Tolerate Silent Data Errors with Coded Computation

Gefei Zuo
University of Michigan
gefeizuo@umich.edu

Jiacheng Ma
University of Michigan
jcma@umich.edu

Andrew Quinn
University of California, Santa Cruz
aquinn1@ucsc.edu

Baris Kasikci
University of Michigan
barisk@umich.edu

***Abstract*—While hyper-scale data centers are reporting a growing number of Silent Data Errors (SDEs), existing techniques alone are still insufficient to build an SDE-resilient system. In this work, we propose the adoption of *Coded Computation* to mitigate SDE computation errors efficiently. Based upon error correcting codes (ECC), which have been proven valuable to the mitigation of SDEs in storage and communication, coded computation performs computations in parallel that are homomorphic to ECC encoding/decoding functions, thus tolerating a configurable amount of computation errors. We first give a motivating example to compare coded computation with traditional triple modular redundancy. Then we introduce state of the art in coded computation, its applications to SDE mitigation mechanisms, and open questions for future work.**

***Keywords*-coded computation; silent data error;**

I. Introduction

Hyper-scale data centers (i.e., hyperscalers) are reporting a growing number of Silent Data Errors (SDEs) during computation [2], [6], [10]. In such cases, a defective CPU produces the wrong result during a computation. For some SDEs, the error is associated with certain instructions; for others, the error is associated with specific operating conditions (e.g., workload, frequency, voltage, temperature). Such errors are difficult to detect with hardware or software and can corrupt and invalidate all of the application's behavior. Moreover, SDEs are expected to be remain pervasive since the scale of data centers is still expanding, while increasing silicon density makes manufacturing and testing of CPUs more challenging [3], [16], [20].

Coping with SDEs requires new techniques since SDEs result in arbitrary failures whereas conventional computation errors are considered to be fail-stop. Current work focuses on detecting SDEs that can arise in specific hardware by monitoring software with a combination of runtime and offline testing strategies [1], [2], [5], [18]. Unfortunately, detection alone is insufficient to build an SDE-resilient system because SDEs that occur between periodic tests will still impact applications and detection will necessarily bias known failure patterns and miss new types of failures.

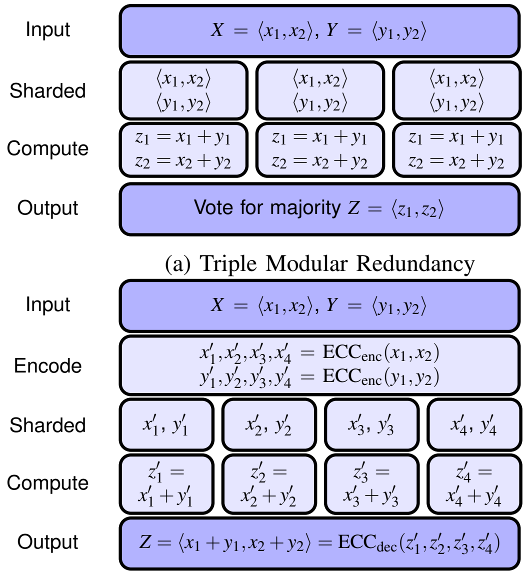
In this work, we investigate mitigation mechanisms for SDEs. A mitigation technique ensures that an application's behavior is correct regardless of the presence of an SDE computation error. For example, Triple Modular Redundancy (TMR) is the conventional approach to tolerate computation errors. TMR leads to redundant computation by a factor of three: the scheme performs the same computation three times and votes for the majority results to tolerate one computation error. Because TMR is inefficient, it is only adopted for "critical applications" [10] and is not a general-purpose solution to handle SDE computation errors. We note that TMR aims to tolerate a failure rate of 33%, which is much higher than the SDE related failure rate observed by hyperscalers (one per several thousand machines) or expected by manufacturers.

To efficiently mitigate SDE computation errors, we propose the adoption of *Coded Computation*. Much like how coding schemes (e.g., checksums, parity codes, erasure coding, etc.) efficiently mitigate SDEs in storage and networking, coded computation uses ECC-like coding schemes to tolerate computation errors with relatively low overhead. Coded computation shards program inputs into chunks and then redundantly encodes the shards into *codewords* using ECC. Computation proceeds on each codeword in parallel; finally, the computation results can be decoded to produce the correct output of the computation result. Due to mathematical properties of the encoding and decoding schemes, the coded computation example using Reed-Solomon Code in §II shows an interesting design different from TMR. The example tolerates a flexible failure rate of $\frac{1}{N}$ (one failure out of N computation nodes), which is suitable for the low failure rate expected in SDEs, and only imposes redundant computation by a factor of $\frac{N}{N-2}$.

II. Motivating Example

Consider the intuitive example of reliably computing element-wise addition of two two-element vectors: $Z = X + Y$ in the presence of SDEs, where $X = \langle x_1, x_2 \rangle$ and $Y = \langle y_1, y_2 \rangle$. In Fig. 1, we demonstrate and compare two solutions to tolerate potential SDEs in one computation entity (e.g., can be one instance of adder circuits, one CPU core, one machine in a distributed setup, etc.).

Fig. 1a shows a TMR-based SDE mitigation. The input vectors X and Y are simply replicated to three entities and each entity performs the same vector addition. Even SDEs occur in one entity, the remaining two entities would still get the same correct results Z and outvote the wrong results. To guarantee two correct additions, TMR performs six additions in total.


Input
$X = \langle x_1, x_2 \rangle$, $Y = \langle y_1, y_2 \rangle$
Sharded
$\langle x_1, x_2 \rangle$ $\langle y_1, y_2 \rangle$
$\langle x_1, x_2 \rangle$ $\langle y_1, y_2 \rangle$
$\langle x_1, x_2 \rangle$ $\langle y_1, y_2 \rangle$
Compute
$z_1 = x_1 + y_1$ $z_2 = x_2 + y_2$
$z_1 = x_1 + y_1$ $z_2 = x_2 + y_2$
$z_1 = x_1 + y_1$ $z_2 = x_2 + y_2$
Output
Vote for majority $Z = \langle z_1, z_2 \rangle$

(a) Triple Modular Redundancy

Input
$X = \langle x_1, x_2 \rangle$, $Y = \langle y_1, y_2 \rangle$
Encode
$x'_1, x'_2, x'_3, x'_4 = \mathrm{ECC}_{\mathrm{enc}}(x_1, x_2)$
$y'_1, y'_2, y'_3, y'_4 = \mathrm{ECC}_{\mathrm{enc}}(y_1, y_2)$
Sharded
x'_1, y'_1
x'_2, y'_2
x'_3, y'_3
x'_4, y'_4
Compute
$z'_1 = x'_1 + y'_1$
$z'_2 = x'_2 + y'_2$
$z'_3 = x'_3 + y'_3$
$z'_4 = x'_4 + y'_4$
Output
$Z = \langle x_1 + y_1, x_2 + y_2 \rangle = \mathrm{ECC}_{\mathrm{dec}}(z'_1, z'_2, z'_3, z'_4)$

(b) Reed-Solomon Code (i.e., ECC)

Figure 1: Example solutions to tolerate one faulty entity for vector additions

Fig. 1b shows an SDE mitigation mechanism based on a particular ECC, Reed-Solomon Code, where input data are encoded before being sharded to different computation entities. ECC_{enc} is the encoding function of the Reed-Solomon Code, which uses the input as the coefficient of a polynomial and encodes the input to be redundant data points on the polynomial. For example, $x'_i = x_1 + x_2 \cdot \alpha_i$, for $i \in \{1,2,3,4\}$, where α_i denotes the location of codewords on the polynomial. Then the addition of two codewords $x'_i + y'_i = (x_1 + y_1) + (x_2 + y_2) \cdot \alpha_i$ also represents the Reed-Solomon Code of the addition results over the original input (i.e., $\langle x_1 + y_1, x_2 + y_2 \rangle$). By decoding (i.e., $\mathrm{ECC}_{\mathrm{dec}}$) the addition results over the codewords, we can reliably get the correct addition results when SDEs affect one of the four computations. To guarantee two correct additions, ECC performs four additions in total.

In the above comparison, the ECC based solution guarantees the same amount of correct additions with fewer additions in total, while assuming a lower (and more realistic) failure rate. ECC is also scalable to tolerate SDEs on one entity out of a configurable N entities, in which case only N additions are required to guarantee $N-2$ correct additions. Although the encoding and decoding steps are more complicated than TMR, their cost could be amortized as the number of computations done in the codewords-domain and the number of involved entities (i.e., parallelism) increase.

III. Coded Computation

As demonstrated in Fig. 1b, SDEs on computations over ECC-encoded data can be tolerated with fewer redundant computations and a more realistic failure rate assumption. Such computation is often referred as *Coded Computation*, which has been studied for a long time [15], [19]. At its core, given a pair of a ECC encoding function $\mathrm{ECC}_{\mathrm{enc}}$ and a decoding function $\mathrm{ECC}_{\mathrm{dec}}$ that can tolerate errors in input data $X_1, \ldots, X_N$, computation errors of function F can also be tolerated if:

$$F(\mathrm{ECC}_{\mathrm{enc}}(X_1, \ldots, X_N)) = \mathrm{ECC}_{\mathrm{enc}}(F(X_1), \ldots, F(X_N))$$

Early research on coded computation has proposed efficient coding schemes for linear functions [15], [19], while non-linear functions were considered challenging. Only recently, researchers proposed new coding schemes to support arbitrary multivariate polynomials [23] (i.e., addition and multiplication) and boolean functions [22]. Prior work mainly focused on computations that are crucial to machine learning or data analytics applications, such as matrix-vector/matrix multiplication [7], [11], [13], [24], least-squares linear regression [23], DNN training [21], Page Rank [8], MapReduce [12], etc. Among these applications, prior work mostly targets stragglers (i.e., improving tail latency via redundant computation), security and privacy (i.e., multiparty computing, federated learning).

However, critical applications that need to tolerate SDEs in data centers are likely to behave differently from those studied in recent coded computation literature. Besides, SDE mitigation should tailor coded computation for high reliability and efficiency, instead of focusing on tail-latency/security/privacy. With the rich literature discussing coded computation applications, we believe that it is a good time to adopt coded computation to develop new SDE mitigation mechanisms. A preliminary but promising example is to apply Lagrange Coded Computation to State Machine Replication [14]. Eventually, we should conduct quantitative analyses on the fault tolerant computation overheads of critical applications and explore the performance-reliability trade-offs between existing TMR based solutions and coded computation.

IV. Open Questions

At this early stage of applying coded computation to tolerate SDEs, we think the following interesting questions are still open and worth investigating:

1) What computations are required in critical data center applications and how can coded computation support them?
2) Coded Computation, which can even be interpreted as Homomorphic Error Correcting Code, resembles Homomorphic Encryption a lot. Can we build a generic framework like FHE compilers [4], [9] that is both programmable and tolerant to SDEs?
3) Hardware accelerators have demonstrated huge speedups in Homomorphic Encryption [17] workloads. Can we also build hardware accelerators for coded computation?

References

[1] "In-Field Scan — The Linux Kernel documentation," https://www.kernel.org/doc/html/latest/x86/ifs.html, Sep. 2022.

[2] D. F. Bacon, "Detection and Prevention of Silent Data Corruption in an Exabyte-scale Database System," in *The 18th IEEE Workshop on Silicon Errors in Logic – System Effects*, 2022.

[3] H. H. Chen, "Beyond structural test, the rising need for system-level test," in *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, Apr. 2018, pp. 1–4.

[4] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 546–561.

[5] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, "Detecting silent data corruptions in the wild," Mar. 2022.

[6] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent Data Corruptions at Scale," Feb. 2021.

[7] S. Dutta, V. Cadambe, and P. Grover, ""Short-Dot": Computing Large Linear Transforms Distributedly Using Coded Short Dot Products," *IEEE Transactions on Information Theory*, vol. 65, no. 10, pp. 6171–6193, Oct. 2019.

[8] S. Dutta, H. Jeong, Y. Yang, V. Cadambe, T. M. Low, and P. Grover, "Addressing Unreliability in Emerging Devices and Non-von Neumann Architectures Using Coded Computing," *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1219–1234, Aug. 2020.

[9] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak, P. Schoppmann, S. Kulankhina, A. Forget, D. Marn, C. Tew, R. Misoczki, B. Guillen, X. Ye, D. Kraft, D. Desfontaines, A. Krishnamurthy, M. Guevara, I. M. Perera, Y. Sushko, and B. Gipson, "A general purpose transpiler for fully homomorphic encryption," https://eprint.iacr.org/2021/811, 2021.

[10] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 9–16.

[11] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding Up Distributed Machine Learning Using Codes," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1514–1529, Mar. 2018.

[13] ——, "A Unified Coding Framework for Distributed Computing with Straggling Servers," in *2016 IEEE Globecom Workshops (GC Wkshps)*, Dec. 2016, pp. 1–6.

[12] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded MapReduce," in *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Sep. 2015, pp. 964–971.

[14] S. Li, S. Sahraei, M. Yu, S. Avestimehr, S. Kannan, and P. Viswanath, "Coded State Machine – Scaling State Machine Execution under Byzantine Faults," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 150–152.

[15] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, "CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework," in *2008 IEEE International Conference on Computer Design*, Oct. 2008, pp. 363–370.

[16] P. G. Ryan, I. Aziz, W. B. Howell, T. K. Janczak, and D. J. Lu, "Process defect trends and strategic test gaps," in *2014 International Test Conference*, Oct. 2014, pp. 1–8.

[17] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 238–252.

[18] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, "SiliFuzz: Fuzzing CPUs by proxy," Oct. 2021.

[19] D. Spielman, "Highly fault-tolerant parallel computation," in *Proceedings of 37th Conference on Foundations of Computer Science*, Oct. 1996, pp. 154–163.

[20] A. J. Strojwas, K. Doong, and D. Ciplickas, "Yield and Reliability Challenges at 7nm and Below," in *2019 Electron Devices Technology and Manufacturing Conference (EDTM)*, Mar. 2019, pp. 179–181.

[21] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. Sydney, NSW, Australia: JMLR.org, Aug. 2017, pp. 3368–3376.

[22] C.-S. Yang and A. S. Avestimehr, "Coded Computing for Boolean Functions," in *2020 International Symposium on Information Theory and Its Applications (ISITA)*, Oct. 2020, pp. 141–145.

[23] Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, and S. A. Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security, and privacy," in *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 2019, pp. 1215–1225.

[24] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler Mitigation in Distributed Matrix Multiplication: Fundamental Limits and Optimal Coding," in *2018 IEEE International Symposium on Information Theory (ISIT)*, Jun. 2018, pp. 2022–2026.