Kernelized-QEMU: A Study of System-Level Virtual Layer in Linux Kernel*

Yongxuan Xu, Xianglan Chen**, Hua-Ping Chen, and Huang Wang

School of Computer Science and Technology, University of Science and Technology of China
No. 443, Huangshan Road, Baohe District, Hefei City, Anhui Province, PRC
{jyoeiken,ustc}@mail.ustc.edu.cn,
{xlanchen,hpchen}@ustc.edu.cn

Abstract. In this paper, we propose a way to implement the cross-platform system-level virtual layer in Linux kernel to achieve a higher performance, because, the virtual layer can be supported directly by the operating system and can directly interact with the real hardware without the performance cost of virtual device and user-space. As an example, QEMU[1] is embedded into the Linux kernel. The tests show that the performance of kernelized-QEMU is better than QEMU.

Keywords: system-level virtual layer, QEMU, Linux kernel.

1 Introduction

Code migration problem [2] becomes increasingly important with the continuous development of computer architecture. In general, if a new architecture cannot be supported widely by software, it will be difficult to get opportunity to survive. Nowadays, an effective method is generally used to solve this problem by building a virtual layer above the architecture through virtualization to allow the common software running smoothly on it [3].

Actually, system-level virtualization research is not a new technology. In the early 60s and 70s of last century, this technology was used to achieve the high-level software to share the hardware resources by IBM mainframe system [4]. At that time, the goal of the virtualization was to make full use of expensive computer resources to allow more people to use the computer through terminal devices. But nowadays, by system-level virtualization, it is possible to unlock the tight coupling between the specific hardware architecture and software system. The virtualization can organize a variety of computing resources flexibly. It's also possible to achieve scalable transparent computing system architecture through system-level virtualization. In addition, it is possible to improve the use efficiency of the computing resources, and provide the personalized and pervasive computing resources usage environment to the users [5].

* Supported by "the Fundamental Research Funds for the Central Universities" (WK0110000016).

** Corresponding author.

H.-Y. Jeong et al. (eds.), *Advances in Computer Science and Its Applications*,
Lecture Notes in Electrical Engineering 279,
DOI: 10.1007/978-3-642-41674-3_121,

Currently, through dynamic binary translation techniques, many system-level virtualization applications such as VMware and QEMU, simulate the full system to construct the system-level virtual layer. Those applications mainly execute in user space with many other applications also executing in the host computer. To the author's knowledge, it may have the following drawbacks:

- The virtual layer cannot take full advantage of performance of the hardware in host platform including the CPUs and peripherals.
- Needs multilayer calls when interacting with the hardware and frequent switches between user mode and kernel mode, therefore, performance loss is very serious.
- It costs more to use virtual devices than the real device at runtime.
- The performance of virtual layer is deeply influenced by the host environment for different kinds of host applications.

Although KVM [6] runs in kernel space, its virtualization requires the support of hardware, such as Intel VT or AMD V technology. Moreover, KVM is based on the homogeneous platform, which limits the possibility of the integration of heterogeneous computing resources.

Based on the above analysis, we proposed an idea that embeds the virtual layer that supports the cross-platform and does not depend on the hardware into the operating system. It is to make the operating system directly support the virtual layer and the virtual layer occupy the hardware exclusively.

In order to prove the idea is feasible, QEMU was embedded into the Linux kernel, which is called kernelized-QEMU. This paper will present how to construct the kernelized-QEMU, focus on solving the interfaces problem and device emulation problem, and the execution performance evaluation of the virtual layer constructed by kernelized-QEMU.

2 QEMU and the Design of Kernelized-QEMU

2.1 A Short Introduction to QEMU

QEMU is a multi-source multi-objective binary translation system, which supports process-level and system-level virtualization operating modes, with the characteristics of high-speed, cross-platform, open source, easy to transplant, etc [1]. The system-level virtualization refers to QEMU emulates the entire computer system, including one or more processors and a variety of peripheral devices.

QEMU translation system consists of front-end decoder, analysis optimizer, back-end translator and control core [1], which is good enough to support the heterogeneous platform. The front-end decoder is responsible to translate the binary code of the source platform into the intermediate code TCG (Tiny Code Generator). The analysis optimizer uses activity analysis method to remove the partial redundant instruction of TCG code. The back-end translator translates the TCG code into binary code of the target platform, so that the code can be run on the host computer. And the control core is responsible for the control flow of the entire system. The four modules work

together which makes QEMU virtual machine can simulate a heterogeneous virtual layer on the host computer.

2.2 The Design of Kernelized-QEMU

Until recently, there is some lack of knowledge about embedding the virtual machine into the operating system kernel to construct the kernel based virtual layer. In this occasion, we proposed to integrate the QEMU source code to the Linux kernel code and only execute the QEMU to build a system-level virtual layer with a target operating system running over the virtual machine when the kernel completes the initialization. Therefore, the host Linux becomes a tool that is only used to provide full service for the virtual layer. To this purpose, many kernel functions are unwanted and should be removed from the kernel. Also, since there is no host application running to share the resources, the QEMU can use the hardware exclusively. Figure 1 is the schematic diagram of ordinary virtual system and kernelized-QEMU system.

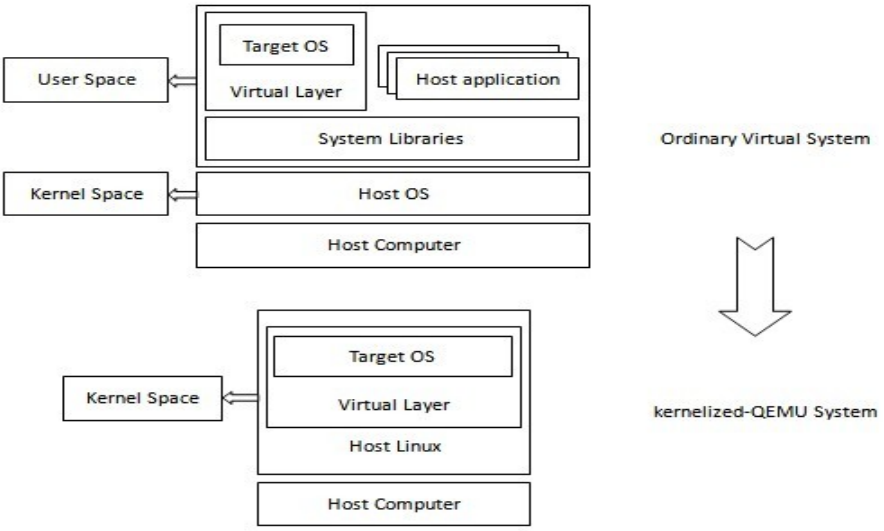

Fig. 1. Ordinary virtual system and kernelized-QEMU system

To make the QEMU embedded into the Linux kernel, there are three major problems: the interfaces problem, device emulation problem and floating point problem. Because the Linux kernel cannot support floating point operation perfectly, we simulated the floating point arithmetic by using fixed point instead of floating point after studied floating point format. The following two chapters will explain in detail about how to solve the interfaces problem and device simulation problem.

3 Interfaces Problem

QEMU in user space needs to call the C library functions, such as open(), close(), read() or write(). Multiple system calls in this process result in frequent switching between user mode and kernel mode, increasing the system overhead. How to get rid of the shackles of the interfaces and various libraries became a problem when the QEMU was embedded into the Linux kernel. After investigation, we proposed that change the interfaces of the library functions that relied by QEMU to the functions implemented by the Linux kernel, and when necessary, implement a part of C library functions in the kernel.

In this paper, the entire modified interfacesare divided into three major categories and analysis in the following three sections.

3.1 Simple Interface

This part of the interfaces almost didn't need to be modified, such as access(), read(), fclose(), fseek(), mprotect(), free(). Most of these interfaces were file operation functions and memory handing functions, because the Linux kernel also need these functions to handle files and memory. The parameters and return values of the interfaces were same as the parameters and return values of user space interfaces.

3.2 Partially Modified Interface

Part of the interfaces could not correspond to the kernel functions. In order to implement the interfaces, some related kernel functions were modified and combined. For example, the kernel function sys_write() and sys_fsync() were combined to construct the interface write(), because the sys_write() does not have the function to make the memory and hard disk in sync; And according to the meaning of the function, sys_read() was used to achieve the interface fgets(). A part of the interfaces were much more complicated to implement, such as open(), fcntl(), fprintf(), pread(), pwrite().

Taking an example of function pwrite(): First, the kernel function sys_write() and sys_fsync() were used to complete basic write function; Then, sys_lseek() was used to change the position of the file pointer; Last, the atomic of pwrite operation had to be guaranteed.

3.3 Rewritten Interface

There were parts of interfaces had to be rewritten because the kernel does not have any related functions. Such interfaces were very few, mainly involved the character conversion functions and the time functions.

4 Device Simulation Problems

Device simulation problems were the key to the experiment. The virtual layer can use the hardware exclusively, so the core idea of the solution was to make the virtual layer interact with the hardware directly.

4.1 Directly Operable I/O

QEMU simulates VGA to control the output of the virtual layer. And there are two main aspects to operate VGA: VGA port that explored to the user and VGA video memory (memory address: 0xA0000 to 0xC0000). The virtual registers of the simulated VGA (not the real VGA) are first changed when the upper operating system wants to change the VGA mode in the ordinary QEMU system. In order to make kernelized-QEMU can operate VGA directly, the two aspects were both modified.

The modification of VGA port operation was associated with the hardware, in the experiment. And the computer of X86 architecture was used as the experiment platform. In such cases, the kernelized-QEMU functions that registered to read and write the VGA ports are as follows:

```
register_ioport_write(0x3c0, 16, 1, cirrus_vga_ioport_write, s);
register_ioport_read(0x3c0, 16, 1, cirrus_vga_ioport_read, s);
```

Therefore, cirrus_vga_ioport_read() and cirrus_vga_ioport_write() were rewritten to achieve the function that operating VGA directly. The function "static void cirrus_vga_ioport_write(void *opaque, uint32_t addr, uint32_t val)" of QEMU writes the value of "val" to the address of "addr". According to the meaning of the function, the parameter "addr" was checked and converted to the real address, and then the "val" was written to the address in kernelized-QEMU. The function lq_inb() was added into the function cirrus_vga_ioport_write() to write the port of VGA. And the modification of the function cirrus_vga_ioport_read() was similar to cirrus_vga_ioport_write(), function lq_outb() was added to read the value of the port. The method of Linux kernel was used for reference to accomplish the specific methods to read or write VGA port.

The virtual layer constructed by kernelized-QEMU is run in kernel space, so it only can access the memory address that above 0xC0000000 (1 GB kernel memory, 32-bit Linux kernel) [8] The address of VGA video memory had to plus the offset of kernel address, otherwise, the program could not access the memory.

The modification of VGA video memory operation was associated with the hardware too. Function groups cirrus_vga_mem_read and cirrus_vga_mem_write were registered to read or write VGA video memory in the experiment. And after a series of calls, cirrus_vga_mem_readb() and cirruss_vga_mem_writeb() were modified to read or write the video memory address.

The pseudocode of function cirrus_vga_mem_writeb():

```
Begin
        char * kernel_vga_offset = (char *) 0xc00a0000
        ifaddr>= 0x0000 AND addr< 0x20000
                returnwriteb(mem_value, kernel_vga_offset+addr);
        else
                ERROR
End
```

The Linux kernel can detect the input event and stores it to the structure input_event. And the traditional virtual layer constructed by QEMU requires the support of specific libraries, such as SDL (Simple DirectMedia Layer), to handle the keyboard events. In order to remove the dependencies of libraries, the way to obtain keyboard events was changed to the host Linux and the keyboard handing function of QEMU was used to process the input through the keycode.

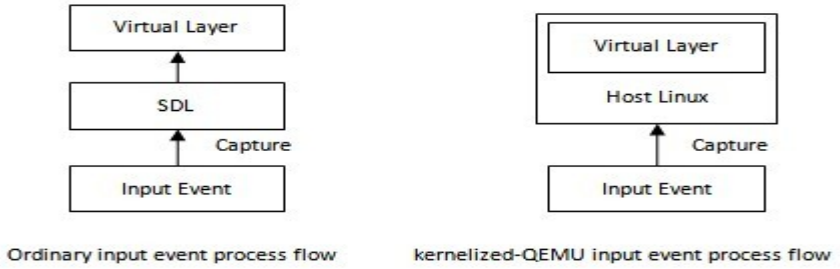

Fig. 2. Ordinary input event process flow and kernelized-QEMU method

Figure 2 is the schematic diagram of the ordinary input event process flow and the improved kernelized-QEMU method. The host Linux processing operation of the input event was truncated and the keycode was delivered to the keyboard event processing interface which provided by kernelized-QEMU. Each keycode had its own state to represent the three states: press, hold or release, and the parameter "value" was used to show the state in kernelized-QEMU. The key was pressed when "value" equaled 1; the key was held when "value" equaled 2; and the key was released when "value" equaled 0. Due to the only keycode was received by the keyboard event processing interface of kernelized-QEMU, an intermediary layer was built to deal with the state. The keycode was delivered to the interface immediately when the key was pressed or held (value > 0); otherwise, the keycode was plus 128 to represent the key was released before delivering.

4.2 DMA Simulation

Asynchronous input and output process was used to simulate DMA in QEMU. And it was initialized when the virtual layer opened the hard disk image at the first time. The

interface paio_submit() was used to submit the asynchronous I/O request to read or write the hard disk when DMA request came. Then, a thread was created to respond to specific request. When the thread had completed the task, a custom signal was sent to notice the main thread. The signal response function which was registered by main thread received the signal, wrote the communication pipe and submitted an event, which was called qemu_event, to the virtual layer. A soft interrupt was implemented through the QEMU event mechanism, and the virtual layer started to check the state of communication pipe that written by the signal response function. If the pipe had been written, function posix_aio_read() was called to read and empty the pipe. At last, function posix_aio_process_queue() was called to complete the remaining work. Above is the process of DMA that simulated by QEMU through asynchronous I/O.

To simulate DMA in kernelized-QEMU, three questions had to be solved: signal, pipe and thread. Because the virtual layer was run in kernel space, through signal and pipe to notice the virtual layer that DMA reading or writing process has been completed was abandoned, function posix_aio_process_queue() was called directly without used QEMU event mechanism while the response of DMA request was completed. The kernel thread was used to instead of pthread (POSIX thread) to create the thread. And the mutex lock of the Linux kernel was used to protect the critical resource.

In addition, another way to simulate DMA is changing asynchronous I/O to synchronous I/O. In this situation, function aio_thread() was called, not created a new thread, when the DMA request was submitted.

5 Experiments and Discussion

Due to the limitation of experimental conditions, kernelized-QEMU was deployed on an X86 platform. And a Linux kernel was run on the system-level virtual layer constructed by kernelized-QEMU. The kernel boot time in kernelized-QEMU system was faster than the time in QEMU system (Figure 3).

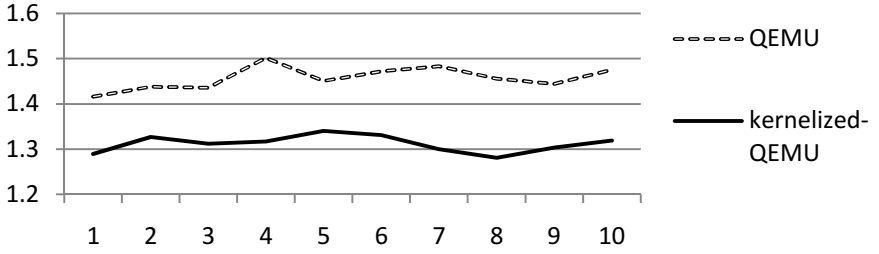

Fig. 3. The kernel boot time (Unit: Second)

The program for calculating the value of π was run to test the performance of the virtual layer. Taylor formula was used to calculate the arctangent value to get the value of π (accurate to 1000 decimal places) in the program. And the virtual layer constructed by kernelized-QEMU used less time than QEMU at all time (Figure 4).

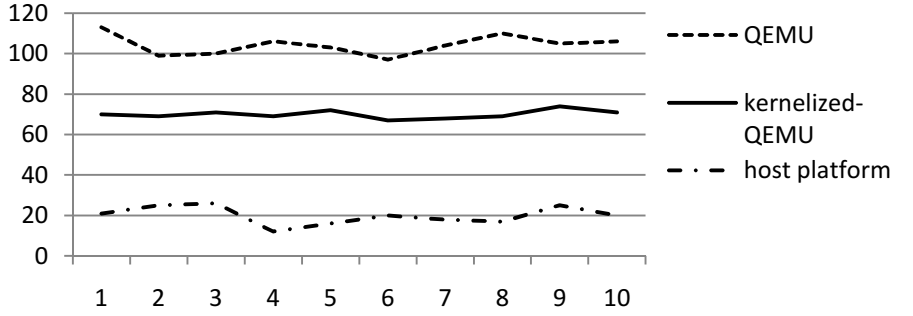

Fig. 4. The time to calculate π (Unit: millisecond)

Figure 3 and figure 4 shows that the operation efficiency of kernelized-QEMU system is better than ordinary QEMU system, which consistent with our vision. The virtual layer implemented in the Linux kernel can reduce the multilayer calls and take full advantage of the hardware performance of the host platform, thereby improve the performance of the virtual layer. And the operating system can be started by kernelized-QEMU automatically and immediately, which means the virtual layer is transparent to the users.

Currently, kernelized-QEMU system cannot apply the memory over the kernel address space. In the future, we will solve this problem, deploy the kernelized-QEMU system to the heterogeneous platform and optimize the virtual layer according to the hardware features of the host platform.

6 Conclusion

The experiments proved that the kernel based system-level virtual layer can improve its operation efficiency. Thus, it may bring up some new ideas for the development of the full system simulation on the heterogeneous platform.

References

1. Bellard, F.: QEMU, a fast and portable dynamic translator. USENIX (2005)
2. Kontogiannis, K., Martin, J., Wong, K., et al.: Code migration through transformations: An experience report. CASCON First Decade High Impact Papers. IBM Corp., 201–213 (2010)
3. Rosenblum, M., Garfinkel, T.: Virtual machine monitors: Current technology and future trends. Computer 38(5), 39–47 (2005)
4. Butts, T.H., Burris Jr, S.H., Clark, S.J., et al.: Server and web browser terminal emulator for persistent connection to a legacy host system and method of operation: U.S. Patent 5,754,830 (May 19, 1998)
5. Vallee, G., Naughton, T., Engelmann, C., et al.: System-level virtualization for high performance computing. In: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 636–643. IEEE (2008)

6. Kernel Based Virtual Machine, http://kvm.qumranet.com/kvmwiki
7. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly Media, Inc. (2005)
8. Yabin, H., et al.: An Optimization Approach for QEMU. In: 2009 1st International Conference on Information Science and Engineering (ICISE 2009), pp. 129–132 (2009)