2015 IEEE International Conference on Cluster Computing

Understanding the Propagation of Error Due to a Silent Data Corruption in a Sparse Matrix Vector Multiply

Jon Calhoun, Marc Snir, Luke Olson, and Maria Garzaran
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
Email: {jccalho2, snir, lukeo, garzaran}@illinois.edu

***Abstract*—With the rate of errors that silently effect an application's state/output expected to increase in future HPC machines, numerous mitigation schemes have been proposed, but little work has been done investigating why these schemes detect some error while other is masked. This paper investigates how silent data corruption (SDC) propagates through a sparse matrix vector multiply (SpMV), a fundamental HPC computation kernel. We discover that analyzing the mathematics of the SpMV limits understanding of SDC propagation. We achieve a more complete understanding by investigating how SDC propagates in a SpMV as it is expressed in machine instructions.**

Keywords*—*Silent Data Corruption; Error Propagation

I. Introduction

As the machine complexity of current and future HPC systems increases due to an increase in component count [4] and technologies such as near-threshold-voltage [3] machine errors both fail-stop and silent are expected to increase. Errors that silently corrupt application state/output are of particular concern. To mitigate such affects resiliency measures have been devised and evaluated. Most research presents some combination of precision, recall, and overhead of their method. Understanding how and why some silent errors are detected while others lead to SDC remains not well understood and is often neglected in works of this type. Some work does look in detail at placing bounds on how much bit-flips can deviate floating-point computation [2]. A more fundamental understanding of silent error propagation will assist in accessing the application's ability to withstand silent error and help application developers with detection placement.

This paper makes the following contributions by

- tests how mathematical intuition of error propagation inside a SpMV compares to a real world scenario using fault injection experiments;
- promotes further study into how SDC emerges and propagates though HPC applications.

II. Example Propagation

Sparse matrix-vector multiplies (SpMV) are fundamental to a large body of HPC applications. Understanding how SDC emerges and propagates inside this HPC kernel is crucial in addressing how error propagates in a large code. Figure 1, illustrates SDC propagation due to a corrupted input vector element. Analyzing the algorithm shows dot products between the rows of the matrix and the input vector. From this, we see the corrupt vector entry at position i will be accessed once for each non-zero entry in column i of the matrix. Therefore, propagation is worse for denser matrices. With a fully dense matrix or column, a single SpMV would completely corrupt the result. Similar analysis can be done on if a matrix entry is corrupted, but it reduces to a form similar to Figure 1.

Fig. 1. Propagation of SDC via a sparse matrix-vector multiply.

III. Tracking Propagation

In order to track SDC propagation, we compare loads and stores of the application to a known good configuration. We concern ourselves only with loads and stores because when the incorrect value is present in the memory, it has been *committed* to the state typically thought of by application programmers. To generate a comparable sequence of loads and stores, we duplicate all instructions except branches labeling the new instructions as *Faulty* and the originals as *Golden* interleaving their execution. Faulty instructions will create and consume their own data, but rely on the Golden instruction's traversal of the control flow graph. One instruction from Faulty will suffer a fault during execution, and through our replication we compare the validity of loads and stores logging the propagation of SDC. In addition, we determine control flow divergence based on branching conditionals.

IV. Experimental Results

A. Testing Methodology

1) Fault Injection: We use the LLVM based fault injector FlipIt [1] to inject faults. Our fault model considers only transient hardware errors that propagate to the application. Because register files and memories such as SRAM and DRAM commonly protected ECC or Chipkill, we do not inject bit-flip errors in these locations. Instead we consider faults arising in processor logic that manifests as a single bit-flip in the instruction's result.

978-1-4673-6598-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CLUSTER.2015.101

IEEE computer society

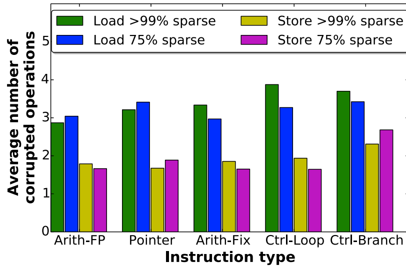

Fig. 2. Average number of loads/stores corrupted in a SpMV due to a single injected fault.

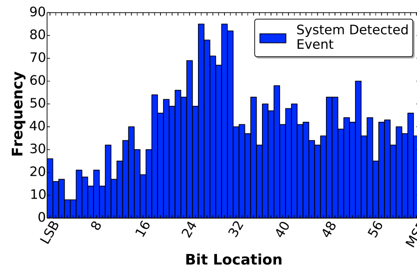

Fig. 3. Frequency of system detectable events – (e.g. segfaults) at various bit locations

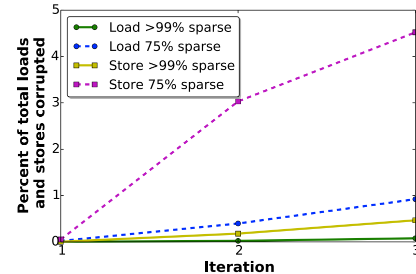

Fig. 4. Propagation of SDC by iterative method style reuse.

TABLE I. BREAKDOWN OF INSTRUCTIONS BASED ON TYPE.

	Floating-Point	Fixed-Point	Pointer	Control-Branch	Control-Loop
Fault Injected Percent	18.7%	29.7%	45.0%	3.4%	3.2%
Dynamic Instruction Percent	18.9%	29.5%	45.0%	3.4%	3.2%

B. Results

To test the mathematical intuition of SDC propagation in a SpMV, we conduct $3,000$ fault injection experiments each on a matrix that is $> 99\%$ sparse and a matrix that is 75% sparse. Table I shows a breakdown of the instruction classifications we inject into. These percentages mirror those of the dynamic instruction count differing by $< 1\%$ due to our experiment count. From this table, we see evidence that mathematical intuition does not fully account for SDC propagation as 18.6 of instructions executed/injected into are floating-point. The reasoning for the high number of non floating-point operations is due to the spare matrix representation requiring several levels of indirection before the floating-point data is accessed. Thus, understanding what happens when SDC occurs greatly depends on how non floating-point instructions propagate it.

Figure 2 shows the average number of loads and stores corrupted during a SpMV due to a single fault injected into a given instruction type for both of our matrices. As we see there is not much of a difference between the two matrices which is to be expected as there is no reuse of the corrupted vector. The average number of loads corrupted is higher than stores due to the SpMV needing several levels of indirection before accessing the data to compute and store.

By checking branching conditions, we discover control flow divergence occurs in 19% of experiments; often resulting in a premature exit of the SpMV. The most damaging instruction types are fixed-point arithmetic and pointers due to their ability to load unaligned/incorrect data. In one case, a corrupted pointer lead to 4391 corrupted loads and 917 stores. These instruction types are also self detecting as bit-flips in the most significant bits of data types i.e., 4 byte integers and 8 byte pointers, easily lead to segmentation faults, Figure 3.

Although there is not a difference in SDC propagation looking at the number of loads and stores corrupted for a single time step, if we feed the corrupted result into the SpMV in iterative method style reuse, Figure 4, a difference dramatically emerges. In this Figure, we see the percent of total loads/stores corrupted at a given iteration. As the vector is reused, SDC propagates due to the mathematical intuition outlined in Figure 1. If we increase the number of iterations, masking starts occurring leading to a reduction in the percent of loads and stores corrupted. This masking is due to convergence of iterative methods schemes. In this case, power iteration.

V. CONCLUSION

Analyzing the mathematics of HPC applications is a good first step at understanding how SDC propagates, but does not account for non floating-point instructions and their interactions. Fault injection experiments show that the non floating-point instructions have the possibility of corrupting a large number of loads and stores. These same instructions also can lead to system detectable events that do not permit propagation. Verifying branching conditions can detect early termination of the SpMV enforcing each component is computed.

ACKNOWLEDGMENT

This work was sponsored by the Air Force Office of Scientific Research under grant FA9550-12-1-0478. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI–0725070 and ACI–1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

REFERENCES

[1] Jon Calhoun, Luke Olson, and Marc Snir. FlipIt: An LLVM based fault injector for HPC. In *Proceedings of the 20th International Euro-Par Conference on Parallel Processing (Euro-Par '14)*, 2014.

[2] James Elliott, Frank Mueller, Miroslav Stoyanov, and Clayton Webster. Quantifying the impact of single bit flips on floating point arithmetic. Technical report, Oak Ridge National Laboratory, August 2013.

[3] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (ntv) design: Opportunities and challenges. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1153–1158, New York, NY, USA, 2012. ACM.

[4] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2):127 171, May 2014.