ABSTRACT
Paravirtualization in Virtual Machine Monitor (VMM) is a virtualization technique which allows a Virtual Machine (VM) to run at a near native speed. It is considered to be a faster alternative than full virtualization. But paravirtualization does not come cheap as it requires modifications (front-end drivers) in the guest Operating System (OS) as well. If a VMM decides to support multiple OSs, it means that the developers need to write front-end drivers for each OS. It is a difficult task to achieve. Fortunately, there is a standardized I/O virtualization framework called Virtio. A lot of Virtio front-end drivers are already available in multiple OSs. Those drivers can be used by any other VMMs as long as it has a standardized Virtio back-end driver. By having Virtio back-end driver, a VMM can support many I/O devices in multiple OSs for their guest VMs. Typical virtio implementation utilizes QEMU to provide device emulation. For pure metal-bare VMM, integrating Virtio backend in the academics and industries requires QEMU for example KVM. And QEMU is not considered to be pure type-1 VMM. In this paper, we designed and implemented Virtio backend in type-1 VMM without QEMU. We also evaluated the performance of our implementation, specifically the block device, with IOZone and compare it to native speed.

KEY WORDS
cloud-based software engineering, virtual machine monitor, i/o virtualization, virtio, paravirtualization

1. Introduction
The main core technology in Cloud Computing is the Virtual Machine Monitor (VMM). It controls everything about Virtual Machine (VM) from its initialization until its termination. One of the enabling technologies in VMM is hardware virtualization. Full virtualization is one of the methods in hardware virtualization. It emulates physical devices to guest VMs. Emulating physical devices are slow and complicated. Another method is paravirtualization. Performance wise, it is a faster alternative than full virtualization. But paravirtualization requires modifications (front-end drivers) in the guest Operating System (OS). It is a daunting task for a VMM to support multiple OSs because it means that the developers need to write a lot of front-end drivers.
Guest VMs are aware that they are running on virtual devices. The drivers for virtual devices are optimized to do less trap and simpler than physical devices. And this technique also doesn't require binary translation as opposed to another technique called as full virtualization. For these reasons, paravirtualization is faster than full virtualization and can reach a near native performance.

In bare-metal VMM, the architecture of paravirtualization can be described by figure 2. A driver in guest VM accesses the back-end driver in VMM. Then it connects to the real device driver either in VMM or Host OS. The device driver will then finally reach physical device.

![Figure 1. Bare-Metal VMM architecture.](image1)

The downside of this method is portability. For an OS to be able to run using paravirtualization, there is a cost to port it. For hardware such as CPU or interrupt, access to OS kernel’s source code is required. Thus, it is not desirable for some proprietary OSs such as Windows or MacOS. But for hardware such as disk, network and random number generator it is possible by installing drivers. Therefore paravirtualization of these hardware is possible even for proprietary OSs.

Paravirtualization technique is used in a lot of VMMs. Xen has its own paravirtualization drivers called as PV Drivers. KVM, VirtualBox[10], Iguest, Xvisor[11], and many others VMMs use an open standard paravirtualization framework called as Virtio. The other techniques in I/O Virtualization are full virtualization and direct device assignment. Full virtualization as mentioned earlier is slower and much more complex to implement than paravirtualization. Direct device assignment virtualization is a slightly faster alternative than paravirtualization. But its downsides are (1) the limitation of supporting hardware and (2) live migration support is hard to implement[12]. Direct device assignment is also referred as hardware-assisted or direct device access.

### 3. Virtio in Bare Metal VMM

#### 3.1 Architecture

In pure bare-metal VMM, an Operating System (OS) that act as a host OS is exist. It is located in a layer above VMM. Note its difference with KVM’s host OS that’s in the same layer of its VMM. Thus, for pure bare-metal VMM, we have two architecture options to place virtio backend. The first option is to put it in Host OS as shown in figure 3. The second option is to put it in VMM as shown in figure 4.

![Figure 2. Paravirtualization in Bare-Metal VMM.](image2)

![Figure 3. Virtio-Backend in Host VM.](image3)

As described in figure 3 and 4, in supporting virtio devices for both architecture, there are 3 main components; back-end, helper, and front-end.

For the first architecture, the front-end is located in guest and it is considered as a normal device driver. There’s no distinguishable difference between virtio front-end device drivers and other physical device drivers. The implementation of front-end driver depends on the running OS of guest. The drivers for Windows and Linux are already created and no changes are needed for them to be used by any VMM with virtio back-end. In our implementation, the transport is done via memory-mapped I/O (MMIO)[15].
To serve front-end’s requests, a back-end driver is necessary in the host. Our back-end driver leverages the already implemented virtualization programs in host’s OS. As an example, in Linux, the back-end driver uses TUN/TAP device to virtualize network devices for guests. The back-end doesn’t influence the OS of guest. The same back-end driver can virtualize devices for multiple guests running multiple and different OSs.

The actions that can be done by back-end in VMM are limited due to security and stability reasons[13]. For instance, the back-end in VMM can’t directly read or write the memory of guests. In that regard, we need software in higher layer to provide such actions. That’s why we have virtio helper. The helper will take requests from both front-end and back-end and process them as necessary.

Meanwhile for the communication of back-end and helper is done by the helper buffer. It is set during the initialization of the host VM. The helper buffer is used to send additional information to back-end. The information inform back-end which virtual device needs attention.

When a guest is booting, guest’s kernel detects a virtio device during device probing. Then the front-end drivers asks the helper for the type, features, and status of the corresponding virtual device. After getting the information, front-end initializes the device accordingly. The initialization is including creating the virtio buffer. The virtio buffer is a memory region shared between host OS and guest OS. The information about the size and location of the buffer in guest OS is transferred to the helper. Helper then map this buffer to a memory region in host OS. The virtio buffer is used as a channel of communication between back-end and front-end.

During the device initialization in front-end, the helper asks back-end driver about the available device features in host OS. By this way of conduct, we can be sure the virtual devices in guest OS work under the capabilities of physical devices in host OS’s. As an example, if the virtual block device in host OS supports SCSI packets then guest’s OS should support too. Otherwise, the SCSI support in guest OS is turned off.

For the virtio operation, front-end writes into the buffer. Back-end reads the buffer and acts properly based on the request and the type of the device. As for the full explanation of virtio’s operation, please refer to a paper published by Rusty Russel[1] as we are strictly following its mechanism.

The mechanism for front-end to contact helper is through memory trapping. VMM traps the MMIO region for virtio device. Any write and read into the region will be handled by the helper. As for back-end, it does hypervisor calls to contact helper. The helper kicks both front-end and back-end by injecting virtual interrupts. Front-end and back-end drivers should register a handler to manage interrupts sent by helper. The interrupt id used by ViMo-S depends on the CPU architecture. For Exynos5250, we arbitrarily use interrupt id 155 to kick back-end. Because interrupt id 155 is not used by the board[14].

In this architecture, a trap (virtio request) from a guest is directly processed by the back-end. The back-end identifies the type of request and reacts accordingly. If the guest is writing to a block device then the back-end is going to open virtio buffer and write the content to the corresponding memory in the host.

As opposed to the previous architecture, putting the back-end in the VMM seems to be beneficial at first because the requests from the guests are handled as soon as possible. But it is actually counter productive for multiple reasons. First reason is the virtio requests are need to be processed by the host also because device emulation is in the host. So after the back-end finished identifying the requests, then the back-end is going to identify the host that the back-end is writing some content to the host so the host can process it. The solution is to put the device emulation in VMM, but the amount of work to be done is daunting.

![Figure 4. Virtio-Backend in VMM.](image)

Another reason is the amount of code that need to be written in VMM. By design, VMM’s size is made to be as minimum as possible. So there should not be a lot of code in VMM. Unfortunately to process virtio buffer, we are going put a noticeable amount of code. This is not only going to be redundant, because we already have it in the host, but also make VMM heavier and more prone to bugs. For these reasons, we are considering the first architecture to be the best choice for our implementation.

### 3.2 Virtio Backend

Inside of virtio back-end, there are at least 4 elements; virtio worker, virtio buffer, helper buffer, and at least one of the virtio devices.

Processes in virtio back-end are activated when virtio IRQ (interrupt) handler detects an interrupt injected by VMM for virtio devices. Therefore, during initialization we should also register an IRQ for virtio. In Linux the function to register an IRQ is `request_irq`.

Interrupt handling is designed to be processed fast but processing virtio requests can be slow. In order to solve this, IRQ virtio handler has only two tasks; wake up virtio worker and return the information that the IRQ is already handled.
Virtio worker is an infinite loop process which handles all requests related to virtio. When virtio worker is activated, it will look at the helper buffer to understand the request. The helper buffer has information about which guest and which device that requesting it. The information in helper buffer is filled by virtio helper in VMM. Virtio worker then will wake the virtio device related to the request and send the other information as well.

All virtio devices (network, block, and others) is also an infinite loop process. Most of the time they are sleeping and they are only awaken by virtio worker if there’s a request. They can access virtio buffer to process the requests from guests. Virtio buffer is a shared memory region written and read by virtio front-end in guest. The region is a table implementing ring buffers scheme called as virtio ring. One of the parts in the table is the address in guest’s regions which contain some data. For example, in a network device, the data may contain network packets sent by guest. While in a block device, they may contain block region written by guest. To be able to read that data, back-end ask virtio helper to copy it into host’s memory region or the reverse. This copy process is very slow thus we should implement zero-copy technique which will be explained in the next part. Once the memory is transferred from and to guest, back-end communicates as necessarily with virtual devices (raw file, tun/tap device, etc.) in host OS. After the request handling is done, back-end modified the table in virtio buffer to inform guest which requests that have been processed. Then back-end does hypervisor call to virtio helper and go back to sleep. Virtio helper then inform guest that there are changes in the virtio buffer so guest can act accordingly.

3.3 Virtio Helper

Virtio helper is a simple component which connects front-end and back-end. It is helping in (1) setting up the virtio devices in guests, (2) mapping the shared memory, (3) reading and writing inter-VM, (4) injecting virtual interrupt to VMs, and (5) handling hypervisor calls from host.

3.4 Virtio Front-end

The beauty of using a framework such as virtio is because to implement back-end, we don’t need to know exactly how the front-end works. All we need to do are to understand the requests and to react accordingly. However if you are interested in how virtio front-end works or are going to write a front-end driver then please refer to references [X, Y].

3.5 Zero-Copy in Virtio

To greatly increase the performance, we implemented zero-copy technique. This is done by mapping guest’s memory region pointed by the table in virtio buffer onto host’s memory region. And for security and stability, un-mapping should be done immediately after reading and writing by virtio back-end. By inserting this technique into our VMM, we noticed a huge improvement in performance. Due to complexity and length that require to explain this implementation of this technique, we will omit its technical explanation in this paper.

4. Implementation

We have implemented virtio back-end in ViMo-S. ViMo-S is a bare metal VMM running on ARM architecture. In this implementation, we used an Insignal Arndale board with a dual core 1.7GHz Cortex-A15 CPU on a Samsung Exynos 5250 SoC. For the host OS and guest OS, we used Linux 3.8.0 kernel. To accommodate virtio, we put our back-end drivers in Host OS. However, guest OS requires no modification since Linux already has virtio front-end drivers for multiple devices. In measuring the performance of virtio-backend as compared to native speed, we ran two scenarios. In both scenario, we ran two VMs. One was the host OS and another one was guest OS. In this evaluation, we thoroughly evaluated I/O performance. We used IOZone 3.420 as the benchmark application.
For the simplicity, the tests that we conducted are sequential write, sequential read and random read. To accurately measure the disk performance, all operations are direct I/O which means they bypass the buffer cache and go directly to disk. For the first part of the benchmarking, we are varying the size of the files. For the second part, we are varying the size of request file.

After conduction multiple tests, we are getting the results that we shown in Figure 7, 8, 9, 10, and 11. In Figure 7 and 8, we are focusing on write operation. While in other figures, we are focusing on read operation. In Figure 7, 9, and 11, we are varying the operated file size. In other figures, we are varying the request/record size. We break the operated file into few files according to request size. For example, if the file is 32 MB in size and we are using 1 MB as the request size, it means that we have 32 files and each of them has the size of 1 MB.

The result indicated that the performance of virtio disk devices in bare-metal VMM. As compared to host VM, which is running directly on top of hardware, guest VM’s performance is around 66% for write operation. While in read operation, the performance of guest VM is higher than host VM. It is possible because the actual read operation is done by Host VM. Even though we have disabled caching in guest VM during reading but in Host VM it is not. This is confirmed during the reading random reading test as shown in Figure 11. In random read operation, the guest VM’s performance is always under the performance of Host VM’s. In this operation, the caching done in host VM is not taking effect for reading performance in guest VM. The caching of host VM is not taking effect during write operating of guest VM because guest VM asking to flush the writing, which is not done during reading operation.
5. Conclusion

The architecture of supporting Virtio in a bare metal VMM has been presented and explained. The mechanisms, role, and interactions of all components in the architecture have been discussed and defined. The solution works without needing QEMU at all. Thus for virtualization solutions running without QEMU, this architecture is easily incorporated. We have implemented the solution in our VMM and run a benchmarking application on it.

Performance of our implementation of virtio backend is below native speed. Performance is important, but not everything. Portability is an important factor in this solution. Hopefully, by following this architecture, the back-end in host VM can be reusable by many VMMs. Thus in the future, the migration is not only possible for guest VM but also host VM.

Acknowledgement(s)

This work was supported by the ICT R&D program of MSIP/IITP [R0101-15-237, Development of General-Purpose OS and Virtualization Technology to Reduce 30% of Energy for High-density Servers based on Low-power Processors].

References