Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Studienarbeit

In Place Migration by Using Pre-Virtualization

Yaowei Yang

Verantwortlicher Betreuer: Prof.Dr. Frank Bellosa

Betreuender Mitarbeiter: Joshua LeVasseur

Stand: March 20, 2006

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Litraturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only citied sources have been used.

Karlsruhe, March 20, 2006 Yaowei Yang

Abstract

The development of hardware virtualization makes it possible to migrate entire operating system instances across distinct physical machines. Different strategies have been discussed recently, and *self migration* is one of those impressive ones. By using this solution the OS migration does not need help from the underlying hypervisor. Furthermore the services downtime during the migration can be minimized. Nevertheless self migration has it own shortcomings.

With the help of *pre-virtualization* technique, we try to keep all advantages of self migration and overcome its main drawback by using a new migration technique, "In Place Migration", in this thesis. We introduce firstly pre-virtualization, compare it with other classical hardware virtualization, show its strength, and then discuss how we use the new concept of "In Place Virtual Machine Monitor"(In Place VMM) from pre-virtualization to design and implement "*In Place Migration*".

Contents

Chapter 1

Introduction

Hardware virtualization makes it possible for one physical machine to host many same or different operating system instances. Normally it will be used to consolidate servers, normalize hardware resources, and isolate applications on the same physical server. Another important advantage of hardware virtualization is that migrating entire operating system instances between hosts is much easier with its support. From the virtualization software's perspective every operating system instance atop the virtualization software is encapsulated in an object named virtual machine(VM). To migrate an operating system people need only copy VM state from source to the destination. Because of this advantage many techniques of migration have been discussed in recent years.

Among variants of migration techniques *live migration* discussed the possibility of migrating a system with very short downtime. The clients can be unaware of services' relocation while the migrating progress is running. On base of live migration, another possibility of migration, called self-migration, has been discussed. This art of migration tries to migrate the entire operating system instance by the operating system itself. Self migration is described to be more secure, more flexible and more portable than VMM-Driven live migration(section 2.2.2). But self migration has its own drawback. This methods embeds a paradox that an operating system must save its suspended state when it is still running. Although the problem is solved with a pseudo final state(that means a machine state checkpoint before the real suspending of machine) by the author, it is better if we can save the real final state of VM for migration. Moreover, as self-migration settles down in the operating system's space, the migration infrastructure must be re-implemented for different operating systems.

"In Place Virtual Machine Monitor"(In Place VMM), a new concept from *pre-virtualization* technique, seems to show us a way for keeping the advantages of self migration and overcoming its drawback as well. "In Place VMM" is originally introduced as a bridge between the world of *pure-virtualization*, which needs minor engineer effort for porting operating system to its virtualization environment, and the world of *para-virtualization*, which has much better performance than pure-virtualization, and try to combine both advantages. It lives in the same VM space together with its supporting operating system, uses high level interface(for performance purpose) provided by the underlying privileged resource monitor, i.e. hypervisor, to emulate the Platform identical interface(for zero engineer effort) for the operating system. Because of this responsibility, we can use the "In Place VMM" to collect all necessary VM states for OS migration. We will show this possibility in this thesis and try to show that Migration by using "In Place VMM", called "In Place Migration", can achieve our goal of keeping Self-Migration's advantages and overcoming its drawback.

In chapter 2 the related works and background of virtualization and migration technique will be introduced. Pure-virtualization, para-virtualization and pre-virtualization will be described and compared. We will also carry out the detailed explanation for live migration and self migration. Chapter 3 figures out the major design part of our "In Place Migration". In the end, in Chapter 4, a part of "In Place Migration" implementation on top of L4 Marzipan, a native resource hypervisor, will be explained.

Chapter 2

Background

2.1 Virtualization

2.1.1 Overview

Virtualization, which was firstly formally discussed in the 70s' [4], is revived in recent years. One obvious motivation is that modern small computer system is powerful enough for hosting more than one operating system to improve its utilization. It can significantly alleviate the hardware cost. But more important reason for rebound of virtualization is that it can provide a clear isolation approach for different services running on one bare hardware. This improves the security, flexibility and portability of services. Furthermore virtualization can also solve a number of problems in many other quite dissimilar contexts, including intrusion detection, debugging, backward compatibility with older or out-of-production hardware and so on [8].

Commonly virtualization replicates the resource into several virtual machines(VMs). Different programs, called "guest", can run in the VM in term of the virtualized platform. These programs range from simple application program to the entire operating system(OS) instance. According to different virtualization type, Instructions Set Architecture(ISA), Application Binary Interface(ABI) and Application Programming Interface(API) will be emulated in the virtualization environment [14]. The emulation normally happens in a layer between bare hardware platform and VMs, called virtual machine monitor(VMM). It has the full control over all hardware resources and distributes the resource among VMs.

Two types of VM were discussed in paper [14] according to the virtualization level. Process VM, one of these two types, is introduces as a virtualization environment with virtual ABI and API. Replication, emulation and optimization will be provided by process VMs [14]. It doesn't concern ISA because from the perspective of process machine is represented only by ABI and API. Also guests in VM is limited to process. The most familiar process VMs are High-Level-Language VMs, e.g. Sun Microsystems Java VM [10]. Process migration can be discussed under this VM definition. But in this thesis, which is about entire operating system migration, more attention should be called to the other type of VM, system VM. In constrast to process VM, system VM duplicates hardware resource by virtualizing the platform ISA for the guests with either the host machine ISA or the host operating system's ABI and API. With this ISA level virtualization infrastructure, a conventional operating system can run in the virtualization environment either with or without modification. As we discussed above, the

VMM of System VM has access to, and manages, all the hardware resource. For sharing the resources among many VMs, it captures or rewrites the privileged instructions or operations executed by the guest OS, intercepts them, checks their correctness and performs them on behalf of the guest [14]. Three properties of virtualization have been argued in [12].

> **The efficiency property** All innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the control program
>
> **The resource control property** It must be impossible for that arbitrary program to affect the system resources, i.e. memory, available to it; the allocator of the control program is to be invoked upon any attempt.
>
> **The equivalence property** Any program K executing with a control program resident, with two possible exceptions, performs in a manner indistinguishable from the case when the control program did not exist and K had whatever freedom of access to privileged instructions that the programmer had intended.

The major problem in the hardware virtualization is to balance the leverage between VM performance and engineer cost for porting an operating system into the virtualization environment, in another words, keeping both efficiency property and equivalence property. For this purpose many hardware virtualization techniques have been developed. But before the appearance of pre-virtualization, only one of these two important properties can be well kept. In the following sections pure-virtualization, which aims at the low engineer cost, and para-virtualization, which in contrast reaches better VM performance, will be introduced. After that pre-virtualization will be discussed and compared with its progenitors.

2.1.2 Pure- and Para-Virtualization

Pure-virtualization is a classical hardware virtualization. It virtualizes the whole platform ISA. All resources are in fact managed and distributed by VMM. From the perspective of guest OS, it would not be aware that they are running in a virtualization environment. Major advantage of this design is that minor engineer effort is needed for porting a guest OS to the virtualization environment, because virtual machine is identical to the raw machine from perspective of guest OS. Nevertheless there are some problems existing for pure-virtualization. While a large portion of instructions can be directly forwarded to the real hardware and executed, a portion of instructions, which can interfere the virtualization environment, should be executed with the intervention from VMM. These instructions are normally privileged instructions, e.g. enable and disable interrupt, I/O instructions, etc. Because VM is mostly running in the non-privileged user space, VMM can control these instructions' execution by catching the system traps caused by illegal accessing from these instructions, but some architectures do not support this pure-virtualization solution, e.g. IA-32. Executing certain supervisor instructions, which should be caught by the VMM, fail silently rather than raising a convenient trap on these platforms [11]. In order to solve this problem, the complexity and overhead of virtualization may be increased dramatically. For example, VMWare ESX server tries to catch these special instructions by dynamically inserting code. More CPU cycles and other resources, e.g. memory, are wasted with this solution. One simple instruction may be executed with tons of additional detection and simulation codes. This definitely increases the executing cost and reduces the performance of virtual machine. A conclusion can be drawn from the discussion about the three properties of virtual machine that pure-virtualization sacrifices efficiency property for the equivalence property on some platform.

In contrast to pure-virtualization para-virtualization [11] pays more attention to the efficiency property rather than equivalence property. It doesn't identically emulate all low-level hardware interfaces; instead, it maps some of them(devices, interrupts) to a high-level interface abstraction, e.g. Xen. Furthermore some para-virtualization technique is only designed for special purpose and eliminates part of ISA for better performance, e.g. Denali. Because of these specialties, VMM of para-virtualization is also called hypervisor. By using the high-level interface abstraction, mode switch and related privileged operation can evidently be reduced. The sensitive instructions, which doesn't cause any trap during their illegal execution, can be replace by customized high level interfaces and doesn't need to be caught any more. Furthermore the guest operating system is also permitted to directly access some privileged low-level interfaces including reading page table, reading real clock time, etc. This makes some operations in the guest OS more precise and effective. With these options the virtual machine performance was improved close to the performance of native system. But a penalty is incurred. In order to use the high level interface provided by para-virtualization technique, the original operating system must be modified to fit the virtualization environment. Intuitively the engineer cost for porting different existing operating system is enormous.

2.1.3 Pre-Virtualization

As we described in last section, efficiency property is sacrificed for equivalence property in pure-virtualization, whereas equivalence property is heavily broken in order to get much better VM performance in para-virtualization. In other words, it seems no way to balance the lever between engineer cost for porting Guest OS to the virtualization environment and high virtual machine performance. But this is true only before the appearance of pre-virtualization technique. By using the API translator in pre-virtualization, which is called In Place VMM, the para-virtualization high level API will be mapped back into Guest-visible Platform API, which is identical to the low-level instruction sets from the Guest OS' perspective. By doing so, the modification of Guest OS can be avoided. From the perspective of Guest OS the underlying virtualization environment is similar to the pure-virtualization. From the hardware abstraction's perspective they will be virtualized by a para-virtualization hypervisor. There are two main concepts introduced by pre-virtualization, i.e. Afterburning and In Place VMM.

Afterburning

In order to redirect the original sensitive instruction to their scapegoat, Pre-Virtualization must know the place where these instructions will be executed. Unlike conventional "Trap Catcher" mechanism in pure-virtualization, Afterburning is used in Pre-Virtualization technique to detect and replace the sensitive instruction. Afterburning occurs only during the compilation stage. pre-virtualization changes assembly language compiler, e.g. gas, so that the compiler can annotate Guest OS source codes, find out sensitive instruction and replace them with customized emulation instructions (a number of macros). By doing so the sensitive instructions doesn't need to be dynamically caught any more when the Guest OS is running. The result is that instruction detection overhead in pure-virtualization was effectively removed and VM performance is improved. We call the binary, that has been afterburned, virtualization friendly binary.

As a result, afterburning replaces sensitive instructions with customized functionaly equivalent macros. According to different types of these macros, afterburning can be divided into two types, static afterburning and dynamic afterburning. For static Afterburning macros will be filled with hypervisor-dependent High Level APIs. That means these macros only be executed atop the corresponding para-virtualization hypervisors. For example, if the macros are written by using Xen-dependent APIs, the afterburned binary with these macros cannot be executed in other virtualization environment. In contrast dynamic Afterburning is more flexible. Macros will be only filled with enough nop operations in necessary place. During the period that the OS binary is loaded into VM, the nops in macros will be dynamically replaced with corresponding hypercall from the underlying para-virtualization VMM. Obviously this macros rewriting requires more startup time than that with static afterburning. But this rewriting process is one-off. It doesn't affect the OS execution later.

Besides sensitive instructions some sensitive memory operations have also been discussed in pre-virtualization. These operations should be discovered and replaced, as well.

In Place VMM

In Place VMM is another brand new concept from pre-virtualization. It lives in the user space and executes in the same isolated domain with its supported guest OS. That means that every VM on top of Hypervisor has its own In Place VMM. As we introduced above, In Place VMM builds a bridge between pure-virtualization and para-virtualization. It maps Guest Platform API into the Hypercall of para-virtualization for the Guest OS. By doing so, the Guest OS can be not aware when they are running on a para-virtualization environment. Another task of In Place VMM is to rewrite the emulated instruction macros for dynamic afterburned guest OS binary. In fact, a database for instruction macro rewriting is stored in a separate ELF section after the source code of guest OS is compiled and linked. In Place VMM uses this database to translate sensitive instructions from original platform API to target macros.

As a bridge, In Place VMM consists of two ends (see figure2.1), front-end and back-end. Front-end is responsible for emulating the platform architecture. Customized compiler uses Front-end platform emulation interface, the macros, to replace the sensitive instructions. On the other hand, back-end translates platform operations to the hypervisor API. That means, macros of front-end are filled with the functions provided by back-end, and these functions are written through the hypervisor API. The performance of pre-virtualization also relies on the division between front-end and back-end. For example, with this division we can pend interrupts in back-end and batch their handling to front-end so that the mode change overhead can be definitely eliminated.

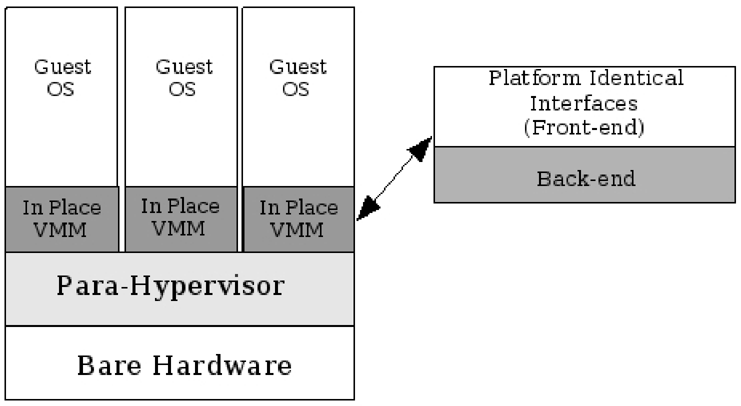

Figure 2.1: Pre-Virtualization Environment and In Place VMM construction

2.2 Migration

2.2.1 Overview

Why do we need migration? We can answer the question by setting up a scenario. For example, we have an emergency and have to go back home. However we are still waiting for the results from an application program that has been running for hours. The program is installed in a desktop and cannot be interrupted. One perfect solution to the problem is to migrate the current work system to your laptop and let the program continue to run after the migration. This example shows us only one potential usage of Migration. Migration can also be used in system management, backup [13], computing grid [7] and so on.

In the beginning, migration was discussed in process level. People tried to migrate processes from one OS environment to another. However, a well-known problem, called residual dependency [6], comes with these solutions, because one process can be related with a lot of other processes, e.g. resource sharing in the source OS environment. It is not easy to construct a similar or identical environment in another OS environment for process migration. Because of this difficulty, migration was not considered as a practical technique in those days.

With the resurgence of hardware virtualization, migration technique also revives recently. This time, migration is discussed on operating system level. People try to migrate an entire operating system along with its applications rather than only to migrate the application processes. Because operating system directly interacts with the hardware interface, the complex relationship reconstruction during process migration doesn't exist any longer during OS migration. Hence OS migration bypasses the residual dependency problem. In this thesis the OS migration will be our discussion point.

OS migration mainly concerns migrating CPU states, VM's memory and VM's local resource. Among them, migrating memory attracts more attention of people because of its indispensability and large capacity. In contrast the states of CPU and other devices, except hard disk , are really small amount of information to be transferred.

Moreover local resource, e.g. hard disk, could be replaced with shared resource. For example, we can use NFS and VFS instead of mounting local file system. We can simply physically unmount the filesystem during the migration and re-mount it after the migration through network, while the logical connection between the NFS or VFS server and the migratee isn't interrupted.

Live Migration

Service downtime is a key criterion in the migration technique. The effect of migration on the service accessing from client should be as little as possible. Pausing a VM during the migration will cause the service inaccessible for long time. Hence live migration appears as a solution to this problem. Live migration is designed for short service downtime. Its rudiment appeared firstly in paper [7] and implemented on L4Linux. Later it was discussed, expanded and re-implemented in Xen para-virtualization environment [2].

Live migration discussed in detail about memory migration since it takes most part of migration time. Generally the memory transferring can be divided into three phases [2].

> **push phase:** The source VM continues running while certain pages are pushed across the network to the new destination. To ensure consistency, pages modified during this process must be re-sent.
>
> **stop-and-copy phase:** The source VM is stopped, pages are copied across to the destination VM, then the new VM is started.
>
> **pull phase:** The new VM executes and, if it accesses a page that has not yet been copied, this page is faulted in ("pulled") across the network from the source VM.

These three phases must not be used together to migrate memory. For example, we can use the pure stop-and-copy for memory migration [2], which halts the VM, copy all pages to the destination and restarts the service. Although this way costs the least migrating time, it brings the unacceptable service downtime. Live migration balances service downtime and migration time by combining a bounded iterative push phase and a very short stop-and-copy phase. During the iterative push phase, Pre-Copy [7] technique is used. Pre-copy copies all memory pages of source operating system to the target machine in the first iterative push-phase. After that only the dirty pages will be transported to the target machine. The iterative push phase won't be finished until the number of dirty page appears to be constant. Then the short stop-and-copy phase will be started to suspend the source machine, collect its running states and transfer them to the target machine. Target machine gets the final state of source machine, rewrites its running environment and restart the service in the end.

Moreover, live migration also discussed about the network device migration and local storage migration. Live migration migrates VM's network device including all protocol state(e.g. TCP PCBs) and its IP Address. An ARP announcement will be trigged after the VM migration. This ARP tells the others that the IP and NIC combination has been changed after migration, the packets should be sent to the new destination. Although some packets will be lost during the switch phase, the number of these will be very small, and they can be retransmitted later. Local storage is a real problem for the migration. It takes too much time and network resource to clone the local storage to another host. Hence the author [2] uses network file system to solve the problem which is uniformly accessible for all host machine. The author also discusses another way of using the RAID protocol to clone the local storage as another local storage migration approach briefly.

2.2.2 Migration Management

In the last section we introduced the idea of migration, discussed migration technique from the perspective of service downtime, and introduced live migration as a technique that balances the migration time and service downtime.

Another point in the migration technique that should be discussed is how to manage the migration progress. We will discuss two main migration management schemes in this section.

VMM-Driven Migration

Virtual machine monitor, as we discussed in section 2.1, normally lives between the VMs and raw hardware interface. It manages the resource allocation and monitors the VM states. Hence using the VMM to manage the migration progress seems very rational. Migration scheme is called VMM-Driven Migration, when people uses VMM as the migration manager. Most mature VMM or hypervisor provides certain VM management interfaces, that can be used to query a VM's running state information, e.g. virtual CPU states, virtual device states, VM's memory contents and so on. So the management unit can be constructed directly in the VMM. But commonly a privileged VM is dedicated for migration management because a commodity operating system running on the privileged VM can provides more conventional development tool and communication methods (e.g. TCP/IP connection).

At the beginning of migration, the migration manager on source machine informs the manager on target machine to start a new container for migration. After the pre-migration work finishes, the source manager VM uses the interfaces or hypercall provided by hypervisor to collect the necessary information for migration and transfer them to the target. The manager on the target machine gets the information, reconstructs the OS running environment in new container. At last the migrated OS is resumed in new container on the target machine. For example, live migration uses the Xen's VMM interfaces to iteratively copy all memory pages of migratee to the target machine. During the short stop-and-copy phase the final state of VM will also be collected and sent to the target machine. After that the migrated OS is restarted to continue running on the new Machine.

Although VMM-Driven migration uses clear and intuitive VM management scheme, adding the migration control interfaces or mechanism to the hypervisor increases the complexity of the hypervisor and reduces the security level. It also reduces isolability between the VMs. How to control the interface access right is a key issue in this scheme. It is possible for a vicious attacker to use the incorrect protected interface to destroy the whole virtualization environment or knock-down the VM performance.

Self Migration

In order to overcome the shortcoming of VMM-Driven migration, another management scheme called self migration [6] was introduced. In contrast to VMM-Driven migration, migration progress will be controlled with a little external help by the migratee itself. As the author described in his paper, operating system can get all necessary migration information by itself, because it is also a resource manager. For example, it is not difficult for a Guest OS to read the current states of registers. It is also no problem for an operating system to scan all memory pages under its control. Furthermore the operating system can also use its own network stack to transfer the migration information. Although self migration is recently designed and implemented in the Xen para-virtualization environment, it theoretically can be started, proceeded and finished without any help from hypervisor. Compared with the VMM-Driven migration, self migration approach has the following advantages [6].

> **Security:** By placing migration functionality within the unprivileged guest OS, the footprint and the chance of programming error of the trusted computing base is reduced.
>
> **Accounting and Performance:** The network and CPU cost of performing the migration is attributed to the guest OS, rather than to the host Environment, hereby simplifying accounting. This has the added benefit of motivating the guest to aid migration, by not scheduling uncooperative (for instance heavily pagefaulting) processes, or by flushing buffer caches prior to migration.
>
> **Flexibility:** By implementing migration inside the guest OS, the choices of network protocols and security features are ultimately left to whoever configures the guest instance, removing the need for a network-wide standard, although a common bootstrapping protocol will have to be agreed upon.
>
> **Portability:** Because migration happens without hypervisor involvement, this approach is less dependent on the semantics of the hypervisor, and can be ported across different hypervisors and microkernels, and perhaps even to the bare hard ware.

Yet nothing is perfect. Self migration has its own drawback. Because self migration is implemented inside Guest OS, it needs to be re-implemented for each type of guest OS. While the implementation for Linux has been proved relatively simple, for other system this may not be general. Otherwise transferring final state of migratee is also impossible. The running OS alters its own states endlessly. To solve the latter problem, pseudo final state has been discussed and used in the self migration. A checkpoint is set before the stop-and-copy phase and all dirty pages, changed states will be saved in a shadow region and transferred in the stop-and-copy phase. After this checkpoint the OS is assumed to be halted. All state changes or dirty pages after the pseudo final state will be ignored.

2.2.3 Migration and In Place VMM

Now if we look back to the section 2.3.2 where the In Place VMM is introduced, we may find out the solution for the two major drawbacks of self-migration. In this section we will firstly discuss the feasibility of using In Place VMM to manage the migration and then take a look at the solutions the In Place VMM brings to us.

Conditions for migration [13] in virtualization environment have been discussed before. Firstly virtual machine monitor should encapsulate all of the machine state necessary to run software and mediates all interactions between software and the real hardware. Secondly this encapsulation allows the monitor to suspend and disconnect the software and virtual device state from the real hardware and write that machine state to a stream. Similarly, the monitor can also bind a machine state to the real hardware and resume its execution. Finally the monitor should require no cooperation from the software running on the monitor.

Accordingly In Place VMM lives between the Guest OS and the underlying Hypervisor, which controls all real hardware resource, and is supported to translate all platform ISA into the hypercall of underlying VMM. Hence every interaction between the Guest OS and real hardware will be mediated firstly by In Place VMM and then by the underlying VMM. Furthermore because Guest OS interacts directly only with the In Place VMM, In Place VMM satisfies the second conditions for migration. Although In Place VMM lives with supported Guest OS in the same address space, it is transparent to the Guest OS and needs no help from the Guest OS atop it. Thus it can be seen that In Place VMM is a very suitable monitor for migration.

Now we take a look at how the In Place VMM solves the two main problems in self migration. For the first issue, because the In Place VMM's duty is only to emulate the Platform API for the Guest OS and doesn't need any cooperation from Guest OS, implementing migration in this layer makes the migration independent on the Guest OS. The only thing that Guest OS should do is to trigger the migration startup by using a special exception provided by In Place VMM. The other problem can also be solved simply, because In Place VMM is not a part of Guest OS, it will not alter the Guest OS final state as self migration does. More interesting is whether using In Place VMM to manage the migration keeps all advantages of self migration. Security of self migration comes from the unprivileged property of Guest OS. This could also be achieved by In Place VMM because it lives also in unprivileged space. "Accounting and performance" advantage is from the perspective of Virtual machine. Certainly the In Place VMM stays in the same virtual machine with its Guest OS, it can also keep this advantage. Although the flexibility advantage of self migration cannot be kept, "Unmodified Device Driver Reuse" [9] project provides us another way to flexibly construct the communication layer in the In Place VMM as well. At last the portability advantage can also be theoretically kept because of the division design concept in In Place VMM. The unit for migration can use the standard interface provided by back-end part of In Place VMM and achieve the in-dependability upon the underlying hypervisor. But we will see later there are still some hypervisor related parts during the implementation, if the In Place VMM doesn't provide a very flawless back-end interface.

2.3 Virtual Machine Environments

2.3.1 L4Ka

The name of L4Ka represents a series of projects from Operating System Group at University Karlsruhe. Microkernel project "Pistachio" is the central project among all of the L4ka projects. Most of the other projects base on the "pistachio" kernel, either show the usage and benefits of microkernel or use the microkernel as its native experiment environment. IDL4 provides a simple and semantic way under the CORBA infrastructure to use the Microkernel. L4Linux shows us how we can port the Linux to the pistachio virtualization environment. pre-virtualization project is the latest project in University Karlsruhe, which uses Pistachio as one of its experiment environments.

Otherwise L4 has also other versions. Fiasco kernel is the one that has been developed by University Dresden. NomadBios is based on this version of L4 kernel.

L4Ka::Pistachio

Pistachio is the 4th generation microkernel [5] from System Architecture Group at University Karlsruhe. It mainly consists of four base abstractions, including threads, address spaces, IPC, and mapping. Fast IPC is one distinguish characteristic of Pistachio. While pistachio is still too small and native to become a virtual machine monitor, it can be used as a very small and fast low level part of the VMM to control the CPU related resource. It also provides primitive APIs to manage the whole machine memory and page abstraction for more effective memory allocation and usage.

L4Ka::IDL4

IDL4 is an IDL compiler, which comes together with Pistachio microkernel. It was designed in order to significantly shorten the development cycle on pistachio. IDL4 supports CORBA and DCE syntax and has an integrated C++ parser which can parse C/C++ header files. IDL4 can generate highly optimized RPC stub code for kernel API and ABI, the hardware architecture, and the C/C++ compiler. The RPC consists of three steps. In the first step all parameters are marshaled into a message buffer. This message buffer is then sent using the system's communication primitives and finally un-marshaled at the destination. Although providing a large memory-based register file usually has a significant impact on the cache footprint of message transfers, using specifically optimized marshaling stubs can completely eliminate the additional costs and result in better overall performance.

Because In Place Migration is firstly tried to be implemented in the virtualization environment comprised of Pistachio microkernel and Marzipan Resource Allocator, IDL4 is a very powerful and useful tool for our implementation.

L4Ka::Marzipan

Marzipan is a very native Resource Separator from L4Ka Team. It uses the pistachio as its underlying hypervisor who manages the CPU resource for the clients. Its duty is to distribute other main local resources (e.g. memory, interrupt) for the virtual machine above it. Marzipan uses grub menu option for booting VMs. Each VM, which will run on the Marzipan, should be declared as a multi-module in the startup menu of grub. Marzipan analyzes the grub menu modules, recognizes the VM module and allocates the resources for the VM. Marzipan uses Sever-client infrastructure to manage the resource, it runs as a server atop pistachio. VMs used idl interfaces provided by Marzipan to access the resources.

Marzipan is very simple. VM management functionality is missing in Marzipan. The Virtual machine can only be allocated by using Grub Modules and be destroyed when the raw machine stops. Actually it is not a good VMM for migration. But on the other hand, because of its simplicity, it is a good VMM for analysis and experimentation. Our approach's implementation is VMM-related. That means for different hypervisor maybe the In Place Migration ought to be re-implemented under the same infrastructure. Implementing on a simple VMM reduces the effort to prove the correctness of our approach.

Unmodified Device Driver Reuse

“Unmodified Device Driver Reuse” (UDDR) discussed how an unmodified device can be unmodified reused in a virtualization environment. The dependability will be improved. This infrastructure uses a dedicated Guest OS to actually host one device driver. This Guest OS is called DD/OS(Device Driver Operating System). It runs as a server to serve the other clients requesting for a specialized device access. The client OS uses a standard driver to access the device. But this access will be replaced by the actual driver of the DD/OS. In other words “Unmodified Device Driver Reuse” project para-virtualizes the device for the client OS.

Pre-virtualization emulates the standard device based on UDDR infrastructure for client VMs. The real Device Driver, which provides service for others, is encapsulated in a dedicated VM. This dedicated VM has the right to directly access the hardware, read and write the register set. On the other hand, it provides high level interfaces for the clients. These clients in pre-virtualization project are not the Guest OS themselves, but the In Place VMM under the Guest OS. In Place VMM first captures the device access platform API from Guest OS(as mentioned before, by using the afterburning). Then it translates these access requests or operations into the Hypercall provided by corresponding DD/OS. After DD/OS finishes the requests, In Place VMM will be informed asynchronously. The Emulated Device states will also be updated.

One thing special in UDDR is that data exchange between server and client will not traverse through the hypervisor. A memory region will be mapped to the DD/OS so that server and client will share the same Memory pages. This increases the efficiency but reduces isolability. Nevertheless DD/OS is only one of choices for the pre-virtualization. Some para-virtualization technique provides already the Device Emulate mechanism, for example, Xen. In Place VMM can directly use the High level interface from these *para-virtualization* environment to emulate the Platform API. In other words, UDDR is a good choice for native hypervisor, but for some mature *para-virtualization* environment, using their original high level interface is a better choice.

NomadBIOS

NomadicBIOS is a virtualization environment based on TU-Dresden’s Fiasco L4-microkernel and hosts L4Linux as its guest OS. Pre-Copy and *live migration* concept was firstly mentioned in this master thesis. Later the technique discussed in NomadBIOS, e.g. Pre-Copy, is used in project “live migration on Xen”, too.

The main purpose of NomadBIOS is to introduce a migratable environment on top of L4 microkernel. The state of Guest OS running in this environment can be checkpointed, transported to and restarted on another instance of NomadBIOS. More Important NomadBIOS achieved live migration with downtime less than one tenth of a second. While its ability is still limited at that time, it provides a very interesting idea for migration and new usage of hardware virtualization. From some respects, NomadBIOS host environment looks like a very native para-virtualization environment. It is due to memory allocation and task allocation for its guest OS. Because of this characteristic, live migration mechanism can be simply transplanted on top of other mature para-virtualization environment as the thing happened on Xen.

2.3.2 Xen

In 2003, the concept of Xen was firstly brought up on SOSP conference by University of Cambridge. People could say that Xen has a successful para-virtualization environment design for the most popular IA-32 architecture. Unlike previous para-virtualization environment, such as Denali, Xen was designed not for a special sense or special mini operating system but for hosting commodity OS operating system, e.g. Linux, Windows XP. In contrast to its precursor it has not only much better at compatibility with popular OS but also with significant performance near that of bare hardware architecture. Four principles run through the whole design of Xen [11]:

1. Support for unmodified application binaries is essential, or users will not transition to Xen. Hence Xen must virtualize all architectural features required by existing standard ABIs.
2. Supporting full multi-application operating systems is important, as this allows complex server configurations to be virtualized within a single guest OS instance.
3. Para virtualization is necessary to obtain high performance and strong resource isolation on uncooperative machine architectures such as x86.
4. Even on cooperative machine architecture, completely hiding the effects of resource virtualization from guest OSes risks both correctness and performance.

Since Xen is a very popular para-virtualization solution in the world, it is used as one of major experiment underlying VMMs in pre-virtualization project. Furthermore live migration was once employed in Xen environment, either with VMM-driven migration's management scheme or with self migration's management scheme. It is also useful for us to evaluate our In Place Migration approach by comparing it with the existing example.

Chapter 3

Proposed Solution

3.1 Design Overview

Firstly, we must consider which core migration scheme we should choose in our virtualization environment. Because live migration is a very good migration mechanism with short service downtime and high migrating efficient, we choose live migration as our core migration scheme. Furthermore since pre-virtualization technique is base on variants of para-virtualization Hypervisor and there are some successful implementations of live migration in para-virtualization environment, e.g. live migration in Xen, they could be good guide for our migration design and implementation.

Secondly, we need to consider the migration management scheme. Although VMM-migration and self-migration scheme can both be implemented in pre-virtualization environment, we still try to design a new management scheme that uses the brand new concept from pre-virtualizaton. In section 2.2.3 we discussed the possibility of using In Place VMM to manage the Migration progress. The discussion told us that using In Place VMM makes it possible to overcome the shortcomings of self migration as well as to keep almost all of its advantages. We call this approach of live migration in pre-virtualization environment "In Place Migration", which uses In Place VMM to manage the whole progress of migration.

It is clear that migrating an OS, no matter which core migration scheme is used, e.g. live migration, pure stop-and-copy migration, how the migration information should be collected and transferred is always the center problem that should be solved. For In Place VMM collecting migration information is not very difficult. It emulates all necessary platform interfaces for the Guest OS and records current states of virtual CPU and virtual devices. The memory or cache modification can also be caught by In Place VMM. In order to collect the information In Place VMM need only read the corresponding virtual device register set or the memory pages, marshals them in the migration data structure, and send to the destination. On the destination side, information is firstly un-marshaled, and then corresponding virtual device registers or memory pages are set or rewritten with this information by In Place VMM. After all necessary information is transferred, In Place VMM on the destination is due to restart the instruction execution of this migrated VM.

The whole migration progress can be described as follows:

Stage 1: starting migration Like self migration guest OS is responsible to start the migration. pre-virtualization technique already uses a special exception number to account some debug information. We also use this mechanism in our approach. A special exception number is added to the exception emulation part in the In Place VMM. Guest OS passes some migration parameter, e.g. target machine hostname or IP address, and then triggers the special exception. In Place VMM gets the exception, reads the corresponding parameter, and prepares for the migration.

Stage 2: finding a new container After "starting" phase, In Place VMM knows at least who the target machine is. After it validates the machine information, it sends a request to the target and asks the target to find or allocate a new container for migration. New container allocation could be treated centralized or autonomically. We can dedicate a VM to deal with the VM allocation request. Or we can also broadcast request in the virtual network domain on the target machine in order to ask all In Place VMM layer, which VM is now in "zombie" state (see section 3.2), if we use the Zombie concept in our virtualization environment. If new container cannot be allocated because of lack of resource or no suitable Zombie machine can be found, the whole migration progress will stop and source VM will be informed to continue running his service. User will get a failed message from In Place VMM. Otherwise next stage comes.

Stage 3: transferring migration data After new container has been found, In Place VMM on source machine starts the iterative Pre-Copy phase. Guest OS' memory pages will be transferred to target. On target side, new container's In Place VMM rewrites its memory by using the memory information from source machine.

Stage 4: stop-and-copy When the number of dirty pages becomes constant, Pre-Copy phase stops. Now In Place VMM will collect the final state of Migratee. It suspends the Guest OS, reads the Virtual CPU states, virtual device registers, and the final dirty pages, packs them, and sends them to the target In Place VMM. Target In Place VMM gets the data, rewrites its own virtual environment.

Stage 5: activating At last In Place VMM on the destination will start its virtual machine and resume the service. If every thing works fine, target In Place VMM sends a successful packet back to the source In Place VMM. Source In Place VMM can then make itself into Zombie state or just inform underlying VMM to deallocate its resource. If restarting VM fails on the destination, target In Place VMM will tell the source In Place VMM this bad news. Source In Place VMM restarts its virtual machine again to resume the interrupted service. A message will be showed to tell the user that migration failed.

3.2 Virtual Machine State

In the classical operation system process and thread have their own state to represent what they are doing so that the resources could be more effectively used and allocated by the operating system. In the real world, there are also states for a bare hardware, for example, running, shutdown and suspended. So the concept of state could be also used in virtualization world. For example, we can give every virtual machine on the hypervisor its own state. These states will not only concern with running, shutdown and suspended but also some state concepts borrowed from process and thread, for example, Zombie. With these states the virtual machine may be more clearly and easily managed. The resource could be more flexibly used, as well.

Zombie State

We introduce the zombie state of virtual machine in this paragraph. It will be used to represent a migrated and suspended but not deallocated virtual machine in the virtualization environment. Normally this virtual machine should be deallocated from the Virtual Machine list by the hypervisor so that its resource can be reallocated for other demand. But keeping this inactivated VM container in the virtual machine list may be helpful in some senses. For example, we need migrate a virtual machine into a virtualization environment where another virtual machine has just been moved out. Using zombie state for the outgoing VM container is a good selection in this situation because we can just pick up the existing container for the incoming VM and get rid of some initialization cost for starting a new container. This is helpful to the In Place migration. The whole migration progress by using a zombie VM could be even proceeded sometimes without any help from hypervisor, i.e. starting a new container, and the 2. stage of migration can be directly handed up to the In Place VMM level. Another motivation for using Zombie state is that some hypervisor doesn't provide the functionality to dynamically start a virtual machine and deallocate them, for example, Marzipan. It may not be very complex to add the management functionalities in the hypervisor. But with the Zombie concept the changes in the hypervisor could be as small as possible.

3.3 Communication between In Place VMMs

We have discussed briefly about the whole migration progress in section 3.1. One thing to be considered now is how to transfer the migration data during the migration. Although we have chosen live migration for migrating system with minimal service downtime, and In Place VMM as the manager, how live migration approach can be implemented by using In Place VMM is still a question. In this section we will answer the question and take a look at how the In Place VMM really communicated.

The most part of communication happens in the second stage. In this stage the In Place VMM of migratee has been informed to start the migration progress and gotten the target machine's information, e.g. host name or IP address. Now the communication channel must be created between the source and destination. But we encounter a problem. In Place VMM hasn't any OS level firmware, e.g. network device driver because it emulates only the platform interface for the Guest OS. In another word it seems that we cannot get the high level communication layer, e.g. TCP layer, as we can get from an operating system. Two ways have been thought about to solve this problem.

In Place Device Driver The first solution is based on the front-end of In Place VMM. In this solution we simply build a Device Driver for the In Place VMM just like operating system does on the bare machine. By doing so we can use an existing conventional device driver rather than trying to add an additional network device handler in the In Place VMM layer. This can definitely reduce the engineer effort.

Because of the transparent property of In Place VMM, This driver is only visible for its In Place VMM. So we call it In Place Device Driver. Grub is a very useful example for this solution, because it also builds a device driver without any help from operating system. Furthermore in the virtualization environment, we can standardize the network device that be emulated. For example, pre-virtualization use dp83820 as its standard network device. This advantage eliminates the additional cost for driver selection. The only driver we should build is the driver for dp83820. Additionally in order to save the critical resource of In Place VMM we decide to choose UDP stack with some confirmation mechanism rather than to implement a TCP stack. Nevertheless planting a device driver in the In Place VMM is still very expensive.

Unmodified Device Driver Reuse Another way is using idea from the Project "Unmodified Device Driver reuse". With help from this project there will be some high-level interfaces that In Place VMM can use. To transfer packets In Place VMM will firstly map the packets to a producer/consumer ring. After the packets have been queued, In Place VMM informs the DD/OS by using the interface from DD/OS. After DD/OS being informed, it gets the data directly from the address space of In Place VMM and transfers them to the destination. In the situation of receiving the packets, DD/OS filters the packets for different In Place VMM, sends them to the corresponding producer/consumer ring and informs the In Place VMM to retrieve them. (See figure 3.1) By using "Unmodified Device Driver Reuse" infrastructure it is simpler for us to construct the UDP communication channel between two In Place VMMs without additional device driver. Problem of this solution also exists, since "Unmodified Device Driver Reuse" technique requires the modification of hypervisor. For example, we need re-implementation the "Unmodified Device Driver Reuse" in the Xen virtualization environment. But if "Unmodified Device Driver Reuse" is a standard part of the pre-virtualization technique, it won't be a big matter for our migration approach.

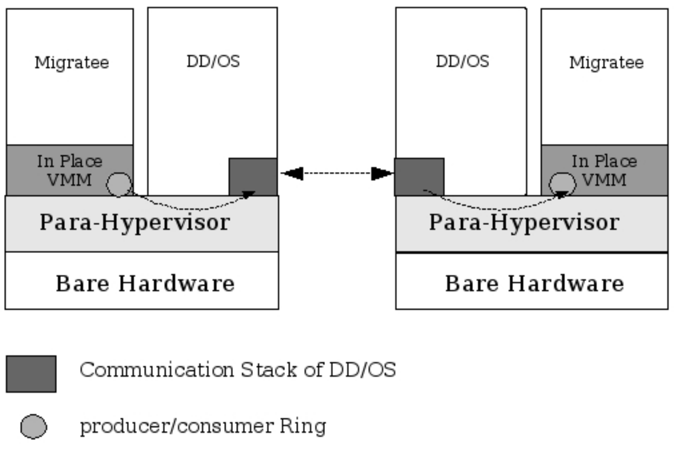

Figure 3.1: Communication with "Unmodified Device Driver Reuse" technique

After we got the way to create the communication channel, the source In Place VMM will try to ask the target machine for an empty container. According to the different designs this will be implemented in different ways. If the target machine has a Dedicated VM for building and finding an empty container, for example Domain0 in Xen, the request will be directly forwards to the Dedicated VM who checks his VM list and looks up firstly a Zombie VM. If a Zombie VM has been found, the Dedicated VM sends the necessary information of the Zombie machine back to the requester, at least the IP address of empty container's In Place VMM. Otherwise dedicated VM creates a new container as empty container. After the source In Place VMM received the empty container's information, a communication channel will be tried to create between the source In Place VMM and the empty container's In Place VMM. After successfully creating this communication channel the subsequent migration stages will be continued progressed. Another way for asking the empty container is directly asking for a zombie machine if there is no centralized VM Manager, e.g. in marzipan. The source machine just sends a request to the target machine and this request will be broadcasted in the target machine's virtual network. If any In Place VMM of the Zombie machine receives the request and considers that it is fit to be an empty container, it will send a reply to the source In Place VMM. After that the communication channel will be created and used as above. One weakness of the second implementation is that if there is no Zombie machine on the target machine, the migration cannot work.

3.4 Memory Migration

Without correct Memory image, a suspended Guest OS cannot be restarted correctly. Thus Memory Migration is the main point in many migration discussions.

In contrast to the small sum of virtual machine state (register, flags), the amount of memory is really big. In recent 256M memory is the lowest recommendable standard for most commercial operating system. 512M or more is also common in real life. On the other hand the memory is also one of the frequently changed parts in computer system. One round transportation is not sufficient for migrating an operating system that is still running during the migration, since the operating system alters its own memory pages rapidly. Thus iterated memory copy is required. Pre-Copy technique from live migration fills these requirements, so we use it in our In Place Migration approach.

Migrating such a large amount of memory from one machine to another will occupy a lot of bandwidth of network and reduce other services response time in the same network environment. Pre-Copy technique also mentioned this problem and tried to solve it with idea "Writable Working Sets". The first Pre-Copy round will copy all memory of source Guest OS to the destination as quickly as possible. In subsequent iteration copy only the dirty pages need be copied. This set of dirty pages is called "Writable Working Set".

For getting the Writable Working Set, We need to be aware of the activity of memory. Page table is one of the best choices for detecting the dirty pages. If the page table provides the dirty page bit, the dirty pages can be simply detected by checking the dirty bit. For setting the dirty bit of a page, every page entry will be set to read only in the beginning of migration. Hence a Page Write operation will cause an exception in the hypervisor. Hypervisor can catch this exception and check if the corresponding page is really writable. If it is writable, the dirty bit of this page will be set and the write operation will be proceeded as normal. When the Pre-Copy round comes, Migration Monitor checks all page tables and transfers the pages whose dirty bit is set. This way of determining dirty pages can be implemented in the pre-virtualization environment, because In Place VMM emulates also the page table operation instruction and maintains the Guest OS's page tables with its own page table structure. We can catch the page table operation, sets the dirty bit in the page table structure of the In Place VMM, and use them to finish our writable working set analysis.

One possible drawback of this method is that Guest OS could not use the Page Table during the copy phase for the persistence of page state. Furthermore the loading and searching the Page Table bring more overheads. So it is meaningful to borrow the idea of bitmap from the paper [2]. Every bit in the bitmap represents one page in the memory. The bit will be set when corresponding page is modified. The copy round of Pre-Copy find the dirty pages by looking up in the bitmap. 6K memory should be allocated for a 256M memory region in order to maintain the page modification information. This is not a big problem when the migration is implemented by using VMM-Driven strategy. But in the In Place VMM resource is very critical. 6K may be a little big for In Place VMM. Besides searching the bitmap and interpreting the bit to page address are also significant overheads for VM.

Since in every round of Pre-Copy, except the first, only the dirty pages should be transported to the destination, we can also transport the pages when they are modified. This implementation doesn't require any external data structure except one dirty page counter. The dirty page counter tells the migration manager if the number of dirty page number is constant. If the number doesn't change rapidly any longer, the Pre-Copy will stopped and final stop-and-copy phase and activating phase finish the whole migration. But this approach shows us another problem. The same page may be transported many times between two Pre-Copy rounds. This affects directly the network performance. Otherwise, the additional transportation of the same page is unnecessary. It is enough for migration to transport only the final changed content of the dirty pages in every Pre-Copy round.

After all we choose the idea of bitmap as our dirty page recording policy, because it has fewer problems than other methods. The problem of critical resource was also tried to be solved by borrowing the pages from guest OS. After the first round of Pre-Copy, we try to find some pages of guest OS, those will potentially be no more changed, map these pages into the address space of In Place VMM, and allocate them for the bitmap who records the dirty pages. Although this seems to be simple to solve the critical resource problem, we still realized that we should be careful when we use the pages from guest OS. If the guest OS suddenly needs these pages again, we should not give the allocated pages back, but allocate some new pages to the guest OS. For these purpose, the pages, which we map to the address space of In Place VMM, will be labeled as protected. Whenever the guest OS try to use these pages again, a page fault exception happens and new page is allocated to the guest OS. One more thing should be done is to request the target machine to transfer the original data of these pages back and the source In Place VMM rewrites them in the corresponding pages.

3.5 Local Resource Migration

Memory can be simply copied to the new host, but some local resource cannot be so easily migrated. For example, local storage is too big to be copied. Live migration argues two ways to solve the problem of migrating disk storage. The first one is to use the network-attached storage(NAS) device instead of local device. By using this method migrating local storage is eliminated. Furthermore NAS can be simply administrated centrally, has widespread vendor support, and reduced failure rate. With this solution the NAS device will be unattached from migratee in the first stage of migration and re-attached to the destination in the final stop-and-copy stage. Although it is also possible to use RAID-5 and iSCSI functionality to mirror the disk for migration, this method isn't mature, yet. During the design progress we also mentioned other possible solutions. One of them is using RAM disk. This solution is only suitable for system that provides tiny service. For some small service 100M space may be sufficient. So using a ram disk for the system is possible. Otherwise ram disk can be migrated as a part of memory. Because disk operation is not so often as memory operation, the most part of ram disk needs to be transferred only once in the first Pre-Copy round. Furthermore we can also combine the NAS and ram disk approach. We can use the ram disk for quickly restarting the Guest OS and recording the system level changes, since the OS kernel is relative small and is the most important part of filesystem during migration. The other files or data changes of service can be recorded in the NAS filesystem. Another way is to pre-migrate the disk storage. That means that we can copy the whole disk pages from source to the target before migrating the OS instance. This pre-migrating phase may take more than one hour because the disk capability normally is very huge. But during this phase the source server doesn't need to degrade performance. After that the disk migration will be similar with the memory migration and use the Pre-Copy mechanism. This way is useful for the migration, which allows relative long time to preparing for the migration, e.g. server maintenance.

Another important local resource, which should be migrated, is network device. After the migration all protocol state (e.g. TCP PCBs) and VM's IP address should be kept. Main problem is the MAC-IP combination. Packets will use this combination to find the final destination. But after the migration, this combination will be broken. In pre-virtualization every VM gets its private MAC Address base on the official MAC address of the host machine, half of the MAC address will be used for private use, so that nice property of decentralized management of virtual MAC addresses can be achieved. Since the migration target machine and migration source machine have different official MAC address, the new VM's MAC address on the target machine is also different from the MAC address of migratee. Hence after migration new MAC-IP must be announced so that packets for the migratee can find its new destination. In the LAN, an ARP packet will be broadcasted to the other machines. This can inform the others the new MAC-IP combination [2].

Chapter 4

Implementation

We didn't implement complete In Place Migration in this work because current version of pre-virtualization is only an experimental version, and some functionality in this environment is still under the development, e.g. pre-virtualization supporting for Xen. Correspondingly L4 pistachio pre-virtualization environment is more mature than the other, so we chose it as our first experiment environment. All changes only concern In Place VMM layer. We didn't change any thing currently in the Marzipan Resource Manager and Pistachio hypervisor. Moreover we used Linux as the guest OS running on top of the In Place VMM.

4.1 Building The Migration Monitor

The pistachio In Place VMM consists of three L4-threads, an exception monitor thread, an irq monitor thread, and the main VM thread. Exception monitor thread is used to deal with the exceptions that happen in the In Place VMM. Irq monitor is used to treat the irq event from the Guest OS. And Guest OS binary will be planted in the VM thread. Similarly, we can construct the migration monitor in the In Place VMM (See figure 4.1). We used the interfaces provided by pistachio In Place VMM to allocate and start the migration thread.

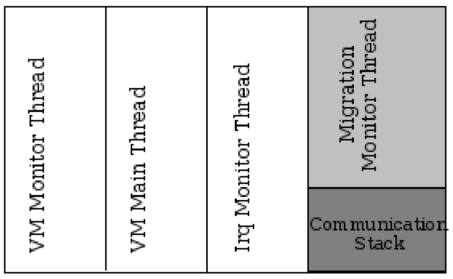

Figure 4.1: In Place VMM with Migration Monitor Unit

The main responsibilities of migration monitor are to start the migration, copy the memory pages, transfer machine final state, suspend migratee on the source side, and receive the memory pages, reconstruct VM running environment, restart service on the destination side. These works will be triggered by L4 IPC from different sources. We simply used an infinitive loop to catch and deal with all IPC events.

4.2 Starting The Migration

Pistachio In Place VMM catches every interrupt instruction and passes them from the back-end part to the front-end part. Then "`backend_handle_user_exception`" deals with the exception and jumps into the corresponding exception handler according to the exception number. Besides the ordinary exceptions, e.g. divided by zero, we can also insert our own special exception number into it in order to treat some special situations. For example, pre-virtualization uses 0x69 and 0x70 to dump the counter number and profile some instructions. So we decided to use 0x68 as special exception number for our approach. If the Guest OS uses "`int0x68`" instruction, this exception will be caught by VM main thread. After that VM main thread will send a L4 IPC to the migration monitor for starting the migration. But to start the migration we need not only trigger the migration monitor but also pass some initial migration parameter to the monitor, e.g. hostname of destination. So it is better to create a module in Linux to manage these things. The module provides an interface for user, e.g. `start_migration()`. It can take one parameter, which is a point referencing to a 'struct' of migration initial information. User can fill the 'struct' with Destination IP, Destination hostname and some flags for migration. Migration Module gets and validates this information, and passes them to the place where migration monitor can read them. By using the module concept we can control the migration better and avoid some attacks. For example, we can also require the user authentication information in the initial struct for the access control.

4.3 Suspending and Restarting the Machine

Before we finish the final migration stage, we need to do two things. Firstly we suspend the VM and secondly we must get all machine states. The latter task is relatively simple, because the emulation part of In Place VMM has already provided data structure for the virtual CPU and virtual devices. What we need to do is to save the necessary information from these structures after the VM has been suspended.

Suspending the VM is more complex. Pistachio In Place VMM maps every thread of Guest OS into a L4-thread for better performance. In other words Pistachio In Place VMM catches all threads or tasks creation event in Guest OS and actually creates them by using the pistachio threads maintenance interface. So we must firstly suspend the running threads one by one before we suspend the main VM thread.

For suspending the threads we need to know how the In Place VMM manages the threads of Guest OS first. Pistachio In Place VMM provides some data structure to manage the thread and task information of Guest OS.

task_manager_t It is used to manage all tasks' information. When address space is going to be changed from one to another, Pistachio In Place VMM uses this structure to find the corresponding task structure and to get the page directory entry address.

task_info_t This structure records the page directory's entry address of every task in Guest OS. We can say that it holds the address space information for the task. Furthermore it manages the utcb space for all threads belonging to the same task. At the same time it holds the main thread id of the task.

thread_manager_t This manages all threads created in the Guest OS and implements the algorithms to find the thread information data structure according to the L4 thread id.

thread_info_t This structure temporarily records the thread environment information. This information is used to feed back some Guest OS requirement. For example pistachio In Place VMM uses it to fill up the initial environment of a new created thread in the Guest OS. Exception also uses these temporary information to jump back it own running routine. Because of these characteristics, we can also get the running thread information by triggering a forced exception and let the In Place VMM get the current environment information of the running threads. Pistachio In Place VMM uses these states to label the threads so that thread event can be treated correctly. The structure defines some different states of thread for different purposes. We add `state_migration` for our migration approach.

We found all existing threads by searching the threads array in `thread_manager_t`, set special IP and SP to every thread by using the L4 interface and changed the thread's state in `thread_info_t` structure to `state_migration`. When the thread runs with the special IP and SP, an exception will happen. This exception will end up with "iret" instruction, and give out the correct running environment information in the Guest OS environment. After getting thread's corresponding `thread_info_t` structure, handler knows that this thread is in the state of migration, then saves its register states, and transfers them to the destination.

To restart the suspended machine on the destination, we did the reverse operation to the suspending machine. Firstly the destination In Place VMM rewrites the structure of virtual devices and virtual CPU by using the received migration packages. Then for reconstructing the running environment on the destination, the threads should be reconstructed on the destination machine. At the beginning there is only one main thread, from the perspective of In Place VMM, running in the virtualization environment. But from the perspective of Guest OS there are several services running in its environment. Every time when a not really existing thread is going to run, an exception will happen. In Place VMM catches this exception, find migration information for this thread, creates a corresponding new L4-thread and starts this thread with the register data from migration information.

4.4 Memory Copy

Reading Pages

To read the memory pages of Migratee we used the mapping concept of pistachio. We mapped the memory pages, which should be copied to the destination, "read only" to the In Place VMM. After that they would be sent by Migration Monitor through the communication channel to the destination In Place VMM.

Restricted Copy

In the first Pre-Copy round all of the migratee's memory should be copied to the destination. Because In Place VMM lives with the Guest OS in the same virtual machine memory, the copy of memory should exclude the part of memory that In Place VMM occupied. We can get the memory range of In Place VMM from some useful interfaces provided by In Place VMM, e.g. vcpu's get_wedget_paddr(), and skip this block.

Dirty Page Detection

After the first round, only dirty pages need to be transferred to the destination. In order to get the dirty pages we tried to monitor the page operation during the migration. Pistachio In Place VMM uses the hardware page table for monitoring the page table operation. Every memory write operation can cause the setting of dirty bit in corresponding hardware page entry. Moreover every time when the address space changes, hardware page table will also be flushed. We use these two characteristics of hardware page table and an additional bitmap for our dirty page detection purpose. When the page table is flushed, the dirty bit of every entry's dirty bit needs to be checked by using `L4_Unmap()` so that the dirty pages could be recorded in the bitmap. In every iterative round of Pre-Copy, the bitmap is used to find all dirtied pages and then reset to the initial state(every bit in the bitmap will be reset to 0). In the last stop-and-copy stage, both loaded page table and bitmap should be checked for dirty pages, because at that time the dirty pages in the hardware page table are still not recorded in the bitmap.

Chapter 5

Conclusion

This thesis introduced the existing primary techniques of hardware virtualization and OS migration, compared them and discussed their advantages and shortcomings. After exploration of these areas we elaborated a migration approach for the pre-virtualization environment. This approach is based on the idea from self migration and uses In Place VMM Unit for implementation.

Main purpose of this approach is to overcome the main shortcoming of self migration as well as keep its advantages. The advantages of self migration, including "security", "accounting and performance", "flexibility" and "portability", are almost be kept in our approach. Furthermore, our approach doesn't need to add many modifications to the guest OS as self migration should do. We simply provide a special exception for guest OS to starting migration rather than writing a lot of OS-dependent codes in the operation system environment. This avoids the high OS-dependency of self migration. Moreover our approach can really save the final VM states, while self migration cannot achieve this.

Due to the constraints in the access of resources, we were only able to implement part of the research design. However, this study managed to demonstrate the feasibility and rationality of the idea.

Future work will focus on standardizing the interfaces of the In Place VMM in order to achieve the flexibility of self-migration. Otherwise the "Unmodified Device Driver Reuse" project will be reintegrated into pre-virtualization environment so that In Place VMM communication can be experimented and achieved.

Bibliography

[1] Marianne Shaw Andrew Whitaker and Steven D. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Boston, MA, December 2002.

[2] Steven Hand Christopher Clark, Keir Fraser and etc. Live migration of virtual machines. In *Proceedings of USENIX NSDI 2005*, Boston USA, 2005.

[3] Scott Devine Edouard Bugnion and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th Symposium on Operating System Principles*, Saint-Malo, France, October 1997.

[4] R. P. Goldberg. Survey of virtual machine research. *Computer*, 1974.

[5] System Architecture Group. The l4ka::pistachio microkernel white paper. Technical report, Fakultät für Informatik, Universität Karlsruhe (TH), May 2003.

[6] J. G. Hansen and E. Jul. Self-migration of operating systems. In *Proceedings of the 11th ACM SIGOPS European Workshop*, September 2004.

[7] Jacob G. Hansen and Asger K. Henriksen. Nomadic operating systems. Master's thesis, Dept. of Computer Science, University of Copenhagen, Denmark, 2002.

[8] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), November 2005.

[9] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[10] T. Lindholm and F.Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[11] Keir Fraser Paul Barham, Boris Dragovic and etc. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles*, October 19–October 22 2003.

[12] Gerald J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17, 1974.

[13] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Boston, MA, December 2002.

[14] James E. Smith and Ravi Nair. The architecture of vritual machines. *Computer*, 2005.