StealthWorks: Emulating Memory Errors

Musfiq Rahman, Bruce R. Childers and Sangyeun Cho

Computer Science Department, University of Pittsburgh, Pittsburgh PA 15260 USA

Abstract. A study of Google's data center revealed that the incidence of main memory errors is surprisingly high. These errors can lead to application and system corruption, impacting reliability. The high error rate is an indication that new resiliency techniques will be vital in future memories. To develop such approaches, a framework is needed to conduct flexible and repeatable experiments. This paper describes such a framework, StealthWorks, to facilitate research on software resilience by behaviorally emulating memory errors in a live system. We illustrate it to study program tolerance to random errors and in the development of a new software technique to continuously test memory for errors.

1 Introduction

Today's computing paradigms and applications owe much of their success to the availability of inexpensive high-capacity main memory. The capacity of computer memories has increased dramatically: A current laptop might have four gigabytes of memory and a server might have tens or hundreds of gigabytes. With the horsepower unleashed by chip multiprocessors, the pressure on memory capacity will only increase as future applications operate on even larger data sets and new execution environments (e.g., virtualization) gain popularity.

The ability to inexpensively construct a many-gigabyte main memory is thanks to increased DRAM chip density (i.e., more memory bits fit in a fixed chip area). Although DRAM improvements are a key enabler to numerous computing advancements, there is a sinister side to the story. As DRAM density improves, the smaller bit cells are more susceptible to manufacturing variations and sensitivities that can cause the cells to malfunction under certain environmental conditions. These malfunctions cause application corruptions, increased service disruption, and decreased system reliability. While it is commonly believed that the probability of "soft errors", which result from background radiation flipping a bit (a single-event upset, or SEU), is increased in large main memories, "hard errors" are *also likely*. Transient and hard errors happen due to intermittent and permanent failures in the memory circuits, rather than external events.

Indeed, a recent study about memory reliability for Google's data centers showed that there are 25,000 to 70,000 errors per billion device hours per year and more than 8% of DRAM chips are affected each year [7]. Of these errors, transient and hard errors were common. This result defies conventional wisdom that application memory corruptions are only plagued by SEUs. An important conclusion from this study is error correction techniques are necessary to achieve

the best reliability. However, these techniques do not come without cost. A typical hardware approach to protect against SEUs is a "SECDED" code, which requires eight extra memory bits per 64-bit word to repair one bit error. Even with this 12.5% information redundancy, the memory is still susceptible to multi-bit errors, which necessitates more sophisticated schemes, like chipkill [1]. Unfortunately, even a simple SECDED scheme is too expensive (in power and dollars) for most machines. Thus, the use of a scheme, like chipkill, is even more unlikely in the competitive marketplace of commodity computing. Given this situation, run-time verification and testing techniques that improve resilience without increasing system cost will serve a vital role [3–5].

To develop new software techniques for both soft and hard errors, a flexible and efficient framework is needed to model, insert and monitor memory errors in an experimentally repeatable and controlled manner. We developed such a framework, called StealthWorks, that can inject and emulate soft and hard errors and observe their impact on applications and the system. StealthWorks is hosted in an actual computer system, and thus, is a fast vehicle for emulation and study of errors. In this paper, we demonstrate StealthWorks with two case studies, one for soft errors and the other for hard errors. The first study illustrates StealthWorks in evaluating application vulnerability to single-event upsets by randomly injecting single bit flips. The second study demonstrates StealthWorks for the development of a novel software-based approach to improve resilience of legacy and commodity systems that cannot use or afford hardware error correction methods for multi-bit hard errors.

2 StealthWorks

Figure 1 shows the components in StealthWorks, which are grouped into the System-under-Test and the User Interface. The System-under-Test emulates memory errors in an actual machine's memory. The User Interface is a remote client for interacting with the System-under-Test.

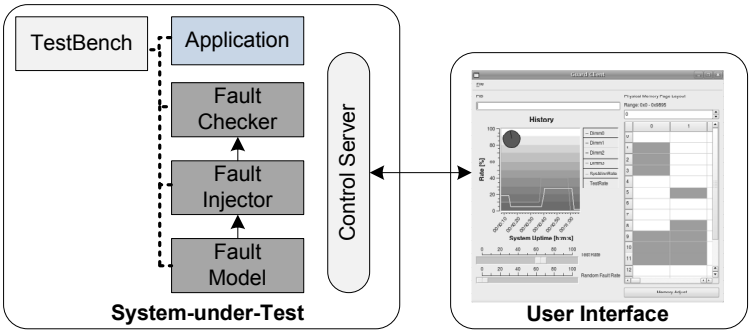

Fig. 1. StealthWorks framework

We focus on the System-under-Test, which has several modules. The Test Bench runs experiments through user scripts. The Fault Modeler determines the errors to inject and the Fault Injector emulates them. The Fault Checker intercepts program memory operations to check addresses for the presence of an error. Finally, the Control Server mediates communication between the System-under-Test and the User Interface. The Fault Modeler, the Fault Injector, and the Fault Checker form StealthWorks' core. Figure 2 shows how these core modules are organized and interact with one another. Next, we explain each module in the System-under-test.

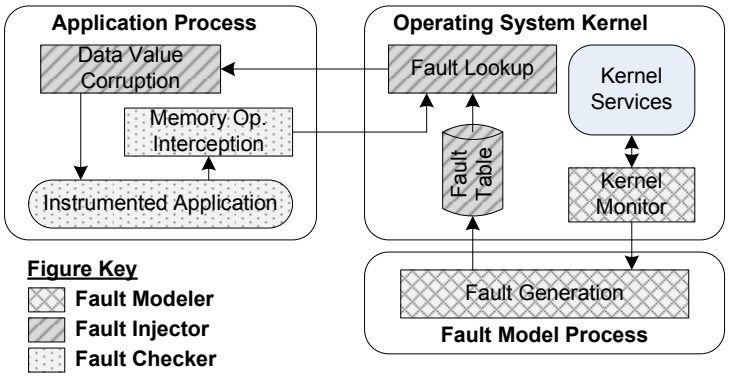

Fig. 2. Components comprising Fault Modeler, Fault Injector and Fault Checker

2.1 Test Bench

This component runs experiments through user-configurable scripts. An experiment includes the workload (how to run it), parameters for modeling memory errors, duration, and the statistics to collect. The scripts interact with the other StealthWorks modules.

2.2 Fault Modeler

The Fault Modeler hosts a user-specified *fault model*. This fault model determines what errors should be present in the memory. As shown in Figure 2, the Fault Modeler has two parts: Fault Generation and a Kernel Monitor. The user implements the fault model in Fault Generation with services provided by StealthWorks, including insert/delete an error, the Kernel Monitor, timers, event triggers, simulated system temperature, usage meters, age meters, and utility data types and functions. As shown in the figure, the fault model interacts with the Fault Injector through a database of errors, called the Fault Table. The fault model can insert and remove errors from the database.

The fault model indicates memory error addresses, error types, distribution, run-time causes, and how to corrupt data values for error types. A memory address with an error is indicated by a physical address since errors occur in hardware resources (i.e., physical pages). In addition to where to place errors, the fault model can optionally indicate how to corrupt data values. To corrupt a data value, the fault model marks a memory error address with a "corruption annotation". The annotation is user-defined. It can be used to specify the way to corrupt a value, such as a random bit flip, a stuck-at-0 error, etc. Fault Generation does not corrupt the actual data values because it cannot access these values directly. Instead, this is done by the Fault Injector (discussed next).

The Kernel Monitor collects information about system operation for the fault model. For instance, the Kernel Monitor can periodically sample memory utilization of different physical memory regions. Because the Kernel Monitor needs access to privileged information (e.g., page tables and allocation lists), it runs in the kernel and system calls are done (via wrapper functions) to interact with it.

We have implemented several example fault models. One model statically determines a fixed set of permanent errors, but it does not corrupt data values. It is useful to study how often a program touches an error location. An extended version can seed errors collected from a live system to create stress tests [5]. Another extension can corrupt data values for studying the inherent resilience of programs to hard faults [4]. A final example model considers the operating conditions identified in the Google study [7] as influential, including temperature, memory utilization, and device age, to determine when to insert errors.

2.3 Fault Injector

The Fault Injector emulates the errors generated by the Fault Modeler. It has three parts illustrated in Figure 2: the Fault Table, Fault Lookup, and Data Value Corruption. The Fault Table is a kernel hashtable of errors, which is indexed by physical memory word address. If an address is in the hashtable, it should be emulated by the Fault Injector as having an error. A hashtable entry records both the error type and the corruption annotation.

Fault Lookup is used by the Fault Checker to check whether a memory address has an error. It is invoked with a system call that passes the address. If an error is found, the system call returns a corruption annotation. If there is no error, a sentinel value is returned to indicate an error-free address. Because the Fault Checker intercepts memory operations, it operates on *virtual addresses*. Thus, Fault Lookup maps a virtual address to a physical one using the program's page table. The physical address is used to access the Fault Table.

Finally, Data Value Corruption changes data values, as indicated by the corruption annotation returned from Fault Lookup. For example, an annotation might indicate a stuck-at-0 fault for a particular bit. Data Value Corruption would set the stuck-at bit to 0 in the data value. Because it needs access to program data (instruction operands), it runs in the program's address space (user space). It is simplest and most efficient to perform the corruption in user space; it also avoids the difficulty kernel modification.

2.4 Fault Checker

The Fault Checker instruments program memory operations (instruction fetches and memory reads/writes) with dynamic binary translation (DBT) to gather an address trace. DBT can efficiently gather these address traces by optimizing instrumentation code in the context in which it is injected [2, 6, 8].

Each operation is intercepted to send the effective virtual memory address and access type (data/instruction, read/write, byte size) to the Fault Injector. Memory operations are rewritten to call an analysis payload shown in Figure 2 as "Memory Operation Interception". The analysis payload does a system call to inform the Fault Injector via Fault Lookup about the access. When Fault Lookup returns a corruption annotation, the Fault Checker invokes Data Value Corruption to determine the actual corrupted data value.

The current implementation of the Fault Checker can use either Pin [6] or Strata [8] DBT systems as the binary instrumenter. Pin offers easy-to-use interfaces to quickly craft the analysis payloads to corrupt data values. Strata provides lower-level facilities to inset and optimize the instrumentation code, which can lead to low instrumentation overhead [2, 8].

2.5 Control Server

This module mediates communication between the System-under-Test and the User Interface. It is a server that accepts connections from remote user interface clients. The Client Server understands commands and queries to control the System-under-Test and report information about an experiment. For example, it has a command to change the emulated temperature and a query to report application error rate.

3 Using StealthWorks

StealthWorks was developed in an ongoing project that aims to improve system reliability with software resiliency strategies. The framework has been used to run hundreds of experiments; we have found it to be robust and quite useful.

To illustrate StealthWorks' usage, we describe two studies. In the first study, we examine single-event upsets, which remain an important source of errors for deep submicron technology. In this study, we used StealthWorks to inject random SEUs into program data. We implemented a simple static fault model that determines ten random memory addresses to receive an SEU. The addresses are annotated with a "one-time bit flip" data value corruption. When a program is run, the Fault Checker determines whether a memory read operand touches an error address. If so, the Data Value Corruption flips a random bit in the word at the error address. Once an error is hit, it is removed. This experiment injects at most ten single-bit errors.

With this setup, we selected two example programs from SPEC2006 to determine whether they would run to completion with a correct result. We picked

tonto and *mcf* because they are expected to touch a large number of memory pages and will likely hit the inserted errors. We ran each program ten times with the same errors. Out of the ten runs, *tonto* crashed eight times and *mcf* crashed five times. From a closer inspection, it appears that *tonto*'s control flow is more data dependent and susceptible to errors than *mcf*.

In the second study, we used StealthWorks to evaluate a new online technique to continuously test memory. This second study illustrates StealthWorks for another memory error source – multi-bit permanent errors, which cannot be corrected by traditional memory error correction. We developed a software-only memory testing and scrubbing technique for computers that cannot support or afford hardware error correction techniques. Our approach constantly tests an application's virtual memory pages; it guarantees that every memory page has been tested within a specified time limit. Pages with permanent errors are retired from page allocation. To keep run-time performance overhead low, the test strategy uses a spare core in a chip multiprocessor to concurrently test memory with program execution. In developing this approach, we relied extensively on StealthWorks for development and experimental evaluation.

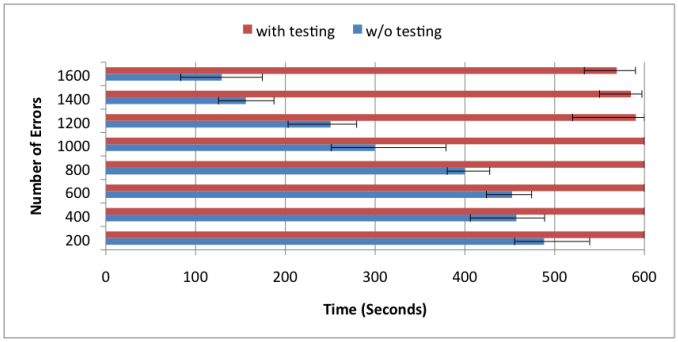

Fig. 3. Time to first fault from StealthWorks

Figure 3 shows one experiment that used StealthWorks to determine whether our online testing technique reduced application vulnerability. To measure vulnerability, we used StealthWorks to find the "time to first fault" (TTFF) of *mcf* when executed under a varying number of emulated errors in a 1-gigabyte memory. The figure compares our technique (bars labeled "w/testing") against a baseline without testing (bars labeled "w/o testing"). The X-axis is the amount of time until the first fault, and the Y-axis is the number of errors injected by StealthWorks. To determine a set of bars, we used the Monte Carlo method to do multiple trials since virtual-to-physical page mappings can change. Each trial was limited to the first ten minutes of *mcf*'s execution. The baseline (without testing) is also instrumented with StealthWorks to ensure that the execution times with and without testing are the same. This permits a fair comparison be-

tween TTFF for runs without and with testing. The figure shows the minimum and maximum of each trial as error bars.

As Figure 3 shows, without testing, *mcf* quickly encounters a memory location with an error. As expected, the time to the first fault decreased (more vulnerable) as the number of errors injected is increased. In comparison, our online testing approach let *mcf* tolerate a higher number of errors before the first fault was encountered.

The increased resilience comes at a small run-time cost; our continuous online testing strategy incurs a modest average 3% degradation in performance. This overhead comes from the additional memory pressure (on both the operating system kernel's memory allocator and the hardware memory subsystem) caused by the testing process. This experiment shows the benefit of StealthWorks – the framework permits development and study of software resiliency techniques with different scenarios.

4 Conclusion

Memory errors are surprisingly common and can lead to application failure. To mitigate errors, new resiliency techniques are needed. In this paper, we described an extensible framework, StealthWorks, that can be used to develop and evaluate methods to tolerate and correct memory errors. StealthWorks emulates memory errors in a live machine. We have found it to be a robust and useful framework for research on software resilience.

5 Acknowledgements

Christian DeLozier, Yang Hu and Yong Li implemented StealthWorks' control server and user interface client. This work is supported in part by the National Science Foundation awards CCF-0811295, CCF-0811352, and CNS-0702236.

References

1. T. J. Dell. A white paper on the benefits of chipkill - correct ECC for PC server main memory. In *IBM Microelectronics Division*, 1997.
2. N. Kumar, B. R. Childers, and M. L. Soffa. Low overhead program monitoring and profiling. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*, pages 28–34, 2005.
3. M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. SWAT: An error resilient system. In *4th Workshop on Silicon Errors in Logic - System Effects*, 2008.
4. M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and its implications on resilient system design. In *Architecture Support for Programming Languages and Operating Systems (ASPLOS'08)*, pages 265–276, 2008.

5. X. Li, M. C. Huang, and K. Shen. A realistic evaluation of memory hardware errors and software system susceptibility. In *USENIX Conference*, 2010.
6. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PDLI'05)*, pages 190–200, 2005.
7. B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *Internaetional Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09)*, pages 193–204, 2009.
8. K. Scott, N. Kumar, S. Velusamy, B. R. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *International Conference on Code Generation and Optimization (CGO'03)*, pages 36–47, 2003.