A Solution for High Availability Memory Access

Chunjing Gan[1], Bin Wang[2,3], Zhi-Jie Wang[1], Huazhong Liu[4], Dingyu Yang[5,7] Jian Yin[1], Shiyou Qian[2], and Song Guo[6]

[1] School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China
[2] Dept. of Compu. Sci. & Eng., Shanghai Jiao Tong University, Shanghai, China
[3] Dept. of Computer Science, Ulster University, United Kingdom
[4] Huazhong University of Science & Technology, Wuhan, China
[5] Alibaba Group, Hangzhou, China
[6] Depart. of Computing, Hong Kong Polytechnic University, Hong Kong, China
[7] Guangdong Key Laboratory of Big Data Analysis and Processing, China
ganchj3@mail2.sysu.edu.cn, binqbu2002@gmail.com,
{wangzhij5,issjyin}@mail.sysu.edu.cn, sharpshark_ding@163.com,
dingyu.ydy@alibaba-inc.com, qshiyou@sjtu.edu.cn, song.guo@polyu.edu.hk

Abstract. Nowadays, in-memory computing has plenty of applications like artificial intelligence, databases, machine learning, etc. These applications usually involve with the frequent access to memory. On the other hand, memory components typically become error-prone over time due to the increase of density and capacity. It is urgently important to develop solutions for high-availability memory access. Yet, existing solutions are either lack of flexibility, or consistently more expensive than native memory. To the end, this paper presents a solution called SC2M. It is a software-controlled, high-availability memory mirroring solution. Our solution can flexibly set the granularity of the memory areas for various levels. Furthermore, it can perform duplication of the user-defined data structures in a high-availability version. The systematic instruction-level granularity for memory duplication reduces the overheads for backup, and lowers the probability of data loss. Experiment results demonstrate the feasibility and superiorities of our solution.

Keywords: High availability, hardware virtualization, system architecutre.

1 Introduction

In modern artificial intelligence (AI) system, it is known that vicious attacks to memory modules are common [33, 27]. An error-prone AI system may cause serious issues in terms of service quality and computation efficiency [15]. Besides the AI system, many other applications are also significantly depended on the correctness of memory. For example, in recent years in-memory computing has been more and more applications such as *Redis* [23], *Memcached* [1], *Spark* [34].

The above applications and many others generate large mission-critical workloads, which need to frequently access memory or cache. Particularly, these applications have strict requirements on the correctness and stability of memory.

Yet, memory components typically become error-prone over time, due to the increase of density and capacity [13]. Eventually, they may no longer guarantee *high availability* (HA). Moreover, although some errors from the *dynamic random access memory* (DRAM) can be detected by the hardware, they usually cannot be corrected instantly. These *uncorrected errors* often incur system crash [18, 5, 29]. All of the above facts have led to a sharp increase in the demand for high-availability memory access.

To provide high-availability memory access, different hardware vendors have implemented their own product-related solutions. For example, the bit-retrieval approach [14] has been widely used in industry, including the HP corporation [13], IBM corporation [7]. In these industries, a handful of motherboards have integrated the *error correcting code* (ECC) memory into their servers [14]. This approach, however, is often ineffective for the block failure. Moreover, another well-known hardware-based approach is *mirror memory* [14]. This approach uses the dual chip (i.e., double chips) to backup data on-the-fly. Although this dedicated approach is useful for many applications, it often generates too much overhead, and is consistently more expensive than native memory [30].

As for software-based solutions, a common method is by simulating hardware checking. For example, software ECC [12] works quite similarly as hardware ECC. Another useful software-based approach is to duo-backup at the application level [5, 18]. This can be witnessed in the Google and Amazon services [27]. Although duo-backup solutions show low latency and fewer interruptions, they usually only consider the application's tolerance, which cannot provide transparent high-availability for operating systems. On the other hand, some *virtual machine* (VM)-based solutions do effectively deal with this shortcoming [31, 30]. For example, a system named Remus [5] uses virtualization checkpointing technology to backup an entire VM. However, checkpointing technology does not backup the system extemporaneously, so data between two checkpoints may be lost during failure; in addition, the overhead of such solutions like Remus is more than 100%, compared to the native memory [5].

Instead of directly repairing above approaches, this paper presents a solution that is a software-controlled memory mirroring (known as SC2M), based on the principles of static binary translation and hardware virtualization. Here static binary translation technology enables our solution to provide a software-controlled high-availability, while virtualization technology uses software/firmware that divides physical hardware equipments into multiple independent virtual instances, it enables our solution to support multiple VMs on the same physical machine. In brief, SC2M explores a redundant memory space (called the *mirror space*) in the physical host machine; it injects instructions into the mirror space to backup the multi-level memory writes, where the *static binary translation* is used. Particularly, to backup data, it implements mirror instructions not only for user mode codes, but also for kernel mode codes. Therefore, when errors happen in the original memory space, the compromised data can be recovered from the mirror space, where only low-cost memory is required to recover data. Our solution allows the application to specify the data structure to be duplicated, and it is

flexible since it can support memory mirroring at different levels (ranging from data level, application level, to system level), and the mirror memory can be easily set to support N-modular redundancy for some specialized, business-critical applications. In addition, it is a lightweight and real-time mirroring solution, compared against traditional methods that use memory-mapped files [27]. Particularly, our solution is implemented by integrating the Intel *shadow page table* (SPT) [35]; this provides even more flexibility in distributed environments or other similar environments, where the *operating system* (OS) level access cannot be perceived by users. To summarize, the main contributions of this paper are: (i) We develop a software-controlled memory mirroring solution, called SC2M, that can achieve high availability, and is cost-effective. (ii) We conduct extensive experiments to evaluate its performance. The experimental results demonstrate the feasibility and effectiveness of our solution.

In next section, we review the related work. Section 3 presents the system architecture of SC2M. The workflow and implementation of our solution are covered in Section 4. We evaluate the performance of our solution in Section 5. Finally, Section 6 concludes this paper.

2 Related work

Existing solutions for high availability (HA) memory can be generally classified into two categories: hardware-based and software-based solutions. We next review prior works related to these two categories.

2.1 Hardware-based HA Memory

Initially, hardware providers adopt extra bits to check and correct memory errors, e.g., the well-known techniques are like *parity checking* [3] and *error correcting codes* (ECC) [16]. Some hardware vendors also promote ECC to support their motherboard services. For example, HP Advanced ECC [13], Google ECC [6], and IBM Chipkill [7]. Besides above approache, there are also various solutions that further improve ECC technique. For example, Odd-ECC [19] is used for conventional 2D DRAMs, DIMMs, or even to 3D-stacked DRAMs. Another notable ECC-based scheme is also proposed, and it introduces the In-DRAM ECC architecture [2]. On the other hand, bit-checking method for large area failures is also investigated [28]. Overall, these methods are effective to deal with limited bit errors, yet they are not suitable for handling massive block failures. To retrieve massive block failures, some works use the mirrored hardware. For example, HP's mirrored channel [13] provides full protection against single-bit and multi-bit errors. The subsystem writes identical data to two channels simultaneously. In case of errors, the system is able to automatically recovery the data from the mirrored memory. In [14], two optimization techniques are developed: *lazy-migration* and *adaptive-activation*. The lazy-migration technique increases the utilization of the rental memory via the volatile page allocations, while the adaptive-activation technique saves the active pages in the rental memory during

the reallocation. To some extent, these hardware-based solutions are usually not cost-effective, and may lack the flexibility.

2.2 Software-based HA Memory

In the literature, there are also many software-based solutions for HA memory. For example, SWIFT [24], a software-only fault-detection technique, duplicates a program's instructions by inserting explicit validation codes, and compares the results of original instructions and their corresponding duplicates. Later, CRAFT [25] adopts the extra hardware structures to improve the SWIFT technique. Compared to these methods, our solution mainly duplicates the memory write instructions, avoiding the extra overhead. Another typical software-based solution is the dual-machine VM replication [18]. As for this method, a backup server is synchronized to the host machine. Besides the methods mentioned above, there are also a lot of works that use VM migration and/or replication [10, 5]. For example, Remus [5], in which the state of the primary VM is frequently recorded and transmitted to the backup server during execution. Remus achieves the high availability memory, yet the compile time of the Linux kernel is doubled. In order to improve the performance and scalability of the Remus, the ReNIC system provides an architectural extension to the Single Root I/O Virtualization (SR-IOV) system that achieves efficient I/O replications [10]. This method requires some new hardware-assisted I/O virtualization (such as SR-IOV). Moreover, a system called Memvisor [9, 22] is proposed, which uses the direct page table (DPT) technique and is tailored for HA memory in cloud environments. Later, a system called kMemvisor [30] is also developed for HA memory in cloud environments and uses also DPT technique. However, this architecture requires the large modifications on the guest OS, while our solution does not need to perform those complicated modifications. In addition, our solution employs the shadow page table, which provides much more flexibility, and so it could be used for more environments besides cloud environments.

2.3 Others

Besides the above works, we also note that there are some works (e.g., *memory error prediction* and *algorithm-based recovery*) that could be complementary to our solution. For example, the authors in [29] propose a scalable and fault tolerant HPL, called SKT-HPL, and validate their method on two large-scale systems. Moreover, a system called Jenga [20] is proposed for protecting 3D DRAM, specifically high bandwidth memory (HBM), from failures in bits, rows, and blocks in the memory. On the other hand, memory error prediction is also benefit to improve the high availability of memory. For example, the work [17] introduces the cache persistence analysis into memory backup for self-powered non-volatile processors. One can integrate this prediction technique to improve the reliability of memory.

3 System Architecture

Figure 1 shows the architecture of SC2M. At the bottom of our architecture, two kinds of memories are deployed: (i) a native memory, and (ii) a mirror memory. The former is mainly used to achieve the high available VMs that can run mission-critical applications; here the VMs can be configured at the data-level, application-level or even to system-level, according to the mirror requirements. Correspondingly, the latter is mainly used to backup the native VM. Generally, SC2M is implemented with a *copy-on-write* manner. That is, whenever a write operation occurs in the native memory, the same write operation should be done in the mirror memory. At the top of our architecture, there are two major modules, which are *memory management module* and *code translation management module*. These two modules shall interact with the components at the middle level of our architecture. Next, we discuss more details about these two modules.

▶ *Memory management module.* It monitors the operations related to page table, and maintains both native and mirror *page table entries* (PTEs). Same to the memory management of modern OS (i.e., operating system), here page table is used to map virtual addresses into physical addresses, so that address space can be extended in memory. A PTE can be created using "syscalls" such as `mmap()`, `malloc()` methods in the kernel space. The memory allocator [4] in the hypervisor maintains the relationship between the *guest physical address* and the *guest virtual address* for each VM, and intercepts all PTE operations. In our solution, we implement the virtual memory allocator via Intel SPT (i.e., shadow page table) technique [35], which is a memory virtualization technology used in full virtualization where host OS can run multiple guest page tables, and the guest OS do not need to be modified [4, 35]; this provides even more flexibility in distributed environments or other similar environments, where the *operating system* (OS) level access cannot be perceived by users.

Denote by Add_{mva} and Add_{nva} the *mirror virtual address* and the *native virtual address*, respectively. The relationship between them is established by the equation: $Add_{mva} = Add_{nva} + offset$, where the *offset* is a constant value. In order to provide data and application-level mirror memory, SC2M defines interfaces by wrapping up the memory area with `getter()` and `setter()` methods. Actually, setting the offset is the most direct way to build the mirror address, since there is no need to modify the memory layout. Notice that, an appropriate value should be carefully selected for the offset, in order to avoid the address conflict and also to ensure the correct creation of the mirror address.

▶ *Code translation management module.* It inserts mirror instructions, identifies all memory write instructions, and replicates them. As we know, in the traditional X86 architecture, an instruction "write destination" is usually translated into a virtual address. Yet, injecting mirror instructions is complicated when some instructions do not have explicit write destinations. For example, as for the atomic and privilege operations, it is difficult to mirror them, due to the reasons above. SC2M alleviates these problems by using two ideas together: (i) translating the explicit write destinations into all write destinations using *static*

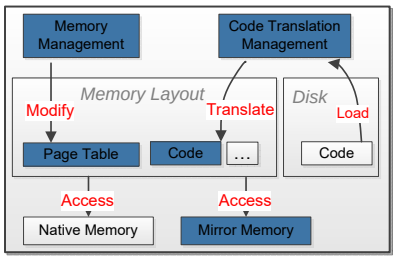

Fig. 1. Architecture of SC2M.

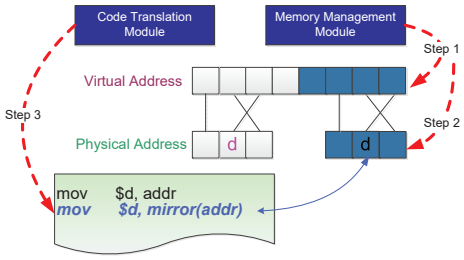

Fig. 2. Workflow of mirror initialization.

binary translation; and (ii) generating mirrored instructions at the instruction compile stage. An extra benefit of the above strategy is that it also reduces the runtime overhead. Specifically, the mirrored instructions are inserted when OS loads the program from the disk (see Figure 1 again).

4 Workflow and Implementation of SC2M

There are several important issues needing to be clarified in the implementation. They are: (i) the creation of mirror page table; (ii) the mapping from physical addresses to virtual addresses; (iii) data synchronization; and (iv) mirroring data for high-availability applications. Before we discuss these issue in detail, we first introduce the workflows of mirror initialization and data recovery, which could be helpful to understand the rest of the paper.

4.1 Mirror Initialization and Data Recovery

As for the mirror initialization, SC2M shall do the following steps (see Figure 2).

Step 1 — Reserve physical memory. When a VM (i.e., virtual machine) starts up, SC2M checks its configuration. When the VM is configured as HA-type, both native and mirror memory are to be created for this VM, and the sizes of the two memories are the same. In other words, in this step a block of physical memory will be reserved when the VM starts up.

Step 2 — Create mirror page table. SC2M intercepts the operations related to page table from the VM, and simultaneously creates mirror page tables. That is, when native PTEs are updated, related mirror PTEs are created as well. In our solution, Intel SPT technique is used, which provides us the *application program interfaces* (APIs) to implement this step.

Step 3 — Write redundant data. The mirror write instruction is replicated through the *static binary translation*, and the redundant data is written using the mirror instruction. In other words, native instructions and mirror instructions will write the same data to different addresses.

As for data recovery, it is mainly for recovering the corrupted data based on the data in the mirror space. Specifically, when a memory failure occurs, the hardware detection module (e.g., parity checking or ECC) notifies SC2M by invoking a *machine check exception*. Unlike the normal response of restarting the

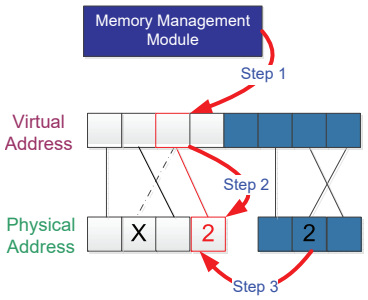

Fig. 3. Workflow of data recovery.

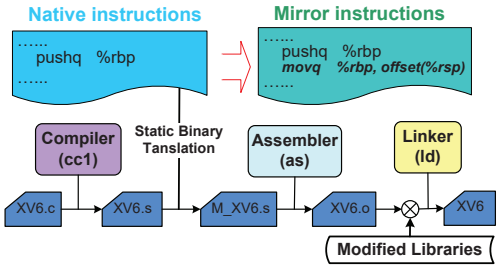

Fig. 4. Data synchronization of SC2M

OS (i.e., operating system), SC2M quickly and effectively retrieves the corrupted data through the following steps (see Figure 3).

Step 1 — Allocate a new page. In the native memory, the memory module allocates a new page from the free zone, blanks the options of the new page, and marks it with "writable"; here the new page will be allocated to the VM.

Step 2 — Remap the new page. The corrupted PTE is rewritten and mapped to the new page (allocated in Step 1). Here the virtual address will be mapped to the new page.

Step 3 — Recovery corrupted data. Data is copied from the *mirror virtual address* to the *native virtual address*. After that, the program continues to execute.

4.2 Creation of Mirror Page Table

As mentioned earlier, we employ the SPT (i.e., shadow page table) technique when creating *mirror page table*. In general, SPT is a technique that maintains the real mapping in the hardware when executing the guest instructions. Here the SPT module is invoked when the guest OS modifies the CR3 register, which is one of control registers. It contains the base address of the page table. In our implementation, SC2M translates the native page table to a new table, and resets the CR3 such that it still points to this new page table. Meanwhile, during the translation, SC2M creates the mirror PTEs (i.e., page table entries) in the new table. The above is the general implementation of creating the mirror page table.

We may need to mention that, in the full virtualization mode of our SC2M, the CPU shall conduct a permission check when an instruction is executed. Specifically, if the check reports an illegal operation, the hypervisor executes a predefined procedure to handle the problem. On the other hand, if a sensitive instruction is not privileged, then a guest OS may obtain (or even modify) resources belonging to the host (or other VMs), yet it does not inform the hypervisor. An immediate example is the `mov()` instruction, which is not privileged in X86-64, and so the hypervisor cannot trap any guest PTE operation from `mov()` instruction. Compared to the instructions like `move()`, updating the CR3 is a privileged instruction that can be used to track the operation of the guest PTEs. In our implementation, the CR3 operations are intercepted by the SC2M.

Naturally, our SC2M can easily read the PTE of the guest OS. This way, a new page can be translated to a new space in the SPT.

Additionally, we observe that OS may not always use the new information changed from the translation. This implies that, during the SPT implementation, there may not need to do a complete translation for every change in the CR3. To improve the performance of SPT, one can set the P (i.e., present) bit to 0 in SPT entries. This way, when a guest OS accesses this address, it causes a page fault due to the nature of the x86 architecture. This page fault can be also captured by SC2M. In this case, SC2M shall find the original value in the CR3, according to the SPT. Then, it determines whether the page fault is caused during the translation. If so, SC2M translates the PTE (i.e., page table entry) to finish the exception handling. Otherwise, it indicates that the page fault is produced by the guest OS, and so SC2M forwards it to the guest OS that shall handle it.

4.3 P2V Mapping

In the virtualization mode, it is necessary to build a mapping between physical addresses and virtual addresses. In our implementation, we rebuild the "*vm_struct*", which is a special data structure that stores the mapping between the virtual addresses and the physical addresses [31, 29, 31]. Our modification is mainly on the layout of guest memory. In our implementation, each virtual memory area has its own mirrored area. The native and the mirrored virtual areas are mapped to different physical memories to ensure that data is replicated physically. This modification allows us to free address space more efficiently. For example, when a process is killed, the virtual address and its related physical address can be released simultaneously. On the other hand, as for the layout of host memory, we exploit the mirror area in both the kernel and user spaces, and so each part of the memory shall reside in different locations of the physical memory. This way, it allows the system to perform physical replication of the data easily. Note that, in our implementation we use the simple memory layout for both guest and host ends. Nevertheless, one can also use more complicated layouts, which are depending on the specific application requirements.

4.4 Data Synchronization

In order to guarantee the data synchronization, the *GNU compiler collection* (GCC) procedure is modified to perform the analysis of the assembly source files and the *static binary translation*, before the assembler handles them. In our implementation, SC2M calculates the mirror memory addresses to determine where to replicate the data. Then, mirror instructions are injected into the source files, which are processed by the assembler as executable files. Note that, since assembling files (created by the GCC) have target addresses represented as labels, these instructions do not cause any problems for indirect `branches` or indirect `jumps`. In addition, once the executable files are generated, they shall have the ability to write data into both native and mirror memories. This way, when the programs run, the data is replicated instantly.

Nevertheless, we may need to note that, there are two types of mode codes, which need to be considered carefully. We next address them respectively.

▶ *User mode code.* In the user mode, some special instructions can change the execution sequence. So, in some cases they can be incorrectly mirrored even if the native instructions are correct; this incurs the inconsistent data. For example, the method `call()` saves procedure linking information on the stack, and branches to the called procedure; in this case, if the mirror instructions are inserted right after the native ones, they will not be executed until the called procedure returns. This creates a long-time data inconsistency between the native and mirror memories. To remedy this, we employ a `copy-on-call` method to replicate the linking information, by inserting mirror instructions at the very start of the called procedure. Since a procedure may be called several times but needs to be implemented only once, the `copy-on-call` method reduces the overall number of mirror instruction calls.

▶ *Kernel mode code.* Codes in the kernel mode are much more complicated than codes in the user-mode programs, although some instructions can be mirrored according to the user mode codes. The complexity is mainly because we need to take into account the atomic operations and privilege changes in the kernel mode. Generally, when an interrupt occurs, the kernel stops the normal processor loop, and starts to process the execution of a new sequence, called an *interrupt handler*. When an interrupt occurs, the processor first saves several registers such as *eip* and *esp*. This is mainly for restoring, when it returns from the interrupt.

4.5 Mirroring Data for High-Availability Applications

The SC2M can achieve fine-grained replication, e.g. data-level memory mirroring. This means that, SC2M can choose a certain address area for the application rather than using the entire VM memory. This can protect the important memory areas from more overhead. In our implementation, SC2M performs data-level replication by wrapping the memory area with `getter` and `setter` methods. Generally, SC2M provides four interfaces for data-level memory mirroring:

- *create*. The *create* operation generates a memory area for mirroring.
- *get*. The *get* operation fetches the content of the mirror memory.
- *set*. The *set* operation configures the mirror memory according to the requirements of the application.
- *destroy*. The *destroy* operation deletes all the mirror PTEs and releases the mirror space.

The steps of these operations can be briefly described as follows. First, developers use the *create* command to create a memory area and then use the *get* and the *set* to access this memory area. Later, they use the *destroy* command to invalidate the memory area. To reduce the complexity of the program, SC2M also provides an approach to mirror the user-defined format of the memory area, based on the user-defined data structure. In this case, users need to declare the structure, implement the *create*, *get*, *set* and *destroy* methods based on specific requirements. SC2M wraps the four methods with binary translation so that every

memory-write operation in the memory area is duplicated to the mirrored area. More importantly, SC2M can also mirror some structures and classes in certain libraries by translating libraries to a redundant version. Therefore, this technique can be used by developers to create high-availability STL *map* and *list* in their applications. In that case, the map and list are used to build parts of the infrastructure library.

5 Performance Evaluation

In this part, we first present the experimental settings (Section 5.1), and then present our main experimental results (Section 5.2), and finally discuss the limitations and summarize our findings (Section 5.3).

5.1 Experiment Setup

As mentioned earlier, the SC2M is developed for high availability. In our experiments, we use the gzip and gunzip software sets to conduct evaluation on the data-level high-availability. We test the *gzip* and the *gunzip* programs with eight different file sizes. They range from 100 MB to 800 MB. Following prior works [5, 11], we use the execution time to measure the performance. The execution time can easily represent the overhead. The execution time for these two applications refer to the compressing/decompressing time. Similar to prior works [5, 14, 11], we compare our proposed solution with the original (native) implementation. As for the former, it is a double write, while the latter a single write. In our experiments we did not extensively compare with existing solutions, since most of them are hardware-based solutions, which are hard to implement in our experimental platform. As for the software-based solutions, most of them are developed for the application level, or rely on disk. Our solution relies on memory and can provide high availability for data-, application-, and system-levels. Its superiorities are obvious, since (i) it is clearly faster than them (due to the difference between memory and disk), and (ii) it has more service domains, ranging from data-, application-, to system-level. Nevertheless, in Section 5.3 we shall give a discussion between our solution and the one most closest to ours.

We use XV6 [32] as the guest OS, and test the syscall `mmap()` function in the kernel space using or without using the SC2M, for evaluation on the system-level high-availability. To compare, we choose three write back frequencies for `mmap()`, they are 2048, 4096, and 8192; and we create an array of characters and a memory-mapped area to test the sequential read performance of our solution and the native system call command `mmap()` with different frequencies.

Our solution SC2M generates mirror instructions at the compile stage. It is necessary and also interesting to investigate the number of inserting mirror instructions, which can reflect the application-level high-availability, to some extent. To evaluate this performance, we use several applications like Memcached [1], Parsec Ferret [21] and SPECjbb [26] as testing benchmarks. Additionally, since our SC2M can support an extra interface that allows us to analyse the mirror instructions in detail, we also present the results related to this feature.

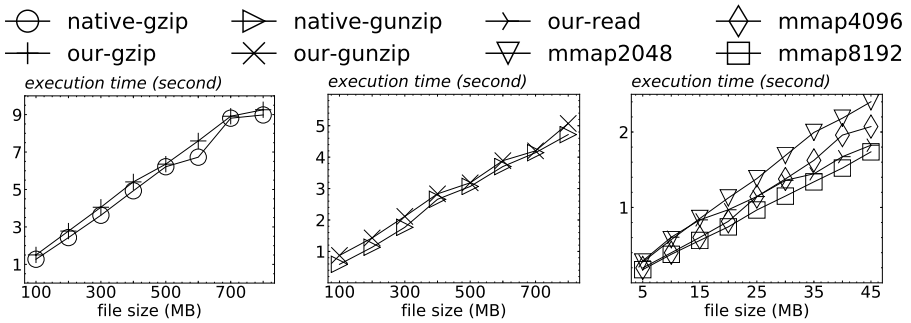

Fig. 5. gzip application **Fig. 6.** gunzip application **Fig. 7.** sequential read

Our goal in this work does not optimize the performance inside the native memory, we shall mainly test how much overhead shall be used for our solution. Ideally, if our overhead is less than two times of the overhead used by native, then our solution should be effective. Our tests are run on a DELL PowerEdgeT30 with a 16-core 3.7GHz Intel Xeon E3-1225v5 CPU with 8 GB DDR3 RAMs with ECC and a 1 TB SATA Disk. Our SC2M is implemented on an Xen-4.6. We use Linux (kernel version 4.3.31) as the host OS, and deploy a lightweight system, Busybox-1.19.2 [8]. Each VM is allocated two virtual CPUs and memories.

5.2 Experimental Results

Data-level. As we know, the gzip program does not occupy too many CPU resources while it is memory-intensive. We first test gzip application to see whether our solution is favourable.

Figure 5 shows the results when the gzip application is used. In this figure, the "our-gzip" means that the hypervisor has already been patched with SC2M, while the "native-gzip" means that the benchmark was run without any modification inside the hypervisor. That is, it does not use the memory mirror and thus no write backup operations are involved. From this figure, we can see that these two curves are almost coincident, although our solution employs the memory mirror that needs the write backup operations and the extra space for mirroring. This result essentially implies that our solution is slightly affected by the mirror space and also the write backup operations. In other words, for the gzip application, adding the mirror leads to only a minor performance degradation. This should be pretty positive and optimistic. The reader could be curious why the overhead of SC2M is too low, instead of two times of the native overhead. The main reasons are (i) SC2M uses redundant memory to backup native memory, avoiding additional I/O reads and writes; and (ii) most of mirror instructions generated by the static binary translation are explicit instructions, SC2M can read the destination address directly, which significantly speed up the process of mirror writes and reads.

Correspondingly, Figure 6 shows the results when the gunzip application is used. In this figure, the "our-gunzip" and "native-gunzip" have the similar meanings with those mentioned in the previous paragraph. We can also see that

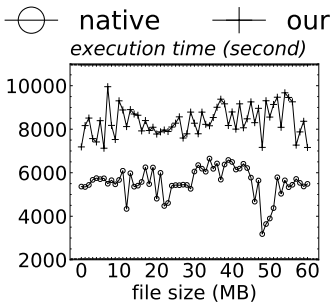

Fig. 8. Compile test

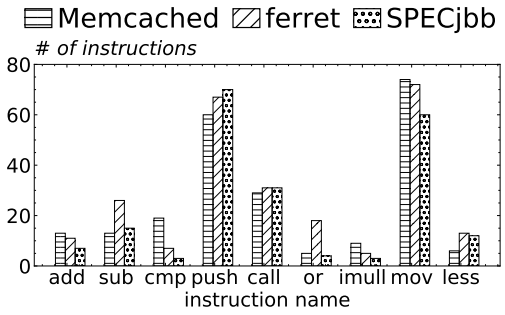

Fig. 9. Mirror instructions detail

the overhead of our solution is also close to that of the benchmark. This further demonstrates the feasibility of our solution. Moreover, one can see that when the file size is equal to 800 MB, our solution presents the worst performance, compared against the benchmark. Nevertheless, the gap is still very small. Specifically, the extra overhead incurred by our solution is less than 10% overhead in the *gunzip* benchmark. This minor extra overhead is fully acceptable and reasonable for most of real applications, since our solution (supporting the memory mirror) provides high-availability memory access.

System-level. As for evaluation on the system-level high-availability, we test the sequential read performance. The comparative results of our tests are plotted in Figure 7. In this figure, the "our-read" means our solution, while "mmap-2048", "mmap-4096" and "mmap-8192" refer to the native system call method *mmap()* with different frequencies. The results show that the execution time of our solution lies between mmap-2048 and mmap-8192. The performance is reasonable since our solution uses the memory mirror that can achieve high-availability. In contrast, for the benchmark, although we execute system calls periodically (every 2048, 4096 and 8192 bytes), the memory-mapped file could still not guarantee that all memory states are retained before the next write sequence. In this case, once a memory error occurs, the new data, which has not been written in the last call, could be lost. This reflects from another perspective that our solution is much more reliable and reasonable than the *mmap* alternative.

Application-level. This experiment is used to study the application-level high availability. Figure 8 shows the number of inserting instructions at the compile stage for the *Memcached* benchmark. In this figure, the "native" denotes the compile test using the original GCC, while "our" denotes the compile test using our solution, SC2M. It can be seen that, on average, our solution at compile stage only leads to about 20% increase, in terms of the number of compile instructions (notice: the famous Remus system leads to 70% increase, as stated in [5]). This essentially reflects that our solution should be favourable. As mentioned earlier, our SC2M can support an extra interface that allows us to analyse the mirror instructions in detail. Note that, since mirror instructions correspond to the native ones, they essentially also provide detailed information for the native

programs, to some extent. Figure 9 shows the detailed information of mirror instructions in Memcached, Ferret and SPECJbb, respectively. We can see that, the numbers of `mov` and `push` instructions alone are about 80% of the total number of instructions. In addition, it can be seen that, among all the memory write instructions, the majority of instructions are comprised of explicit write instructions (e.g., `mov`). The implicit write instructions are fewer (e.g., `imull, sub, or, les`) but usually are more complicated, which easily incur the extra overhead.

5.3 Discussion and Summary

Although all the experimental results show the feasibility and benefits of our solution, we would like to point out that, our solution may not be absolutely better than some existing (hardware-based or software-based) solutions. This is mainly because different solutions rely on different configurations, they may have their own advantages. A good example could be Remus [5]. It backs up the whole virtual machine in disk, has the full checkpoints and recovery mechanisms. In this regard, our solution does not always perform better than Remus. Yet, Remus generates 100% (double write) overhead using duo-backup, which is significantly larger than our overhead. Our solution uses directly the memory to backup memory data, it avoids additional disc I/O read and write. In this regard, our solution has its largest superiority than almost all the previous memory fault-tolerant solutions. Besides, our solution is also equipped with the following advantages: (i) it can retrieve the data quickly because it does not need any I/O operation from external devices; (ii) a hypervisor in the physical host does not need to launch new VMs for the backup of native VMs; and (iii) redundancy is narrowed down to one general machine or single server, so it does not involve networking resilience or migration maintenance; that is to say, large-scale deployment of our proposed system affects bandwidth utilization only minimally. Therefore, on the whole our solution is still competitive and attractive.

Summary. We find that (i) although our solution employs the memory mirror that needs the write backup operations for mirroring, it only leads to a minor performance degradation (e.g., executing gzip and gunzip applications), instead of two times of the native overhead. Specifically, in most of cases it curs about only 1.5% performance degradation. Such a performance is very positive, since our solution uses the memory mirror that can achieve high-availability for data level. (ii) Our solution can efficiently perform system-level operations (e.g., sequential read), and its performance is optimistic. Specifically, its performance is close to native system call method, yet it is much more reliable than native method. (iii) As for application-level (e.g., *Memcached*), our solution incurs a little more (about 20%) inserting instructions at the compile stage, and most of instructions are explicit write instructions (e.g., mov), which usually take less overhead, compared implicit instructions (e.g., imull).

6 Conclusion

In this paper, we proposed SC2M, which provides software-controlled memory mirroring based on hardware virtualization and static binary translation. SC2M

duplicates memory by implementing mirror instructions in both user and kernel modes. It is able to simultaneously support high-availability and native virtual machines. For each VM one can choose to use data-, application-, or system-level high availability. SC2M can also increase the memory copies on demand, and so is more flexibility than hardware-based approaches. We conducted experiments to validate the feasibility and superiorities of our solution.

Acknowledgement

This work was partially supported by the National Key R&D Program of China (No. 2018YFB1004400), the NSFC (No. U1811264, U1501252, U1611264, U1711261, U1711262, U1711263, 61972425, 61702320, 61867002, 61772334, 61872310), and the Key R&D Program of Guangdong Province (No. 2018B010107005, 2019B010120001), and the Opening Project of Guangdong Province Key Laboratory of Big Data Analysis and Processing (No. 201807).

References

1. ACME Laboratories. memcached-ahigh-performance, distributed memory object caching system. http://www.memcached.org/about, 2018.
2. S. Cha, H. Cho, J. H. Ahn, N. S. Kim, O. Seongil, H. Shin, S. Hwang, K. Park, S. J. Jang, J. S. Choi, et al. Defect analysis and cost-effective resilience architecture for future dram devices. In *HPCA*, pages 61–72, 2017.
3. C. Chen and M. Y. B. Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.
4. D. Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.
5. B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, pages 161–174, 2008.
6. J. Deegan and K. Gower. High reliability memory subsystem using data error correcting code symbol sliced command repowering, 2005. us Patent App. 10/723,055.
7. T. J. Dell. Ecc-on-simm test challenges. In *ITC*, pages 511–515, 1994.
8. Denys Vlasenko. BusyBox: The Swiss Army Knife of Embedded Linux. http://www.busybox.net/, 2013.
9. H. Dong, W. Sun, B. Wang, H. Sun, Z. Qi, H. Guan, and Y. Dong. Memvisor: Application level memory mirroring via binary translation. In *CLUSTER*, pages 562–565, 2012.
10. Y. Dong, Y. Chen, Z. Pan, J. Dai, and Y. Jiang. Renic: Architectural extension to sr-iov i/o virtualization for efficient replication. *ACM Trans. Archit. Code Optim.*, 8(4):40:1–40:22, 2012.
11. U. Ferraro-Petrillo, F. Grandoni, and G. F. Italiano. Data structures resilient to memory faults: An experimental study of dictionaries. *ACM Journal of Experimental Algorithmics*, 18:1–6, 2013.
12. D. Fiala, K. B. Ferreira, F. Mueller, and C. Engelmann. A tunable, software-based DRAM error detection and correction library for HPC. In *Euro-Par Workshops*, pages 251–261, 2011.
13. HP Corporation. HP advanced memory protection technologies. http://h18000.www1.hp.com/products/servers/technology/memoryprotection.html, 2013.

14. J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng. Rigorous rental memory management for embedded systems. *ACM Trans. Embed. Comput. Syst.*, 12(1s):43:1–43:21, 2013.
15. S. Khan, D. Paul, P. Momtahan, and M. Aloqaily. Artificial intelligence framework for smart city microgrids: State of the art, challenges, and opportunities. In *FMEC*, pages 283–288, 2018.
16. L. Levine and W. Myers. Special feature: Semiconductor memory reliability with error detecting and correcting codes. *IEEE Computer*, 9(10):43–50, 1976.
17. J. Li, M. Zhao, L. Ju, C. J. Xue, and Z. Jia. Maximizing forward progress with cache-aware backup for self-powered non-volatile processors. In *DAC*, pages 1–6, 2017.
18. H. Liu, C.-Z. Xu, H. Jin, J. Gong, and X. Liao. Performance and energy modeling for live migration of virtual machines. In *HPDC*, pages 171–182, 2011.
19. A. Malek, E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis. Odd-ecc: On-demand dram error correcting codes. In *MEMSYS*, pages 96–111, 2017.
20. G. Mappouras, A. Vahid, R. Calderbank, D. R. Hower, and D. J. Sorin. Jenga: Efficient fault tolerance for stacked dram. In *ICCD*, pages 361–368, 2017.
21. Parsec. Parsec - a unit of measure. http://parsec.cs.princeton.edu/, 2019.
22. Z. Qi, H. Dong, W. Sun, Y. Dong, and H. Guan. Multi-granularity memory mirroring via binary translation in cloud environments. *IEEE Trans. Network and Service Management*, 11(1):36–45, 2014.
23. Redis. redis - an open source, BSD licensed, advanced key-value cache and store. http://www.redis.io/, 2014.
24. G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *CGO*, pages 243–254, 2005.
25. G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. *SIGARCH Comput. Archit. News*, 33(2):148–159, 2005.
26. SPEC. SPEC benchmark. https://www.spec.org/benchmarks.html, 2018.
27. V. Sridharan and D. Liberty. A study of dram failures in the field. In *SC*, pages 1–11, 2012.
28. S. Stuart, L. G. H., S. Karin, and B. Doug. Use ecp, not ecc, for hard failures in resistive memories. *SIGARCH Comput. Archit. News*, 38(3):1–12, 2010.
29. X. Tang, J. Zhai, B. Yu, W. Chen, W. Zheng, and K. Li. An efficient in-memory checkpoint method and its practice on fault-tolerant hpl. *IEEE Trans. Parallel Distrib. Syst.*, 29(4):758–771, 2018.
30. B. Wang, Z. Qi, H. Guan, H. Dong, W. Sun, and Y. Dong. kmemvisor: Flexible system wide memory mirroring in virtual environments. In *HPDC*, pages 251–262, 2013.
31. B. Wang, Z. Qi, R. Ma, H. Guan, and A. V. Vasilakos. A survey on data center networking for cloud computing. *Computer Networks*, 91:528–547, 2015.
32. XV6. XV6 Doc. http://pdos.csail.mit.edu/6.828/2011/xv6.html, 2011.
33. K. Ye, Y. Liu, G. Xu, and C.-Z. Xu. Fault injection and detection for artificial intelligence applications in container-based clouds. In *CludCom*, pages 112–127, 2018.
34. M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.
35. H. Zheng, Z. Zhu, X. Dong, B. Chen, and C. Liu. Studying shadow page cache to improve isolated drivers' performance. *Concurrency and Computation: Practice and Experience*, 29(10), 2017.