Evaluating application vulnerability to soft errors in multi-level cache hierarchy

Z. Ma
T.E. Carlson
W. Heirman
L. Eeckhout

Report 09.2011.1, September 2011

This work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT).

Evaluating application vulnerability to soft errors in multi-level cache hierarchy

Zhe Ma*[13], Trevor Carlson[23], Wim Heirman[23], and Lieven Eeckhout[23]

[1] Imec
Kapeldreef 75, 3000 Leuven, Belgium
mazhe@imec.be
[2] Ghent university
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{tcarlson,wheirman,leeckhou}@elis.ugent.be
[3] Intel ExaScience lab
Kapeldreef 75, 3000 Leuven, Belgium

Abstract. As the capacity of caches increases dramatically with new processors, soft errors originating in cache memories has become a major reliability concern for high performance processors. This paper presents application specific soft error vulnerability analysis in order to understand an application's responses to soft errors from different levels of caches. Based on a high-performance processor simulator called Graphite, we have implemented a fault injection framework that can selectively inject bit flips to different levels of caches. We simulated a wide range of relevant bit error patterns and measured the applications' vulnerabilities to bit errors. Our experimental results have shown the differing vulnerabilities of applications to bit errors in different levels of caches (e.g. the application failure rate for one program is more than the doulbe of that for another program for a given cache); the results have also indicated the probabilities of different failure behaviors for the given applications.

Keywords: Soft error, processor simulator, fault injection

1 Introduction

Two trends are observed in the ongoing development of the future generations of high performance computing systems: 1) the processor is fabricated with the CMOS processing technology that is constanly scaling down and 2) commodity high-end processors, rather than customized processors, are more widely employed to reduce the total cost. The combination of these two trends makes the reliable working of hardware much more difficult [16]. Unreliable hardware behaviors can be roughly split into hard errors and soft errors. While a hard error is a persistent hardware failure, a soft error is a transient failure and hence harder to detect and analyze.

* This work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT).

Soft error is a well known reliability concern. However, it is known that applications have intrinsic masking of soft errors (error rate derating) [12, 14]. Thus many bit errors are filtered out and are not visible at the application level. To find a cost-effective soft error mitigation strategy, it is necessary for system designers to have fault injection tests in order to obtain a good estimate of application level soft error rate. How to perform error injection is an important topic in a computer system reliability analysis. Different approaches are developed and can be roughly grouped into the following categories:

hardware built-in injection These approaches [10] depend on specific units built into hardwares, usually in the form of Built-In Self-Test. They are only available in existing hardware.

accelerated hardware emulation These approaches [13, 6] employe a detailed model of a target hardware, and simulate this model on an accelerator (usually a FPGA platform). The requirement of a detailed model usually means this is performed already at a later stage of a design workflow.

software based injection These methods [5, 2] run an altered software application natively on its target processor. The modifications inside the application can inject faults to the application visible memory addresses.

simulated hardware based injection These approaches [17] employe a high level model of a target hardware and simulate this model on a general purpose computer. The high level model is usually less accurate but is easier to modify in order to model innovative features in a yet to be implemented processor. This high level model is usually fast enough to test relevant software applications and hence to have more application specific estimate of the effect of faults.

We choose simulated hardware based injection because 1) it has relatively low development and deployment cost and 2) it provides flexibility to explore novel processor architectures. Thanks to a high performance processor simulator called Graphite [11], we can efficiently simulate an application's responses when soft error induced bit errors take place in a processor's cache hierarchy.

Previous studies [5, 2] investigated the scientific applications derating ratio to bit errors from the (off-chip) memory; they did not compare the effects of soft errors from a multi-level cache hierarchy. However, as its capacity increases dramatically, cache has become a major source of soft error induced bit errors on high end processors [1]. To have an accurate estimation of an application's response to soft errors in cache, it is necessary to have an application's access patterns to different levels of caches. However, application cache accesses patterns are difficult to determine from a static analysis, especially for multi-threading applications.

In this paper we present a simulation based analysis that can better reveal realistic cache access patterns. We then perform fault injection directly into the caches when they are accessed by simulated applications. In this way we can have the best approximation of the soft error's effect to an application; and can distinguish this effect from different levels of caches.

In the rest of this paper we first describe the processor simulator that we used for fault injection and how we can conduct fault injections in the cache hierarchy (Section 2); then we present our motivation for various bit error patterns used in the fault injection (Section 3); next, we describe the experimental process (Section 4) and present the experimental results (Section 5); we finally discuss the collected results and draw some conclusions (Section 6).

2 Graphite simulator and bit error injection

We want to obtain the applications response to cache bit errors by directly observing the simulated results after the fault injection. This simulation should faithfully execute the target application's instructions; and the error bits are only injected to the instructions that are accessing the specific cache that we select for injection during the specified time period. We have chosen a processor architecture simulator called Graphite developed at MIT [11]; and used the extensions made by University of Ghent [3] .

2.1 Fast processor simulation

Graphite is a high performance processor simulator. Based on the dynamic instrumentation tool called Pin [9], Graphite can directly dynamically re-compile and execute instructions from the x86 executable binary of a target application. This feature allows us to easily evaluate scientific applications that are already compiled for x86 target machines. Also, Graphite can be configured with different processor parameters about cache, such as levels, capacities, latencies and replacement strategies. This is especially true after we applied the extension described in [3]. One thing to keep in mind is that Graphite is only for application space simulation. All system calls in the simulated applications are intercepted and handled by Graphite. Although these emulated system calls can affect the insight into OS kernel's responses to fault injection, our experiments are not greatly affected because we are mainly interested in evaluating scientific computing applications where most processor time is spent in application space.

2.2 Cache error injection

We use a partly modified Graphite that allows us to more flexibly configure the cache hierarchy of a modeled processor. Based on a user defined cache model, this Graphite simulator can determine for each memory access if it is hit in a specific cache. If it is a hit, Graphite can also determine which cache line is accessed.

Our Graphite can read in a separate configuration file with information about the location and time in the cache hierarchy to insert a bit flip. An illustration of such a configuration file is shown below.

```
...
[fault_injection_model/L3]
start_cycle = 12022450
total_faults_nr = 1
err_bit_nr = 2
multi_byte_upset = false
...
```

A random configuration generator has been made to generate a large number of fault injection configuration files. While the injection location and time are randomized by the generator, the bit error pattern (see Section3) in the configuration files is given as an input to the generator. Because soft error is a rare event and is unlikely to hit the same application more than once during its execution with the input size used in our simulation, we only inject one soft error with a single injection configuration file. One simulation is launched for each individual fault injection configuration file. During every simulation, the Graphite simulator flips the error bits that are specified in this configuration file providing that a cache access at the selected cache level takes place during the specified time period in the configuration file. The injected error bits stay as long as they are not overwritten or flushed out of the cache.

3 Multiple cell upset and bit error patterns

When processing technology scales down, the probability of Multiple Cell Upset (MCU) increases dramatically [15, 7]. We simulate the 2-, 4- and 6-cell upsets which are known to increase in SRAMs and are hard to detect and/or correct by a conventional ECC mechanism.

Because caches can use physical bit interleaving, A MCU cannot be directly translated into a Multiple Bit Upset (MBU). Instead, how a MCU is translated into a MBU depends on the physical bit interleaving implemented in the cache array. Due to high energy overheads associated with physical bit interleaving in large cache arrays [8], we only simulated physical bit interleaving for L1 and L2 caches (with different degrees of interleaving):

L1(D+I) 4-way interleaving For both L1 data and instruction caches we assume a 4-way physical bit interleaving. Then the 2-, 4- and 6-cell MCUs are translated into single bit upset and 2-bit upset in two consecutive bytes.

L2 2-way interleaving For L2 cache the 2-, 4- and 6-cell MCUs are translated into single bit upset, 2- and 3-bit upset in two consecutive bytes.

L3 no interleaving When no physical bit interleaving is present, the 2-, 4- and 6-cell MCUs are directly translated into 2-, 4, and 6-bit upset in a single byte.

4 Simulation setup

4.1 Applications

We use the SPLASH-2 benchmarks [18] as our applications for the fault injection simulation. SPLASH-2 benchmarks have a variety of scientific computing programs that are widely used with processor simulators. Because computational kernels usually account for the main execution times of scientific computations, we only present the results from three computational kernels from SPLASH-2 suite in this paper. The selected kernels are a sparse matrix factorization (Cholesky), a fast fast Fourier transform (FFT) and an integer radix sort (Radix). The problem sizes used for each benchmark are listed in Table 1. All benchmarks are compiled by GCC in 64-bit mode, with the -O3 optimization.

Table 1. Simulated SPLASH-2 benchmarks and problem sizes

Benchmark	Program size
Cholesky	tk25.O
FFT	256K points
Radix	256K integers

4.2 Simulation parameters

We simulate our application with a processor model that resembles the Intel Xeon Dunnington processor (X7460). An X7460 has six cores; each core has private L1 data and instruction caches (32KB + 32KB). Every two cores share a L2 cache with the size of 3MB. All cores share a single L3 cache with the size of 16MB.

Because of the large numbers of cache accesses, it is too expensive to do an exhaustive fault injection at each cache access during simulation. Hence we apply statistical sampling techniques to estimate the responses from simulated applications. Suppose a cache is accessed for X times during the execution, the total population space for this cache is X. What we need to determine is a sampling size x that is (much) smaller but can still give a reasonable estimation of the probabilities associated with different application responses. For applications that run for an extended period of time, the cache access numbers are very large (up to several hundreds of millions for L1 caches). Thus we consider the application responses follow the normal distribution. Based on the sampling theory [4], and the baseline profiling results, we can calculate the sampling number that we need for each cache is around 500. Such a sampling size can obtain an error margin less than 5% with a statistical confidence interval of 95%. Therefore, we repeat the fault injection for each individual cache for 500 times (i.e., with 500 different random fault injection files for each cache when simulating an application).

5 Simulation results

We first profile our target applications by running them on the simulator without fault injection. These profiling results are called baseline results. We use baseline results 1)to setup the normal execution time for each application and 2) to collect the correct output if applicable. We repeat each simulation for 10 times and obtain consistent profiling results as shown in Table 2. Note that as a Dunnington processor has six L1 I/D caches and three L2 caches, the access numbers in the table are averaged access numbers for each individual L1 and L2 cache. The total execution cycle is the largest execution cycle number among six cores.

Table 2. Baseline profiling information

Benchmark	L1-I cache accesses	L1-D cache accesses	L2 cache accesses	L3 cache accesses	Total exec. cycles
Cholesky	147344731	48832936	1776513	1147177	296651192
FFT	34513627	7581382	368357	572405	54740721
Radix	12408179	1878307	150693	62016	16366277

In the second step we simulate the applications with the randomly generated fault injection configuration files. We have observed different responses from the simulations with injections. In the rest of this section, we first describe four different responses caused by fault injections; then we compare the responses from different benchmarks for fault injections from each cache level.

5.1 Application response to fault injections

The four types of responses that we observed from the applications are listed below. These responses have different frequencies in our simulations. We summarize the occuring percentages of each response in the Table 3, 4,5,6. Also, we compared the vanished fault percentages of all applications in Figure 1. As shown in this figure, applications have different levels of vulnerabilities to injected bit errors in different caches.

fault vanished This is a response in which a target application finishes its execution successfully.

application crash This is a response in which a target application aborts its execution. This usually happens with an error return value from the application or from the libraries (such as glibc) used by the application. In most cases the application gets a segmentation fault.

application hang Because each run of simulation for a given application takes the same input data, and the applications have no intrinsic reasons to show very different execution times, we consider an application becomes a hanging application if it has been running for three times as long as its baseline execution time.

silent data corruption This response is defined as a target application finishes its execution successfully without exceeding three times of its baseline execution time. But the final output cannot pass the correctness test. We only perform the test for FFT in this paper.

Table 3. Applications responses percentages for L1 instruction cache fault injection simulations; as explained in Section 3, two types of bit error patterns are simulated: 1-bit upset in single byte(SBU1) and 2-bit upset in consecutive bytes(MBU2)

		Cholesky	FFT	Radix
SBU1	Crash	10.5	12.1	11.2
	Hang	1.1	1.9	1.5
	SDC	–	0.1	–
	Vanished	88.4	85.9	87.3
MBU2	Crash	10.8	11.6	11.5
	Hang	1.2	2.2	1.5
	SDC	–	0.3	–
	Vanished	88.0	85.9	87.0

Table 4. Applications responses percentages for L1 data cache fault injection simulations: 1-bit upset in single byte (SBU1) and 2-bit upset in consecutive bytes (MBU2)

		Cholesky	FFT	Radix
SBU1	Crash	12.1	7.9	5.0
	Hang	7.3	4.2	2.2
	SDC	–	1.5	–
	Vanished	80.6	86.4	92.8
MBU2	Crash	16.0	9.0	6.7
	Hang	5.2	4.1	2.3
	SDC	–	1.5	–
	Vanished	78.8	85.4	91.0

6 Conclusions

We present a cache fault injection framework based on a fast processor simulator. Running several scientific computing programs on this simulator with injected cache bit errors, we have observed various responses from the simulated programs with different probabilities. All programs show that a large percentage of errors are filtered out and hence invisible at the application level. For the errors that do

Table 5. Applications responses percentages for L2 cache fault injection simulations; three types of bit error patterns are simulated: 1-bit upset (MBU1), 2-bit upset (MBU2) and 3-bit upset (MBU3), all three cases in consecutive bytes

		Cholesky	FFT	Radix
MBU1	Crash	8.1	6.9	8.0
	Hang	2.8	1.0	3.0
	SDC	–	0.5	–
	Vanished	89.1	91.6	89.0
MBU2	Crash	9.0	7.5	8.9
	Hang	4.2	1.7	4.2
	SDC	–	0.9	–
	Vanished	86.8	89.9	86.9
MBU3	Crash	9.4	7.6	9.5
	Hang	4.9	1.9	4.9
	SDC	–	0.9	–
	Vanished	85.7	89.6	85.6

Table 6. Applications responses percentages for L3 cache fault injection simulations; three types of bit error patterns are simulated: 2-bit upset (SBU2), 4-bit upset (SBU4) and 6-bit upset (SBU6), all three cases in a single byte

		Cholesky	FFT	Radix
SBU2	Crash	5.0	8.8	4.8
	Hang	1.0	3.0	1.1
	SDC	–	0.7	–
	Vanished	94.0	87.5	94.1
SBU4	Crash	5.2	11.4	5.6
	Hang	1.3	4.8	1.4
	SDC	–	1.1	–
	Vanished	93.5	82.7	93.0
SBU6	Crash	5.5	13.0	5.7
	Hang	1.2	4.9	1.5
	SDC	–	5.1	–
	Vanished	93.3	77.0	92.8

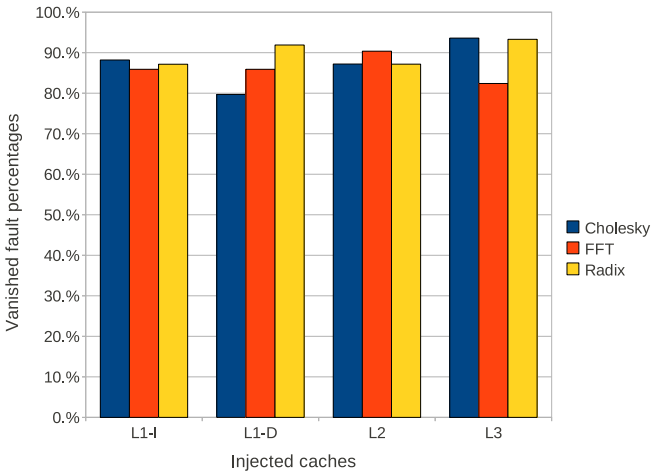

Fig. 1. Comparison of consolidated percentages of vanished fault for all applications; vanished fault percentage for each application is the average of its vanished fault percentages, assuming each bit error pattern has the same occuring probability

cause an application failure, application crash is the most likely type of failures (4.8% – 16.0%); while silent data corruption, though relatively rare, is still not negligible (up to 5.1% for FFT). Moverover, our results indicate that different programs have different levels of vulnerability to bit errors injected in different caches (e.g., 6.4% application failures for Cholesky *vs*. 17.6% for FFT in L3 cache fault injection simulations). These results suggest that the benefits of protecting an individual cache depends on the application program that is running on this processor.

References

1. Robert Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, 2005.
2. Greg Bronevetsky and Bronis R. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *SELSE*, 2007.
3. Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Exploring the level of abstraction for scalable and accurate parallel multicore simulation. In *SC*, 2011.
4. William G. Cochran. *Sampling Techniques, 3rd Edition.* John Wiley, 1977.
5. Charng da Lu and Daniel A. Reed. Assessing fault sensitivity in MPI applications. In *SC*, page 37. IEEE Computer Society, 2004.
6. Jean-Marc Daveau, Alexandre Blampey, Gilles Gasiot, Joseph Bulone, and Philippe Roche. An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip. In *IRPS*, pages 212–220, 2009.

7. David Heidel, Paul Marchal, and *et al.* Single-event upsets and multiple-bit upsets on a 45nm SOI SRAM. *IEEE TRANSACTIONS ON NUCLEAR SCIENCE*, 56(6):3499–3504, 2009.
8. Jangwoo Kim, Nikos Hardavellas, Ken Mai, Babak Falsafi, and James C. Hoe. Multi-bit error tolerant caches using two-dimensional error coding. In *MICRO*, pages 197–209, 2007.
9. Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.
10. T. M. Mak, Subhasish Mitra, and Ming Zhang. DFT assisted built-in soft error resilience. In *IOLTS*, page 69, 2005.
11. Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, pages 1–12, 2010.
12. Shubhendu S. Mukherjee, Christopher T. Weaver, Joel S. Emer, Steven K. Reinhardt, and Todd M. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO*, pages 29–42. ACM/IEEE, 2003.
13. Pradeep Ramachandran, Prabhakar Kudva, Jeffrey W. Kellington, John Schumann, and Pia Sanda. Statistical fault injection. In *DSN*, pages 122–127. IEEE Computer Society, 2008.
14. Sonny Rao, Pia Sanda, Jerry Ackaret, Adrian Barrera, Jorge Yanez, and Subhasish Mitra. Examing workload dependence of soft error rates. In *SELSE*, 2008.
15. Franz X. Ruckerbauer and Georg Georgakos. Soft error rates in 65nm SRAMs–analysis of new phenomena. In *IOLTS*, pages 203–204, 2007.
16. Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *DSN*, pages 249–258, 2006.
17. Nicholas J. Wang, Michael Fertig, and Sanjay J. Patel. Y-branches: When you come to a fork in the road, take it. In *IEEE PACT*, pages 56–, 2003.
18. Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.