **Figure 6** plots the progress of FoxyLargo and Native Linux in various CPU speeds. The x-axis represents the ratio of the produced CPU speed to the underlying CPU speed (%), and the y-axis represents the progress (million loops per msec). This figure suggests that FoxyLargo makes progress very similar to that of Native Linux. The progress of FoxyLargo is almost identical to Native Linux between 25% and 62.5% speed rates. Although FoxyLargo makes slightly faster progress in speed rates over 62.5%, the progress is 5.1% faster than Native Linux in the worst case. When the CPU speed rate is 12.5%, FoxyLargo makes progress slower than Native Linux. However, we believe SpeedStep does not pro-

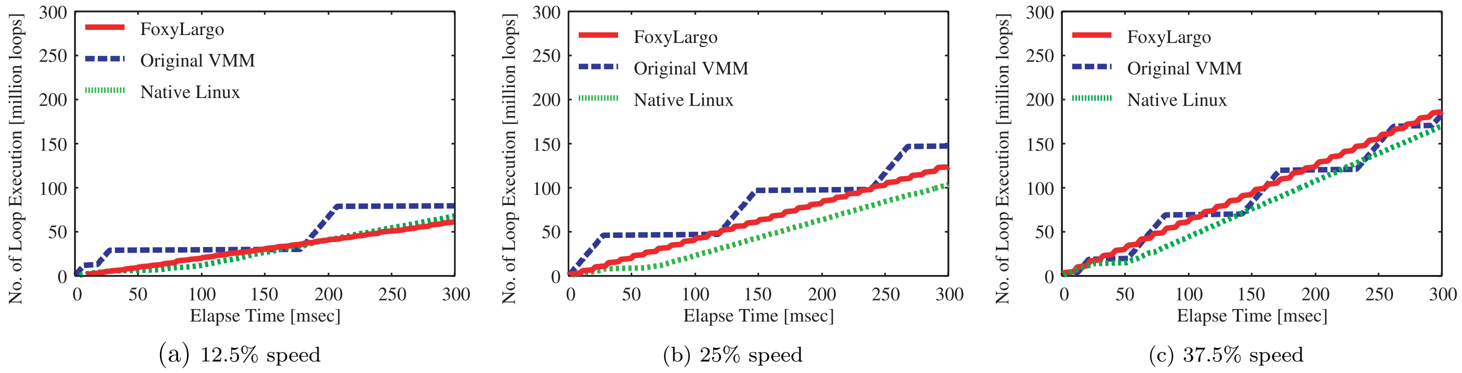


Fig. 5 Comparison of loop executions (CPU speeds supported by SpeedStep). FoxyLargo shows linear progress, which is very similar to that in Native Linux in every case, while Original VMM shows non-linear progress in all cases.

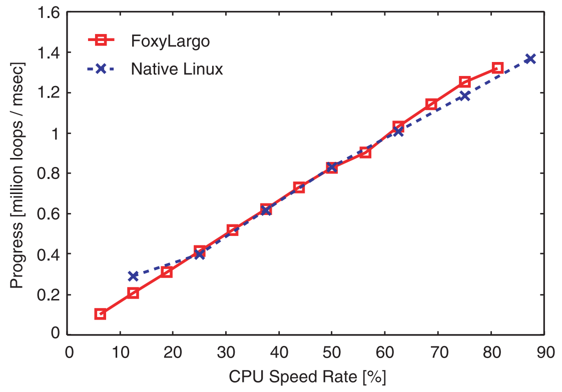


Fig. 6 Comparison of the progress of loop executions. The progress of FoxyLargo is close to that of Native Linux at every CPU speed rate.

duce the desired CPU speed rate appropriately, because the data point of Native Linux is not on the line extended from the line segment of Native Linux from 25% to 87.5%. On the other hand, the data points (6.25%, 12.5%, and 18.75%) of FoxyLargo are all on the extended line.

5.1.2 Interrupt-Sensitiveness of FoxyLargo

To demonstrate that FoxyLargo improves the interrupt-sensitiveness to network interrupts, we ran another benchmark to measure network response time. For comparison, we measured the response time on Native Linux, Original VMM, and FoxyLargo. In this benchmark we measured the response time of HEAD requests used in HTTP. This benchmark sends 50000 HEAD requests to a web server and records the response time of each request. We used an Apache web server (version 2.2.6) running on Linux 2.6.22. The client and the server machines are connected via a Gigabit Ethernet switch. We ran this benchmark in the three configurations for comparison, producing 12.5%, 25%, 37.5%, 50%, 62.5%, 75% speed of the underlying CPU. The clock-tick interval was 10 msec in this experiment.

Figure 7 shows a portion of the results where the CPU speed is 12.5% of the underlying CPU. Note that the scale of graphs for Original VMM is different from the scale for Native Linux and FoxyLargo; the former is 20 msec and the latter is 1 msec. The x-axis represents the number of HEAD requests, and the y-axis represents response time (msec).

This figure suggests two important things. First, FoxyLargo shows behavior that is very similar to Native Linux. The response times are less than 5 msec; most

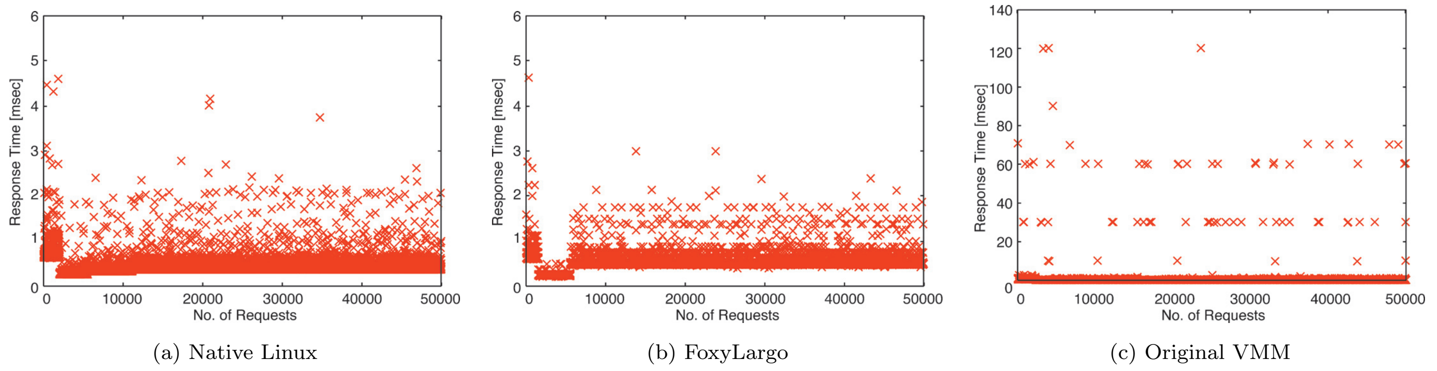


Fig. 7 Comparison of response time for HEAD requests (12.5% CPU speed). FoxyLargo recorded similar results to Native Linux. In the contrast, the default Xen sometimes recorded quite large response times compared to Native Linux.

of them are between 0.5 msec and 2 msec. FoxyLargo also showed similar results in the other CPU speed settings. Thus, we can say that FoxyLargo successfully improves the interrupt sensitiveness.

Second, Original VMM shows behavior that is quite different from that of Native Linux. The response times are up to 120 msec in Original VMM whereas they are up to 5 msec in Native Linux. Original VMM also showed similar results in the other CPU speed settings. This is because the guest domain of Original VMM is likely to be scheduled out for a long time, and thus network interrupts are tremendously delayed until the VM is scheduled again.

If we do not implement the interrupt sensitiveness, FoxyLargo behaves very differently from Native Linux. To confirm the usefulness of the interrupt-sensitiveness, we prepared FoxyLargo *without* interrupt sensitiveness and ran the same benchmark with the CPU speed ratio set to 12.5%. **Figure 8** shows the result. As shown in this figure, the FoxyLargo behavior is quite different from Fig. 7 (a) (and (b)). Almost all the response times are 10 msec, and the rest is more than 10 msec. This behavior is caused by the feature of Xen. In Xen a guest domain receives I/O requests from the driver domain. Thus, the guest domain must wait for the driver domain to be scheduled to receive the I/O requests. Since FoxyLargo, if interrupt sensitiveness is turned off, schedules guest domains every 10 msec in this experiment, the response time of HEAD requests are more than 10 msec.

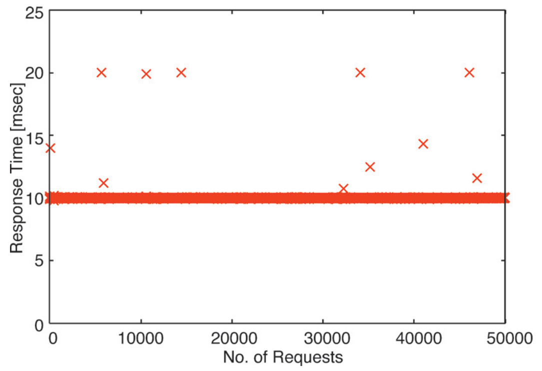
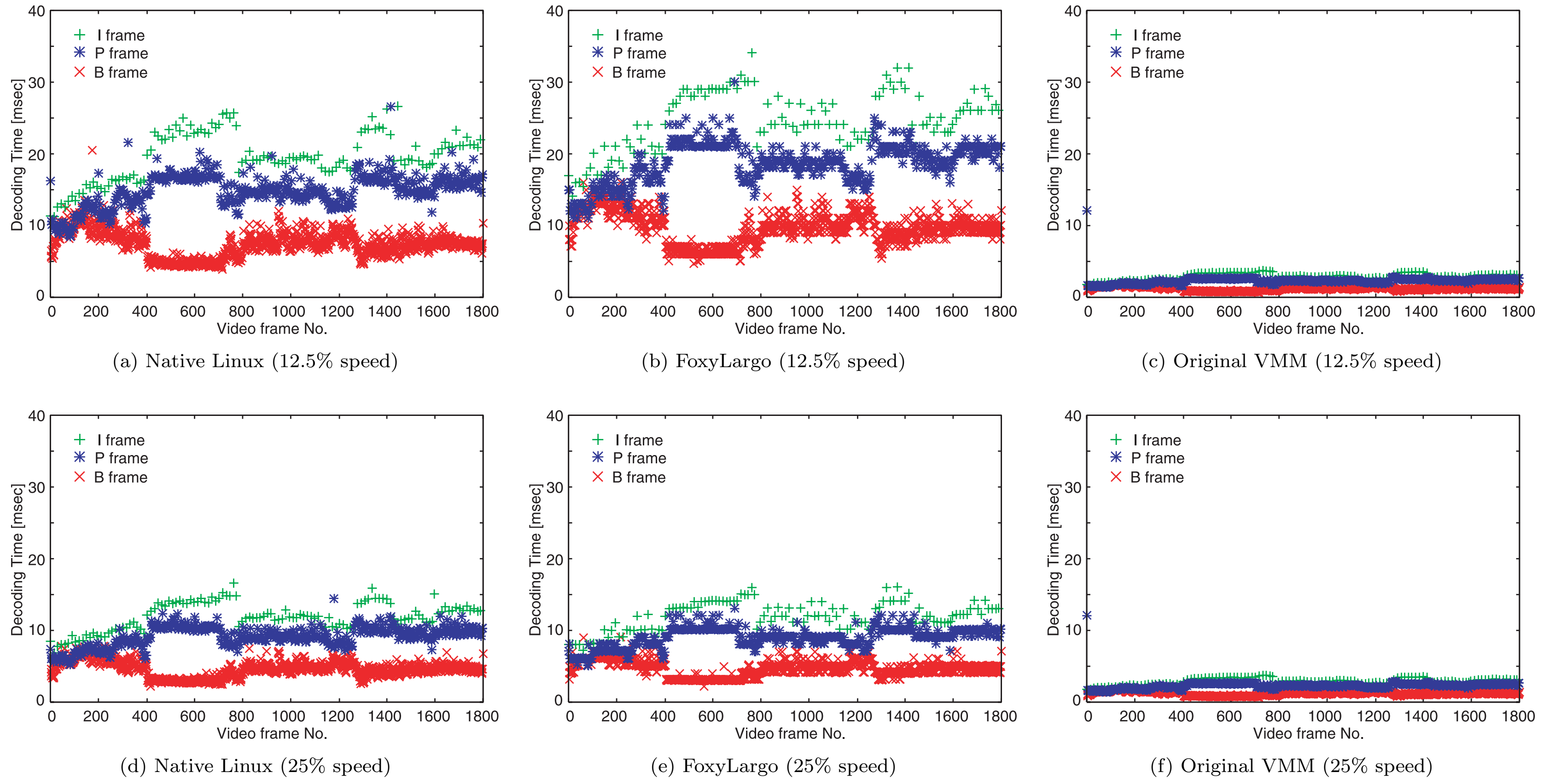


Fig. 8 Response time of HEAD requests on FoxyLargo without interrupt sensitiveness. Without interrupt sensitiveness, the response times are not less than 10 msec because of the mediation of the driver domain.

5.2 Macrobenchmark

To demonstrate that our approach is useful for real applications, we examined an MPEG-1 decoder named *mpeg play* [14]. We conducted the experiments in the three configurations for comparison, producing 12.5%, 25%, 37.5%, 50%, 62.5%, 75% speed of the underlying CPU. The clock-tick interval is set to 1 msec in



(a) Native Linux (12.5% speed) (b) FoxyLargo (12.5% speed) (c) Original VMM (12.5% speed)

(d) Native Linux (25% speed) (e) FoxyLargo (25% speed) (f) Original VMM (25% speed)

Fig. 9 Comparison of decoding time of MPEG video frames. The figures show the decoding time of each MPEG frame. FoxyLargo successfully reproduces behavior that is similar to Native Linux. On the other hand, Original VMM failed to do so.

order to schedule the VM in a finer-grained fashion. We used a video stream composed of 1800 frames.

5.2.1 Decoding Time

We measured decoding times of the video frames to demonstrate that Foxy-Largo reproduced the application's behavior in a fine-grained fashion. **Figure 9** shows a portion of the results where the CPU speeds are 12.5% and 25% of the underlying CPU. The x-axis represents the amount of video frames, and the y-axis represents the decoding time of each video frame (msec). Note that MPEG video streams consist of three types of frames, I frames (intra-picture), P frames (predicted picture) and B frames (bib-directional predicted picture)[4)]; I frames are complete images, P frames are composed of the differences from the preceding frame, B frames are composed of the differences from both the preceding and the following frames. We plotted three types of frames separately in Fig. 9 to show the decoding time of each frame type.

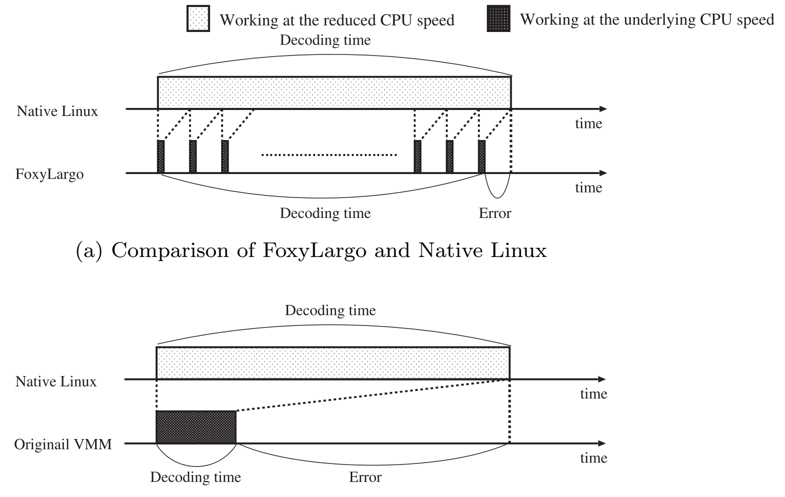
We can see two things from Fig. 9. First, FoxyLargo reproduces results that are very similar to those of Native Linux in all cases. The differences between the decoding times of FoxyLargo and Native Linux are between 0.149% and 28.2%, and the differences of the standard deviations are between −2.00% and 22.2%. In addition, FoxyLargo reproduces the decoding time distribution of each frame type. Decoding time of a frame depends on its frame type, and FoxyLargo gracefully reproduces a distribution similar to Native Linux; I frames are the most time-consuming, B frames are the fastest, and P frames are in the middle.

Second, Original VMM generates results that are completely different from Native Linux. The differences between the decoding times of Original VMM and Native Linux are between −84.15% and −43.77%, and the differences of the standard deviations are between −85.04% and −44.70%. Since the decoding times of Original VMM are much faster than the ideal ones, the video frames are displayed much more smoothly on Original VMM than on Native Linux. In addition, much smaller standard deviations reveal that we cannot reproduce the difference of each frame type on a slow CPU. We therefore can suggest that we cannot verify the behavior of *mpeg play* on a slow CPU with Original VMM.

Figure 10 illustrates the difference of the frame decoding process between FoxyLargo and Original VMM. In FoxyLargo, the working time given to the VM is short enough to pretend a continuous time flow. Figure 10 (a) depicts that FoxyLargo executes the VM every clock-tick only for a short period. The amount of CPU cycles consumed during one time slice is equivalent to that of Native Linux during the clock-tick interval. Therefore, the error in decoding time is up to the clock-tick interval (1 msec in this experiment). On the other hand, Original VMM gives the VM a longer working time during which the VM can decode the entire frame. In this experiment, the decoding time is 1.62 msec on Original VMM on average, but the default Xen runs each VM for 30 msec. Thus, the error in decoding time becomes much larger than FoxyLargo as shown in Fig. 10 (b).

5.2.2 Display Period

Embedded systems often perform several tasks simultaneously. For instance, we can enjoy e-mailing while listening to music with a modern cellular phone. Embedded software developers therefore have to test time-sensitive software with



(a) Comparison of FoxyLargo and Native Linux

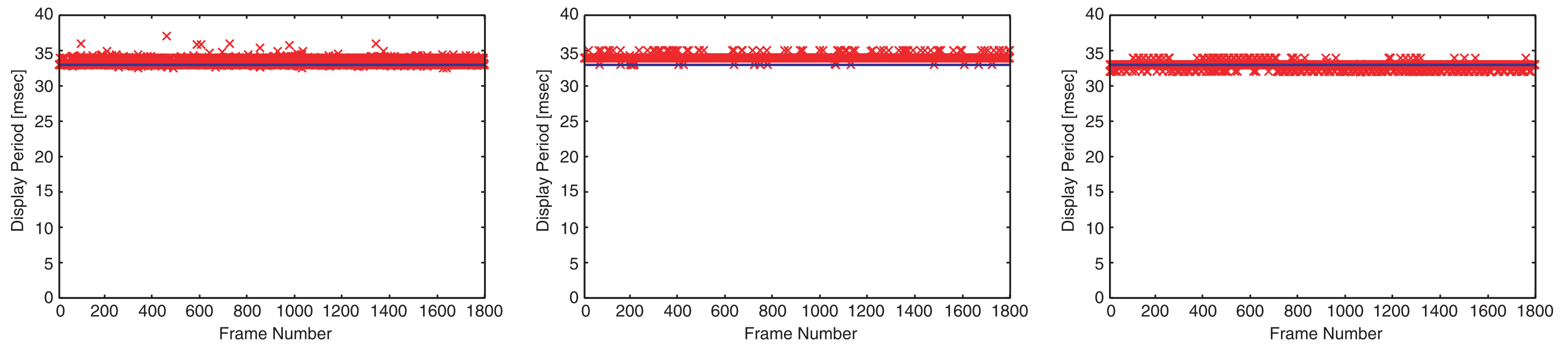
(b) Comparison of Original VMM and Native Linux

Fig. 10 The difference of the frame decoding processes in FoxyLargo and Original VMM. The working time given to the VM on FoxyLargo is short enough for the VM to pretend to run continuously. On the other hand, Original VMM gives the VM a much longer time. Thus the VM cannot pretend to run continuously.

competitive tasks. For such verification, the test environments of embedded time-sensitive software need to be capable of reproducing time-sensitive software's behavior even when competitive tasks exist.

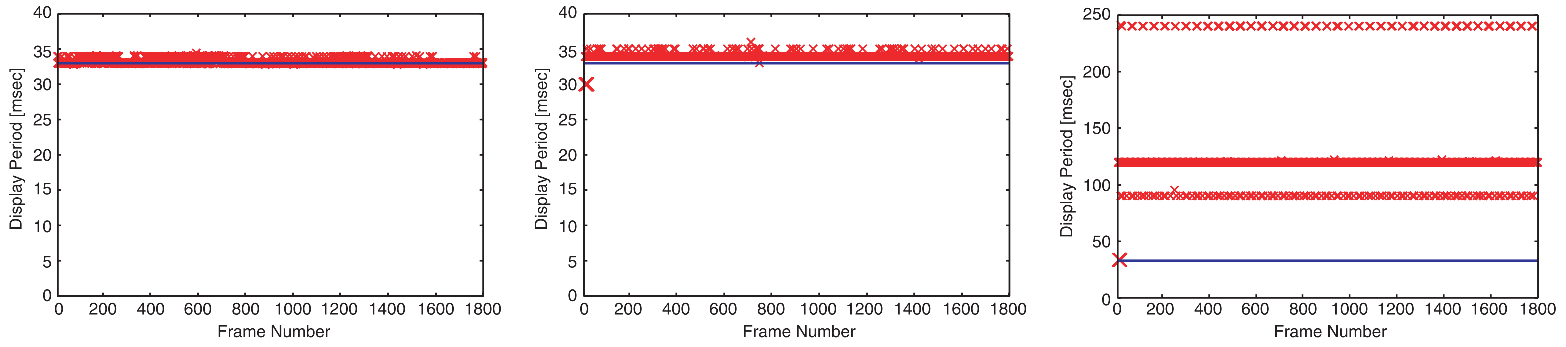
To demonstrate that FoxyLargo has the capability, we examined *mpeg play* while running the CPU-intensive loop benchmark used in Section 5.1.1. We performed this experiment under two situations for comparison. First, we ran *mpeg play* alone. Second, we ran it with the loop benchmark. In this experiment, we set the frame rate to 30 frame/sec; each frame is ideally displayed every 33 msec.

Figures 11 and **12** show the experimental results. In this experiment the CPU speed is set to 12.5% of the underlying CPU. The x-axis represents frame numbers, and the y-axis represents display periods. The blue lines drawn in the graphs show the ideal display period, or 33 msec. Figure 11 shows the results in the first case, running *mpeg play* alone. In this case, all the three configura-



(a) Native Linux (12.5% speed) (b) FoxyLargo (12.5% speed) (c) Original VMM (12.5% speed)

Fig. 11 Comparison of display periods of MPEG video frames (w/o competitive task). The figures show the period of displaying the video frames when mpeg play runs alone. All the three configurations successfully display the video frames in the correct interval



(a) Native Linux (12.5% speed) (b) FoxyLargo (12.5% speed) (c) Original VMM (12.5% speed)

Fig. 12 Comparison of display periods of MPEG video frames (w/- competitive task). The figures show the period of displaying the video frames when a competitive task exists. Native Linux and FoxyLargo display the frames at the correct timing. In contrast, Original VMM failed to do so since the boost function does not work.

tions show durations similar to the ideal case. It is curious that Original VMM successfully displays the frames in the correct interval, because the VM running on the default Xen would sleep for too long a time to do so; since the time unit of Xen's VM scheduler is 30 msec, the VM would repeat to run for 30 msec and sleep for 240 msec when we set the CPU share of a VM to 12.5%. We thus expect that *mpeg play* would fail to display the video frames every 33 msec in the setting. The reason that Original VMM successfully displays them against the above expectation is that Xen's credit scheduler has a boost function. With the boost function the scheduler preferentially schedules the VM that yielded the CPU before running out of its time slice. If *mpeg play* runs alone, the VM yields the CPU and sleeps until an interrupt notifies the time to display the decoded frame, since the VM becomes idle after finishing decoding the frame. Because of the boost function the VM is immediately scheduled when the interrupt occurs, and *mpeg play* consequently can display the decoded frame at the correct time.

We next show the results under the second situation in Fig. 12, running *mpeg play* with the loop benchmark. Native Linux displays almost all the frames in the correct interval as shown in Fig. 12 (a); the average display period is 33.24 msec and the standard deviation is 0.4358 msec. This result demonstrates that the CPU speed of 12.5% of the underlying CPU is sufficient to process this video stream at the frame rate of 30 frame/sec even if *mpeg play* runs with the loop benchmark.

As shown in Fig. 12 (b), FoxyLargo shows a similar duration of display periods to Native Linux; the average display period is 34.04 msec and the standard deviation is 0.2138 msec. This result demonstrates that FoxyLargo can reproduce the display behavior of *mpeg play* even if it runs *mpeg play* with the CPU-intensive benchmark.

On the other hand, Original VMM shows a quite different result as shown in Fig. 12 (c); almost all the plots are on 90, 120 or 240 msec. In this case, the VM does not yield the CPU since it is always busy because of the loop benchmark. Therefore, the boost function of the VM scheduler does not work unlike the first case, and as a result the VM fails to display the frames at the correct frame rate.

6. Related Work

There are many studies for improving embedded software testing. Seo, et al.[15] focused on the testing of the embedded software interface between embedded system layers, and proposed an automated scheme of embedded software testing based on the emulated target board. Hong, et al.[10] presented a case study of creating and applying a virtual platform in the development of a firmware of a hard disk system. They demonstrated that the virtual platform is useful to optimize the software code, but the simulation speed of the virtual platform is very slow. These techniques are very useful to verify the functional behavior of embedded systems. To the best of our knowledge, there is no technique that can be used to verify the sensory responsiveness of embedded time-sensitive applications in real time.

The technologies for improving the resource managements of VMMs have been investigated during the past decades. The balloon driver[20] manages the amount of memory resources allocated to guest OSes. The balloon process[18] achieves fair load balancing of VMs in multiprocessor environment. Communication-aware CPU scheduler[6] reduces network delay and improves VM consolidation by scheduling VMs with taking the communications between the guest OSes into consideration. The objective of these researches is to improve the efficiency of resource management in virtualized environments. In contrast, we set our sights on restricting CPU speed accurately even if there are surplus CPU resources.

Some researchers use virtual machine technology for software testing. Time travel virtual machine (TTVM)[13] enables developers to debug OSes with replaying arbitrary segments of the past execution of the OSes. TTVM is used to test low level software like drivers and OSes. In contrast, our system is tailored to testing the sensory responsiveness of time-sensitive applications. Time dilating[8] and DieCast[7] exploit a VMM to slow down the passage of time on OSes by delaying the notifications of the clock-tick interrupts to the guest OSes. These two systems change the speed of the software in simulation time rather than in real time. On the other hand, our system slows down the speed of time-sensitive applications and enables us to observe the behavior of them in real time.

7. Conclusion

Embedded software is usually developed in parallel with real hardware devices due to a tight time-to-market constraint. It is therefore quite difficult to verify the sensory responsiveness of time-sensitive applications until the device has been developed. To verify the responsiveness, the mechanism to slow down CPU speeds is useful. Although there are some existing mechanisms to do so, they do not meet the requirements for testing embedded time-sensitive applications.

This paper presented FoxyLargo, a technique to smoothly slow down the speed of the underlying high-end CPU with a virtual machine technology. FoxyLargo schedules the VM in such short periods that the time-sensitive applications running on it can process their periodic tasks on time. In addition, FoxyLargo schedules the VM for short time when a hardware interrupts occurs, so that the applications can handle the interrupts immediately. To demonstrate that FoxyLargo provides the target CPU speeds, we conducted three experiments. The first benchmark showed that FoxyLargo scheduled the VM on short periods and produced various CPU speeds. With the second benchmark, we showed that FoxyLargo enabled applications to handle the NIC interrupts immediately even if the CPU speeds were slowed down. Finally, we examined a real application, *mpeg play*, and showed that FoxyLargo reproduced decoding behavior that is very similar to the behavior on the machine slowed down by DVS. Through these experiments, we demonstrated that FoxyLargo can slow down CPU speeds suitably for testing the sensory responsiveness of embedded time-sensitive applications.

References

1) Advanced Micro Devices, inc. *AMD PowerNow!^{TM} Technology* (2000). http://www.amd.com/epd/processors/6.32bitproc/8.amdk6fami/x24404/24404a.pdf
2) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield. A.: Xen and the Art of Virtualization. *Proc. Symposium on Operating Systems Principles* (*SOSP '03*), pp.299–310 (Oct. 2003).
3) Freescale Semiconductor, Inc. *Reducing Time to Market: Parallel Software Development with Emulation and Simulation Tools for Mobile Extreme Convergence* (*MXC*) *Architectures* (Aug. 2005). http://www.freescale.com/files/wireless_comm/doc/brochure/MXCSWDEVWP.pdf
4) Le Gall, D.: MPEG: A video compression standard for multimedia applications, *Comm. ACM*, Vol.34, No.4, pp.46–58 (1991).
5) Goldberg, R.P.: Survey of Virtual Machine Research, *IEEE Computer Magazine*, Vol.7, No.6, pp.34–45 (June 1974).
6) Govindan, S., Nath, A.R., Das, A., Urgaonkar, B. and Sivasubramaniam, A.: Xen and Co.: Communication-aware CPU scheduling consolidated xen-based hosting platforms, *Proc.Coference on Virtual Execution Environments* (*VEE '07*), pp.126–136 (June 2007).
7) Gupta, D., Vishwanath, K.V. and Vahdat, A.: DieCast: Testing Distributed Systems with an Accurate Scale Model, *Proc. Symposium on Networked Systems Design & Implementation* (Mar. 2008).
8) Gupta, D., Yocum, K., McNett, M., Snoeren, A.C., Vahdat, A. and Voelker, G.M.: To infinity and beyond: time-warped network emulation, *Proc. Symposium on Networked Systems Design & Implementation* (*NSDI '06*), pp.87–100 (May 2006).
9) ARM Holdings. http://www.arm.com
10) Hong, S., Yoo, S., Lee, S., Lee, S., Nam, H.J., Yoo, B.-S., Hwang, J., Song, D., Kim, J., Kim, J., Jin, H.S., Choi, K.-M., Kong, J.-T. and Eo, S.: Creation and Utilization of a Virtual Platform for Embedded Software Optimization: An Industrial Case Study, *Proc. International Conference on Hardware/Software codesign and System Synthesis* (*CODES+ISSS '06*), pp.235–240 (Oct. 2006).
11) INTEL CORPORATION, http://www.intel.com/design/processor/manuals/253668.pdf, *Intel® 64 and IA-32 Architectures Software Developer's Manual* (Nov. 2007).
12) Jones, M.B. and Saroiu, S.: Predictability requirements of a soft modem, *Proc. ACM SIGMETRICS International Conference on Measurement and modeling of computer systems* (*SIGMETRICS '01*), pp.37–49 (2001).
13) King, S.T., Dunlap, G.W. and Chen, P.M.: Debugging Operating Systems with Time-Traveling Virtual Machines, *Proc. USENIX Annual Technical Conference* (*USENIX '05*), pp.1–15 (Apr. 2005).
14) Rowe, L.: The Berkeley MPEG Player. http://bmrc.berkeley.edu/frame/research/mpeg/mpeg_play.html
15) Seo, J., Sung, A., Choi, B. and Kang, S.: Automated Embedded Software Testing on an Emulated Target Board, *Proc. International Workshop on Automation of Software Test* (*AST '07*), pp.9–9 (May 2007).
16) Sugerman, J., Venkitachalam, G. and Lim, B.-H.: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, *Proc. USENIX Annual Technical Conference* (*USENIX '01*), pp.1–14 (June 2001).
17) Tam, D., Tsang, W. and Drula, C.: Dynamic Voltage Scaling in Mobile Devices, Csc2228 project final report, University of Toront (Dec. 2003).
18) Uhling, V., Le Vasseur, J., Skoglund, E. and Dannowski, U.: Towards Scalable Multiprocessor Virtual Machines, *Proc. Coference on Virtual Machine Research and Technology Symposium* (*VM '04*), pp.43–56 (May 2004).
19) Vast Systems Technology Corporation: *1-SOURCE^{TM} Virtual Prototyping for Embedded System Design* (July 2006). http://www.vastsystems.com/docs/1_Source_Virtual_Prototyping_VaST.pdf
20) Waldspurger, C.A.: Memory Resource Management in VMware ESX Server, *Proc. Symposium on Operating System Design and Implementation* (*OSDI '02*), pp.181–194 (Dec. 2002).
21) Whitaker, A., Shaw, M. and Gribble, S.D.: Scale and Performance in the Denali Isolation Kernel, *Proc. Symposium on Operating Systems Design and Implementation* (*OSDI '02*), pp.195–210 (Dec. 2002).

(Received January 27, 2009)
(Accepted March 22, 2009)

Tetsuya Yoshida was born in 1983. He received his B.E. degree from University of Electro-Communications in 2005, his M.E. degree from Keio University in 2008. He is currently a Ph.D. candidate in School of Science for Open and Environmental Systems at Keio University, since 2008. His research interests include virtual machine technology, operating systems, and system software. He is a member of IPSJ, ACM.

Hiroshi Yamada was born in 1981. He received his B.E. and M.E. degrees from University of Electro-Communications in 2004 and 2006, respectively. He received his Ph.D. degree from Keio University in 2009. He is currently a research associate of the Faculty of Science and Technology at Keio University. His research interests include virtual machine technology, operating systems, and system software. He is a member of IPSJ, ACM, USENIX and IEEE/CS.

Kenji Kono received the B.Sc. degree in 1993, the M.Sc. degree in 1995, and the Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE/CS, ACM and USENIX.