Multi-Granularity Memory Mirroring via Binary Translation in Cloud Environments

Zhengwei Qi, *Member, IEEE,* Haoliang Dong, Wei Sun, Yaozu Dong, and Haibing Guan, *Member, IEEE*

***Abstract*—As the size of DRAM memory grows in clusters, memory errors are common. Current memory availability strategies mostly focus on memory backup and error recovery. Hardware solutions like mirror memory needs costly peripheral equipments while existing software approaches reduce the expense but are limited by the high overhead in practical usage. Moreover, in cloud environments, containers such as LXC now can be used as process and application-level virtualization to run multiple isolated systems on a single host. In this paper, we present a novel system called Memvisor to provide high availability memory mirroring. It is a software approach achieving flexible multi-granularity memory mirroring based on virtualization and binary translation. We can flexibly set memory areas to be mirrored or not from process level to the whole user mode applications. Then, all memory write instructions are duplicated. Data written to memory are synchronized to backup space in the instruction level. If memory failures happen, Memvisor will recover the data from the backup space. Compared with traditional software approaches, the instruction level synchronization lowers the probability of data loss and reduces the backup overhead. The results show that Memvisor outperforms the state-of-the-art software approaches even in the worst case.**

***Index Terms*—Application level memory mirroring; multi-granularity high availability; virtualization.**

I. Introduction

GOOGLE'S latest research shows memory (DRAM) errors are common in modern compute clusters, and more than 8% of DIMMs affected by errors per year [1]. This rate is unacceptable for commercial cloud service providers, e.g., Google App Engine [2] and Amazon EC2 [3], which require availability with an annual uptime percentage at least 99.95%. Compared with system crash, what is worse is that memory errors can lead to data corruption, which is a disaster in financial, medical and scientific fields [4].

There are soft errors (e.g., a bit flipped by cosmic rays) and hard errors (memory damaged physically) [1] in modern complex computing environments. To achieve memory High Availability (HA), hardware HA approaches are proposed at first. Parity [5] and ECC [6] are the earliest way to enhance memory availability. The parity memory uses parity check code [7] to detect a single bit error, while ECC memory uses Hamming Code [8] to detect and restore a single bit error. The disadvantage of them is that they cannot handle hard errors. Mirror memory solution [9] is proposed to solve the problem, which uses doubled memory chip to back up the data on the fly. However, both ECC memory and mirror memory are expensive and need the support of special hardware.

Software HA approaches are developed quickly in recent years. Software ECC [10] works like the hardware ECC, which provides a library for developers to enhance specific blocks of memory. However it incurs a high overhead. Google and Amazon have designed some special systems for HA, such as Google File System [11] and Amazon Dynamo [12]. They store same data in two or more physical machines to prevent data loss. But both of them are designed and run in the application level, which cannot provide transparent HA for operating systems or other applications. Some Virtual Machine (VM) based solutions address this shortage. Now Xen hypervisor supports Remus [13] and tools (included with Xen 4.0+). So VM-level High Availability is a critical feature for network and service management in cloud environments. Today, LXC (http://lxc.sourceforge.net/) is the user space control package for Linux Containers, a lightweight virtual system mechanism. LXC builds up from *chroot* to implement complete virtual systems, adding resource management and isolation mechanisms to Linux's existing process management infrastructure. So application-level and VM-level High Availability play an important role in these containers. However, Remus is a system using virtualization technology to back up the whole VM by checkpointing, while it does not back up the system on the fly. So the data between two checkpoints will loss if a memory error happens, and the overhead of Remus is 103% with 40 checkpoints/second.

To efficiently improve the availability at a low cost, we have previously presented Memvisor in CLUSTER 2012 [14], an application level cost-effective software memory mirroring. In this paper, we extend it to support flexible user-mode high availability via process or application level multi-granularity memory mirroring. Memvisor uses binary translation to modify applications to replicate the data to mirror memory so when a memory failure happens, the data could be restored from the replica.

Memvisor is a hypervisor providing process-level and application-level high availability based on hardware virtualization. Our design is based on two core technologies, i.e., Direct Page Table (DPT) based mirroring space and Static Binary Translation. Compared with Shadow Page Table (SPT), DPT can reduce the overhead without keeping the

Manuscript received June 23, 2013; revised October 27, 2013. The associate editor coordinating the review of this paper and approving it for publication was G. Martinez Perez.

Z. Qi, H. Dong, and W. Sun are with the School of Software and the Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai, China (e-mail: {qizhwei, donghl, zmsw2008129}@sjtu.edu.cn).

Y. Dong is with the Open Source Technology Center, Intel, Shanghai, China (e-mail: eddie.dong@intel.com).

H. Guan is with the Shanghai Key Laboratory of Scalable Computing and Systems, and Department of Computer Science, Shanghai Jiao Tong University, Shanghai, China (e-mail: hbguan@sjtu.edu.cn).

Digital Object Identifier 10.1109/TNSM.2014.031714.130415

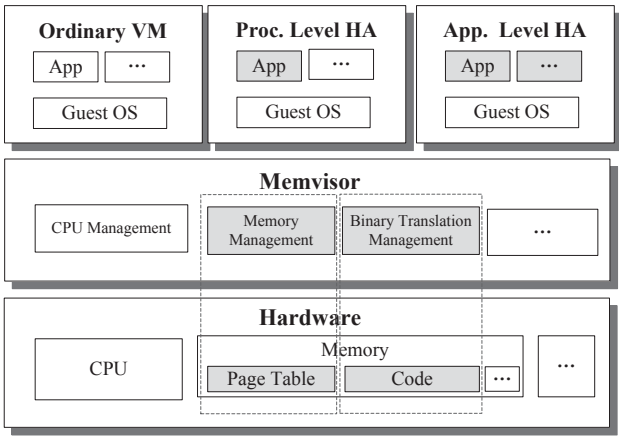


Fig. 1. The high-level architecture of Memvisor. The gray blocks indicate the difference between Memvisor and a typical hypervisor. The VMs can be classified as three categories: ordinary VM, process level HA VM, and application level HA VM.

synchronization between original virtual memory and shadow memory per process switch. Compared with static binary translation, dynamic binary translation often needs more system resources including cache or runtime support components, which also incur more overhead. Also, we use hypervisor based solution to provide multi granularity memory mirroring, because hypervisor such as Xen, VMWare, and KVM are popular in cloud environments which are easy to be integrated. Moreover, hypervisor controls the low-level resources, which will not leak some sensitive information, thus providing a strong isolation in cloud applications.

Generally, Memvisor instruments mirror instructions not only for user mode code but also can be extended to kernel mode code. Only low-cost memory is required to recover the data when the memory corrupts. Also Memvisor is a more flexible and low-priced approach than the hardware solutions. The virtualization technology assists it with the ability of supporting VMs with or without mirror memory feature on a same physical machine. And the binary translation technology offers the choice of process level or application level high availability. Also the mirror memory could easily be set to more than two copies for some special mission-critical systems.

Memvisor is also more efficient than other software approaches. The results show that the performance of CPU intensive tasks is unaffected, and even in the worst situation. Our stressful memory write benchmark shows the backup overhead of 80%, less than 100%+ of Remus and other software approaches.

The rest of the paper is organized as follows. The overall architecture and the process of Memvisor are introduced in Section II. Section III gives detailed implementation and Section IV evaluates the performance and overheads. Section V has a technical discussion and Section VI covers the related work. At last Section VII concludes the paper.

II. Design

Memvisor is a modified hypervisor which provides multi-granularity application level high-availability memory features. Memvisor inserts instructions (mirrored instructions) through binary translation technology and replicates data to this space. The replica can be used to recover the data when a memory error happens.

A. Architecture

Figure 1 illustrates the high-level architecture of Memvisor. Virtual machines can be configured as native or HA VMs, i.e., process level or application level high availability VM. The different types of VMs can be run on the same physical machine. Memvisor can be divided into two components. First is the memory management module, and second is the binary translation module. However, compared with hardware memory mirroring solution, the shortcoming of using modified hypervisor is the overhead caused by mirror instructions. Another problem is that this modification is not transparent to the hypervisor, so it is not compatible with non-Memvisor based environment.

Memory management module monitors the page table related operations to create mirror page table. Memvisor maintains two sets of page tables. One is native page table which deals with native address mapping. The other is mirror page table that handles the mirror address mapping. We choose the hypervisor instead of the operating system (OS) because the memory can be replicated at the VM granularity. And it is flexible to switch VMs between high available memory and native memory. If the data of an OS are corrupted, the OS will crash and become unable to recover the data by itself. But using the hypervisor overcomes this shortage, because hypervisor has less code size than OS with a small trusted computing base, which are robust and secure in most cases.

Binary translation module inserts mirror instructions, which identifies all memory writing instructions and replicates them. The difference between original instruction and replicated instruction is the destination address. Memvisor defines a concept "mirrored virtual address (*mva*)" in addition to the "native virtual address (*nva*)". An *mva* is a virtual address mapped to an additional physical area. This physical area is created by Memvisor to store the redundant data, and we call an address in this area a mirrored physical address (*mpa*). We define an abstract function *mirror* to describe the relationship of these two kinds of virtual addresses as:

$$mva = \mathbf{mirror}(nva) \tag{1}$$

The variable *mva* stands for mirrored virtual address and variable *nva* stands for native virtual address. With the function **mirror**, we can distinctly describe the workflow of data replication in Memvisor.

B. The workflow of redundancy in Memvisor

Memvisor uses the redundant data for recovery, and the redundant data are created by mirrored instructions. The process of this workflow includes several steps:

1) **Startup a virtual machine:** Initially, Memvisor creates a candidate VM or a process and allocates the physical memory for it. In our system, it also allocates another equal size memory as the mirrored physical memory for the target VM.

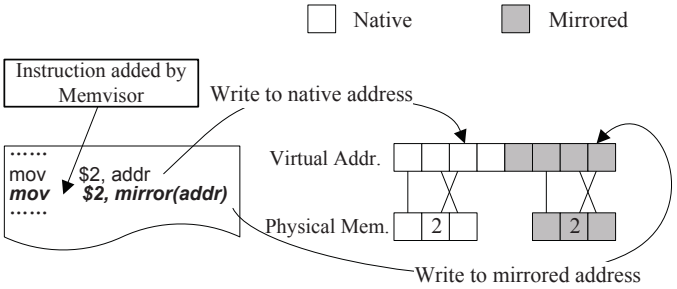


Fig. 2. A mirrored instruction will be created to write the same data to the mirrored address.

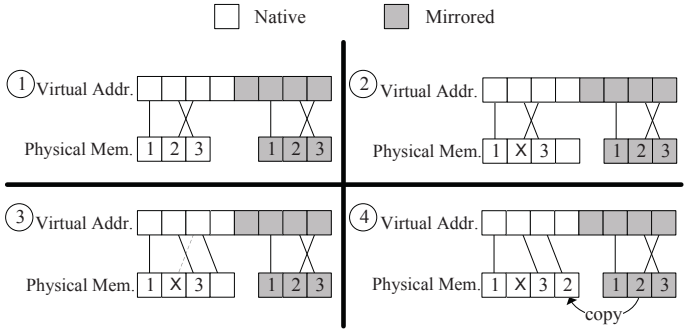


Fig. 3. The process of memory recovery: 1. The correct memory status. 2. If a page is corrupted, a new page will be allocated for the VM. 3. Map the virtual address to the new page. 4. Copy data from the mirrored address.

2) **Create a page table:** When the OS creates a page table entry (PTE) to map *nva* to *npa* (native physical address), Memvisor will intercept this process and create a mirrored PTE map *mva* = **mirror**(*nva*) to *mpa* (mirrored physical address).
3) **Write data to memory:** When a write instruction writes data to *nva*, another instruction will be inserted by binary translation technology, and write the same data to *mva*. Since *mva* is mapped to another physical address: *mpa*, the data can be physically copied (Figure 2).

When a memory failure happens, a hardware detection mechanism (e.g., parity check [5] or ECC) will notify Memvisor, and Memvisor will quickly and effectively retrieve the corrupted data by the following steps (Figure 3):

1) **Prepare a new physical page:** If a page with memory errors is detected, the data should be recovered in another mirror physical block. First, we should prepare a new memory page to recover the data.
2) **Remap the new page:** Rewrite the corrupted PTE and map it to the new page just allocated, so we build a new map between the new physical page and the corrupted virtual memory address.
3) **Recover the corrupted data:** Copy the mirrored data from *mva* to *nva*. After data recovery, the program will continue to run.

C. Binary translation

Memvisor uses binary translation technology to duplicate memory write instructions. When a memory write instruction appears, we should insert a mirror instruction to back up the memory and keep the native and mirror virtual memory space consistent. We choose static binary translation, because the redundant instructions are added at the compile time which reduces the performance overhead. Dynamic binary translation which translates the instructions at the run time is also an alternative design.

There are some side effects after the mirrored instructions are inserted. For example, if a hardware interrupt happens between one native instruction and the following mirrored instruction, the data will not be replicated for a relatively long time. We will discuss this issue in Section V.

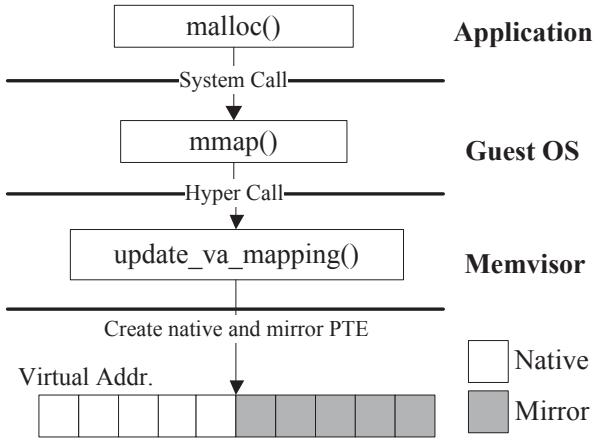


Fig. 4. The process of memory allocation in Memvisor. During the hypercall processing, first the native page table and then the mirrored page table are created.

III. Implementation

We have implemented the Memvisor prototype in Xen-3.4.2 with 231 lines of code. Those modifications include two parts: First, a block of physical memory should be allocated as the mirrored physical memory when a virtual machine starts up. Second, a mirrored PTE should be created when the native PTE is created. We choose Linux 2.6.30 as the Guest OS, and the guest OS memory management strategy also needs to be changed slightly. Finally we will introduce the approach to add mirrored instructions writing data to the mirrored memory.

A. Mirrored page table

Although we can fill in the mirrored PTE in the page fault process, it will result in a low performance because of the overhead of intercepting frequent page faults. So we need to modify the hypervisor memory management to ensure that the mirrored PTE and native PTE are created simultaneously. There are three memory virtualization technologies: direct page table (DPT), shadow page table (SPT) [15] and extended page table (EPT) [16]. EPT is a hardware assistant memory virtualization, which cannot be modified by software. Meanwhile, DPT and SPT are the software implementations which can be employed in Memvisor. So we will introduce both technologies briefly:

- **Direct page table:** When a guest OS modifies sensitive page tables, it will uses a hypercall to notify Memvisor. Then, Memvisor intercepts the hypercall and fills in the PTE if the operation is legal. So a mirrored PTE can be filled in after the native PTE is created (Figure 4).

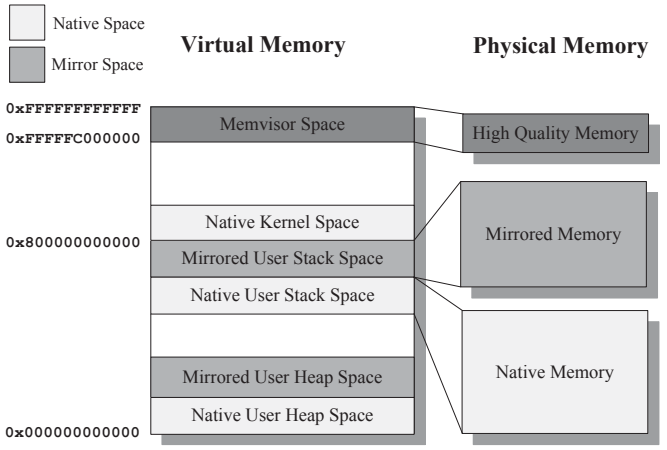


Fig. 5. A simple memory layout of the Memvisor implementation. The light grey area means native area while deep gray means mirrored. Virtual addresses are mapped to the physical addresses by the page table.

- **Shadow page table:** DPT needs to modify the source code of guest OS. On the contrary, once the guest OS modifies *cr3* to point to the new page table, it is a sensitive operation which will notify Memvisor. Then Memvisor will translate the native page table to a new table and reset *cr3* to point to the new page table with mirrored PTEs. During this translation process, SPT needs not modify the source code of guest OSes.

In SPT, due to frequent process switches, the *cr3* will be accessed very busy. Once *cr3* is changed, it will cause a page table update to keep the consistency with the mirrored page table, which incurs a big overhead. Although we have implemented both DPT and SPT, we finally choose the DPT technology to evaluate its performance and correctness.

B. Modified guest OS

Memory management in Linux should be modified to support Memvisor. One of them is the *vm_struct* which records the page table including the map between virtual and physical address [17]. When a process is killed, Linux will iterate through the page table with the help of *vm_struct* to release the virtual memory. However, we modify the page table through Memvisor, but the guest OS does not know this modification. Thus, an error will occur because there is no *vm_struct* for the mirrored page table. To avoid this problem, we add a special flag in a PTE for mirrored memory, which can be used by the guest OS to identify a native PTE from a mirrored PTE. When the guest OS finds that a PTE is mirrored, the guest OS just ignores it.

C. Modified memory layout

In this paper, we employ the plus operation in the **mirror** function, that means:

$$mva = \textbf{mirror}(nva) = nva + offset \tag{2}$$

$offset$ is a relatively large integer constant, and the value of it will be discussed later. The plus operation does not need additional instructions on the x86 platform, because it can be integrated into *mov* instructions.

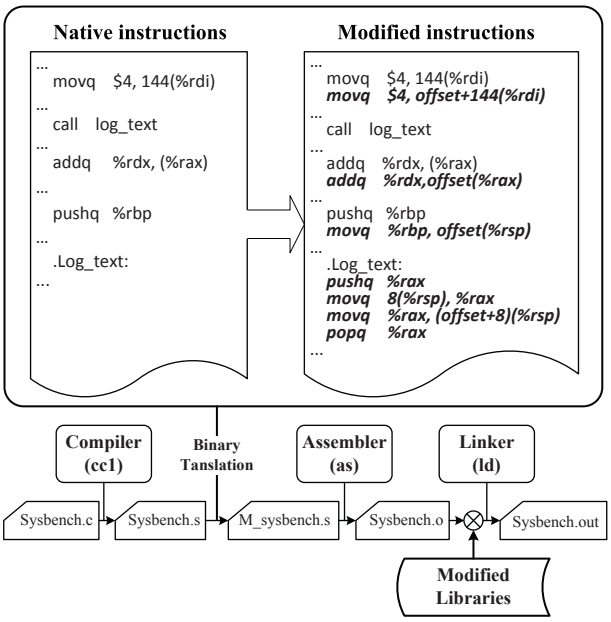


Fig. 6. The process of static binary translation.

Figure 5 shows a memory layout in our implementation. The lower virtual addresses are for user space which includes heap and stack space, and the higher addresses for the kernel. Memvisor occupies the highest virtual addresses. Each user mode virtual memory area has its own related mirrored area. The native and the mirrored virtual area are mapped to different physical memory to ensure that data is replicated physically. This figure is a simple layout, the actual memory layout will be more complicated, e.g., a multi-threaded program has more stack areas, and the dynamic link area is also not shown in this figure.

D. Synchronize the native memory and the mirror memory

To ensure memory is replicated synchronously, a writing operation to the native memory should be followed by an identical writing operation to the mirrored memory. This can be done via dynamic or static binary translation technology. We employ the latter to implement the process.

Memvisor performs static binary translation with the help of the GNU compiler collection (GCC). After the compiler (CC1) generates the assembly-language source files. Memvisor analyzes them and adds mirrored instructions. Then Memvisor passes the modified source files to the assembler (AS) which translates them into object files. Finally the linker (LD) merges the object files into an executable file with modified libraries.

As we described in the previous section, now the mirrored space has been created and original instructions have been fetched. Then there are two problems needs to be solved: (1) What kind of mirror instructions should be injected? (2) Where to inject mirror instructions?

For the first problem, choosing a mirror instruction is based on the type of original instructions. Only memory write instructions should be mirrored. If an original instruction is relatively simple, so the virtual address can be calculated in an explicit way, then injecting a mirror instruction is relatively

simple. That is, we just copy the original instruction types and formats, and then replace the original address with mirrored address. If an original instruction is relatively complex, so memory address cannot be calculated in this simple static translation. We should need some special instructions to duplicate the original space. Generally, an original instruction needs one mirror instruction. But for some complex original instructions, such as int, we may need more than one instruction to back up mirrored memory.

For the second problem, once original instructions modify the original memory data, we then update the mirrored memory. Generally, mirroring instructions always follow the corresponding original instructions, such as mov instruction. But for some instructions will change the execution sequence of program, such as $call$, mirror instruction should not be inserted after just original instructions. Because $call$ will push a return address into the stack, so it is a challenge to keep the mirror stack consistent with original stack. We will discuss this case soon.

E. Handle various kinds of write instructions

Most instructions access registers and many of them access memory. Instructions that change the data in memory can be classified into two main types: explicit write instructions and implicit write instructions. Memvisor can handle both precisely with specific approaches.

1) Data transfer instructions: Most of the data transfer instructions are memory-related, i.e., mov, $push$, $pusha/pushad$, and $xchg$. According to the style of memory access, they can be divided into two kinds: explicit and implicit write instructions.

Explicit write instructions: Most memory writing instructions are explicit write instructions (i.e., *mov*, *push*) which store data directly to a memory address. Memvisor can get the native virtual address and calculate the corresponding mirrored virtual address via equation (2). Then a mirrored instruction should be added to write the same data to the mirrored virtual address.

Take mov as an example, we can inject the mirror instructions as follows,

```
......
movl %eax, (%esp)
movl %eax, offset(%esp)
......
movb $0, (%eax)
movb $0, offset(%eax)
......
```

Implicit write instructions: Some instructions manipulate the data in memory indirectly (i.e., *call*, *int*, etc.). Sometimes, the native virtual address is invisible within the scope of the instruction. For example, when a *push* instruction saves the current data into the stack, the native virtual address is represented by the value in the stack pointer (SP) register (i.e., *rsp* register). Memvisor should first get the native virtual address from the SP register and add an instruction that saves the data to the mirrored memory address calculated via equation (2). These instructions generally consist of many complicated procedures in which Memvisor should catch the memory writing ones.

Take $push$ as an example, we can inject the mirror instructions as follows,

```
......
pushl %eax
movl %eax, offset(%esp)
......
pushl %esp
pushl %eax
movl (%esp), %eax
movl %eax, offset(%esp)
popl %eax
......
```

2) Arithmetic, logical and shift instructions: This kind of instructions include arithmetic, logical and shift operations as follows,

Arithmetic	add/sub/imul/idiv/inc/dec
Logical	and/or/xor/not
Shift	sar/shr/sal/shl/ror/rol
Bit operations	bt/btc/btr/bts

The format of these instructions is Opcode Data Address or Opcode Address. So the address of these instructions can be obtained explicitly. Therefore, we can deal with them as the same to previous explicit write instructions.

3) Control transfer instructions: Unlike data transfer instructions, "jmp", "$call$", and "int" are complex to mirror. For example, all *call*s access registers and memory. They save the return address on the stack and branch to the called procedure. Memvisor should save these data which are pushed onto the stack. But this problem cannot be solved by just adding a mirrored instruction after the *call* instruction, because it will not be executed until the called procedure returns. So Memvisor saves this data right at the beginning of the called procedure and writes it to the mirrored memory.

Take $call$ as an example, we can inject the mirror instructions as follows,

```
......
call grep
......
grep:
pushl %eax
movl %eax, offset(%esp)
movl 4(%esp), %eax
movl %eax, offset+4(%esp)
popl %eax
......
```

For system call instruction int, if we only focus on user-mode mirroring, it can be ignored. However, it is not hard to obtain a kernel-mode mirroring. If the privilege level is changed, i.e., from user mode to kernel mode, int will push five registers, i.e., SS, ESP, $EFLAGS$, CS, and EIP into the stack. If not, only latter three registers will be pushed into the stack. So it is a change to mirror this instruction. Therefore, we should know the context before int. Another problem is where to inject the mirror instructions. Once int is called, we have no chance to inject mirror instruction to back

up the current return address. Similar to $call$, we inject the code at the beginning of interrupt handler.

Take int as an example, we can inject the mirror instructions as follows,

```
.....
int $64
.....
alltraps:
 pushl %eax
 movl %eax, offset(%esp)
 movl 4(%esp), %eax
 movl %eax, offset+4(%esp)
 movl 8(%esp), %eax
 movl %eax, offset+8(%esp)
 movl 12(%esp), %eax
 movl %eax, offset+12(%esp)
 movl 16(%esp), %eax
 movl %eax, offset+16(%esp)
 movl 20(%esp), %eax
 movl %eax, offset+20(%esp)
 movl 24(%esp), %eax
 movl %eax, offset+24(%esp)
 popl %eax
 .....
```

4) String manipulation instructions: X86 has a kind of special instruction called string manipulation, i.e., $movsl$, $stosb$, and rep. These instructions use the register RSI as source operator, and EDI as target operator. It is repeatedly executed by the loop counter register ECX.

Because these instructions need to modify the value of ESI, EDI, and ECX in original instructions. So we should mirror the destination operand address with an offset of EDI.

For example, we can inject the mirror instructions as follows,

```
.....
movl $19, %eax
movl %edx, %edi
movl %ebx, %esi
movl %eax, %ecx
rep movsl
addl $offset, %edx
movl %edx, %edi
subl $offset, %edx
movl %ebx, %esi
movl %eax, %ecx
rep movsl
.....
```

5) Other instructions: Some binary instructions will use a special prefix symbol, such as $lock$, $addr16$, etc. It is simple to deal with these special instructions according to the prefix notation. Moreover, we should only handle all memory write instructions.

This paper only presented Intel 32 bit instruction set, but the binary translation can be extended to the X86-64 instruction set in a similar way, with the modification of the respective length and name of registers. Actually, we have implemented the binary translation of 64 bit instruction set.

6) Merge with modified libraries: An application will use Application Programming Interface (API) provided by the OS and libraries provided by various third parties. Some function in libraries (i.e., C language standard library "printf" function) will finally call APIs (i.e., $write$ in Linux) or other library functions. Meanwhile, some functions will not call any APIs (i.e., $strlen$). Whether it includes APIs or not, it does not affect the injection of mirror instructions. Static binary translation should cover all the source code of the executable file.

So Memvisor should modify not only the application source code but also the standard libraries so that no memory writing operation is beyond Memvisor's reach. We first use static binary translation to obtain a mirrored version of libraries, then link them to get a final executable file. So the standard libraries should be dealt with via the same approaches for write instructions. Then the linker will merge object files and the modified libraries into an executable file. In theory, all the data in the mirrored memory should be identical to that in native memory.

IV. Evaluation

Memvisor adds additional instructions to the system, which will lead to some performance impact. In this section, we focus on the overhead of our approach. Before we measure the performance, we will verify the correctness of Memvisor. After that, a micro-benchmark will test the memory related operations. Then we choose Sysbench [18] as the benchmark to measure the performance impact. Finally, we test a practical application, SQLite [19] (represent for memory-intensive tasks), to evaluate the performance and concurrency impact in real world applications.

A. Test environment

Our tests are run on a Dell PowerEdge T610 server with a 6-core 2.67GHz Intel Xeon CPU with 12MB L3 cache. There are two Samsung 8GB DDR3 RAMs with ECC and a 148GB SATA Disk. As we described in the previous section, Memvisor is implemented on Xen-3.4.2. We choose Linux as the guest OS, the kernel version is 2.6.30, and we also deploy a lightweight system, Busybox-1.19.2 [20]. Each VM is allocated two virtual CPUs and 64MB memory. We use Sysbench-0.4.12 as our benchmark, and choose SQLite representing real world applications.

Busybox, Sysbench and SQLite have had mirrored instructions inserted throughout, while the *libc* library and other libraries have been added on-demand only.

B. Correctness verification

To verify the correctness of Memvisor, we added a hypercall and code in Memvisor to compare the native memory and the mirrored memory bit-by-bit. Given that not all kernel instructions have been modified, we only verify the lower virtual addresses, the user space. We choose Sysbench to do the verification. When Sysbench is running, we invoke the syscall and hypercall to verify the data, the result is that about 99.7% of data are exactly same. Especially in the user heap area, the correctness is 100%. We also print the position of

incorrect data and find that all of them are located in the static data area and stack area. We believe the incorrect static data is due to the OS loader, and the incorrect data on the stack is due to the *libc* library; neither of them has all write instructions doing mirror because of a large engineering effort. We also conducted our user-mode memory mirroring on the XV6 platform, with Sysbench-0.4.12. We test the memory from 5,120 bytes to 20,480 bytes, all user-mode memory are 100% identical to original memory. So our solution is practical to back up the user-mode memory mirroring.

C. Case study: error recovery

Although memory errors are common in cluster, they are scarcely happened in a single machine at a short period time. To validate the availability of Memvisor, we design a case study to emulate the memory error in VM and observe the recovery.

In order to simulate the memory failure, we use a hypercall to pollute a specific block of memory area (i.e., 5M bytes) with garbage data. In real case, if memory corrupts, it will notify OS to restart machines, but this notification is captured by Memvisor. So in simulation test, it will invoke the recovery process to restore data, as we described in previous sections. We take a new page to replace the corrupted page, then we remap the virtual memory to these new pages and recover the data from mirrored pages. So when the recovery succeeds, we can get the original context from the above polluted memory with garbage data. This case study had been tested on user mode applications. We invoke the hypercall when an application is running. As we expected, when the recovery process is finished transparently, the correctness of the application or even the whole VM are unaffected. In our case, it takes about 24000 CPU cycles to restore each page. That means if the size of error memory is less than 500KB, it can be fixed in 1ms. In practice, both user and application will not be aware of the failure due to such short recovery time.

D. Micro-benchmark

Memvisor modifies the hypervisor memory management module to create the mirrored space which may lead to some performance impact when creating the page table. For a user application that means the *malloc()* function may take more time. The additional mirrored instructions may also double the time for a memory write. We therefore design a study case to measure those operations. To observe the impact of cache, memory tests are split into four categories: first-time write cost, average sequential write cost, average random write cost and average sequential read cost.

In *malloc()* operation, the overhead is mainly ascribed to the creation of additional PTE (Page Table Entry) and the cost of TLB (Translation Lookaside Buffer) flush. The impact on performance is somewhat large when the allocated memory is less than 256KB but much more limited when the size is larger. This is because the initialization overhead for the allocation of new memory (i.e., PTE, TLB). However, the overhead is amortized over the larger allocation so the overall effect is small.

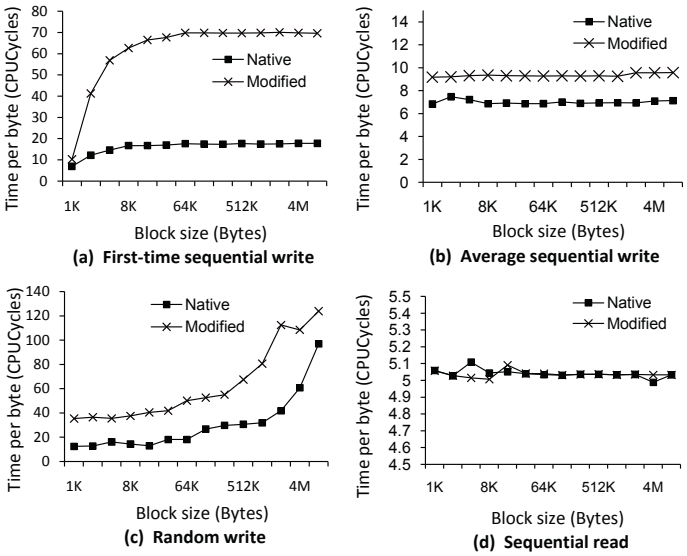


Fig. 7. (a) The overhead for first-time write is somewhat high at about 500%. (b) The average overhead for sequential write is small at about 80%. (c) The average overhead for random write is between those two at about 100%. (d) The overhead for sequential read is almost none.

Figure 7 shows the performance result with different cache impact. Without the assistance of cache, the overhead of first-time write is somewhat high especially when the block size is larger than 10KB. For blocks less than 10KB, the load is not memory-intensive so the overhead of Memvisor is not significant. The average overhead of the sequential write is much lower than that of first-time write due to the high cache hit rate. The average overhead of a random write is between first-time write and sequential write because of the cache miss rate; the larger the block size is, the higher the miss rate will be. There is no big difference in the average overhead of the memory read as expected.

E. Sysbench

Sysbench consists of several test cases for CPU, memory, IO, etc. Figure 8 plots the geometric mean for Sysbench tests. The IO test is configured as direct IO (no additional memory as cache) to eliminate the memory impact. It is obvious that Memvisor incurs a negligible overhead for CPU and IO intensive applications. The mutex test has some memory operations which cause a little overhead. The memory read test has almost zero overhead while the memory write test shows non-trivial impact. But the overhead is still acceptable as the inevitable cost for high availability.

In the experiment, we fixed a bug in Sysbench-0.4.12; it would do nothing in the memory read test due to the *GCC* -O2 optimization. We also notice that the block size in a memory-intensive test will affect the result. To illustrate this relationship, we designed a test to measure the impact of read and write memory with different block size ranging from 1MB to 10MB. The result is shown in Figure 9. The overhead in the memory read test is negligible and remains the same as the block size increases. The overhead in the memory write test is in line with the block size approximately at 80%.

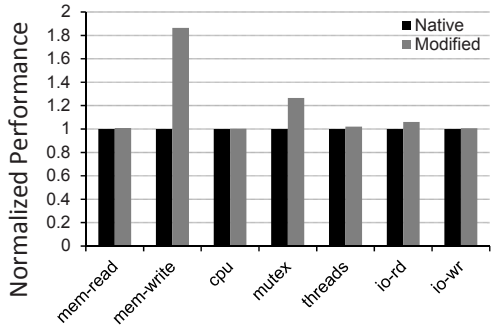


Fig. 8. The overall result of Sysbench test suits.

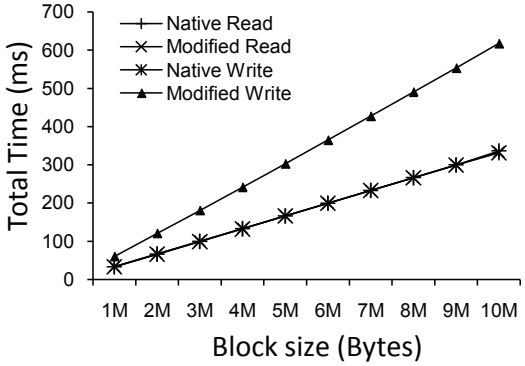


Fig. 9. Sysbench memory-intensive test with different block size.

F. Database

Databases are widely used in cloud services, and it can be also used to represent memory-intensive applications. SQLite-3.7.11 is chosen to measure the performance impact. We write a test suite including five kinds of operations: *create*, *insert*, *select*, *update* and *delete*. Then, we measure the total execution time (including the time of resolving SQL statements) for each test with the database size of 50M bytes. In order to compare the overhead, we use the normalized performance benchmark in Figure 10. That is, the execution time of all native operations are normalized as 1. So the higher of Y-axis for modified operations means the larger overhead of mirror memory. Figure 10 shows the results.

The results indicate that different operations present different overheads in Memvisor.

The *insert* operation causes only little overhead while the *create* operation needs double time to finish the same thing. This difference depends on the implementation of SQLite, e.g., the cache mechanism will cause frequent memory writing operations. Luckily, the *create* operation is scarcely used in a real world server, and the other four operations are used frequently but cause a little performance impact. We can conclude that Memvisor is also suitable for a database. Moreover, in this result, the relative overhead on *select* is larger than *insert*, which is counter-intuitive as select should mostly consist of memory read operations while insert must consist some memory write operations. However, SQLite has many memory operations. For example, there are more than fifty thousand $movl$ instructions. Also there are a lot of $call$ and $push$ instructions that require a lot of processing of system resources, thus causing a different overhead between *select* and *insert*.

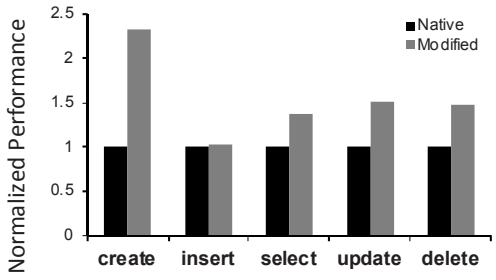


Fig. 10. The performance impact for *create*, *insert*, *select*, *update* and *delete* in SQLite.

V. Discussion

The current Memvisor still needs to resolve the following issues. First, some memory areas cannot be made redundant or cannot be recovered if an error happens, e.g., Memvisor area, page table area, etc. Second is that the native virtual address and the mirror virtual address may conflict. Last, we will discuss a solution for the problem in a multi-threaded environment which has been mentioned in Section II.

A. Special memory mirror areas

There are several memory areas such as page table, device, DMA, and Memvisor own memory, which cannot be made redundant. For example, in the x86 platform the page table is located in physical memory. The problem is that replicating a mirrored page table would need to write a new mirrored page table, which would cause an infinite loop. For I/O-mapped areas used by input/output devices, it is also unnecessary to mirror because they are special I/O ports. In traditional X86 architecture, we use in and out instructions to access devices, while using mov instructions to access device for I/O-mapped areas. Unfortunately, during binary translation phase, we cannot distinguish mov instructions between I/O mapped operations and normal memory access. Fortunately, these special memory areas are located on the kernel mode space, which do not affect our use mode application level memory mirroring. So it is a great challenge to support kernel mode memory mirroring for a system wide high availability.

B. Guest OS's modification

Currently, we use DPT in the hypervisor level to optimize performance. Although the major modification of source code is in Memvisor, the guest OS such as Linux is also modified to support DPT, because the page table created by DPT is not transparent to a guest OS. That is, when Linux lookups its page table, it will cause potential data inconsistency. So we should modify the source code of Linux to ensure that it is transparent to lookup operations when accessing the backup page table. However, if we use SPT, this problem disappears because it is transparent to the modification of the page table in Linux.

Another problem is address conflict, which means two or more memory area are overlapping. As mentioned above, the

mirror function in Memvisor's current implementation is a simple addition, *mva* = *nva* + *offset*. The offset is a constant and needs to be set properly. There is a simple solution to address this problem. That is, we use small applications and choose *offset* carefully, which are useful in practice. But this problem is not solved completely, because some applications with large memory are prone to have address conflict. One possible solution is to reorganize the layout of memory with the backup space reserved in guest OSes, which breaks the transparency of guest OSes. This solution works for open source guest OSes such as Linux, but not for Windows.

C. Static and dynamic binary translation

In order to achieve process-level and application-level high reliability, we can use static or dynamic binary translation to inject mirror instructions. Compared with static translation, dynamic binary translation often needs more system resources including cache or runtime support components, which also incur more overhead. Furthermore, if we extend it to kernel mode mirror operations, it is difficult to deal with interrupt exception. Meanwhile, static translation is handling code at the compile time, so we can also deal with kernel mode code in a similar way. However, it is also limited to self-modification code. Fortunately, Linux is an open source OS and we use GCC to insert mirror instructions during the compile time.

D. Multi-threaded environments

In a user mode multi-threaded environment, if the binary translation module finds a lock-prefix instruction, to assure the atomicity after the code is modified, an explicit mutex lock should be added for each lock-prefix instruction and its mirrored instruction, which incurs a large performance overhead. If we extend Memvisor to support kernel model memory mirroring, it needs to check the instruction after an interrupt happens. Similar to function calls, if the next instruction is a mirrored instruction, it should emulate this instruction before executing the interrupt handler. That ensures all the mirrored data is created immediately.

VI. Related Work

High Availability plays an important role in computer system's architecture. Memory HA is also considered as a part of the whole system HA. Google File System [11] and Dynamo [12] are the most famous application level HA systems. They employ special algorithms to distribute data on two or more physical machines, and can recover the data immediately from the backup. Another typical model of system HA is dual-machine VM replication [21]. A backup server is used to get synchronized to the primary host. Nowadays, Xen, VMware [22], and KVM have implemented their live migration to take over failures, with the optimization of memory page pre-copy to reduce downtime. Furthermore, Remus [13] uses synchronous replication and speculative execution based on Xen to optimize the performance. In this solution, the state of the primary VM is frequently recorded and transmitted to the backup server during execution. Therefore, Linux kernel compilation time was doubled, and SPEC-web benchmarks suffer more slowdown when doing 40 checkpoints per second using a 1 Gbit/s network connection for transmitting changes in memory state. Kemari [23] aims to keep VMs transparently running in times of hardware failures. VMWare has provided another replication model based on replaying [22], but it can be applied only to uniprocessor VMs and is highly architecture-specific. Memvisor has better performance, which brings only about 55% overhead with the most affected memory-write-intensive workload. Moreover, without the limitation incurred by checkpoints, Memvisor replicates the data on the fly. Finally, Memvisor is a solution based on single machine backup, so the network bandwidth is no longer a problem.

Hardware redundancy: Besides the above software solutions of high availability, hardware providers consider using extra bits to check and correct memory errors. A typical technology is Parity [5] which uses one extra bit to check one byte data. Another solution is ECC [6], which uses Hamming Codes [8] to detect and correct the internal data corruption. Some machine providers promote ECC to support their motherboard services (e.g., HP Advanced ECC [9], Google ECC [24], and IBM Chipkill [25]). However, both Parity and ECC can only retrieve limited bit errors rather than massive block failures and more error check bits are needed with the increment of native data bits.

In order to address single-bit and multi-bit errors, HP mirrored channel [9] and Dell mirrored memory [26] on PowerEdge 1850 provide full protection against single-bit and multi-bit errors. The subsystem writes identical data to two channels simultaneously. And it automatically retrieves the data from the mirrored when errors happen. However, the high cost and strict environment requirement become a problem for providers. The providers may possibly pay a double or triple cost to guarantee their memory availability. Meanwhile, hardware solution also represents a poor flexibility in mission-critical servers. If mission-critical clients and non mission-critical ones coexist in one cluster environment, extra work needs be done to schedule mission-critical clients only to such hardware mirrored servers. With Memvisor, each VM only needs a configuration of whether to support memory HA or not. Compared to the hardware mirrored, Memvisor is a cost-effective and flexible solution.

VII. Conclusion

Memvisor is a software solution achieving a multi-granularity memory mirroring to enhance the process- and application-level high availability. By duplicating the memory write instructions, Memvisor replicates the whole memory, and recovers the data when memory failures are detected. With the help of virtualization technology, Memvisor could support high available VMs and native VMs simultaneously. VMs can easily choose to use process level or application level high availability. Memvisor also can increase memory copies on demand, which is more flexible than hardware approaches. Moreover, Memvisor leverages the binary translation technology to guarantee the data are replicated on the fly, which is a large improvement over other software HA solutions.

The current Memvisor is implemented with static binary translation and Direct Page Table technology. Compared with current software HA systems, e.g., Remus configured with 40

checkpoints per second incurs 103% overhead, our instruction level rewriting avoids unnecessary data copy. The results show that the performance of CPU-intensive tasks is unaffected, and even in the worst case, our stressful memory write benchmark shows the backup overhead of 80%. This paper also discusses the method to implement Memvisor with dynamic binary translation and shadow page tables, and we hope these features can be done in the future. Moreover, our implementation is limited on the user-mode applications, we will extend it to the kernel-mode memory mirroring to implement a system-wide availability. Moreover, the comparison between Direct Page Table over Shadow Page Table will be conducted. Meanwhile, we will implement our prototype on more operating systems, such as embedded systems and mobile systems and evaluate its usage and performance on these platforms. We will also develop the user mode plug-in for applications to develop a mirrored backup for critical data structures.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (No. 61272101), NRF Singapore CREATE Program E2S2, National Science and Technology Major Project of China (No. 2013ZX03002004), and National Key Project of Basic Science and Technology Research of China (No.2013FY111900).

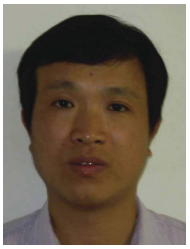
REFERENCES

[1] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," *Commun. ACM*, vol. 54, no. 2, pp. 100–107, 2011.
[2] Google, "App Engine Service Level Agreement," https://developers.google.com/appengine/sla.
[3] Amazon, "Amazon EC2 Service Level Agreement," http://aws.amazon.com/ec2-sla/.
[4] . Panzer-Steindel, "Data integrity," *CERN/IT*, 2007.
[5] C. L. Chen, "Error-correcting codes for semiconductor memory applications: a state-of-the-art review," *IBM J. Research and Development*, 1984.
[6] L. Levien and W. Meyers, "Special feature: semiconductor memory reliability with error detecting and correcting codes," 1976, pp. 43–50.
[7] R. Gallager, "Low-density parity-check codes," *IRE Trans. Inf. Theory*, pp. 21–28, 1962.
[8] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical J.*, 1950.
[9] HP Corporation, "HP advanced memory protection technologies," http://h18000.www1.hp.com/products/servers/technology/memoryprotection.html.
[10] D. Fiala, K. B. Ferreira, F. Mueller, and C. Engelmann, "A tunable, software-based dram error detection and correction library for hpc," in *Proc. 2011 Euro-Par*, vol. 7156, pp. 251–261, 2012.
[11] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *SOSP*, 2003, pp. 29–43.
[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and Werner Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP*, 2007, pp. 205–220.
[13] B. Cully, G. Lefebvre, D. T. Meyer, M. Feeley, N. C. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication (best paper)," in *NSDI*, 2008, pp. 161–175.
[14] H. Dong, W. Sun, B. Wang, H. Sun, and Z. Qi, "Memvisor: application level memory mirroring via binary translation," in *CLUSTER (poster)*, 2012.
[15] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*, 1st ed. Prentice Hall, 2007.
[16] G. Wu, J. Gao, H. Zhang, and Y. Dong, "Improving PCM endurance with randomized address remapping in hybrid memory system," in *CLUSTER (poster)*, 2011, pp. 503–507.
[17] M. Gorman and P. Healy, "Supporting superpage allocation without additional hardware support," in *ISMM*, 2008, pp. 41–50.
[18] Sysbench, "Sysbench Doc," http://sysbench.sourceforge.net/docs/.
[19] SQLite, "SQLite Web Site," http://www.sqlite.org/.
[20] D. Vlasenko, "BusyBox: The Swiss Army Knife of Embedded Linux," http://www.busybox.net/.
[21] C. Clark, K. Fraser, S. H, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI*, 2005, pp. 273–286.
[22] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *Operating Systems Rev.*, vol. 44, no. 4, pp. 30–39, 2010.
[23] Y. Tamura, K. Sato, S. Kihara, and S. Moriai, "Kemari: virtual machine synchronization for fault tolerance," in *USENIX ATC*, 2008.
[24] J. M. Deegan, "High reliability memory subsystem using data error correcting code symbol sliced command repowering," US Patent 7,206,962, Google Patents.
[25] T. J. Dell, "Ecc-on-simm test challenges," in *ITC*, 1994, pp. 511–515.
[26] Q. Li and U. Patel, "Enabling memory reliability, availability, and serviceability features on Dell PowerEdge servers," http://www.dell.com/downloads/global/power/ps3q05-20050176-patel-oe.pdf.

Zhengwei Qi received his B.Eng. and M.Eng degrees from Northwestern Polytechnical University, in 1999 and 2002, and Ph.D. degree from Shanghai Jiao Tong University in 2005. He is an Associate Professor at the School of Software, Shanghai Jiao Tong University. His research interests include program analysis, model checking, virtual machines, and distributed systems.

Haoliang Dong received the B.E degree from Shanghai Jiao Tong University in 2010. He is a graduate student at Shanghai Key Laboratory of Scalable Computing and Systems, School of Software, Shanghai Jiao Tong University. His research interests mainly include virtual machine, distributed systems, and mobile computing.

Wei Sun received the B.E degree from Northwestern Polytechnical University in 2010. He is a graduate student at Shanghai Key Laboratory of Scalable Computing and Systems, School of Software, Shanghai Jiao Tong University. His research interests mainly include virtual machines, cloud systems, and binary translation.



Yaozu Dong is currently a Ph.D. candidate at Shanghai Jiao Tong University under the supervision of Professor H.B. Guan. He is also a software architect of Open Source Technology Center at Intel Corporation, who works on Linux virtualization including KVM and Xen. Yaozu Dong is an active participant of both Industry and academia event, and a frequent presenter of Xen, KVM and academia conferences. Before that Yaozu worked in Linux kernel debugger and other OS enabling work for Xscale architecture in his 10+ years Intel working experience.

Haibing Guan received the PhD degree in computer science from the Tongji University, China. He is currently a professor with the Faculty of Computer Science, Shanghai Jiao Tong University, Shanghai, China. He is a member of the IEEE and ACM. His current research interests include, but are not limited to, computer architecture, compiling, virtualization, and hardware/software co-design.