License

MH-QEMU: Memory-State-Aware Fault Injection Platform *

Hideyuki Jitsumoto[1], Yuya Kobayashi[2], Akihiro Nomura[1], and Satoshi Matsuoka[1,3]

[1] Tokyo Institute of Technology, Tokyo, JAPAN
jitumoto@gsic.titech.ac.jp, nomura.a.ac@m.titech.ac.jp, matsu@acm.org
[2] Digital Media Professionals Inc., Tokyo, JAPAN
yuya.kobayashi@dmprof.com
[3] RIKEN Center for Computational Science, Kobe, JAPAN

Abstract. As we move towards higher-density, larger-scale, and lower-power computing hardware, new types of failures are being experienced with increasing frequency. Hardware designed for the post-Moore generation are also bringing about novel resiliency challenges. In order to improve the efficiency of resiliency methods, fault injection plays an important role in understanding how errors affect the OS and application. Memory-state-aware fault injection, in particular, can be used to investigate the memory-related faults caused by using current and future hardware under extreme conditions and assess the costs/benefit trade-off of resiliency methods. We introduce MH-QEMU, a memory-state-aware fault injection platform implemented by extending a virtual machine (VM) to intercepting memory accesses. MH-QEMU supports collecting the physical and virtual addresses of memory accesses and defining appropriate injections condition using the collected information. MH-QEMU incurs a 3.4× overhead, and we demonstrate how row-hammer faults can be injected using MH-QEMU to analyzing the resiliency modified NPB CG's algorithm.

Keywords: fault injection · resilience · virtual machine

1 Introduction

As computing systems increase in scale while simultaneously trending towards higher-density and lower-power hardware, new types of failures are becoming more prevalent and significant. Failures due to Silent Data Corruption (SDC) are examples of such failures that are increasing in frequency. SDC produces incorrect results without raising any errors during an application's execution. Furthermore, hardware designed for the post-Moore generation, such as 3D-stacked memories [6,13], and their usage may introduce new failures, such as the

* This work was partially supported by JST CREST Grant Numbers JPMJCR1303 and JPMJCR1687, Japan and conducted as research activities of AIST - Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory (RWBC-OIL)

degradation of flash memory devices caused by frequent writes to a specific location. In order to improve the efficiency of resiliency strategies, it is necessary to know how errors affect the OS and application in order to apply the appropriate resiliency method based the target and the impact of the resiliency method.

Fault injection is an important technique that is used for investigating the effectiveness of resiliency strategies. However, fault injection on real hardware is very costly since injecting hardware faults typically involves breaking the hardware. Previous work [8, 10] has achieved low-cost fault injection by emulating hardware fault with VM's software fault. As a result, some simple faults can be injected easily, such as bit-flip on CPU and memory. Nonetheless, faults that depend on memory state, such as the flash memory degradation mentioned above, is difficult simulated using only the approach described in that work.

We introduce a new fault injection platform, MH-QEMU, which can inject memory-state-aware fault. MH-QEMU is implemented by extending the memory management system of VM and can achieve the following:

- **Injection and flexible description of memory-state-aware fault**: MH-QEMU can emulate various hardware faults affected by memory state and access patterns, such as Row-Hammer [14] on DRAM and the cell degradation on flash memory, by the VMM which can call external modules from VM memory manager.
- **Physical-Virtual placement aware fault injection**: MH-QEMU can modify memory access pattern and its mapping to the physical location by calling external module for each memory access. MH-QEMU also can define the next generation memory module.
- **Supporting analysis of the effects of faults on the system**: For observing the effects of fault on OS and application in the target architecture, it is important to locate the virtual memory address of faults injected by physical memory addresses. MH-QEMU can map such memory addresses and get the information of OS and application without using processes that are executed on a target node.

1.1 Necessity for state-aware memory fault injection

MH-QEMU aims to simulate SDCs, especially the ones depending on memory access patterns. We focused primarily on the physical location and frequency of access patterns. Examples of this class of corruptions are as follows:

- **Disturbance Error**: As the density of DRAM increases, access to a specific memory cell causes electric interference to surrounding cells, which destroys data.
- **Row-hammer fault in DIMM**: Frequent access to a specific memory row causes fluctuation of the signal voltage of the row-selection line, leading to an increase discharge rate of surrounding rows and loss of data.
- **Deterioration of flash memory**: Memory cells of flash devices are known to become unreliable after a limited number of erase cycles. Either the unreliable cells cannot serve as memory elements or they work as memory elements but cannot provide correct value.

To emulate such hardware-specific errors, it is important to consider the physical properties of the hardware, the electrical and magnetic interactions between multiple components. Flexibility in the descriptions of relationships among components is also required in order to adopt not-well-known error mechanism of emerging hardware architecture, like next-generation memory. Examples of possible new error mechanisms are as follows:

- **Hierarchical usage of different memory architectures**: As a result of the trade-off between cost, speed, and capacity, we often use multiple memory architectures in combination, such as DRAM and NVMe, . In such memory systems, memory performance and the error mechanism depend on which physical memory address is accessed.
- **3D structured memory**: Memory architectures achieving high-bandwidth and high-capacity by stacking memory cells vertically to form a 3D structure are currently under active development. As the physical structure is completely different from traditional DIMM, new kinds of disturbance error can occur.

2 Related Work

2.1 Fault Injection to Physical Hardware

Some work simply inject errors by causing physical damage to hardware. Other work inject errors to hardware by neutron beams [19], electromagnetic field [12], heavy-ion beams [9], and so on. Additional hardware has also been employed to inject faults in some manner [1,18]. In these approaches, a fault can be injected easily, but with higher cost: the high cost of procuring additional hardware or causing unrecoverable damage to the system. Furthermore, it is hard to control the location and intensity of the faults being injected.

2.2 Fault Injection by Program Modification

It is possible to inject a code snippet that emulates certain fault behavior into a user program [16]. LLVM-based methods [20] can automatically encode errors to an application without source code modification. These methods can analyze fault effects easily because a user can get detailed information of application processes, such as how the values in memory are used. On the other hands, this method cannot inject hardware specific fault because hardware specific access patterns cannot be determined at program modification time.

2.3 Fault Injection by Virtual Machine (VM)

Error injection to VMM's memory and CPU manager can produce fault on the system executed on VM. In addition, because VMM can dump the state of CPU register and memory value, fault effects can be analyzed in this method without any modification to source code of OS and applications. However, it takes a long time to analyze the effects of injected faults because VMM have to emulate all hardware behavior by software. F-SEFI [8] can inject errors to the logic circuit of CPU, register, and memory modules by this method. D-Cloud [10] is another fault injector by QEMU [4] for hard disk and network controller. D-Cloud can also inject a bit-flip error in memory. Our method, MH-QEMU, also follows this method and the difference from F-SEFI is that MH-QEMU focuses on memory module faults caused by memory state and access pattern. MH-QEMU has APIs that help to analyze memory access behavior, such as a function which maps physical address to virtual address and the reverse in real-time.

3 Design

MH-QEMU is a platform for analyzing memory access patterns of applications and OS and injecting faults depending on the characteristics of the memory modules. The analysis is important for selecting which memory region needs resiliency and what types and levels of resiliency are requested. We assume the following requirements for MH-QEMU's fault injection: 1) no damage to the physical hardware, 2) emulating memory module faults flexibly including those dependent on the memory state and access pattern, and 3) supporting the analysis of the effects of a fault on the OS and application. We choose the VM approach to meet requirement #1 as in previous work (described in 2.3).

3.1 Emulation of fault injection to memory module

In order to emulate faults that are dependent on memory state, MH-QEMU gathers memory access pattern, analyses them to create an appropriate fault injection plan, and applies it to target VM memory. To avoid side effects to the target system, the analysis and injection should be done from host OS. To achieve these functionalities, MH-QEMU consists of the following three modules, which is illustrated in Fig. 1:

Memory Mapper of VM to Host (MM) In order to access the VM's memory from the host environment, the MM identifies where the VM's physical memory is located in host's address space and exposes its content to the host. The VM's physical memory is modified when a process on the host OS modifies the exposed place.

Memory Access Handler (MH) User-defined handler functions (MH) can be registered as hooks to load and store accesses to the target VM's memory space. The MH is invoked with trapped memory addresses and arbitrary operations can be executed. Users can collect and analyze memory access patterns and inject faults from MH function.

Fault Injection Scheduler (FS) The FS manages the MM and the MH by following a scenario file that describes the time of fault injection and configurations of MH. To avoid expensive performance losses in the MH execution, FS can enable and disable MH.

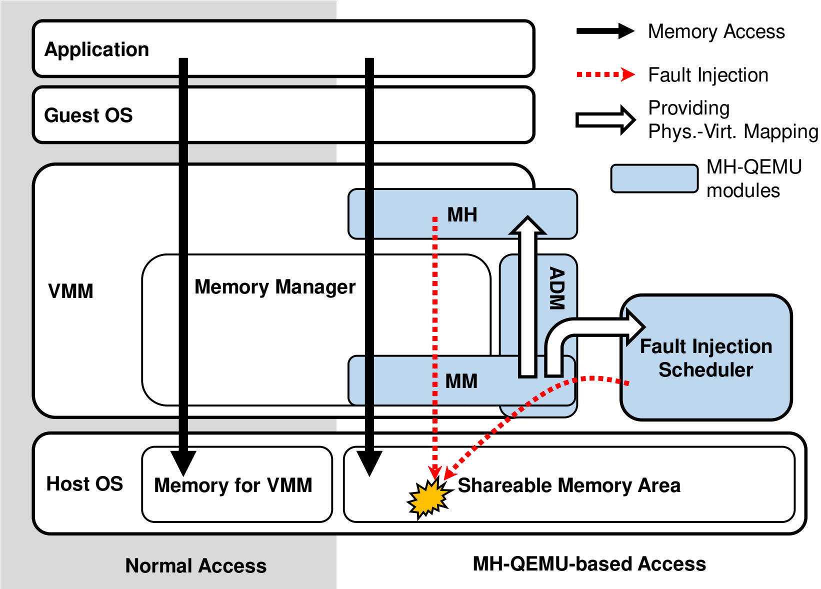


Fig. 1: Overview of MH-QEMU

3.2 Assistance API for analysis of fault effects inside VM

For detailed analysis and well-controlled injection of faults, MH-QEMU needs to know how the physical memory is used by the guest OS. In addition, MH-QEMU should inject fault based on the memory usage of the guest OS.

Address-Data Mapper (ADM) The ADM retrieves information about the guest OS, such as memory page table and process information. A user can use the ADM from the MH via an API. The ADM can also be called in the configuration script invoked by the target VM for initializing other MH-QEMU modules. In addition, the ADM can dump the process information to storage for off-line data analysis.

3.3 Fault Injection Scenario on MH-QEMU

MH-QEMU invokes the user fault injection code defined by the MH by extended the VMM. For memory-state-aware fault injection, MH-QEMU uses each component in the following manner:

1. User starts a VM extended by MH-QEMU and enables FS.
2. At the appropriate time, the FS invokes the target application in the VM and enables MH using the extended the VMM.
3. When an application process accesses memory, VMM calls the MH with the physical and virtual address of the memory that has been accessed.
4. The MH injects errors and collects various information in cooperation with the ADM.
5. For better performance, the FS disables the MH that will no longer be used since the error is injected only once.

Moreover, the MH should not be used for transient and non-memory-state-aware fault injection since the calling the MH has a high cost. In this case, MH-QEMU can inject fault via the FS as follows:

1. FS suspends the VM via the VMM.
2. FS injects faults via the MM following the user-specified fault injection scenario.
3. FS resumes the VM via the VMM.

To illustrate how the MH can simulate a specific type of fault, we show how a fault can be triggered in the frequently-accessed region of an application's heap. The pseudocode is presented in fig.2

1. MH retrieves the heap memory region by using the ADM with the target application name.
2. MH records (position, counts) of the access to heap region.
3. MH injects an error to frequently-accessed memory bit via MM.
4. MH dumps the process information of the target application and the address where the error was injected. The MH gets the process information from ADM using target application's name.
5. MH reports the injection to the FS.

4 Implementation

MH-QEMU is implemented on top of QEMU 2.3.1. Due to the ADM's implementation, Linux is the only supported guest OS on MH-QEMU. The implementation of each MH-QEMU module (MM, MH, FS, and ADM) is described in the following subsections. API functions for calling other modules from MH module are described in Table 2.

4.1 MM: Memory Mapper

The memory space of a QEMU VM can be mapped into a file in host OS (`-mem-path` option). The MM uses this functionality to enable access to guest OS's memory image from host OS. For performance reason, guest OS's memory space will be mapped to files in `tmpfs`, which is a virtual file system that uses the host system's memory as a data store.

```
memory_access_handler(physaddr, virtaddr){
  range←ADM_get_heep_addr(target_name)
  if (virtaddr is in range){
    count[virtaddr]++
  }
  for(addr←each range){
    if (count[addr] >= threshold){
      records addr
      MM_flipbit(addr)
      ADM_write_processinfo(target_name)
      FS_turnoffMe()
    }
  }
}
```

Fig. 2: Pseudo code of MH

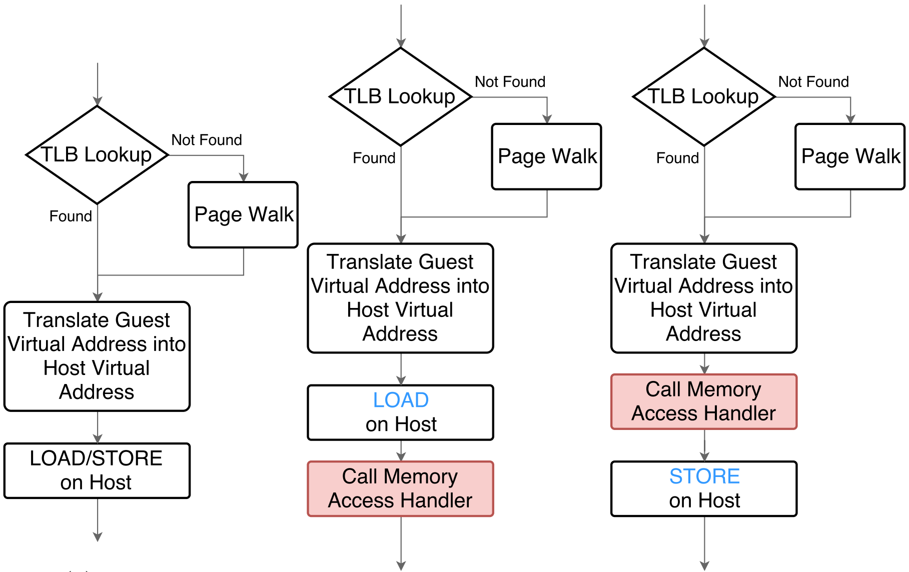
Table 1: Structure of **MHinfo**
`target_ulong` is the alias of `unsinged long`

Name	Type	
val	uint64_t	a value which is stored or loaded
dsize	size_t	a size of data which are accessed
gvAddr	target_ulong	virtual memory address on guest OS
gpAddr	target_ulong	physical memory address on VM
hvAddr	uintptr_t	physical memory address on host server
isLoad	bool	true:on load operation, false:on store operation
isBigEndian	bool	GuestVM's endian: true:Big, false:Little

4.2 MH: Memory Access Handler

The MH is implemented as an extension to TCG (Tiny Code Generator), which is a part of QEMU. TCG is a virtualization module for CPU operations. In the TCG layer, all memory access operations are expressed as either `ld`(load) or `st`(store) operations. We added call to the MH (Fig. 3) in the implementation of these operations. The MH is called either before an actual memory store occurs or after an actual load finishes. The MH function takes an argument that is a pointer to the **MHInfo** structure. This structure contains the information on memory access listed in Table 1.

In KVM [15], which utilizes hardware virtualization extensions of CPU to accelerate VM emulation, the TCG is replaced by hardware extensions and MH implementation does not work. However, MH-QEMU can benefit from the accelerated performance in KVM by incorporating other code insertion method. Specifically, memory accesses must be trapped with binary level translator such as Intel Pin [11, 17] or dyninst [2].


TLB Lookup
Not Found
Found
Page Walk
Translate Guest Virtual Address into Host Virtual Address
LOAD/STORE on Host
LOAD on Host
STORE on Host
Call Memory Access Handler

(a) MH disabled

(b) MH enabled, load operation

(c) MH enabled, store operation

Fig. 3: Code generation by TCG

4.3 FS: Fault Injection Scheduler

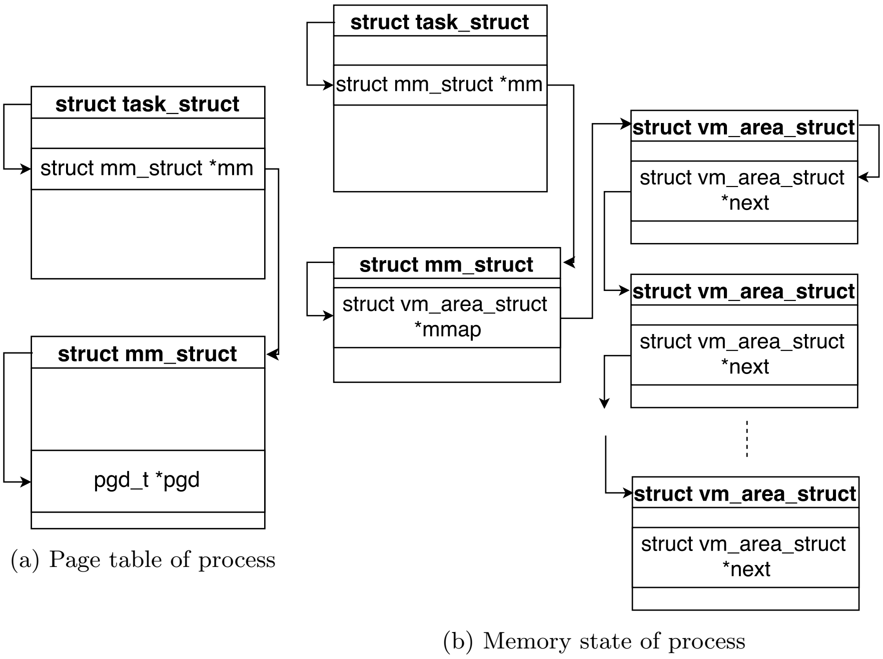
The FS is a process using extended QMP (QEMU Machine Protocol) and HMP (Human Monitor Protocol). QMP and HMP are protocols for controlling the state of QEMU such as shutdown, making a snapshot, and adding new virtual hardware. We add new entry points to manage MH-QEMU components and the FS calls them to interact with MH-QEMU.

4.4 ADM: Address-Data Mapper

The ADM gets page table and process states from the guest OS. Although this information can be obtained easily in the guest OS, the ADM read them from the outside of VM in order to not modify the memory state of guest OS. The ADM analyzes the VM's memory, via the MM, and gets process information and their page table as follows:

Page Table The ADM gets the physical address of the kernel page table from the symbol table of the kernel binary by using QEMU and the GDB function. The ADM is able to convert physical memory addresses to virtual memory addresses using this page table if the memory has not been reallocated.

Process Information Process information in the Linux kernel is managed by a circular list. The ADM can get all process information in the guest OS if ADM accesses the process information structure of any process. The ADM uses information of the idle process of Linux, since the location of idle process information is stored in a global variable. The ADM can also get process information from the kernel binary with symbols by using QEMU and the GDB function (Fig. 4) in a similar manner as with the page table information described above. The same limitation that memory cannot be reallocated also applies to process information retrieval.



(a) Page table of process

(b) Memory state of process

Fig. 4: Process Information of Linux Internals: a) Page table of process, b)Memory state of process

5 Evaluation and Use case

We present the overhead of the MH-QEMU platform using the NAS Parallel Benchmark, and we use the CG kernel to illustrate how to use MH-QEMU.

Table 2: API to MH module from other modules

MM	
MM_set(st,fin,val)	Write value to memory
FS	
FS_turnoff_me()	FS disable MH
ADM	
ADM_write_pagetable(app)	Write a page table to storage
ADM_write_filemapping(app)	Write a file mapping info. to storage
ADM_get_addrange(app, file)	Get addr. range used by app.
ADM_conv_virtaddr(physaddr)	Convert virt. addr. to phys. addr.
ADM_conv_physaddr(app, virtaddr)	Convert phys. addr. to virt. addr.

5.1 Evaluation Environment

All evaluations described in this section use a single host server. Eight MH-QEMU VM instances are executed on the server. The specification of the host server and the VM are shown in Fig. 3.

Table 3: Execution Environment

Host Server	
CPU	2 * Intel X5650 (2.67GHz, 6core/12thread) with VT-x
Memory	ECC DDR4 SDRAM 46GB
OS	CentOS 7.1 (Linux Kernel 3.10.0)
VM Server (8VM/host)	
CPU	x86_64 Architecture
Memory	512MB
OS	Scientific Linux 7.4 (Linux Kernel 3.10.0)

5.2 Overhead of MH-QEMU platform

To evaluate the overhead of MH-QEMU platform, we compared the execution time of NAS Parallel Benchmark on native QEMU and on MH-QEMU with empty an MH function. We decomposed the overhead of MH-QEMU to overhead caused by the MM and the overhead caused by MH; the overhead of MM was found to be negligible. Therefore, the overhead of MH-QEMU is almost the same as the overhead of MH. The EP, CG, MG, FT and IS kernels of NAS Parallel Benchmark 3.3.1 were used with the class B problem size. The average execution time of five runs for each kernel is shown in Fig. 5 and Table 5. The overhead of MH-QEMU is normalized to the overhead of naive QEMU. MH-QEMU was up to 3.4 times slower than native QEMU.

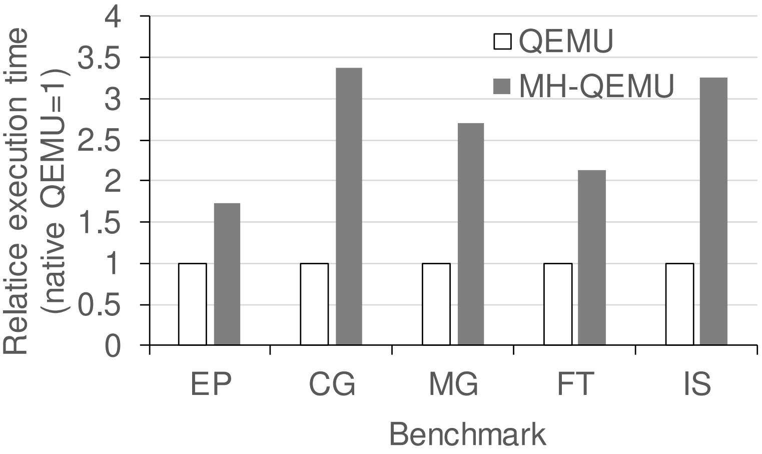


Fig. 5: MH-QEMU overhead toward native QEMU

Table 4: Execution Time of QEMU and MH-QEMU(sec.)

	QEMU	MH-QEMU
EP	345.65	596.626
CG	38.676	130.652
MG	98.844	266.458
FT	201.47	428.078
IS	24.38	79.378

5.3 Use case: resiliency analysis of modified NPB CG

We use NPB CG [3] to demonstrate the usage of MH-QEMU for resiliency analysis. We expect CG already has some algorithm-level resilience to SDC because it uses the inverse power method, an iterative method. In this scenario, we want to reveal which memory region is weak due to SDC. In the original NPB CG implementation, the number of iteration is fixed as it is intended to be used as a performance benchmark. To evaluate resiliency of the iterative method, we modified NPB CG to continue the iteration until it converges, that is, until the residual becomes less than the 10^{-20}. We inject Row-Hammer faults, which corrupts data in the memory line next to a frequently accessed memory line. We executed 2443 CG runs for this analysis.

Implementation of Row-Hammer MH The Row-Hammer MH injects the fault as follows:

1. The physical address of each memory access is decomposed into the locations of the physical memory channel, the bank, and the line, following the mapping rule of Intel 82955X-MCH memory structure [5] described in Fig. 6. The MH counts the access for each memory line.
2. If the access counter exceeds the threshold α, the MH determines whether an error is injected with probability λ.
3. If the error is to be injected, the MH retrieves the process memory information using the ADM and randomly changes a single bit in the adjacent line of the accessed region to 0. We choose parameters as $\alpha = 1000$ and $\lambda = 5 \times 10^{-10}$

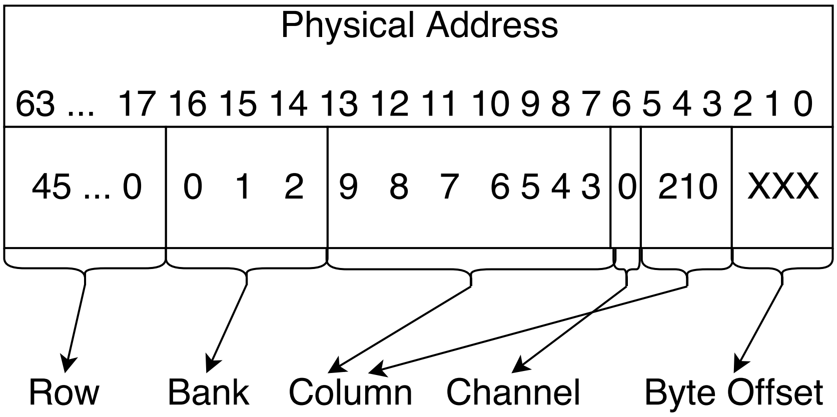


Fig. 6: Mapping rule of Intel 82955X-MCH

Distribution of computation error A histogram of the computation errors in the results is shown in Fig. 7. The last category labeled as ”Abort” represent the number of detectable failed executions. These include when the VM hangs, abnormal termination of the application, and the result containing NaN. Other than such failed execution, all the results fall into one of two categories. We judged that the results with more than 5% error is caused by SDC. In most SDC results, the error is around 166%. It is unknown why they converge to that value as the inverse power method does not have a local solution. On the other hand, 60% of execution return the correct result even after the injection of memory hammer error. This means the CG algorithm has a certain resiliency to SDC.

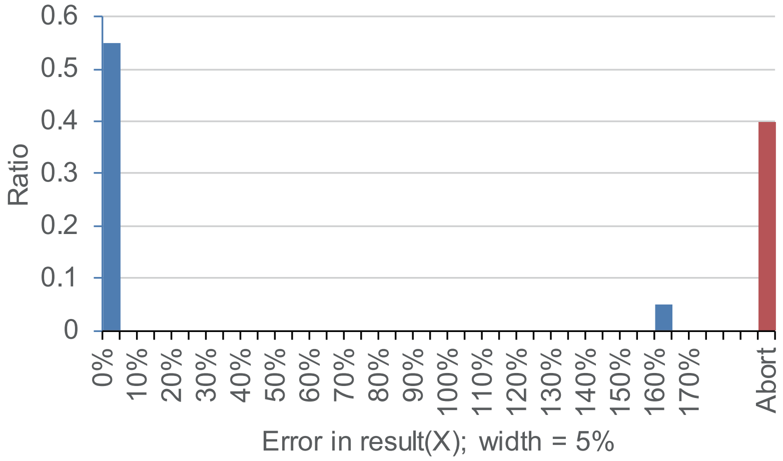


Fig. 7: Histogram of errors

Relationship between fault and process memory region To investigate the cause of SDC, we select 825 runs at random and mapped the modified data region and execution results, as shown in Fig. 8. The results show that SDC occurs only when the BSS section of CG’s binary is modified. The BSS region stores global and static variables with an initial value. Most of the data in BSS region of the CG application kernel are input matrices and intermediate data, modification to which does not lead the application to abnormal termination. In the execution of CG, most of the data are stored in the BSS region, not in the stack. If we analyze the access pattern of each variable to determine its importance, we can specify which variables should be protected to avoid SDCs, without knowledge of CG’s algorithm.

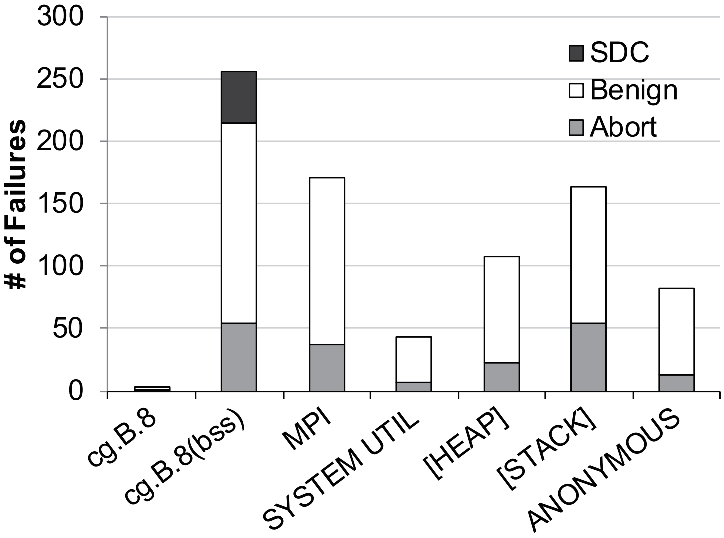


Fig. 8: Effect of error for each memory region in process

6 Conclusion

Brand-new hardware architectures, which has different usage and characteristics from current architectures, are emerging in the post-Moore era. We need fault injectors that can emulate errors in such new architectures in order to develop resiliency methods with the appropriate scope.

We developed MH-QEMU, a fault injector that can generate errors by emulating memory access patterns and the physical structures of memory modules, to accommodate new memory architectures. With MH-QEMU, we can verify resiliency against SDCs brought by architecture-specific properties as well as incidental SDCs.

Currently, the overhead of MH-QEMU is significantly large. It can be reduced by narrowing the memory region that is monitored by the memory handlers. MH-QEMU can also be accelerated by employing hardware-level VM acceleration in KVM when supported by other code insertion methods like Intel Pin [11,17] and dyinst [2].

We are focusing on the the flexibility of fault injection and obtaining the memory location of injected errors at the process level; MH-QEMU does not trace application behavior after fault injection. In future work, we are planning to evaluate application level resiliency for new memory architectures, such as flash memories, 3D stacked memories [6,13], and hierarchical combination with them and DIMMs [7], after enhancement of MH-QEMU for such tracing functionality. If CPU state can be controlled with tools like F-SEFI [8], MH-QEMU approach can be generalized to other types of devices, including network devices and emerging hardware architectures.

References

1. Arlat, J., Crouzet, Y., Laprie, J.C.: Fault injection for dependability validation of fault-tolerant computing systems. In: Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on. pp. 348–355. IEEE (1989)
2. B, B., J, K, H.: An api for runtime code patching. High Performance Computing Applications **14**
3. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., et al.: The nas parallel benchmarks. The International Journal of Supercomputing Applications **5**(3), 63–73 (1991)
4. Bellard, F.: Qemu, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track. pp. 41–46 (2005)
5. Corporation, I.: Intel 82955x memory controller, https://ark.intel.com/products/27727/Intel-82955X-Memory-Controller
6. Corporation, I.: Intel optane technology, https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html
7. Endo, T.: Realizing out-of-core stencil computations using multi-tier memory hierarchy on gpgpu clusters. In: 2016 IEEE International Conference on Cluster Computing (CLUSTER). pp. 21–29 (Sept 2016). https://doi.org/10.1109/CLUSTER.2016.61
8. Guan, Q., Debardeleben, N., Blanchard, S., Fu, S.: F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. pp. 1245–1254. IEEE (2014)
9. Gunneflo, U., Karlsson, J., Torin, J.: Evaluation of error detection schemes using fault injection by heavy-ion radiation. In: Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on. pp. 340–347. IEEE (1989)
10. Hanawa, T., Koizumi, H., Banzai, T., Sato, M., Miura, S., Ishii, T., Takamizawa, H.: Customizing virtual machine with fault injector by integrating with specc device model for a software testing environment D-cloud. Proceedings - 16th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2010 pp. 47–54 (2010). https://doi.org/10.1109/PRDC.2010.37
11. Intel Corporation: Pin - a dynamic binary instrumentation tool, https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
12. Karlsson, J., Folkesson, P., Arlat, J., Crouzet, Y., Leber, G., Reisinger, J.: Application of three physical fault injection techniques to the experimental assessment of the mars architecture. Dependable Computing and Fault Tolerant Systems **10**, 267–288 (1998)
13. Kim, J., Kim, Y.: Hbm: Memory solution for bandwidth-hungry processors
14. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental

study of dram disturbance errors. In: ACM SIGARCH Computer Architecture News. vol. 42, pp. 361–372. IEEE Press (2014)

15. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: kvm: the linux virtual machine monitor. In: Proceedings of the Linux symposium. vol. 1, pp. 225–230 (2007)
16. Li, D., Vetter, J.S., Yu, W.: Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. p. 57. IEEE Computer Society Press (2012)
17. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 190–200. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065034, http://doi.acm.org/10.1145/1065010.1065034
18. Madeira, H., Rela, M., Moreira, F., Silva, J.G.: Rifle: A general purpose pin-level fault injector. In: European Dependable Computing Conference. pp. 197–216. Springer (1994)
19. Michalak, S.E., DuBois, A.J., Storlie, C.B., Quinn, H.M., Rust, W.N., DuBois, D.H., Modl, D.G., Manuzzato, A., Blanchard, S.P.: Assessment of the impact of cosmic-ray-induced neutrons on hardware in the roadrunner supercomputer. IEEE Transactions on Device and Materials Reliability **12**(2), 445–454 (2012)
20. Thomas, A., Pattabiraman, K.: Llfi: An intermediate code level fault injector for soft computing applications. In: Workshop on Silicon Errors in Logic System Effects (SELSE) (2013)