**QUEEN'S
UNIVERSITY
BELFAST**

Userspace Hypervisor Data Characterization in Virtualized Environment

Wang, B., Vandierendonck, H., & Karakonstantis, G. (2019). Userspace Hypervisor Data Characterization in Virtualized Environment. In *The 24th IEEE International Conference on Parallel and Distributed Systems* (24 ed., pp. 638-645). Institute of Electrical and Electronics Engineers Inc..
<https://doi.org/10.1109/PADSW.2018.8644612>

Published in:

The 24th IEEE International Conference on Parallel and Distributed Systems

Document Version:

Peer reviewed version

Queen's University Belfast - Research Portal:

[Link to publication record in Queen's University Belfast Research Portal](#)

Publisher rights

Copyright 2018 IEEE. This work is made available online in accordance with the publisher's policies. Please refer to any applicable terms of use of the publisher.

General rights

Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy

The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Userspace Hypervisor Data Characterization in Virtualized Environment

Bin Wang, Hans Vandierendonck and Georgios Karakonstantis

Centre for Data Science and Scalable Computing

Queen's University Belfast, Belfast, United Kingdom

{B.Wang, h.vandierendonck, G.Karakonstantis}@qub.ac.uk

Abstract—Memory-intensive applications have grown rapidly to adapt to the changing needs of businesses. Memory errors become even more concerning, since configuration at extended operating points makes hardware more susceptible to system failures. Some data structures may be more sensitive to errors and may cause system crashes more easily than others. A failure in a critical data structure can cause a complete system crash. In this paper, we provide a sensitivity characterization of hypervisor structures. First, we implement an error-injection framework using *ptrace* syscall. We profile both static and dynamic data structures through our framework. Then we provide the detailed analysis and provide data characterization to QEMU hypervisor. Finally, we discuss the full checkpointing and selective mechanism in case the QEMU hypervisor crash.

Index Terms—Reliability, data characterization, virtualization, error injection.

I. INTRODUCTION

In-memory computing technology has grown rapidly, to adapt to changing needs of businesses. A growing number of in-memory applications have gained popularity in recent years, for example, the database Redis [1], the key-value store Memcached [2], the computing framework Spark [3], and others [4]. These applications and their quality of services will generate a large number of mission-critical workloads that frequently access memory or cache [5], [6], [7]. As a result, requirements on the correctness and stability of memory are stricter in these types of applications. Consequently, demand for high-availability memory access has increased considerably.

On the other hand, typical memory components gradually become error-prone under increasing density and capacity [8], and thus fail to guarantee high availability (HA) for these applications. Moreover, uncorrected errors from DRAM, i.e. those detected by hardware and cannot be corrected, are a significant cause of system downtime [9]. For example, Google's research showed that memory (DRAM) errors are common in modern computing clusters, and that more than 8% of DIMMs are affected by errors per year [10]. This rate is unacceptable for today's commercial in-memory computing service providers, such as Google App Engine [11] or Amazon EC2 [12], because they require availability with an annual uptime percentage of at least 99.95%. In the cloud era, with ever increasing deployments of virtualization technology, one server's memory is sometimes shared between thousands of virtual machine (VM) instances. If the executing instances

access the damaged memory block, data corruption and system crashes will inevitably occur in the cloud services [13].

Finally, introducing more reliability into a system, at any level, invariably reduces performance. This means that a practical design has to strike a balance between these two attributes in the design. When we inject errors to sensitive structures, QEMU [14] and KVM [15] are significantly more prone to crashes. We conducted a set of experiments in which we performed static data characterization of QEMU hypervisor, in order to find an appropriate balance for the modern cluster.

In this paper, we propose the error-injection approach which could emulate the crash of QEMU hypervisor, and provide data characterization of different sensitive structures inside QEMU hypervisor. Our experiments considered the reliability of memories, determined which software components can safely remain in the relaxed side of the system and how dynamic adaptation of the system configuration can help us limit performance degradation to a minimum. This kind of information is very important to the design decisions and the overall approach of the high availability system. For example, it can be used as an input to the predictor module and the checkpoint mechanism. In details, this paper has the following architecture:

- We use Syscall *ptrace* [16] to find an address of a sensitive data structure in the process memory, bit-flip a random bit to that address, and finally observe the impact to the running state of QEMU hypervisor.
- We read the symbol table from *.elf* file using *objdump*. We run a benchmark VM (e.g., Specjbb benchmark [17]), use *pmap pid* or */proc/pid/maps* to find the virtual address of the static area.
- We use both manual and automatic dumps to find the address of the dynamic data structure from the heap area. The manual dump is implemented with the pointer while the automatic dump is implemented with Syscalls *malloc hook* and *backtrace*.
- We provide good data characterization in our experiment result. Using our error injection approach, we can rank the most sensitive structures during the running period of the QEMU hypervisor and the benchmark VM.
- Using Libvirt commands, we implement a various checkpoint and judge mechanism so that we can determine when we setup and insert a checkpoint and which size should a checkpoint duplicate.

- We discuss the approach of selective checkpointing mechanism for memory backup.

The rest of the paper is organized as follows. The overall framework of error injection is introduced in Section III. This section also provides details regarding its implementation, and Section IV provide data characterization about QEMU hypervisor. Section V discusses full and selective checkpointing mechanisms using Libvirt. Section VI covers related work, and Section VII concludes the paper.

II. BACKGROUND

This section demonstrates the background of the used techniques in this paper.

A. DRAM Architecture for System Level Data

1) *DRAM architecture*: Dynamic random-access memory (DRAM) [18] is a type of random access semiconductor memory that stores each bit of data in a separate tiny capacitor within an integrated circuit. DRAM is usually arranged in a rectangular array of charge storage cells consisting of one capacitor and transistor per data bit. In morden DRAM chips, it has various memory control units (MCUs) [18] which manage the allocation from the system or software layers. An MCU lies in the DRAM bank which could be regarded as the basic control unit.

2) *Address Management*: Modern operating system use physical address and virtual address to store the data inside the memory. A physical address is the actual location in the hardware memory. A virtual address is determined by OS kernel and process. They have the mapping relationship with segment and page table. A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. A translation lookaside buffer (TLB) [18] is a memory cache that is used to reduce the time taken to access a user memory location. It is a part of the chips memory-management unit (MMU) [18]. The TLB stores the recent translations of virtual memory to physical memory and can be called an address-translation cache

B. Virtual Address Space

A virtual address space is the set of ranges of virtual addresses that an operating system makes available to a process. In Linux system, the layout is handled by *mm_struct* [19]. To any running process, the operating system allocates the memory space so that it can maintain the normal events for the application. Shown in Figure 1, the process memory area could be divided to library space, static global space and heap area. The library space stores the dynamic libraries such as *libso*, *libnss*, *libpthread* [19]. The storage of dynamic library do not rely on the running state of any process. On the other hand, a process will build a redirected mapping to that process area. So different processes have different virtual addresses of dynamic library, but the physical address of dynamic library to the different processes are the same.

Fig. 1. The layout of a memory space to a process

The static global data stores the data defined in the application including initialized and uninitialized symbols, functions, data structures. All the offsets of static data have already confirmed during the compile time. And if a process for that application is running, the virtual address of any static data is confirmed by the global base address + offset. During the whole running period of that process, the virtual address of any static data structure is not changed. The dynamic data is stored in the heap area. The heap is memory set aside for dynamic allocation. There is no enforced pattern to the allocation and deallocation of blocks from the heap. Operating system can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time; there are many custom heap allocators available to tune heap performance for different usage patterns. To any dynamic data, they are defined by the application as non-static and non-global in application source code. And during the running time, they will be executed by *malloc*/*kalloc*.

C. System Level Approach to Detection Virtual Address

1) *Ptrace*: *ptrace* [20] is used by debuggers and specialized programs to patch running programs, to avoid unfixed bugs or to overcome security features. By attaching to another process using the *ptrace* call, a tool has extensive control over the operation of its target. This includes manipulation of its file descriptors, memory, and registers. Based on the *ptrace* functionality, we can use it to profile the data to a given virtual address in another process. For example, *ptrace* provides the *PEAK_DATA* to read a data from an address and *POKE_DATA* to revise that data. Hence, *ptrace* is an easy Syscall to build up an error injection framework in virtual address space.

2) *Objdump*: *objdump* [19] displays information about one or more object files. The options control what particular information to display. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work. In this paper, we use *objdump* to print all the offsets of data structures inside QEMU hypervisor. After that, we run the hypervisor and acquire the base address

in a process space. Finally, all the virtual addresses of static structures could be derived.

3) *Pmap*: The *pmap* [19] command reports the memory map of a process or processes. In the Linux system, we can also use */proc/pid/maps* to print the process virtual space. The space shown from *pmap* means the operating system has already been allocated from physical memory, and has the mapping from the MMU. In this paper, we use *pmap* to get the base address for static structures, and the heap address for dynamic structures.

III. ERROR INJECTION ARCHITECTURE

QEMU is a userspace hypervisor which can create the execution of KVM for various usecase. QEMU can also emulate different instruction sets and machine architectures and support more than 300 kinds of kernels. So QEMU data structures could be chosen as the typical observed target and we can first read the data structures from QEMU hypervisor and error inject the sensitive data structures inside QEMU running process. In this section, we will demonstrate our error injection framework and the profiling approach to QEMU sensitive data structures.

A. Error Injection Framework

In our error injection scenario, our plan is to find an address of a sensitive data structure in the process memory, bit-flip a random bit to that address, and finally observe the impact to the running state of QEMU hypervisor. Normally, the memory address includes virtual address and physical address. And they have the mapping relationship (page table entry) and can be looked up in memory management unit (MMU). Hence, we can bit-flip either virtual address or physical address of the target data structure in a process memory space.

In term of error injection to the physical address, we can manually configure the underlying cell or bank inside DRAM. For example, you can revise the cell voltage so that some bits could be disabled. However, this approach requires the high frequent interaction between memory controller and software layer. Even we can find the approach to couple these elements, we cannot address the interleaving functionality of memory controller. An interleaving of memory controller enables the memory store the data from different processes. An incorrect revision to a cell will undermine the execution of another process. Therefore, we can also use software/system approach to the physical address. Modern Linux kernel allocates memory through */dev/mem* module. We can dump the footprint of */dev/mem* and find the physical address of the target data structure. Then we can use page table entry to get its virtual address and finally read the process space to get the target data.

Another approach is to use Syscall *ptrace* to operate the virtual address in the userspace memory. Syscall *ptrace* provides the functionality that a process can profile and even update the execution of another process. That includes the examination and revision of memory and registers from the target process. Compare with the footprint of */dev/mem*, *ptrace*

approach require no kernel modification and recompiling but it performs the same functionality with */dev/mem/*. In this paper, we adopt *ptrace* to build up our error injection framework.

B. Ptrace Module

Figure 2 shows the overall error injection framework using the Syscall *ptrace*. We will introduce the modules through bottom-up sequence. The substrate module is implemented by *ptrace*. We configure the parameter with *PEAK_DATA* and *POKE_DATA*. *PEAK_DATA* can enable *ptrace* dump the data of the target virtual address while *POKE_DATA* can write the new value to the target address. The workflow of this module is as follows:

- **Step 1:** When we confirm a virtual address of a data structure, we configure *PEAK_DATA* to get the data of that virtual address.
- **Step 2:** We choose a random bit and flip it using XOR 1.
- **Step 3:** We write the new data to the virtual address and then profile the execution of the hypervisor.

With *ptrace*, we can simply revise data of any virtual address in a process memory space. In the next step, we will think about how to find the virtual address of any data structure. We can classify the data structures of hypervisor by the static and dynamic. We will use the following two subsections to provide the details.

C. Profiling Static Data Structure

In a process memory space, the addresses from all static data structures are generated during the compiling period. After the generation of an executive file (e.g., .elf file), we can use Linux command *objdump* or *readelf* to browse the symbol table of all static data structures. In this symbol table, we can lookup the offset of any data structure. The offset is never changed so long as the source code is not recompiled. Actual, an offset in the symbol table is not the real virtual address because operating system does not always allocate memory through the 0 base address. To this point, the virtual address of a given static data structure is the basic address plus offset. Therefore, we should find the base address of the running hypervisor. This reply on the read of hypervisor process area. On the other hand, the virtual address of any static data structure is not changed during the execution period because both base address and offset are fixed in a process.

Shown in Figure 2, we can simply read any process area through its pid with *pmap* command or */proc/pid/maps*. Hence the address and corresponding data could also be read through QEMU process area. We show the workflow of error-injection to the static data structure as follows:

- **Step 1:** Compile the QEMU hypervisor. After generating the .elf file, use *objdump* (or *readelf*) to get the symbol table of all static data structure.
- **Step 2:** Find the offset of a given data structure through symbol table.

Fig. 2. The error injection framework to static data structure using *ptrace*

- **Step 3:** Run a benchmark VM (e.g., Specjbb benchmark), use *pmap pid* or */proc/pid/maps* to find the base address of the static area.
- **Step 4:** The virtual address of a given static data structure is its base address + offset.
- **Step 5:** Use *ptrace PEAK_DATA* flag to read the address and *POCK_DATA* to revise the data. Then profile the variation of the hypervisor execution.

D. Profiling Dynamic Data Structure

The dynamic data structure is defined inside a function without static and global property. Its address is confirmed during the running time. It is usually generated through the calling of *malloc/kalloc* etc. And during the running period, the address lies in the heap area of a process space. Finding the virtual address of dynamic data structure are more complicated than that of static structure. There are two challenges to capture the virtual address of a given dynamic data structure:

- **Challenge 1:** A virtual address of the dynamic data structure only alive with the calling or the execution situation of the function. For example, when hypervisor call a function which invokes a given dynamic data structure, the operating system will allocate its memory space for that function in the heap area. However, after finishing the calling, process will release its function space in the heap. That cause the problem that a dynamic data structure tends to various virtual addresses. We can only capture its instant virtual address of the dynamic structure. Even we hope to error inject it, we will bit-flip the data immediately after finding the virtual address.
- **Challenge 2:** A dynamic data structure associates with various functions (unlike static structure). For example, the execution of the function relies on the result from a dynamic data structure inside the other function. Because

different functions have the interleaving relationship. That increases the difficulty of data characterization when we observe the error injection to the dynamic data structure.

Motivated by the two challenges, In this paper, we use two approaches to read and error-inject data from a heap address: manual dump and automatic dump.

Fig. 3. The dynamic error-injection using manual dump

1) *Manual Dump:* In the manual dump approach, we patch every sensitive data structure with a pointer before the compile and then all the values of pointers are the virtual address. In

the next step, we can error-inject this address use `ptrace`. The steps could be shown in Figure 3.

- **Step 1:** Range the dynamic data structures with frequency. Linux GDB [21] or Vagrind [22] can provide such functionality.
- **Step 2:** To a given dynamic data structure, we can use tools to find all the callers and their initialized function.
- **Step 3:** We patch all the callers with pointer so that when QEMU hypervisor runs, it can print the pointer value (the virtual address). This could also be recorded in QEMU log file.
- **Step 4:** Use `ptrace POCK_DATA` to revise the data immediately when we get the address. Then profile the variation of the hypervisor execution.

This approach is the most direct way to find heap area address. However, it has large drawback. First, it requires large source code modification. Second, we can only find dynamic data structure through our read. The code read work is huge. The reading also includes 1) finding the complex relationship between interleaving data structures; 2) finding all the variation from a data structure.

2) *Automatic Dump:* In the automatic dump approach, we can create a dynamic link library to intercept all the malloc operation. For example, we can use malloc hook to do that. Figure 4 shows the dynamic error-injection using automatic dump to the sensitive data structure. The malloc call tool can record all the malloc /realloc addresses in the heap. The tool can also record which function call the malloc in the heap. Hence, we provide the workflow as follows:

- **Step 1:** Revise the QEMU (or other userspace hypervisor) makefile using `rdynamic` configuration. `rdynamic` can enable the linker to show all symbols so that we can get the address information.
- **Step 2:** Create a dynamic library using malloc hook. The malloc call tool can trap all the malloc/realloc addresses in the heap.
- **Step 3:** Print the caller use backtrace Syscall, so that when malloc hook traps a malloc from a dynamic data structure, we can find its caller.
- **Step 4:** Run the QEMU hypervisor using `LD_PRELOAD = XX.so` so that we can enable the QEMU process dynamically load our malloc hook library. When QEMU runs, the standard output will show all the virtual address of malloc function and its caller in a process.
- **Step 5:** Use `ptrace POCK_DATA` to revise the data immediately when we get the address. Then profile the variation of the hypervisor execution.

IV. RESULTS AND HYPERVISOR DATA CHARACTERIZATION

In this section, we propose the evaluation of our data characterization framework.

A. Used Benchmark and Testing Tools

We mainly targeted the ARM XGene2 architecture [23] but we used also the Intel architecture. The Libvirt is used as the

Fig. 4. The dynamic error-injection using automatic dump

monitor APIs. We executed three typical Cloud benchmarks on the VMs: Parsec [24], Memcached [2], Specjbb [17]. We inject all errors when running the benchmarks and we trace only QEMU crashes. In our experiments, we discovered that some kinds of errors trigger QEMU crash and thus we record all such errors. We choose 80 typical modules which affect 3,000 QEMU functions. In this report, we list most sensitive modules and functions. For each trace, we run our simulation 50 times (1 minute per time)

B. Static Data Characterization

Figure 5 shows the failures of modules in case the Parsec, Memcached, and Specjbb benchmarks are used as a workload on top of QEMU. In this experiment, we found that the memory section has many sensitive modules. For example, the page module incurs 60 system failures, while the memory module incurs 16 failures. In Memcached experiment, we use CloudSuite to emulate Memcached client and server. Memory modules such as mem, page, they incur higher failures than that in Parsec benchmark. For experiments with Specjbb benchmark, we create 2 JVMs inside the guest virtual machine and setup the warehouse threshold as 1 minute. We found that rcu failures are higher than the previous two benchmarks. That means Specjbb JVM needs more read-copy update for communication.

C. Dynamic Data Characterization

In this subsection, we use manual dump approach to measure the dynamic data characterization. We create three VMs which run Parsec, Specjbb, and Memcached benchmark separately. We choose 100 typical dynamic structures which cover 22 modules of QEMU hypervisor. The time threshold is

Fig. 5. Failures of QEMU modules in different benchmarks to static structure

Fig. 6. Failures of QEMU modules in different benchmarks to dynamic structure

Fig. 7. Frequency of dynamic data structures under Parsec, Memcached and Specjbb benchmarks

set up with 1 minute. We run and error-inject these benchmarks 100 times, and then count the number of crash. We can get

the following results.

Figure 6 shows the failures of modules in case the Parsec, Memcached and Specjbb benchmarks to these modules who contain the dynamic structures. Similar to the case of static data structure, *tb*, *tcg* and *page* module have various crashes when we error injection the corresponding data structures. That is because QEMU setup more exception handler to handle the error of those structures. On the other hand, ferret similar search does not take too much RCU computation. And *tlb* module shows the error because it manages the PTE lookup for virtual address translation. The wrong result will cause the exception in memory module of QEMU. On the other hand, Memcached benchmark requires various memory translation through guest virtual address to the host virtual address from QEMU. The memory management module in QEMU maintains the relationship between guest physical memory (GPM) and guest virtual memory (GVM) for each VM and intercepts all PTE operations. So we find that the structures in terms of *page*, *tcg*, *mem* becomes higher than those in Parsec situation. In Specjbb benchmark, because the 2 JVMs inside the guest VM periodically communicate with each other. It requires more RCU computation than Parsec and Memcached. On the other hand,

D. Frequency of Calling Modules

Actually, we need to measure the frequency of used data structures of QEMU hypervisor. The reason is because if a data structure is never accessed through the running period but it cannot be regarded as a non-sensitive as if we error-inject this structure, it still has the possible to crash the whole hypervisor. Thus, we choose the typical data structures from the above modules and profile them in 1 minutes. Figure 7 shows the results. We found that *AioContext* appears most times and the next is *kvm_pic*. *AioContext* issues the QEMU multi-thread mechanism while *kvm_pic* provides the segment for QEMU interrupt. That means during a running time of any given benchmark (Parsec, Memcached and Specjbb in this paper), the hypervisor has plenty of events for multi-thread and interrupt. On the other hand, we found that many memory allocation events also show high frequency such as structures *PhysPageEntry* and *kvm_userspace_memory_region*. The CPU events show lower frequent than the memory events.

V. BACKUP AND CHECKPOINTING SOLUTION

Checkpoint mechanism keeps the state of disk, CPU, memory of a VM and even hypervisor (QEMU) itself in a short time. We expect to use checkpoint to keep the sensitive structures of a hypervisor and VM so that whenever the native hypervisor or VM occurs crash. The backup file can quickly be migrated to replace the native one so that the working state of the hypervisor has slightly been affected by the underlying DRAM and CPU errors. The other approach is copy-on-write mechanism, where the native memory, image, CPU register has a write operation, the backup memory, image, CPU register will do the same operation as well. We prefer the checkpoint solution.

We found that the Libvirt has already embedded the checkpointing APIs and we can simply use these to implement full checkpoint and backup. In this section, we will discuss the checkpoint and backup implementation.

A. Full Checkpoint Duplication

Using Libvirt commands, we implement a various checkpoint and judge mechanism so that we can determine when we setup and insert a checkpoint and which size should a checkpoint duplicate.

Fig. 8. The architecture of full hypervisor backup

Figure 8 gives the whole checkpoint architecture in this paper. We use QEMU as the userspace hypervisor to manage the events from VM. A VM instance will link to its image and then we can backup its image using *qemu-monitor-command*. On the other hand, the hypervisor itself is regarded as the application files which could also be backedup with Libvirt. In this figure, through the QEMU hypervisor command *qemu-monitor-command*, we can acquire both VM image information and hypervisor file information.

B. Discussion of Selective Backup

In our current implementation, full hypervisor backup still needs large optimization. First of all, in the cloud network, some application is becoming even larger, when something is slightly updated in the VM image, we cannot backup the whole image on the fly due to the size restriction. Second, during the running time, we produce each checkpoint which could have high possibility to get an inconsistent checkpoint. That is because we raise a live backup, and during the backup period, the program is still running and needs sometime to walk through the whole image.

C. Performance Evaluation about Checkpoint

In this subsection, we present the performance evaluation about checkpointing and backup mechanism.

Fig. 9. Syscalls *read* and *write* from ferret benchmark

Fig. 10. CPU throughput between ferret and specjbb benchmarks

In Figure 9, we use Parsec ferret similar search to record the times of read and write between non-checkpointing and checkpointing. We setup every 15 seconds our checkpointing mechanism raise a checkpoint. The caption with the backup suffix means we add the checkpoint and backup in our experiment. The caption with the normal suffix means we measure the read and write Syscall normally. Actually, duplication will generate the double write overhead. However, we found that read and write are slightly affected by the overhead. We propose the reason that Linux will create another process for our checkpoint architecture. The checkpointing itself does not need too much read and write (syscall).

In Figure 10, we use Ferret and Specjbb benchmarks to record the CPU throughput between non-checkpointing and checkpointing. We setup every 15 seconds our checkpointing mechanism raise a checkpoint. The caption with the backup suffix means we add the checkpoint and backup in our experiment. The caption with the normal suffix means we run the ferret or specjbb benchmarks normally. We use perf metrics to measure the CPU throughput. We found that a checkpoint sometimes generate large overhead during each backup. That is because of disk check period before each snapshot sometimes takes a long time to finish. And the duplication creates another process to finish. Finally, the process will consume the

CPU throughput. In the future, we should optimize the CPU throughput in our checkpointing mechanism.

VI. RELATED WORK

This section describes the related work about error injection and checkpoint.

A. Error Injection

Generally, existing error injection solutions for data characterization can be divided into two categories: hardware and software. Hardware solutions enhance availability with special motherboards and/or memory chips. Software solutions are mostly based on the emulation and system level error injection. In hardware error injection, the system SASSIFI [25] provides efficiency by allowing instrumentation code to execute entirely on the GPU. Rizwan A. Ashraf [26] et al. propose a fault propagation framework to analyze how faults propagate in MPI applications and to understand their vulnerability to faults. Software level error injection is more general in this area because it relies on system-level support which can be deployed easier. In this field, a tool called MEI [27] was implemented in physical address space for memory errors injection and manipulation. Guanpeng Li et al. [28] observed the GPUCPU interaction boundary naturally limits error propagation in GPU applications compared to traditional non-GPU applications.

B. Checkpoint Solution for Hypervisor

Initially, hardware providers typically considered using extra bits to check and correct memory errors. Parity [29] checking is a common technology, which uses one extra bit to check data. Another solution is ECC [30], which uses Hamming Codes [31] to detect and correct internal data corruption. Certain hardware vendors promote ECC to support their motherboard services, with HP Advanced ECC [8], Google ECC [32], and IBM Chipkill [33] among them. Another bit-checking method for large area failures is ECP [34], which corrects a failed bit in a memory line by recording the position of the bit in the line and its correct value [35]. Bit-based checking can only deal with limited bit errors rather than massive block failures, however, and more error check bits are required along with increases of native data bits.

Memory-based HA is also considered a part of the entire system's HA. Redundancy has been applied at different levels of granularity, such as the hardware, thread, and instruction levels [36]. SWIFT, a software-only fault-detection technique, duplicates a program's instructions, inserting explicit validation codes to compare the results of original instructions and their corresponding duplicates [37]. CRAFT [38] later improved SWIFT's approach by adding extra hardware structures. In order to direct the level of high reliability, PROFIT [39] adjusts the level of protection and performance at fine granularities based on SWIFT. However, SWIFT incurs unwanted performance overhead as the number of instructions can be easily doubled, mainly due to the full duplication of instructions [36].

VII. CONCLUSION

In this paper, we propose an error-injection framework which can provide software-controlled error-injection and data characterization to the hypervisor structure. Basically, we used Syscall *ptrace* to implement an error injection framework to both static and dynamic data structures. We provided good data characterization in our experiment result. We ranked the most sensitive structures during the running period of the QEMU hypervisor and the benchmark VM. Using Libvirt commands, we implemented a various checkpoint and judge mechanism so that we can determine when we setup and insert a checkpoint and which size should a checkpoint duplicate.

In the future, we will provide more results in terms of both static and dynamic data structures. And we will try to use fork to create the backup process which duplicate the sensitive data from the hypervisor so that we can finally implement the selective checkpointing mechanism. At that time, we will also design the experiment to verify the performance of the selective checkpointing.

REFERENCES

- [1] Redis, "redis - an open source, BSD licensed, advanced key-value cache and store." <http://www.redis.io/>, 2014.
- [2] Memcached, "memcached - a high-performance, distributed memory object caching system," <http://www.memcached.org/about>, 2018.
- [3] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 423–438.
- [4] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based dram controller for mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*. IEEE, 2015, pp. 317–326.
- [5] T. R. Kumar, R. Govindarajan, and C. Ravikumar, "On-chip memory architecture exploration framework for dsp processor-based embedded system on chip," *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 1, Apr. 2012.
- [6] K. Bai and A. Shrivastava, "Heap data management for limited local memory (llm) multi-core processors," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2010 IEEE/ACM/IFIP International Conference on*. IEEE, 2010, pp. 317–325.
- [7] P. R. Panda, S. Roy, S. Chandrasekaran, N. Sharma, J. Kaur, S. K. Kandam, et al., "High level energy modeling of controller logic in data caches," in *Proceedings of the 24th edition of the great lakes symposium on VLSI*. ACM, 2014, pp. 45–50.
- [8] HP Corporation, "HP advanced memory protection technologies," <http://h18000.www1.hp.com/products/servers/technology/memoryprotection.html>, 2013.
- [9] S. Vilas and L. Dean, "A study of dram failures in the field," in *SC*, 2012, pp. 76:1–76:11.
- [10] 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.
- [11] Google, "App Engine Service Level Agreement," <https://developers.google.com/appengine/sla>, 2013.
- [12] Amazon, "Amazon EC2 Service Level Agreement," <http://aws.amazon.com/ec2-sla/>, 2013.
- [13] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian, "Mitigating soft error failures for multimedia applications by selective data protection," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2006, pp. 411–420.
- [14] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.
- [15] I. Habib, "Virtualization with kvm," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.
- [16] P. Padala, "Playing with ptrace, part i," *Linux Journal*, vol. 103, no. 5, 2002.
- [17] SPEC, "SPEC benchmark," <https://www.spec.org/benchmarks.html>, 2018.
- [18] B. Keeth and R. J. Baker, *DRAM Circuit Design: A Tutorial*. Wiley-IEEE Press, 2000.
- [19] P. Raghavan, A. Lad, and S. Neelakandan, *Embedded Linux system design and development*. Auerbach Publications, 2005.
- [20] P. Padala, "Playing with ptrace, part i," *Linux Journal*, vol. 2002, no. 103, p. 5, 2002.
- [21] R. M. Stallman and R. H. Pesch, *Debugging with Gdb: The Gnu Source-level Debugger Fifth Edition, for Gdb Version, April 1998*. iUniverse Com, 2000.
- [22] Valgrind Developer, "Valgrind tool," <http://valgrind.org/>, 2018.
- [23] APM, "X-Gene: Worlds first ARMv8 64-bit server on a chip solution," <https://www.apm.com/products/data-center/x-gene-family/x-gene/>.
- [24] Parsec, "Parsec - a unit of measure," <http://parsec.cs.princeton.edu/>, 2018.
- [25] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 249–258.
- [26] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. ACM, 2015, pp. 72:1–72:12.
- [27] X. Wang, X. Wang, F. Zhu, Q. Zhou, and R. Zhou, "Mei: A light weight memory error injection tool for validating online memory testers," in *2016 International Symposium on System and Software Reliability (ISSSR)*, Oct 2016, pp. 129–136.
- [28] G. Li, K. Pattabiraman, C. Y. Cher, and P. Bose, "Understanding error propagation in gpgpu applications," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 240–251.
- [29] Chen, C. L., "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," in *IBM Journal of Research and Development*, 1984.
- [30] L. L. and M. W., "Special feature: Semiconductor memory reliability with error detecting and correcting codes," in *Computer*, 1976, pp. 43–50.
- [31] RW Hamming, "Error detecting and error correcting codes," *Bell System technical journal*, 1950.
- [32] J. Deegan and K. Gower, "High reliability memory subsystem using data error correcting code symbol sliced command repowering," May 26 2005, uS Patent App. 10/723,055. [Online]. Available: <https://www.google.com/patents/US20050114750>
- [33] Timothy J. Dell, "Ecc-on-simm test challenges," in *ITC*, 1994, pp. 511–515.
- [34] S. Stuart, L. G. H., S. Karin, and B. Doug, "Use ecp, not ecc, for hard failures in resistive memories," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, Jun. 2010.
- [35] Qureshi Moinuddin K., "Pay-as-you-go: low-overhead hard-error correction for phase change memories," in *MICRO-44*, 2011, pp. 318–328.
- [36] J. Lee, Y. Ko, K. Lee, J. M. Youn, and Y. Paek, "Dynamic code duplication with vulnerability awareness for soft error detection on vliw architectures," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 48:1–48:24, Jan. 2013.
- [37] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '05, 2005, pp. 243–254.
- [38] 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*, vol. 33, no. 2, pp. 148–159, May 2005.
- [39] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Software-controlled fault tolerance," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 4, pp. 366–396, Dec. 2005.