2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops

SDC is in the Eye of the Beholder: A Survey and Preliminary Study

Bo Fang*, Panruo Wu‡, Qiang Guan†, Nathan DeBardeleben†, Laura Monroe†,
Sean Blanchard†, Zhizong Chen‡, Karthik Pattabiraman*, Matei Ripeanu*
*Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, Canada
Email: {bof,karthikp, matei}@ece.ubc.ca
†Ultrascale Systems Research Center
Los Alamos National Laboratory[2], Los Alamos, USA
Email: {ndebard,qguan,lmonroe,seanb}@lanl.gov
‡Department of Computer Science and Engineering
University of California Riverside, Riverside, USA
Email: {pwu011,chen}@cs.ucr.edu

***Abstract*— Silent data corruptions (SDCs) are one of the most critical issues in modern HPC systems, as they are "silent" by definition and raise no warnings to users and application developers that a calculation has been corrupted. A significant amount of effort has been made to characterize, detect, and tolerate SDCs. However, current approaches do not share the same understanding of SDC, hence it is not only difficult to evaluate their effectiveness, but also to compare with each other. This position paper argues that SDCs should be discussed at each layer of the system and are confined within the goal of the approach. We provide a preliminary result to differentiate data corruptions across system layers, and show that application-specific correctness checks can tolerate about 50% of the errors that appear in the application output.**

Keywords: Silent Data Corruption (SDC), Application-specific Correctness

I. INTRODUCTION

Hardware faults are one of the major issues future HPC systems face [1], [2], and are estimated to become an even more pressing issue at exascale [3]. Silent data corruptions (SDCs) are considered as one of the most serious outcome types of hardware faults for HPC systems and applications. For example, the computational results of an application can be corrupted due to hardware faults and returned to the users without any notification. Given the application domains massive supercomputers are used for (climate, energy, national security, scientific discovery, etc.) and the real-world implications of making decisions based on corrupted outputs, it is easy to see how dealing with correctness is challenging. While most HPC applications use checkpoint/restart to recover from *detectable* failures, this technique cannot cover undetectable, silent corruptions (SDC). Protecting a system and/or an application from SDCs requires additional techniques which incur overheads in the hardware (such as higher voltages, additional die area used for redundancy), in the software (such as algorithm-based fault tolerance and invariant checking), or as a combination of both.

There is a long list of studies exploring solutions to mitigate the impact of hardware faults and focusing on SDCs. It is typical for these studies to balance between effectiveness (detection and coverage) and the overhead incurred. This is particularly demonstrated in the scope of software-based fault tolerance approaches such as [4]–[16]. However, although these studies all use "SDC" to refer to silent deviations from the data's correctness, several categories can be observed in terms of different levels of correctness. For example:

1) An SDC is classified as whether the application output is different from the golden run [4], [7]–[9], [11]. A "golden run" is a previously known to be correct execution of the application with a set of output information that are checked for bit-for-bit equality (Class I).
2) An SDC is classified as whether the application output does not pass an application-specific correctness check [6], [10], [12], [17] (Class II).
3) An SDC is classified as whether the application state (i.e., not the final output) violates program-level properties such as a numerical threshold or an invariant check [5], [13]–[16] (Class III).

Importantly, approaches that focus on different program data may result in wildly different estimation of their occurrence (thus leading to different risk estimated). Additionally, since SDCs are being classified in different ways, it is not obvious that how one can directly compare approaches that use different classifications. For example, if a simple application-specific correctness check only considers the correctness of a floating-point result up to 3 digits after the decimal point, then it would be more tolerant than a general method that performs binary comparison on the program outputs. This can also impact the evaluation of an application's error detection efficiency. For example, both SDCTune [6] and SWIFT [9]

[2]A portion of work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S. Department of Energy contract DE-FC02-06ER25750. The publication has been assigned the LANL identifier LA-UR-16-22870.

DOI 10.1109/DSN-W.2016.46

use heuristic-based instruction-level duplication and program analysis to detect errors (i.e. data corruptions), but the former evaluates SDC detection coverage using an application-level specific correctness check, if one is provided (Class II), while the latter simply checks if the output differs from the golden run (Class I). This leads to a gap in SDC detection coverage.

Therefore, in this paper, we confine ourselves to discussing silent data corruptions from architectural states (i.e. memory) that directly expose program data to the application specific layer. We provide a classification of prior studies in the context of the whole system stack. We believe that the classification can help researchers from different backgrounds (i.e working at different layers of the system stack) be aware of the boundary of each layer and reason about the error resilience characteristics of systems/programs given the SDCs that occur at a specific layer, which can eventually enable cross-layer fault tolerance analysis when protecting the system from SDCs.

II. Survey: Related Work and Classification

Figure 1 shows a simplified version of components in a typical computer system that includes the memory system, the operating system and the executed applications. While the microarchitectural layers certainly have corruptions that do not propagate to higher levels of the system, in this work, we only consider the faults that transfer from this layer into the memory layer. The shrinking size of the squares along the data path in the computing system implies the inherent fault masking from lower layers to the top, which relates to the application-specific correctness verification. In this section, we discuss some studies that provide important insights about the fault tolerance characterization and mechanisms for each layer along the data path.

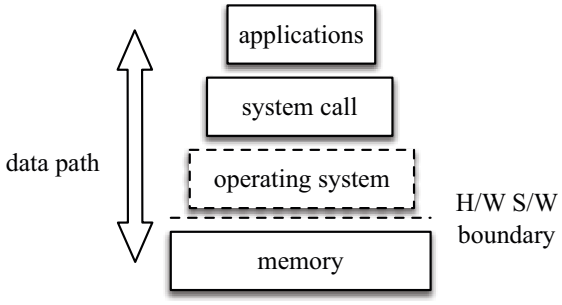

Fig. 1: A software view of the data path in the computing system. The operating system layer is marked with a dotted line because techniques that have operating system support are outside the scope of this paper. Memory is the interface between the architectural states and applications.

Memory layer Checking memory states for data corruption is an effective way to see if a hardware fault escapes from microarchitectural masking and affects the program execution. Recently, Ashraf et al. [18] proposed a fault propagation model to track transient hardware faults within a process and across MPI processes until they reach the memory units. Their model computes both the potentially-corrupted value when the memory gets affected by a fault and its pristine value, and compares them when the value gets stored in the memory. Another study performed by Wang et al. [19] compares all the architectural state (memory, PC and register) between the reference simulation and a fault-injected simulation of a program to determine the effects of a fault that flips a program branch.

System call A system call is the channel for applications to communicate with the underlying operating system, and also for the operating system to pass information to the application. Therefore, monitoring system calls can be a direct approach to check if a fault propagates to program-visible states from the memory layer. Shye et al. [20] propose a software-centric technique that replicates processes of a program to perform fault detection and recovery. In particular, it emulates the system calls on redundant processes, and compares the output and parameters of emulated system calls as one of the fault detection mechanisms. This process level redundancy (PLR) technique demonstrates the effectiveness of the fault tolerance approach on the system call level to provide an accurate conservative error detection mechanism.

Application layer - output Scientific applications report their final computational results in I/O devices and/or in files. To check if a hardware fault affects the output of an application, many studies such as [4], [8], [11], [21], [22] use binary comparison of outputs between the fault-free run and faulty runs as a common practice. That is, any difference in bits of two outputs is considered as a deviation from the correct output.

Application layer - application-specific correctness check Another approach is where applications define an "unacceptable result" in a way that the final output or even intermediate states/outputs do not pass some pre-defined checkers. These checkers, depending on the application, can be numerical thresholds, physical properties or rules derived from algorithm-invariants. A typical and common example of such acceptance test is for generic floating-point calculations where fault masking can occur for certain bit positions and operations [23]. For example, High Performance Linpack (HPL) solves a linear system of order n using LU decomposition [24] and tests the correctness of the result by checking the residual of the linear system as a norm-wise backward error based on [25]. Such residual calculation potentially may not easily distinguish transient errors from round-off errors that occur in floating point computation. A more application-specific example of the correctness check is the Livermore Unstructured Lagrange Explicit Shock Hydro*(LULESH)* [26] which represents a typical numerical algorithm and data motion in many scientific applications. LULESH defines three metrics and their correctness requirements [26]: a) the number of iterations should be exactly the same as expected, b) the final origin energy after the simulation should be correct to at least 6 digits, and c) the three measures of symmetry should all be 10^{-8} or smaller in

double precision floating point.

In summary, hardware faults can affect different layers in the system, but whether or not they affect the system as a whole depends on what and how the impact is made on each system layer by the fault. It is possible to observe mismatches during fault tolerance characterization across layers because of the different fault masking characteristics. Understanding this effect is significantly important for HPC systems and applications. For example, error detection mechanisms that detect erroneous states of a program can use the understanding of such cross-layer fault masking to improve the overall efficiency by checking if a detected error is actually going to lead to an application unacceptable result. Similarly, checkpoint/recovery schemes based on anomaly data monitoring can also determine if a roll-back is needed by predicting the final outcome of an intermediate data corruption.

III. Experimental Results and Discussion

In this section, we perform a preliminary study to investigate how much data corruption changes across layers. Our hypothesis is that the amount of hardware faults that propagate from the memory layer to the application layer in the system is likely to reduce across the layers. To test our hypothesis and study how many errors are masked at each layer of the system, we conduct fault injection experiments on three HPC applications which are part of the DOE (Department of Energy) mini-apps, namely *HPL, LULESH* and *CLAMR [27]*. For cross-layer analysis, we compare the final states of the memory layer and the application layer between a fault-injected run and the fault-free run to check if any difference is caused by fault injection[1].

First, we provide detailed information about the fault injection experiments, and then we discuss the results.

Fault model: We model faults that affect the architectural states of a program such as register files and memories. In particular, we randomly select one instruction during execution of a program and inject a single-bit flip fault into the destination register of that instruction, or the register that stores the memory address. We only consider faults that are read by the program as activated faults - this is similar to prior work.

Outcome: We categorize the final outcome of a program after fault injection into benign (correct/acceptable results), crash, hang and SDC (in this paper, we consider data corruptions that cause the application to fail the application specific correncess check). In addition, we also measured how many times the injected faults cause data corruptions in the program output. Table I defines the output and the application-specific check used in this paper for each application. To determine if a program output is corrupted, we use binary comparison as discussed earlier.

Experimental setup: We run all applications in sequential mode with a fault injection tool called PINFI [28], which is based on the Intel dynamic instrumentation tool PIN [29]. For each benchmark, 3,000 activated fault injection runs are conducted to gain a statistical significant estimation (we calculated error bars at the 95% confidence interval).

[1]We combine the output of the write() system calls and the *stdout* to form the program output

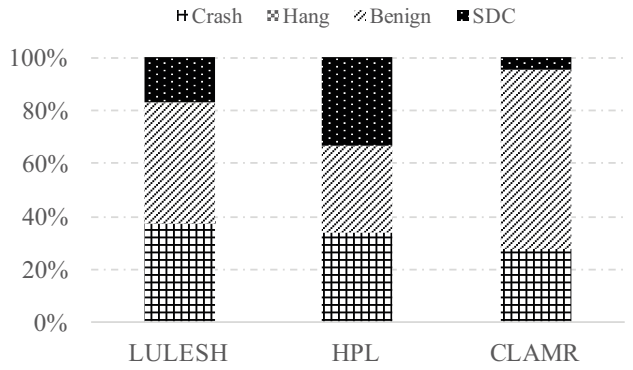

Fig. 2: The fault injection results on three HPC mini-apps. Hangs are less than 1% across all three benchmarks.

Figure 2 shows the fault injection results for three HPC applications. Crashes and benigns are the most dominant types of outcomes, and hangs constitute the smallest category, which are less than 1% across all benchmarks. In terms of SDCs, which we define as outcomes that do not pass the application-specific correctness check, *CLAMR* has the lowest SDC rate among the three (4.5%). On the other hand, in *HPL* nearly 33% of fault injection runs lead to SDCs. The reason for such a huge difference is likely that *HPL* implements LU decomposition which is a direct solver, while the other two are based on iterative behaviors which have the tendency to mask transient errors algorithmically.

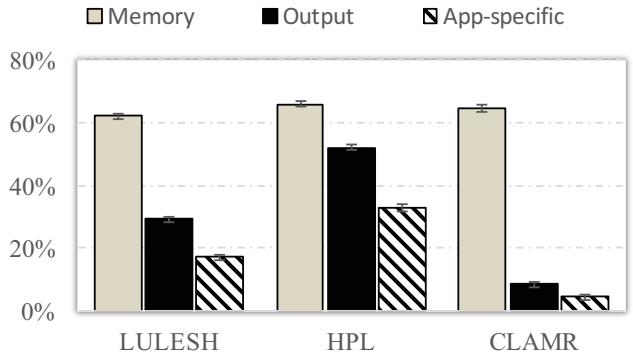

Fig. 3: The percentages of data corruptions across system layers for three HPC applications. The 95% confidence interval is shown for the estimation in each level. Cases where corruptions are in output (defined in Table I) is categorized into 'Output', and cases where the program results do not pass the application-specific correctness check are categorized into 'App-specific'.

Figure 3 shows that the portions of faults causing data

TABLE I: Application output and application-specific correctness check for each application.

Applications	Output	Application-specific correctness check
LULESH	Number of iterations Final origin energy Measures of symmetry	Number of iterations: exactly the same Final origin energy: correct to at least 6 digits Measures of symmetry: smaller than 10^{-8}
HPL	Solution vector x	Residual check on x
CLAMR	Number of cell units Mass change per iteration	Threshold for the mass change per iteration[a]

[a] $https://github.com/losalamos/CLAMR/blob/master/clamr\_cpuonly.cpp : 443$

corruptions for each layer of the three mini-apps. We do not consider the cases where faults lead to crashes as they are essentially system level detection. Overall, there is a significant amount of fault masking between memory level and output level, and a relatively smaller amount of fault masking between output and the application-level checks (i.e., SDCs), depending on the application. For example, in *LULESH* 63% of fault injection runs (out of 3,000) lead to non-crash outcomes, and 62% out of 3,000 (i.e. 99% of non-crash runs) cause memory corruptions, 30% out of 3,000 cause output difference, and 17% out of 3,000 actually lead to unacceptable results (i.e., SDCs). For all three applications, the average difference between the memory corruption rate and the SDC rate is 46%, which means that 46% of 3,000 runs a fault that causes memory corruptions would not impact the final correctness of these applications. In addition, the average difference between output corruption rate and the SDC rate is about 12%. This indicates that in average around 50% of output corruptions do not lead to SDCs for three mini-apps. For many studies that use the program output to represent the correctness of programs [5], [7]–[9], [11], [20], this gap may essentially affect the effectiveness of their techniques, as if application-specific correctness checks are employed.

Using relatively simple experiments and tools on proxy-apps, we have shown that differing rates of "SDC" can be obtained depending on the level at which the SDC is considered. It is important that discussions of this topic are precise, and are clear at what layer the SDC is being considered, as there are a number of different strategies for tolerating data corruption. Without this clarity, it is difficult to reason about the insights of vulnerabilities to "SDC" provided by such techniques.

IV. CONCLUSION

In this paper we argue that fault tolerance and resilience characterization techniques need to classify SDCs with respect to the level of the system stack where the measurements were conducted. Cross-layer analysis is important, and more emphasis needs to be placed on this aspect to effectively evaluate the impact of SDC on systems. We discuss several important studies which look at these different ways of classifying SDC. Using three sample HPC workloads, we demonstrate the the determination of how tolerant (or not) the applications are depends largely on where the detection is done. Evaluating SDC rates by observing applications running in production can be useful but using application specific correctness checking as a metric can make it complex to determine the SDC rates seen at lower layers of the system due to the masking effects. We hope that this position paper is useful to both practitioners and researchers to emphasize the importance of clarity in this area.

As future work, we want to expand the scope of the applications considered, as well as the application-specific checks. Our ultimate goal is to provide a detailed characterization of different applications and their correctness checks to understand how much the SDC rate varies based on these factors, and develop efficient fault tolerance techniques for these applications.

ACKNOWLEDGMENT

We would like to thank the reviewers of RSDA 2016 for their insightful feedback. This work was funded in part by Discovery grants from the Natural Science and Engineering Research Council (NSERC).

REFERENCES

[1] B. Bode, M. Butler, T. Dunning, W. Gropp, T. Hoe-fler, W.-m. Hwu, and W. Kramer, "The blue waters super-system for super-science. contemporary hpc architectures, jeffery vetter editor," 2012.

[2] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly." *ASPLOS*, vol. 43, no. 1, pp. 297–310, 2015.

[3] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, 2014.

[4] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2012, pp. 1–12.

[5] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello, "Lightweight silent data corruption detection based on runtime data analysis for hpc applications," ser. HPDC '15, 2015.

[6] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "SDCTune: A Model for Predicting the SDC Proneness of an Application for Configurable Protection," in *CASE 2014*.

[7] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," *SIGPLAN Not.*, vol. 45, no. 3, Mar.

[8] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[9] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *International Symposium on Code Generation and Optimization*. IEEE, 2005, pp. 243–254.

[10] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, March 2014, pp. 221–230.

[11] S. Hari, R. Venkatagiri, S. Adve, and H. Naeimi, "Ganges: Gang error simulation for hardware resiliency evaluation," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, June 2014, pp. 61–72.

[12] K.-H. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *Computers, IEEE Transactions on*, vol. C-33, no. 6, pp. 518–528, June 1984.

[13] Z. Chen, "Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods," *SIGPLAN Not.*, vol. 48, no. 8, pp. 167–176, Feb. 2013.

[14] P. Wu and Z. Chen, "Ft-scalapack: Correcting soft errors on-line for scalapack cholesky, qr, and lu factorization routines," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14, 2014, pp. 49–60.

[15] J. Sloan, R. Kumar, and G. Bronevetsky, "Algorithmic approaches to low overhead fault detection for sparse linear algebra," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, 2012, pp. 1–12.

[16] ——, "An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, 2013, pp. 1–12.

[17] S. E. Michalak, W. N. Rust, J. T. Daly, A. J. DuBois, and D. H. DuBois, "Correctness field testing of production and decommissioned high performance computing platforms at los alamos national laboratory," ser. SC '14, Piscataway, NJ, USA, 2014, pp. 609–619.

[18] R. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in HPC applications." *SC*, pp. 72–12, 2015.

[19] N. Wang, M. Fertig, and S. Patel, "Y-branches: when you come to a fork in the road, take it," *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pp. 56–66, 2003.

[20] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures." *IEEE Trans. Dependable Sec. Comput.*, vol. 6, no. 2, pp. 135–148, 2009.

[21] V. Sridharan and D. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, Feb 2009, pp. 117–128.

[22] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pp. 1–10, 2013.

[23] J. Elliott, F. Mueller, M. Stoyanov, and C. Webster, "Quantifying the impact of single bit flips on floating point arithmetic," 2013.

[24] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. (2008) Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. [Online]. Available: http://www.netlib.org/benchmark/hpl

[25] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[26] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong, "Lulesh programming model and performance ports overview," Tech. Rep. LLNL-TR-608824, December 2012.

[27] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey, "Cell-based adaptive mesh refinement implemented with general purpose graphics processing units," 2012.

[28] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, June 2014, pp. 375–382.

[29] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel itanium programs with dynamic instrumentation," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, Dec 2004, pp. 81–92.