Toward Under-Millisecond I/O Latency in Xen-ARM

Seehwan Yoo, Kuen-Hwan Kwak, Jae-Hyun Jo and Chuck Yoo
Korea University
{shyoo, khkwak, jhjo, hxy}@os.korea.ac.kr

ABSTRACT

This paper addresses the I/O latency issue within Xen-ARM. Although Xen-ARM's split driver presents reliable driver isolation, it requires additional inter-VM scheduling. Consequently, the credit scheduler within Xen-ARM results in unsatisfactory I/O latency for real-time guest OS. This paper analyzes the I/O latency in Xen-ARM's interrupt path, and proposes a new scheduler to bound I/O latency. Our scheduler dynamically assigns priorities to guest OSs so that Xen-ARM ensures to schedule the most urgent task within the system. The experimental results show that Xen-ARM with our new scheduler reduces delay spikes, latency larger than 1ms, from 16% to 1% while retaining the split driver model.

Categories and Subject Descriptors

D.4.7 [**OPERATING SYSTEM**]: Organization and Design

General Terms

Performance

Keywords

Virtual machine, Real-time system

1. INTRODUCTION

Recently, virtualization is drawing attention to embedded systems such as mobile phones or smart pads, and several hypervisors for those mobile systems are presented because virtualization systems have several advantages over traditional embedded operating systems. First of all, those hypervisors allow multiple heterogeneous operating systems to run over a single physical machine. They support both a real-time guest OS for mobile communication and a general-purpose OS for rich applications like Web browsers or office programs. In addition, virtualization protects user data from security threats, and enhances reliability by isolating an insecure or faulty domain from the others.

Although embedded virtualization has diverse advantages over traditional embedded operating systems, performance issues have been continuously challenged because those systems have inherent performance requirements while having limited hardware resources. Particularly, I/O latency in a virtual machine needs to be carefully addressed since it affects the performance of a real-time guest OS. For example, over 1ms interrupt latency leads to call misses in mobile phones, which is regarded as a serious defect.

Embedded hypervisors take several different approaches in order to achieve satisfying I/O latency for a real-time guest OS over a virtual machine. One of the most representative embedded hypervisors is OKL4 microvisor [2]. The microvisor presented unique interrupt handling originated from L4 microkernel's user-level I/O driver model [6]. The microvisor achieves low latency by architectural optimizations enough to run with commercial mobile phones. Another commercial hypervisor VirtualLogix-VLX presents different I/O model [7]. Device drivers are located within the hypervisor so that the drivers can timely handle physical interrupts, and are easily shared among multiple guest OSs.

Xen [1] is another hypervisor that is popular in server systems, and recent Xen-ARM [5] presented how Xen can be adapted to mobile devices. It enhances reliability of user data by isolating a private domain from insecure domains. In addition, it is small enough to fit in a mobile platform. For I/O handling, Xen-ARM provides *split driver model*, which isolates unreliable device drivers from guest OSs. In addition, the model makes physical devices to be easily shared among guest OSs.

However, the current credit scheduler, the default scheduler of Xen-ARM, is insufficient to support a real-time guest OS due to its long I/O latency. Note that split drivers require an additional inter-VM scheduling between the driver domain and a user domain. It is known that the credit scheduler has limitations on supporting time-sensitive applications although it has additional *BOOST* priority. Two major limitations are: First, the scheduler allows multiple domains to be boosted simultaneously, so BOOST can be easily ignored. Second, a guest domain is not boosted in some cases because it primarily focuses on fairness among CPU-bound domains.

Adding another priority such as REAL_BOOST would not completely resolve the I/O latency issues because Xen-ARM has a hierarchical scheduling structure. Since the hypervisor cannot impose task scheduling within a guest OS, the hypervisor prioritizes the entire guest OS rather than a task within the guest OS. This results in increased latency because the hypervisor cannot distinguish urgency among tasks within different guest domains. In addition, we found that physical interrupt handling can be delayed by the virtualization because a guest OS can only disable/enable virtual interrupts; and thereby the driver domain can be preempted by a user domain within the code with interrupts disabled. For example, the driver domain can be preempted by another user domain even when it handles interrupts.

So, this paper analyzes the latency in the interrupt path of Xen-ARM, and proposes new mechanisms to bound I/O latency. This paper consists of five sections. In Section2, we review the current credit scheduler within Xen-ARM. Section 3 presents how the interrupt handling can be delayed over a virtualization system. Section 4 presents our approach to guarantee I/O latency in order to run a real-time guest OS over a virtual machine. Finally, Section 5 concludes the paper.

2. CREDIT SCHEDULER OF XEN-ARM

The current Xen-ARM uses the credit scheduler, which is based on the WRR (Weighted Round-Robin) scheduling algorithm. The credit scheduler primarily tries to keep fairness of CPU utilization among guest domains. In the credit scheduler, a VCPU[1] has one of three priorities, UNDER, OVER, or BOOST. VCPU begins from UNDER, at which the VCPU has remaining credits, and is ready to run. Xen-ARM periodically distributes credit over all guest OSs, and debits credit as much as the guest OS consumes the physical CPU time. When a VCPU consumes all its credit, then the priority of VCPU is changed from UNDER to OVER, and the guest OS is not scheduled to run until it becomes UNDER again. Each guest OS can specify the weight so that Xen-ARM can differentiate the CPU utilization among the guest OSs.

Regarding I/O operation, the credit scheduler has additional priority called BOOST. A guest OS can be temporarily prioritized when the guest OS has pending I/O events. In Xen-ARM, BOOST is the highest priority so that the guest OS can handle I/O events in a timely manner. When a guest OS is boosted, it preempts the currently running guest OS.

3. I/O LATENCY IN XEN-ARM

To test I/O latency over the current Xen-ARM hypervisor, we measured interrupt handling latency at real-time domain using ping tests. Three guest OSs are running over Xen-ARM: dom0 for driver domain, dom1 for real-time I/O domain, that is a icmp receiver with light CPU load (20%), and dom2 for CPU-intensive application (100% CPU load). Then, another external server sends icmp request packets to dom1 with random interval (0∼40ms), of which average of ping interval is 20ms. The measurement is based on ARM-cortexA9 processor, and 100Mbps network device is attached via usb interface.

Figure1 shows the cumulative latency distribution for ten thousand interrupts in the two different intervals: 1) from the physical interrupt handler at Xen-ARM to network backend driver (netback) within dom0, and 2) from the netback handler to the icmp handler within dom1.

Graph in Figure1 shows two important characteristics of latency distribution within Xen-ARM. First, latency from the physical interrupt handler at hypervisor to dom0 (Xen-netback latency) is small in most cases, but the worst-case latency is significantly large. In the figure, more than 97% of icmp requests are handled within 1 ms at dom0, but rest 3% of requests are handled with very long delay up to 60ms. So, the driver domain cannot handle interrupts in a timely manner, which means I/O latency cannot be guaranteed even though dom0 serves as a real-time domain.

Second, latency from dom0 to dom1 (netback-domU latency) can be considerably large. In the figure, although more than 84% of icmp requests are handled within 1 ms at domU, more than 16% of requests are delayed more than 1ms. This is due to the inter-VM scheduling for the communication between two domains (dom0 and dom1). To support real-time, this latency has to be bounded.

In the following sections, we focus on the latencies in these two intervals.

3.1 Xen-netback Latency

In Xen-ARM, all physical interrupts are handled within the hypervisor, and a guest OS handles only virtual interrupts. The hypervisor delivers virtual interrupts to the designated guest OS by setting the corresponding bit in the event channel, and it finishes interrupt handling as soon as possible in order to minimize the interrupt-disabled region. Then, the hypervisor performs scheduling so that the designated domain can quickly handle interrupts. The designated domain is boosted by the credit scheduler so that the interrupts can be handled in a timely manner.

However, as we have seen in the previous section, interrupt handling at the driver domain presents large worst-case I/O latency. To guarantee I/O latency, the worst-case execution has to be time-bounded.

3.1.1 BOOST in Credit

Although the credit scheduler supports BOOST priority, it is well known that it lacks in supporting time-sensitive applications[4, 3]. BOOST in the credit scheduler has limitations to guarantee I/O latency because boost is easily negated. Namely, multiple domains can be boosted simultaneously. This is called *multi-boost*, which negates the original boost mechanism. Since the current credit scheduler doesn't distinguish the urgency among them, the scheduler can select less urgent domain. Multi-boost is largely observed in I/O intensive systems, and it affects the Xen-netback latency.

For example, when a driver domain finishes the backend

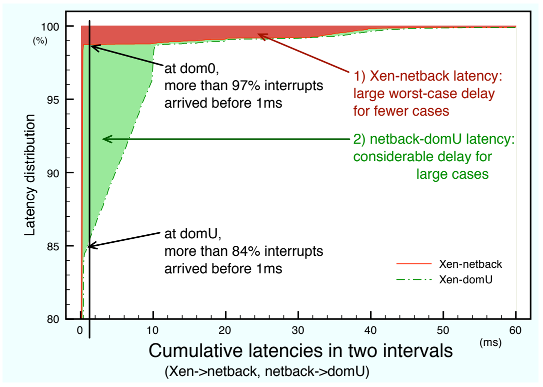
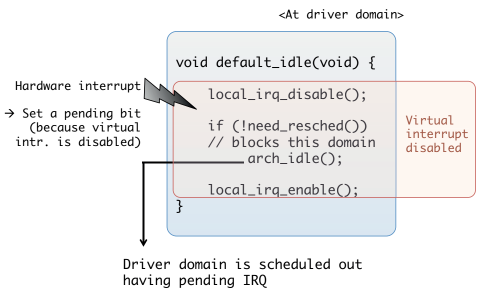


Figure 1: Latency distribution for two intervals

[1]In this paper, we interchangeably use VCPU, guest OS and domain if it is clear in the context.


<At driver domain>
Hardware interrupt
→ Set a pending bit (because virtual intr. is disabled)
Virtual interrupt disabled
Driver domain is scheduled out having pending IRQ

```
void default_idle(void) {
    local_irq_disable();

    if (!need_resched())
    // blocks this domain
        arch_idle();

    local_irq_enable();
}
```

Figure 2: Wrong idle paravirtualization

driver handling, it awakens the designated domain, at which the VCPU is boosted. If an additional interrupt occurs, the driver domain is boosted again. So, both domains are temporarily in BOOST priority at the same time. If the hypervisor schedules the designated domain, then the interrupt handling at the driver domain can be delayed.

In addition to multi-boost, the credit scheduler doesn't boost the VCPU if the domain is OVER or the VCPU is on the run queue (we name it *non-boosted VCPU*). For example, if the I/O workload is so intensive that it leads its credit shortage, then the driver domain will not be boosted. The driver domain needs to wait until the credit is recharged, and the domain becomes UNDER again. As shown in Figure 1, this latency can vary up to multiple quantum.

3.1.2 Virtualized interrupt handling

Interrupt handling within a guest OS also affects I/O latency. Since interrupt handling within a guest OS is different from a native (non-virtualized) system, latency is differently characterized. Originally, virtual interrupt makes a guest OS fully preemptible because guest OS cannot disable physical interrupts, and allows preemption at arbitrary points. Namely, all the guest OSs run in user-level, and interrupt is disabled only within the minimal code inside the hypervisor. Thus, the entire system becomes responsive because a guest OS can be preempted even though it disables virtual interrupts. Furthermore, misbehavior with handling interrupt effects locally, and never affects the other domains.

However, virtualized interrupt handling can cause additional delay since a guest OS cannot fully control physical interrupts. The driver domain can be preempted by a user domain even though it has pending interrupts. Note that the hypervisor can still receive physical interrupts although a guest OS disables virtual interrupts. If the guest OS disables virtual interrupts, the received virtual interrupts are stored in the event channel, and the guest OS handles pending interrupts only after it re-enables virtual interrupts again. In this situation (that the driver domain has pending virtual interrupts while disabling virtual interrupts), if the hypervisor receives additional physical interrupts that might trigger inter-VM scheduling, the driver domain would be scheduled out. Consequently, the pending virtual interrupt handling is delayed until the driver domain is re-scheduled by the hypervisor, and the driver domain re-enables virtual interrupts.

For example, in an idle paravirtualization of XenoLinux, a guest OS disables interrupts just before the entering the hypercall, *do_block()*, as shown in Figure2, and re-enable after the call in order to reduce unnecessary wake ups during the idle.

If an interrupt occurs just after it is disabled, then the

vcpu state	priority	intr. state	sched. out count
Blocked	BOOST	Enabled	0
		Disabled	275
	UNDER	Enabled	0
		Disabled	13
Unblocked	BOOST	Enabled	0
		Disabled	664,190
	UNDER	Enabled	8
		Disabled	41

Table 1: Schedule out counts for network interrupts

interrupt handling is delayed until the guest OS re-enables interrupts. Since *arch_idle()* invokes a hypercall that blocks itself, the domain is scheduled out. The domain has to wait until it receives additional event at a later time. Namely, the pending interrupt is handled with a considerable delay, which will show as latency spikes in experiment results.

This contradicts to the original intention of interrupt virtualization because the interrupt handling can be delayed by virtualization even though virtual interrupt handling makes the guest OS more preemptive. Since no physical interrupt are blocked by a guest OS (although virtual interrupts are disabled within the guest OS), the guest OS has become fully preemptive. However, it does not lead to the reduced interrupt handling latency because the physical interrupts are still received by the hypervisor; and some interrupts allows another guest domain to preempt the driver domain even though it has pending interrupts to process.

3.1.3 Experiments

To analyze the delayed interrupt handling at the driver domain, we measure the actual latency between physical interrupt handler (i.e. *handle_interrupts()*) at Xen and the netback handler (i.e. *net_be_start_xmit()*) at the driver domain. The experimental parameters are the same as in Section 3. To categorize the reasons of delay spikes, we count the number of different cases when the driver domain is schedule out to another domain. For each schedule out events, we check the VCPU states 1) whether it is blocked or not, 2) the priority is BOOST or UNDER, and 3) whether virtual interrupts are disabled or not. If the VCPU is blocked, and interrupt is disabled, then it implies that the driver domain invoked idle with virtual interrupts disabled. This causes the delay by virtualized interrupt handling. If the VCPU is unblocked and BOOST, it implies multi-boost because there should be another VCPU that has BOOST priority. If VCPU is unblocked and UNDER, it implies that the VCPU was not boosted because it was on the run queue.

Table 1 summarizes the results for 343,335 ping tests. Among total 1,003,321 number of interrupts, 6,327 interrupts are handled after 1ms, which is regarded as delay spike. That is, about 0.63 percent of network interrupts experiences large interrupt handling latency more than 1ms. Table shows that the virtualized interrupt handling causes 288 (= 275+13) number of schedule outs, multi-boost causes more than 0.6 million schedule outs, and not-boosting VCPUs causes 49 (= 8 + 41) schedule outs.

3.2 Netback-domU Latency

Besides Xen-netback latency, another large latency can occur between netback driver and frontend driver. The major reason for this latency is due to *non-boosted VCPU* of the credit scheduler. The credit scheduler doesn't boost a

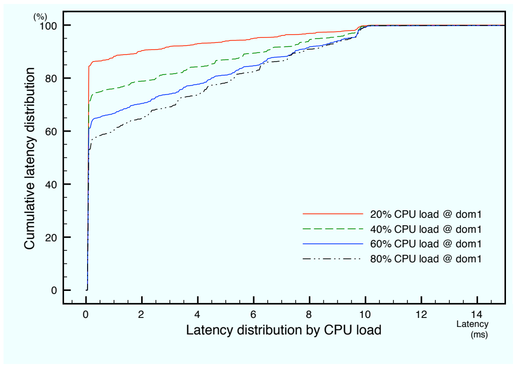


Figure 3: Impact of CPU load on I/O latency

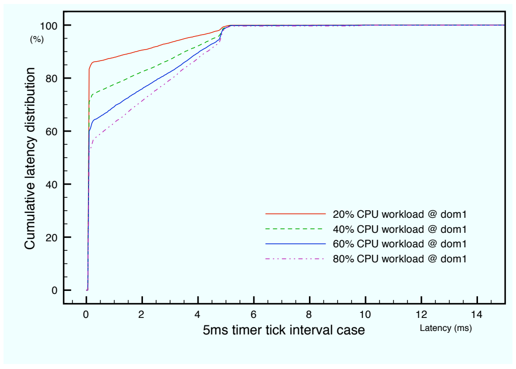


Figure 4: Timer tick interval and I/O latency

VCPU when the designated VCPU is in the run queue or has consumed all its credit. Xen-ARM regards those VCPUs as CPU-bounded, and doesn't boost them so that the CPU fairness can be preserved. In reality, a guest OS usually has CPU-I/O mixed workloads, and long I/O latency is observed when CPU-intensive job is running. Thus, I/O latency increases as the CPU utilization of a user domain. Namely, when a user domain utilizes more CPU, then its I/O latency is very likely to be longer. The following experiment shows the impact of CPU utilization to the I/O latency of a domain.

We run three guest OSs as follows: Dom0 for the driver domain, dom1 for CPU-I/O mixed workload, and dom2 for CPU-intensive workload to ensure the work-conserving. Then, we measured I/O latency from the interrupt handler within Xen-ARM to the netfront driver at dom1 by varying the CPU utilization (20%, 40%, 60%, 80% of CPU load) at dom1. The graph in Figure3 shows the distribution of the measured latency.

According to the graph in Figure3, latency could be increased by CPU utilization of dom1. In the case that dom1 has 20% CPU utilization, 85% of interrupts are handled immediately. The rest 15% of interrupts are delayed because the VCPU has not been boosted by CPU workload. On the other hand, for 80% CPU utilization case, only 50% of interrupts are handled immediately, which means that the probability to have small latency decreases according to the CPU utilization.

Delay from CPU workload is closely related with timer tick interval. In general, system becomes less responsive and the overall performance overhead increases as timer tick interval increases. Namely, if the timer tick interval is small, scheduler is more frequently invoked, and scheduler is able to dispatch the most urgent thread as soon as possible. However, frequent context switch introduces additional overhead from cache pollution, TLB flush, etc. The same rule applies to inter-VM scheduling in Xen-ARM. As we can see in the Figure 3, the latency is bounded by timer tick interval, 10ms.

Figure 4 shows the same experiments with 5ms timer tick interval. As we can see in the figure, the delay impact from CPU-bound work also decreases as the timer tick interval.

4. I/O SCHEDULING FOR REAL-TIME GUEST OS

As we have observed in the previous section, the current credit scheduler is not suitable for supporting real-time I/O in its current form. Two major latencies are 1) Xen-netback latency, and 2) netback-domU latency. To bound the interrupt handling latency due to Xen-netback latency at Xen-ARM, we introduce the following methods to the hypervisor so that the designated backend driver can handle interrupts in a timely manner.

Firstly, we mitigate the latency by virtualized interrupt handling. We changed the Xen-ARM's scheduler so that the driver domain is not scheduled out when the driver domain disables virtual interrupts. Since the driver domain is not scheduled out during it disables interrupts, it can re-enable interrupts as soon as possible, and backend driver can immediately process the pending interrupts. This resolves all the virtual interrupt-related problems including wrong idle paravirtualization. Note that we do not disable physical interrupts, but pinning the driver domain when it disables virtual interrupts.

Secondly, we differentiate the interrupt handling for real-time I/O devices (i.e. netdevice in our case) from the other devices. We use FIQ[2] of ARM processor so that the real-time devices take advantage of architectural support. FIQ has higher priority than IRQ, and this largely removes misbehavior through disabling interrupts. For example, the driver domain is able to handle network interrupt with FIQ, even in code with IRQ disabled.

Thirdly, we further optimized the Xen-ARM's scheduler so that the driver domain can be scheduled at the highest priority particularly when it has pending interrupts. The highest priority is remained until the domain finishes the interrupt handling, and the inter-VM scheduling is postponed until the driver domain triggers end-of-interrupt via a hypercall *pirq_guest_eoi()*.

Figure 5 shows the enhanced I/O latency by our modifications. Three domains run at the same time: dom0 for driver domain, dom1 for handling icmp requests, and dom2 for cpu-intensive operation for ensuring work-conserving scheduling policy. Each line presents the cumulative distribution for the presented delay spikes. (1), (2), and (3) represent our modifications regarding virtual interrupt preemption, FIQ handling, and prioritized driver domain scheduling, respectively. Originally, 0.8% interrupts are handled over 1ms, (2) and (3) reduce the spikes to 0.71% and 0.65%, respectively.

Finally, we modified the Xen-ARM's scheduler so that it can schedule domains with strict priority, based on urgency. By providing the strict scheduling priorities, the interrupt la-

[2]FIQ is an ARM-specific architectural mode for faster interrupt handling

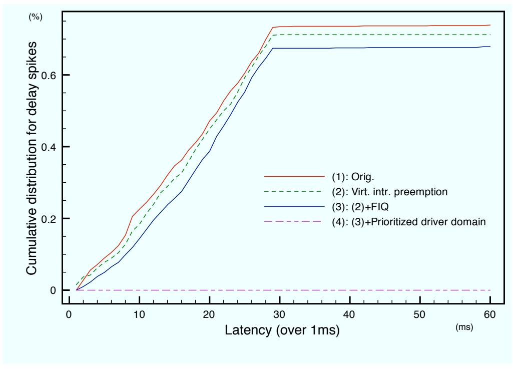


Figure 5: Reduced delay spikes within Xen-netback

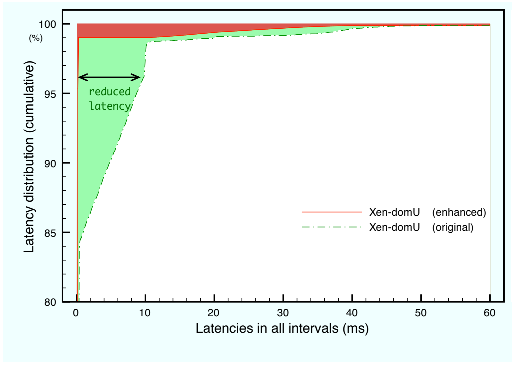


Figure 6: Latency enhancement by proposed methods

tency can be deterministically bounded. Priority orders are: 1) driver domain within interrupt, 2) dom1 within interrupt context, 3) dom1 without interrupt context, 4) driver domain without interrupt context, and rest of the cases are handled based on the existing credit scheduler's priority. Strict priority-based scheduling overcomes the two major limitations in the current credit scheduler (from multi-boost and non-boosted VCPU). It can distinguish the urgency among the BOOST domains, and eliminates I/O latency affected by CPU load. Therefore, a real-time guest OS can handle interrupts with strict time-bound while retaining the split driver model.

The overall cumulative latency distribution is presented in Figure 6. In the graph, more than 99% of network interrupts are handled within 1ms at dom1. Note that dom1 has 20% CPU load, and rest 1% interrupts are measurement error because of the complex interrupt generation of the device. Considering this measurement error, our modification guarantees interrupt handling with 1ms time-bound. Finally, after applying all the modifications, all the delay spikes are eliminated for one million of network interrupts.

5. CONCLUSION AND FUTURE WORK

This paper addresses I/O latency in a Xen-ARM-based virtual machine. To support a hard real-time guest OS, the hypervisor has to ensure that a real-time guest OS can handle I/O with strict time-bound. The current Xen-ARM has limitations regarding its scheduler and unique split driver model. BOOST priority in credit is easy to be negated, and domains are sometimes not boosted. In addition, virtual interrupt handling at the driver domain can be delayed because another domain can preempt the driver domain even within the ISR. To guarantee time-bounded latency, we modified the hypervisor scheduler in order to give strict priorities to domains. With the modified schedulers, we achieved guaranteed I/O handling within 1ms time-bound. The result shows that the real-time I/O handling can be possible with Xen-ARM.

Our prototype implementation focuses on single-core platform, but we believe that our mechanisms are orthogonally applicable to multicore systems. Fairness is another concern in heavy I/O environment, and we are considering fairness among multiple VMs with different virtual I/O devices environment as future work.

6. ACKNOWLEDGEMENT

The authors appreciate reviewers for their helpful comments and feedback. This research work was financially supported by the Seoul Research and Business Development Program (No. WR080951) and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0029180).

7. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*, pages 164–177. ACM, 2003.

[2] G. Heiser and B. Leslie. The okl4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM.

[3] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for i/o performance. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110, New York, NY, USA, 2009. ACM.

[4] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the xen hypervisor. In *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 97–108, New York, NY, USA, 2010. ACM.

[5] S.-M. Lee, S.-B. Suh, B. Jeong, S. Mo, B. M. Jung, J.-H. Yoo, J.-M. Ryu, and D.-H. Lee. Fine-grained i/o access control of the mobile devices based on the xen architecture. In *MobiCom '09: Proceedings of the 15th annual international conference on Mobile computing and networking*, pages 273–284, New York, NY, USA, 2009. ACM.

[6] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sept. 2005.

[7] S. Sumpf and J. Brakensiek. Device driver isolation within virtualized embedded platforms. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1–5, Jan. 2009.