

# Using Unreliable Virtual Hardware to Inject Errors in Extreme-Scale Systems

Scott Levy, Matthew G. F. Dosanjh and  
Patrick G. Bridges  
Department of Computer Science  
University of New Mexico  
slevy,mdosanjh,bridges@cs.unm.edu

Kurt B. Ferreira\*  
Scalable System Software Department  
Sandia National Laboratories  
kbferre@sandia.gov

## ABSTRACT

Fault tolerance is a key obstacle to next generation extreme-scale systems. As systems scale, the Mean Time To Interrupt (MTTI) decreases proportionally. As a result, extreme-scale systems are likely to experience higher rates of failure in the future. To mitigate this, significant research has focused on developing and validating fault tolerance techniques. However, evaluating techniques for withstanding hardware failures at large scale is challenging because replicating those failures on small-scale testbeds is difficult. In this paper, we propose a virtualization-based framework for creating testbeds with unreliable virtual hardware. Our proposed approach allows for comprehensive evaluation of fault tolerance techniques in a broad range of failure regimes. Although there are many other approaches for mimicking unreliable hardware, none of them offer the breadth, scalability, and performance that a virtualization-based solution does.

## Categories and Subject Descriptors

D.4.5 [Operating Systems]: Reliability—*fault-tolerance*;  
D.2.5 [Software Engineering]: Testing and Debugging—*error handling and recovery, testing tools*

## Keywords

fault-tolerance; resilience; virtualization; high-performance computing

## 1. INTRODUCTION

Fault tolerance is a key obstacle to next generation extreme-scale systems. As we build larger and more powerful systems, socket counts continue to grow. The most powerful

---

\*Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FTXS'13, June 18, 2013, New York, NY, USA.

Copyright 2013 ACM 978-1-4503-1983-6/13/06 ...\$15.00.

HPC systems today contain tens of thousands of processor sockets [18]. Current predictions suggest that the first exascale system will require nearly an order of magnitude more sockets [2]. Because system Mean Time To Interrupt (MTTI) is inversely proportional to socket count [26], we expect future systems to encounter failures more frequently than current systems do. As a result, fault tolerance and resilience in large-scale systems has been an important and increasingly active area of research.

Error injection techniques are an increasingly important tool for evaluating techniques for improving the fault tolerance of large scale systems. These techniques are necessary because while large-scale systems fail frequently failures are rare in the smaller systems that are commonly available for research. Current error injection techniques are limited in their breadth, performance, and scalability. In particular, no error injection framework supports the broad range of devices and failure modes that may occur, including failures in memory, processor logic, disks and network interfaces. In addition, many error injection techniques have performance costs that preclude their use in long-running applications or on large-scale systems.

In this paper, we propose a virtualization-based framework for injecting hardware errors to evaluate the effectiveness of fault tolerance strategies for extreme-scale systems. Virtual machine monitors (VMMs) designed for HPC systems (e.g., Palacios [19]) allows us to inject errors across an entire system with minimal performance costs. In addition, because virtualization enables fine-grained control the hardware visible to a guest, it can inject a broad range of hardware errors from the virtual machine monitor (VMM). Overall, virtualization provides a platform that enables us to inject errors with the breadth, performance and scalability required.

In the remainder of this paper, we describe the demands of error injection for large-scale systems and outline our proposed framework for meeting these demands. To demonstrate the viability of this approach, we present two examples using our proposed framework to mimic unreliable hardware. In addition, we discuss potential directions of future research that build upon this framework.

## 2. MOTIVATION

Platforms for evaluating fault tolerance in extreme-scale systems need to be able to accurately mimic real hardware errors. Although small-scale tests and microbenchmarks play an important role, comprehensive evaluation requires techniques that: mimic the full spectrum of hardware er-

rors, present errors across multiple nodes, and operate with low overhead. Unlike other approaches, virtualization allows us to fully address all three of these criteria.

## 2.1 Breadth

Memory [16, 27], processor logic [8], network interfaces [21] and disks [26] have all been recognized as important sources of hardware errors in large-scale systems. Comprehensive evaluation platforms must be able to mimic errors in any of these components. Moreover, they must also support error injection across an entire component. For example, considerable effort has been expended evaluating the effect of silent errors in applications [5, 21, 8]. However, these approaches are targeted at application memory and are not easily extended to evaluate approaches for recovering from detected errors [3] or errors in kernel memory [10].

One of the most common causes of node failure is DRAM errors [27]. The prevailing strategy for handling these failures is coordinated checkpoint/restart. However, as socket counts increase this technique may no longer be adequate [11]. Additionally, recent evidence suggests that errors may occur more often in kernel memory than in user memory [16]. As a result, developing and evaluating alternative fault tolerance strategies for memory errors that occur outside of the application or that are detected by hardware is critically important.

Because virtualization precisely controls the hardware visible to the guest, it offers an ideal platform for mimicking the full range of hardware errors. For example, in the context of memory errors, virtualization enables us to mimic silent data corruption as well as detected memory errors (e.g., DRAM ECC errors) and errors in both application and kernel memory.

## 2.2 Scalability

The next generation of extreme-scale systems will expose an enormous amount of concurrency. The first exascale system may be composed of hundreds of thousands of nodes [2]. Effective evaluation of fault tolerance strategies targeted for such an environment requires a platform that can mimic hardware errors throughout the system. VMMs designed for large-scale systems (e.g., Palacios) allow virtualization to be deployed throughout an entire HPC system with very low overhead [19]. In addition, they are being used to emulate next-generation large-scale systems [4]. As a result, a virtualization-based approach can readily mimic hardware failure regimes in which errors occur not just on a single node but throughout the entire system.

## 2.3 Performance

Workloads on extreme-scale systems will frequently run for days or weeks. Fault tolerance strategies need to be evaluated over the entire lifetime of a realistic job. As a result, any infrastructure for mimicking hardware errors must minimize the runtime overhead it imposes on the application. The Palacios VMM has been shown to allow near-native performance (less than 5% overhead with nested paging) in HPC systems [19]. Virtualization can, therefore, allow us to examine fault tolerance techniques over the lifetime of realistic workloads.

## 3. FRAMEWORK

### 3.1 Basic Framework

To facilitate comprehensive evaluation of fault tolerance techniques, we propose a virtualization framework for mimicking a broad range of unreliable hardware environments. Our framework is composed of three types of components: a Front End, an Error Scheduler, and Error Injectors.

#### 1. Front End

The Front End provides a simple interface to allow users to describe the errors that they wish to inject. In addition to specifying simple, deterministic hardware errors, this interface allows the user to describe stochastic processes that allow for non-determinism spatially, temporally and behaviorally.

#### 2. Error Scheduler

The Error Scheduler sits between the Front End and the Error Injectors and is responsible for managing the Error Injectors. In particular, it schedules errors to be injected as specified by the user through the Front End.

#### 3. Error Injectors

The low-level interface between our framework and the VMM are low-level Error Injectors that handle particular types of errors in a specific VMM. Our framework could be ported to another VMM by rewriting the Error Injectors to work with the target VMM.

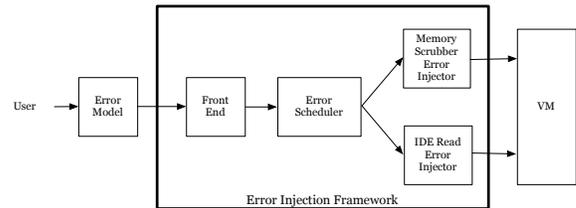


Figure 1: The high-level architecture of our Error Injection Framework

### 3.2 Error Scheduling

Accurately mimicking system-wide errors requires the ability to precisely control the manner in which errors occur. Support for simple, deterministic errors (i.e., single-event upsets) allows us to support methodical approaches for exploring system sensitivity to errors (e.g., [21]). However, we believe more complex stochastic models are also important. For example, the occurrence of memory errors is commonly modeled as an exponential process. But recent evidence suggests that this is a poor model for the failures that we see in the field [16]. By supporting complex stochastic models, our framework can help develop and validate more accurate models of memory errors that may in turn enable more effective resilience techniques.

Our design supports models that describe error regimes spatially, temporally and/or behaviorally. Spatial models describe how errors occur within a given hardware device. The canonical example is specifying the address at which a memory error will occur. Temporal models describe when

errors occur. Behavioral models describe when and where errors occur based on system behavior. For example, memory scrubbers have little impact on the rate at which memory errors are encountered [16]. This suggests that memory failures are dependent on access patterns. A model that allows error occurrences to be defined in terms of memory access patterns (e.g., number of writes to a particular location) may be useful for more accurately modeling real memory errors.

### 3.3 Error Injection

There are a large number of errors that are potentially interesting in the context of system resilience and fault tolerance. We describe a number of these types of errors below.

#### 3.3.1 Memory Scrubber Errors

Memory faults are one of the most commonly observed faults in distributed systems [24]. Therefore, the first Error Injector that we implemented injects a Memory Scrubber DRAM ECC error at a user-specified guest physical address.

To generate an error, the Memory Scrubber Error Injector uses a physical address supplied by the user to populate the processor’s machine check registers. The contents of these registers inform the guest of the nature of the virtualized exception that has occurred. The Error Injector then injects an asynchronous Machine Check Exception (MCE) into the guest. From the guest’s perspective this sequence mimics what would happen if a memory scrubber encountered an uncorrectable memory error.

This discussion raises a potential challenge presented by virtualization. By using this Error Injector we can easily define errors that mimic failing memory anywhere in the guest’s physical memory. However, there exists a semantic gap between a VMM and its guests: the VMM sees undifferentiated physical memory. Using well-established introspection techniques [12] allow us to bridge this gap.

#### 3.3.2 Synchronous DRAM ECC Errors

DRAM errors account for a significant fraction of node failures [24]. Frequently, these failures are a result of errors that exceed the ability of the ECC or *chipkill* [9] to correct. When the MMU detects an uncorrectable memory error, it raise a synchronous Machine Check Exception. Currently, Linux makes some effort to avoid crashing unnecessarily [17]. But, if the error occurs in memory that is being used, the kernel kills the applications that reference the corrupted memory. Strategies for avoiding this outcome (e.g., [3]), are difficult to fully evaluate without the ability to cause memory errors to occur.

There is also evidence to suggest that memory errors occur more frequently in kernel memory than in application memory [16]. When an error is detected in kernel memory, Linux reboots the entire node. Based on this evidence, strategies for hardening kernel data structures against errors are being explored [10]. However, evaluating the effectiveness of a hardened kernel is difficult in the absence of an effective way to induce errors throughout the entire memory footprint.

Because of this empirical evidence on the significance of memory errors, our Error Injector supports injection of synchronous DRAM errors throughout the entire memory footprint, thereby more realistically mimicking memory failures.

#### 3.3.3 IDE Read Errors

Given the relative frailty of mechanical devices, another

common fault is hard disk failure [25]. As a result, we also implemented a Error Injector that injects an IDE read error at a user-specified logical block address (LBA). From the guest’s perspective this appears to be a balky ATA hard drive.

To inject a fault, the IDE Read Error Injector snoops the IDE bus to determine when the guest is accessing an LBA at which an error has been injected. The Error Injector relies on the fact that each time the guest reads from an IDE device, it will query a status register to determine whether the read was successful. Therefore, to virtualize an IDE Read error, the Error Injector intercepts reads of the status register. It also intercepts writes to IDE control registers so that it can maintain internal state about the commands that have been issued. As a result, it is able to determine whether the most recent request was a read from a faulty sector when the guest reads the status register. If it determines that a error should be injected, it sets a bit in the status register to indicate that an error occurred.

#### 3.3.4 Additional Errors

Our VMM-based approach supports many other types of errors, including many of the most frequently studied types of errors. For example, virtualization-based failure injectors can mimic errors in processor logic [8], corrupt messages received from the network [21], and silent corruption of memory [5].

## 4. PRELIMINARY RESULTS

We evaluated our framework qualitatively by examining the fidelity of virtualized hardware errors as they are experienced by a guest operating system. This approach is difficult, however, because we lack a good benchmark for comparison and guest operating systems can mask some simple memory errors. For this same reason, it is difficult to fully assess whether our framework accurately virtualizes hardware errors.

As a result, we used operating system logs to validate the operation of our framework. Specifically, we carefully inspected the guest’s logs and its kernel’s source code to verify that the errors that we injected are good approximations of actual hardware faults. All evaluation of our framework was conducted using the Palacios virtual machine monitor and a simple Linux 2.6.37 kernel busybox guest.

### 4.1 Memory Scrubber errors

As discussed earlier, the semantic gap between our VMM and our guest makes the isolated evaluation of this error injector difficult. Linux implements several strategies for withstanding DRAM ECC errors. For example, if an error occurs on a clean page in the page cache, Linux can simply flush the page from the cache and proceed. However, if an error occurs in a page containing kernel data structures, the guest will simply crash.

Therefore, to validate the DRAM ECC Error Injector, we needed to identify a guest physical address such that an error at the address would elicit observable behavior without crashing the guest. A crash does not represent meaningful validation of our Error Injector. To accomplish this goal, we constructed a simple application (`hello_world`) that `mallocs` a block of memory, displays the (guest) virtual address of the newly acquired memory, and enters an infinite loop. We also wrote a small utility program (`pagemap`) that uses

/proc/<pid>/pagemap and /proc/kpagefiles. This utility allows us to translate a virtual address from a process' address space to a guest physical address that can be then provided to the Error Injector. The output of this utility provided us with an interesting guest physical address to use as a target. When an error occurs at this address, Linux should terminate our `hello_world` application.

We then used the Memory Scrubber Error Injector to inject a error at the target address. Using `dmesg` from within the guest enabled us to validate that the guest responded to this error appropriately. The results are shown in Figure 2. The log shows that the guest detected a Machine Check Exception and the page on which we injected an error was in fact dirty. As a result, we see that the guest killed the `hello_world` process.

```
[ 3.370000] Freeing unused kernel memory: 360k freed
[ 3.380000] Freeing unused kernel memory: 1620k freed
[ 36.310000] Disabling lock debugging due to kernel taint
[ 36.310000] [Hardware Error]: Machine check events logged
[ 36.320000] MCE 0x19c3: dirty LRU page recovery: Recovered
[ 36.320000] MCE: Killing hello_world:60 due to hardware memory corruption fault
```

Figure 2: The guest’s kernel log after a DRAM ECC error was injected at 0x19c3000

## 4.2 IDE read errors

Validating the IDE Read Error Injector presented similar issues. Again the semantic gap makes it difficult to identify an LBA that will elicit observable behavior. An LBA that maps either to an empty section of the disk or to a file that is never read is not instructive. Using existing files is problematic because reading from a file that has been read recently will not result in the desired device access. This is due to the fact that the requested sectors will be read from the buffer cache rather than from the underlying hardware device. Therefore, to validate the IDE Read Error Injector, we began by adding a simple text file to the virtual hard drive. We can guarantee that this file will not be read from disk until we have injected our error.

To identify the LBA of a block within this file, we constructed a simple utility (`blockmap`) that uses a `FIBMAP ioctl` call on the target file to determine a target LBA. One additional complication is that our guest filesystem used 1024 byte blocks but our virtual ATA hard drive has 512 byte sectors. Thus, the final step to obtaining a target LBA was to account for this difference.

We then used our IDE Read Error Injector to inject a error within our target file. Next we entered the guest and attempted to read the file using `vi`. The result is shown in Figure 3. The snippet of the log shown here reflects a single attempt by the guest to read from the faulty sector. Although the remainder of the log is omitted to conserve space, we observe that the guest makes several attempts to read the sector before giving up and reporting an error to the user.

## 5. RELATED WORK

Extensive research has been conducted on effective methods for injecting hardware faults in operating systems and applications running on reliable hardware. These approaches fall roughly into three categories.

```
[ 3.360000] Freeing unused kernel memory: 360k freed
[ 3.370000] Freeing unused kernel memory: 1620k freed
[ 83.540000] ata1.01: exception Emask 0x0 SAct 0x0 SErr 0x0 action 0x0
[ 83.540000] ata1.01: BMDMA stat 0x4
[ 83.540000] ata1.01: failed command: READ DMA
[ 83.540000] ata1.01: cmd c8/00:18:53:01:00/00:00:00:00/f0 tag 0 dma 12288
[ 83.540000] res 51/00:18:53:01:00/00:00:00:00/a0 Emask 0x1 (device timeout)
[ 83.540000] ata1.01: status: ( DRDY ERR )
[ 83.560000] ata1.00: configured for MWDMA1
[ 83.570000] ata1.01: configured for MWDMA2
[ 83.570000] ata1: EH complete
```

Figure 3: The guest’s kernel log after a IDE read error was injected

## 5.1 Hardware

One of the earliest approaches was to induce actual hardware faults in the systems being tested. Several novel techniques for inducing actual hardware faults have been explored. For example, [1] explored creating hardware faults by manipulating the pin-level voltages of chips in the target system. Additionally, in [14] the authors developed methods for injecting faults by exposing the target system to a Californium-252 radiation source. However, as these approaches suggest, creating conditions in which an actual hardware fault occurs is invasive, time-consuming and expensive. Other hardware-based approaches have been explored more recently. For example, in [23] the authors explored fault injection in a specialized system constructed from a network of FPGAs. Although this approach can potentially mimic a broad range of hardware errors, it is limited to a computing platform that is not widely used. Our virtualization-based approach can be used on the real hardware that is commonly used in extreme-scale systems.

## 5.2 Simulation

Another approach for evaluating the fault tolerance of a system is to use simulation. One common approach is to specify a system using VHDL and simulate the system in the presence of various hardware faults. In addition to being slow, another drawback of this approach is that the evaluation of a full-featured system is challenging. To overcome these limitations, simulation has been coupled with virtualization in several hybrid approaches. In these cases, virtualization is used to run a guest and simulators are invoked to model the behavior of faulty devices [15, 22]. Although this approach has many benefits over full VHDL simulation, writing and maintaining device simulators is a significant undertaking.

## 5.3 Software

Due to the shortcomings of these two approaches, recent research has focused on software techniques that are broadly referred to as Software-implemented Fault Injection (SWIFI). In [6], errors are injected using UMLinux and `ptrace`. UMLinux allows a guest operating system to be run as a user-level Linux process. `ptrace` is a system call that provides debugging features that are used by common tools such as `gdb` and `strace`. Because `ptrace` has complete access to the registers and memory of a given process, it can create the appearance of a wide range of hardware faults. Augmented application fault-injection, e.g., [20], takes an application-centered approach, injecting faults such as bit-flips into the application’s data structures. Unlike these approaches, our virtualization-based approach works with unmodified applications and operating systems. Another approach is to use the performance monitoring and debugging

features of the CPU [7, 13]. However, the range of faults that can be injected are limited (e.g., corrupting memory is difficult).

Finally, emulation and virtualization have seen limited use in failure injection. In [8], the authors use the QEMU emulator to inject logic errors into an application. Although an emulator can potentially mimic many important classes of hardware faults, emulators are typically exceedingly slow and do not scale well to multi-node systems. There are several examples of pairing virtualization with hardware simulators to model the behavior of faulty devices [15, 22]. However, like emulation, simulation has similar performance and scalability challenges. Our approach, in contrast, relies on VMMs with proven scalability and allows meaningful and (relatively) precise errors to be injected without requiring full device simulation.

## 6. FUTURE WORK

Numerous options are available for carrying this work forward. Given the relative prevalence of memory errors, our immediate focus is on investigating the impact of memory errors on system behavior. To this end, we are currently implementing a fault injector that will enable us to inject synchronous DRAM ECC errors and silent data corruption. Coupled with the support for asynchronous DRAM ECC that we already have, this framework will enable us to more accurately mimic a wide range of memory errors. In particular, this allows us to explore error sensitivity across the entire memory footprint, not just within a single application.

Additionally, we are adding support for stochastic models for injecting errors. This will allow us to begin to experiment with different models of memory failure. Specifically, we plan to investigate techniques for incorporating system behavior (e.g., memory access patterns) into our models of memory failure. Our ultimate goal is to develop a model of memory failure that more accurately represents patterns of errors that we observe in the field.

Finally, we plan to use this framework to explore the effect of memory errors across multiple nodes. In particular, we are interested in examining the impact of error recovery on overall system performance.

## 7. CONCLUSION

Successfully building next-generation extreme-scale systems will require innovative strategies for withstanding increasingly frequent errors. In this paper, we propose a framework that uses virtual unreliable hardware to mimic complex error regimes. Virtualization allows us to mimic hardware errors with the breadth, scalability and performance that is required for comprehensive evaluation of fault tolerance techniques. Additionally, this framework enables us to thoroughly explore the effect of hardware errors that occur on an individual node and across the nodes of a system. To illustrate the viability of this approach, we built two Error Injectors and demonstrated that they could be used to virtualize Memory Scrubber errors and IDE read errors.

## 8. REFERENCES

- [1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *Software Engineering, IEEE Transactions on*, 16(2):166–182, 1990.
- [2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, P. Kogge, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snively, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. [http://www.science.energy.gov/ascr/Research/CS/DARPA/exascale-hardware\(2008\).pdf](http://www.science.energy.gov/ascr/Research/CS/DARPA/exascale-hardware(2008).pdf), Sept. 2008.
- [3] P. Bridges, M. Hoemmen, K. B. Ferreira, M. Heroux, P. Soltero, and R. Brightwell. Cooperative application/os DRAM fault recovery. *Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids in conjunction with the Euro-Par Conference, Lecture Notes in Computer Science*, pages –, 2011.
- [4] P. G. Bridges, D. Arnold, K. T. Pedretti, M. Suresh, F. Lu, P. Dinda, R. Joseph, and J. Lange. Virtual machine-based emulation of future generation high-performance computing systems. *International Journal of High Performance Computing Applications*, 26(2):125–135, May.
- [5] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS '08*, pages 155–164, New York, NY, USA, 2008. ACM.
- [6] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including os and network aspects. In *High Assurance Systems Engineering, 2001. Sixth IEEE International Symposium on*, pages 95–105. IEEE, 2001.
- [7] J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *Software Engineering, IEEE Transactions on*, 24(2):125–136, 1998.
- [8] N. DeBardeleben, S. Blanchard, Q. Guan, Z. Zhang, and S. Fu. Experimental framework for injecting logic errors in a virtual machine to profile applications for soft error resilience. In *Euro-Par 2011: Parallel Processing Workshops*, pages 282–291. Springer, 2012.
- [9] T. J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. IBM Microelectronics Division, Nov. 1997.
- [10] K. Ferreira, K. Pedretti, P. Bridges, R. Brightwell, D. Fiala, and F. Mueller. Evaluating operating system vulnerability to memory errors. in *Proceedings of the International Workshop on Runtime and Operating Systems for Supercomputers*, 2012.
- [11] K. Ferreira, R. Riesen, J. Stearley, J. H. L. III, R. Oldfield, K. Pedretti, P. Bridges, D. Arnold, and R. Brightwell. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis, (SC'11)*, Nov 2011.
- [12] T. Garfinkel, M. Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, 2003.

- [13] W. Gu, Z. Kalbarczyk, R. K. Iyer, Z. Yang, et al. Characterization of linux kernel behavior under errors. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, pages 459–468, 2003.
- [14] U. Gunneflo, J. Karlsson, and J. Torin. 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*, pages 340–347. IEEE, 1989.
- [15] T. Hanawa, H. Koizumi, T. Banzai, M. Sato, S. Miura, T. Ishii, and H. Takamizawa. Customizing virtual machine with fault injector by integrating with SpecC device model for a software testing environment D-Cloud. In *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pages 47–54. IEEE, 2010.
- [16] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don’t strike twice: understanding the nature of DRAM errors and the implications for system design. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’12*, pages 111–122, New York, NY, USA, 2012. ACM.
- [17] A. Kleen. mcelog: memory error handling in user space. In *Proceedings of Linux Kongress 2010*, Nuremberg, Germany, September 2010.
- [18] P. Kogge and T. Dysart. Using the top500 to trace and project technology and architecture trends. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 28. ACM, 2011.
- [19] J. R. Lange, K. T. Pedretti, T. Hudson, P. A. Dinda, Z. Cui, L. Xia, P. G. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *IPDPS’10*, pages 1–12, 2010.
- [20] D. Li, J. S. Vetter, and W. Yu. 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, SC ’12*, pages 57:1–57:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
- [21] C. Lu and D. Reed. Assessing fault sensitivity in MPI applications. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 37. IEEE Computer Society, 2004.
- [22] S. Potyra, V. Sieh, and M. D. Cin. Evaluating fault-tolerant system designs using FAUmachine. In *Proceedings of the 2007 workshop on Engineering fault tolerant systems*, pages 04–04. ACM, 2007.
- [23] R. Sass, R. R. Sharma, and N. DeBardeleben. Towards a hardware fault-injection testbed to support reproducible resiliency experiments. In *Proceedings of the 2009 workshop on Resiliency in high performance*, pages 15–22. ACM, 2009.
- [24] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, June 2006.
- [25] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, 2007.
- [26] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1):012022, 2007.
- [27] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. *Communications of the ACM*, 54:100–107, February 2011.