Dynamic Translator-based Virtualization

Yuki Kinebuchi[1], Hidenari Koshimae[1], Shuichi Oikawa[2], and Tatsuo Nakajima[1]

[1] Department of Computer Science, Waseda University
{yukikine, hide, tatsuo}@dcl.info.waseda.ac.jp
[2] Department of Computer Science, University of Tsukuba
shui@cs.tsukuba.ac.jp

Abstract. Microkernels and virtual machine monitors are both utilized as platforms for running operating systems. Although there are many similarities in their designs and features, they have opposite advantages and drawbacks. A microkernel based system is highly portable. However, the interface it exposes is inflexible and incompatible with other real hardware interfaces. In contrast, a virtual machine monitor interface is identical to a specific real hardware interface. However, the implementation of virtual machine monitors highly depends on processor architectures and specific hardwares.
In this paper, we present a new model of virtual machine monitor, a flexible dynamic translator constructed on a portable microkernel. Our model offers both high portability and compatibility. Moreover, its flexible interface could be reconfigured to support various types of hardware interfaces. The results of the evaluation show that the performance of our prototype system is unsatisfactory, so we propose some techniques to improve its performance as future work.

1 Introduction

Microkernels and virtual machine monitors (VMMs) have common purpose which is to run operating systems on them. There are many similarities in their designs and features. However, since their primary aims differ from each other, they have opposite advantages and drawbacks in a point of portability and compatibility.

Microkernels started with the redesigning of conventional operating systems. In order to reduce the size and the complexity of a kernel, the number of its functions was minimized and some traditional kernel functionality were moved to the application level. The resulting system realizes a moduled and highly portable structure. Since their interfaces differ from real hardwares interface, there is a drawback that a guest operating system needs to be modified to run on a microkernel.

In contrast, VMMs aim to the reuse of commodity operating systems. Like a microkernel, VMM is a small and simple program running in privileged mode, but provides an interface identical or almost identical to the underlying real hardware. Thus, operating systems can run on VMM without any or with minimum modification. Since its implementation highly depends on a processor architecture and a specific hardware, the portability of itself is low.

In this paper, we propose a new model of constructing a VMM, which is a flexible dynamic translator constructed on a portable microkernel. The past virtual execution platforms have implemented their interfaces directly on hardware interfaces. They have a hardware dependent, not portable and inflexible interface implementation. In our model the hardware dependent layer is split from the interface implementation. The underlying hardware interface is abstracted by the microkernel, which provides a uniform interface on different architectures to the machine emulator running on it. The machine emulator provides an flexible interface implementation, which enables the execution of unmodified operating systems and can be reconfigured to support different hardware interfaces. We implemented a prototype system by porting an existing portable machine emulator to an existing microkernel, and made some evaluations on its performance.

The next section compares microkernels and VMMs. Section 3 introduces some related work. Section 4 introduces the implementation of our prototype system. Section 5 proposes some applications using our model. Section 6 introduces the results of the evaluation. Section 7 discusses some performance issues and future directions. Finally Section 8 concludes the paper.

2 Microkernel vs VMM

This section compares advantages and disadvantages of microkernels, VMMs, and propose our new virtualization model which integrates the advantages of both of them.

A microkernel is a small operating system supporting only a minimum set of API. Microkernels are used as bases for constructing operating systems. The interface provided by a microkernel is an abstract hardware interface, which is different from any existing hardware interfaces. Therefore, when running an existing operating system on a microkernel, its architecture dependent part must be modified as shown in Figure 1 (a). This is the drawback of the microkernel-based approach. The advantage of using microkernels is their portability. Since a microkernel-based system splits the hardware dependent layer and operating system services, the system could be supported on a different hardware by porting only a part of the microkernel.

A virtual machine monitor is a software that enables multiple operating systems to run on a single hardware by giving the illusion of using a whole hardware to each operating system. The interface provided by a virtual machine monitor is almost identical to a specific existing hardware interface. The advantage to use VMMs is that they do not require the modification of guest operating systems to be run on it. Figure 1 (b) shows the operating system for architecture A running directly on the hardware of architecture A. The operating system could run on the virtual machine monitor without modifying its architecture dependent part as shown on in Figure 1 (c). The drawback of the virtual machine monitor-based approach is strong dependency to the underlying hardware interface. Moreover, the architecture offered by the virtual machine monitor and the interface of the underlying hardware should be the same.

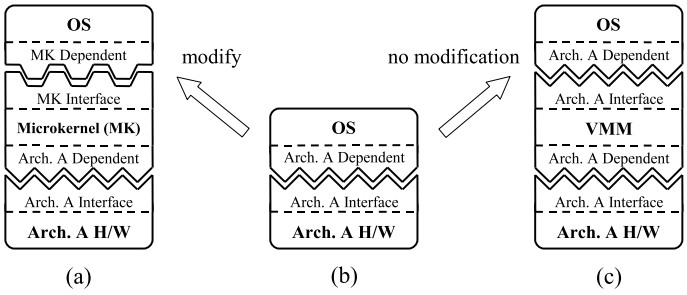

Fig. 1. OS on VMM and a microkernel

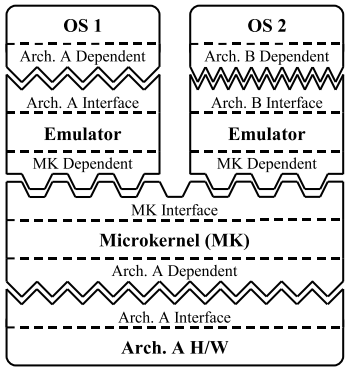

Fig. 2. Machine Emulator on a microkernel

We propose a new virtualization architecture, which has the advantages of both microkernels and virtual machine monitors, the portability and the compatible interface with existing hardware architectures. Figure 2 shows the overview of our model. Flexible machine emulators are running on a portable microkernel. The emulator provides an interface compatible with an existing architecture interface. The emulator on the lefthand side of the figure offers the interface of architecture A, which executes an unmodified operating system. In addition, the emulator could be reconfigured to execute operating systems on various different architectures. The emulator on the lefthand side of the figure is reconfigured to offer the interface of architecture B. The implementation of the emulator depends on the microkernel interface but not the host architecture. The underlying microkernel hides the hardware interface from the emulator and offers a uniform interface. When the underlying hardware changes, only the small architecture dependent part of the microkernel is modified. Therefore the porting cost of the system is decreased dramatically.

3 Related Work

In this section, we introduce an existing machine emulator and some existing virtual machines.

Bochs[6] is a machine emulator which emulates the x86 architecture machine. It has a capability to run guest operating systems built for the x86 architecture without any modifications. The code of Bochs is written in C++, which can be compiled to run on various operating systems. Although it supports a high portability over operating systems, the portability of supporting new hardware platform depends on the host operating system implementation.

Xen[1] is VMM leveraging a virtualization technique called para-virtualization. Using para-virtualization increases the performance of guest operating system, but it requires the modification of guest operating systems to be virtualized. In addition, the implementation of Xen highly depends on the x86 architecture, it has low portability.

VMware Workstation[8] is VMM that can run unmodified operating systems built for the x86 architecture. It runs as an application running on commodity operating systems such as Linux and Windows. It installs VMM running in the privileged level as a device driver in order to use privileged level instructions. This is to increase the performance of running guest operating systems. At the same time it increases the dependency on both the host operating system and the host hardware architecture.

4 Constructing Machine Emulator on Microkernel

4.1 Overview

We implemented a prototype system of our proposed model by porting the QEMU machine emulator[2] to the L4Ka::Pistachio microkernel[9] (L4 for short) with the Kenge[4] environment. The architecture of the prototype system is shown in Figure 3. QEMU, originally running on Linux, is modified to run as an application on L4. Each of QEMU can run a single guest operating system on it. By running multiple QEMU, multiple guest operating systems can run simultaneously on a single hardware. Kenge provides some libraries and servers that facilitate the constructions of applications on L4.

The following subsections briefly introduce QEMU, L4 and Kenge followed by the description of virtual devices.

4.2 QEMU

QEMU is a portable machine emulator, which emulates entire computer interface including CPU, memory and hardware devices. It runs as an application on commodity operating systems such as Linux, Windows, Mac OS X and Free BSD. Since QEMU emphasizes a portability, it supports various processor architectures as the host and the guest architecture. Currently, it supports x86,

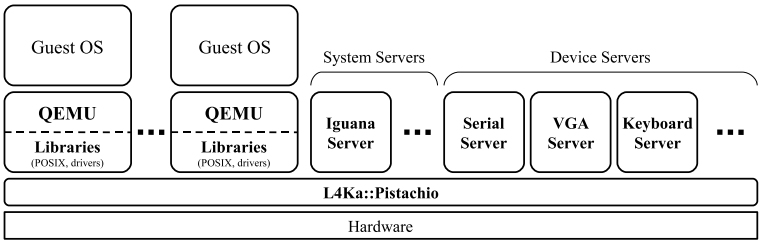

Fig. 3. A machine emulator on a microkernel

x86_64, ARM, SPARC, PowerPC and MIPS for the guest architecture, x86, x86_64 and PowerPC for the host architecture. In addition, the host and the guest architecture can be different.

In order to provide a virtual CPU running guest programs, QEMU leverages the technique of a dynamic translation. It splits a guest instruction into pseudo microcodes that consists of host instructions. The translation is continued up to the next jump instruction, and the chunk of translated codes is put in a buffer as a unit of translated block (TB). TBs are reused when corresponding codes are executed again. Each microcode is written in the C language that is compiled to native code by GCC on the building stage of QEMU. Since the C codes can be compiled for various architectures by using different compilers, the porting costs are kept low.

QEMU also provides virtual devices that offer interfaces of existing hardware devices. The virtual devices for original QEMU are implemented using functions and libraries of the host commodity operating system. For example, data contained in a virtual hard disk is saved to a file on a host filesystem as a hard-disk image. In addition, the inputs and outputs of a guest operating system for a display, keyboard and mouse are processed using host graphic libraries and window systems. Since these libraries are not supported on L4, we modified the virtual devices to run in the L4 environment. The implementation of virtual devices on L4 is described in Section 4.5.

4.3 L4Ka::Pistachio

L4Ka::Pistachio is a portable microkernel. L4 itself only provides primitive functions to support thread management, address space management and IPC. Facilities supported by modern operating systems are moved to the user level and implemented as servers and libraries. For example, device management is moved to the user level, and implemented as device servers. Applications running on the microkernel interface with device servers to access hardwares.

4.4 Kenge and Iguana

Since L4 offers only primitive functions, we worked on the Kenge environment which helps the development of applications for L4. Kenge consists of a system build environment, libraries and servers, including the Iguana server[3] and some device servers. The libraries provide some POSIX functions, device drivers and some RPC stubs to interact with servers. The detail of device servers is described in the next section.

Iguana is a privileged server which manages resources such as memory, CPU and capabilities to access those resources. It also provides some high-level functions for applications running in L4 to create and delete threads, map and unmap memory regions.

QEMU on L4 uses some POSIX functions provided by these libraries, but not all the POSIX functions used by QEMU are provided by Kenge. Therefore we added some POSIX functions used by QEMU to help the porting.

4.5 Virtual Devices

This section describes the two different models of virtual hardware implementation. The first implementation is the model which virtual devices interact with device servers running beside QEMU and other applications. The second implementation is the model which links virtual devices with device driver libraries which let them directly interact with hardwares.

Device Server. A device server is a special application running on L4 which manages a device I/O to a specific hardware device. Although device servers are running in unprivileged mode, they are given a permission to access hardware devices. Therefore device drivers contained in device server can directly interact with hardware devices.

QEMU interact with device servers through the IPC function provided by L4. When a guest operating system writes to a virtual hardware, it transfers the written data to a corresponding device server using IPC. The device server receives the data, it invokes the device driver function, and perform an actual device output (Figure 4 (a)).

When multiple guest operating systems are sharing a single real device, the device server should arbitrate inputs and outputs. For example, the console server we made, can switch to which QEMU it transfers data, and let multiple guest operating systems to share a single display and keyboard. The drawback of this model is that frequent IPC between a guest operating system and device servers triggers frequent context switch.

Internal Device Driver. The other is the model using device driver library. QEMU, using the module linked to itself, interfaces with hardware directly (Figure 4 (b)). In this model there is no overhead of IPC because the data does not go through the device server, however a real device cannot be shared among

multiple guest operating systems. For example, the ported QEMU accesses the VGA device using VGA device driver library. In this case, the guest operating system directly writes to real VRAM.

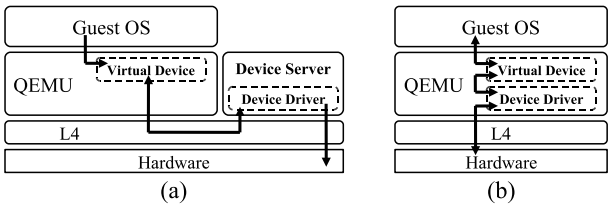

Fig. 4. The implementation of virtual devices

5 Applications Using QEMU on L4

This section propose some applications using our system.

5.1 Emulating Multiple Architectures

The primary use of our system is the reuse of existing operating systems on top of various types of architectures. Since QEMU exposes an interface compatible with existing hardwares, guest operating systems can run on top of it without any modifications. QEMU can run a guest operating system even if the host and the guest architecture are different as shown in Figure 5. The console server splits the monitor into four parts and makes each guest Linux to use one of them.

5.2 Anomaly Detection/Recovery

Alex Ho et al. proposed a taint-based protection using a machine emulator[5]. In normal times, an operating system runs on VMM. When the CPU is executing a code that interacts with data downloaded through the internet, the execution is dynamically switched on to the machine emulator. In this way, it reduces the performance degradation and protect the system from tainted data.

Using our system, we propose a similar system that offers an anomaly detection and recovery (Figure 6). In normal times, applications run directly on L4. When the system finds the symptom of application anomaly, the application is migrated to run on QEMU. QEMU runs the application and analyzes it in detail. When it detects an anomaly, it stops the execution of the application, and if possible, it recovers the application and puts it back to run directly on L4 again. In this way, the system can realize the anomaly detection and recovery with near-to-native performance.

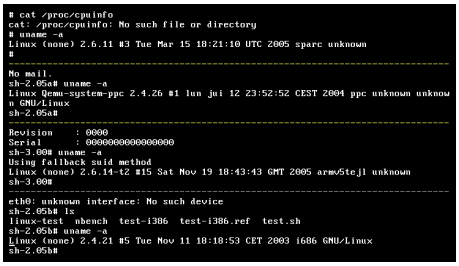

Fig. 5. SPARC, PowerPC, ARM and x86 Linux on the prototype system

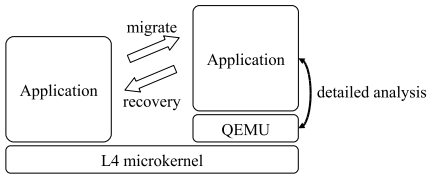

Fig. 6. Anomaly detection

Table 1. LMbench measurement result

	x86(Native)	x86	ARM	SPARC
lat_syscall (μsec)	0.2634	3.0967	18.9526	3.0504
lat_ctx (μsec)	0.54	48.10	80.13	83.09
bw_mem rd (MB/s)	9328.49	1051.80	618.39	508.50
bw_mem wr (MB/s)	5509.96	597.21	436.50	379.23
bw_file_rd (MB/s)	1557.27	36.29	45.42	41.47

6 Evaluation

In this section, we evaluate the performance of Linux running on QEMU on L4. We used LMbench[7] to measure their performance. LMbench is a cross platform benchmark to measure the performance of operating system primitives. We built LMbench for three different architectures; x86, SPARC and ARM. The measurements were performed on non-virtualized (native) Linux for x86 architecture, virtualized Linux for x86, ARM and SPARC. For non-virtualized and virtualized Linux for x86, we used the same kernel and root filesystem. The machine we used for the measurement is IBM ThinkPad R40, with 1.3GHz CPU and 768MB memory. Dynamic frequency control is disabled for accuracy.

Table 1 shows the result of the measurement. We measured the system call latency, context switch latency, the bandwidth of reading from and writing to the memory, and the bandwidth of reading a file. The first row of the table shows the system call performance. Comparing non-virtualized and virtualized Linux for x86, the performance is decreased by a factor of 11. The second row shows the latency of the context switch. The performance decreased by approximately a factor of 90. The third and forth row show the bandwidth of reading and writing a memory. The throughput decreased by a factor of 10. The memory access speed of programs running on QEMU is the one of the major overhead of QEMU, which we describe in more detail in Section 7.2. The last row shows the bandwidth of reading a file. Comparing non-virtualized and virtualized Linux for x86, the performance is decreased by approximately a factor of 40.

7 Discussion

Running QEMU on L4 realizes the running of multiple operating systems simultaneously on a single hardware, isolation between the guest operating systems with giving them the illusion of using a hardware by itself, and the reuse of operating systems without modification even for the operating system for different architectures. However, as shown in Section 6, the performance of guest operating systems is degraded comparing to native Linux. In this section we propose techniques to improve the performance of dynamic translators running on microkernels. Furthermore we propose future directions of this research.

7.1 Hypercall

As described in Section 4.2, QEMU translates guest codes to host codes with dynamic translation before executing them. The dynamic translation produce two types of overheads, the direct and the indirect. The direct overhead is the processing time of dynamic translation itself. Since every single instruction is translated to corresponding microcodes at the execution time, the execution of a code is significantly delayed. When the guest code is executed again, it is executed without any translation by reusing TB. The indirect overhead derives from the inefficient code contained in TB. Since the guest code is translated to microcode which consists of several host codes, the number of instructions contained in TB is longer than the corresponding original guest code.

In order to reduce these overheads, we propose to implement a *hypercall* by extending an instruction set provided by QEMU. In time of execution an extended instruction is translated to a host instruction which directly calls a QEMU function. For instance, we propose the implementation of efficient device driver for guest operating systems using these extended instructions. We replace the code included in a function exposed by the device driver, say `write()`, with a single extended instruction. When the function is executed, the instruction is translated to a host instruction which directly invokes a function in QEMU which may be a device driver function or RPC stub calling a device server function. In this way, the direct and the indirect overhead of the dynamic translation can be decreased.

7.2 MMU with Map Function

Since many modern processors has MMU, virtual machines and emulators needs to support a function equal to MMU in some fashion.

QEMU provides *software MMU* which emulates MMU only with software function. Software MMU interposes every single memory access done by guest programs running on top of QEMU. When the guest program accesses a memory, software MMU perform a lookup through the page tables constructed by a guest operating system and translates a virtual address to a physical address. Therefore, single memory access expands to multiple memory accesses including

the access to page tables. The resulting time for memory access would be factor of ten.

Unlike commodity operating systems, L4 provides APIs to manage address spaces. Using these APIs, an application on L4 can create new address spaces and map memory section into them. We propose the implementation of new virtual MMU which employs L4 APIs. The virtual MMU creates and maps memory regions into the space according to page tables constructed by a guest operating system. A guest program is executed in a separate address space, so it can access memory directly without interposed by software MMU. The implementation should dramatically decrease the overhead of software MMU.

8 Conclusion

In this paper, we proposed the model of running a dynamic translator on a microkernel and implemented the prototype system. We also proposed some sample applications using our model and evaluated the performance.

The model we proposed has a greater flexibility and higher portability than existing VMMs and microkernels. The prototype has shown it by running multiple guest operating systems for different architectures simultaneously on a single hardware. Our model is expected to be useful for the basis for reusing existing operating systems and applications, debugging and a system that requires high degree of security and reliability.

References

1. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.
2. F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2005.
3. Embedded, Real-Time, and Operating Systems. Iguana. `http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/`.
4. Embedded, Real-Time, and Operating Systems. Kenge. `http://www.ertos.nicta.com.au/software/kenge/`.
5. A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation and intel research cambridge. In *Proceedings of the EuroSys 2006*, Leuven, Belgium, Apr. 18–21 2006.
6. K. P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux J.*, 1996(29es):7, 1996.
7. L. McVoy and C. Staelin. Lmbench - tools for performance analysis. `http://www.bitmover.com/lmbench/`.
8. J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.
9. System Architecture Group. L4Ka::Pistachio microkernel. `http://l4ka.org/projects/pistachio/`.