Hybrid Hardware/Software Detection of Multi-Bit Upsets in Memory

Robin Thunig
TU Dresden, Germany
robin.thunig@tu-dresden.de

Christoph Borchert
Osnabrück University, Germany
christoph.borchert@uni-osnabrueck.de

Urs Kober
TU Dresden, Germany
urs.kober@tu-dresden.de

Horst Schirmeier
TU Dresden, Germany
horst.schirmeier@tu-dresden.de

***Abstract*—Bit flips in main memory can be caused by a multitude of environmental effects, such as heat or radiation, as well as by malicious actors exploiting Rowhammer-style hardware vulnerabilities. The industry-standard countermeasure is SEC-DED ECC memory, which can reliably correct single- and detect double-bit flips in a data word. However, larger multi-bit upsets (MBUs) regularly occur in real-world systems, and – as shown by an analysis in this paper – have a high probability of being miscorrected. Software-implemented hardware fault tolerance (SIHFT) mechanisms can flexibly handle MBUs, but incur significant runtime costs.**

In this paper, we propose to combine hardware ECC as a low-cost detector and SIHFT as a handler for miscorrected MBUs that recategorizes them as uncorrectable. A preliminary evaluation on the basis of differential checksums shows a 98.5 % reduction in miscorrected silent data corruptions with a very moderate execution-time overhead.

I. Introduction

Soft errors pose a threat to all modern computer systems. As the number of transistors at the system level continues to increase, the soft-error rates are expected to increase in the future [1]. A widely used method of dealing with soft errors in DRAM are error-correcting codes (ECC). This enables efficient fault-tolerant systems, mainly used for professional applications. However, nowadays ECC memory with single-error correction and double-error detection (SEC-DED) is commonly used. Accordingly, no guarantees are given for larger multi-bit upsets (MBUs). Such MBUs regularly occur in large systems [2]–[4].

One possible hardware measure against MBUs in DRAM is IBM's *chipkill correct* [5] and similar trademarks. Chipkill scatters an ECC word over multiple memory chips and enables fault tolerance even for MBUs that affect more than two bits locally. However, chipkill reduces performance and requires up to 30 % more energy, since a large number of memory chips must be activated per memory access [6]. In addition, there are studies that show that, despite the use of chipkill, soft errors occur that could not be corrected [3]. A different, well-studied solution is software-implemented hardware fault tolerance (SIHFT) with a strong flexibility regarding MBU detection and correction capabilities, but at the cost of high runtime overheads even for highly optimized implementations [7].

In this paper, we aim for an efficient detection of MBUs in DRAM using a *combination* of hardware SEC-DED ECC and SIHFT. We propose to use a cascade of 1) ECC hardware as a detector with low performance cost and 2) a SIHFT mechanism to handle miscorrected MBUs properly by recategorizing them as uncorrectable. We also provide a brief analysis of the characteristics of SEC-DED ECC memory for MBUs larger than two bits. This analysis shows the feasibility of using ECC as a low-cost detector also for larger MBUs, and why the combination with SIHFT methods is promising.

As a proof of concept, we demonstrate our approach in a prototype implementation, applied to a selection of TACLe [8] benchmarks.

In summary, our contributions are the following:

- An analysis of the characteristics and usefulness of SEC-DED codes for MBUs larger than two bits up to a size of eight bits.
- A hybrid hardware/software approach to efficiently detect MBUs affecting more than two bits while preserving the ability of SEC-DED ECC to correct single-bit upsets.

II. SEC-DED Codes

ECC memory with SEC-DED makes no guarantees for the correction or detection of MBUs with three or more corrupted bits. To better understand how exactly SEC-DED ECC behaves for MBUs, we briefly analyze this scenario in the following and compare the results with the literature.

We examine Hsiao codes [9] with 64 data bits and 8 check bits as ECC, since these are widely used in commercially available ECC memory [10]. In order to determine how MBUs of different sizes affect the behavior of the Hsiao codes, we simulate all possible $\binom{72}{n}$ combinations of an n-bit MBU within an ECC word of 72 bits and measure the effect. We show the results in Tab. I. We observe that for three-bit MBUs no error can be corrected. In fact, 56.45 % of the injected errors are *miscorrected*. This means that the ECC hardware corrupts one *more* bit in addition to the present three-bit MBU. However, 43.55 % are detected as uncorrectable. For four-bit MBUs, 99.18 % are classified as uncorrectable, but 0.82 % are not detected at all. These values are in line with Hsiao [9].

For this paper, we also evaluate MBUs with sizes of 5–8 bits. We find that the pattern that emerged for three-bit MBUs also applies to other MBUs of odd-numbered size. Similarly, we find that the pattern measured for four-bit MBUs is also consistent for other even-number sized MBUs. This means that for an odd MBU, all errors can be detected, but about 56 % are miscorrected. For even MBUs, almost all errors can be detected, with less than 1 % not being detected at all. At first glance, one could assume that ECC alone is a good MBU

2325-6664/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN-W60302.2024.00029

TABLE I
HSIAO CHARACTERISTICS FOR THREE-BIT UP TO EIGHT-BIT MBUS, TESTED FOR ALL POSSIBLE MBUS IN A 72-BIT ECC WORD WITH 64-BIT DATA AND 8-BIT CHECK BITS. FOR AN ODD NUMBERED MBU AROUND 56 % OF ALL POSSIBLE MBUS ARE MISCORRECTED. FOR AN EVEN NUMBERED MBU NEARLY ALL ARE DETECTED AS UNCORRECTABLE, BUT LESS THAN 1 % ARE NOT DETECTED AT ALL.

MBU	Miscorrected	Uncorrectable	Not detected
3	56.45 %	43.55 %	0.00 %
4	0.00 %	99.18 %	0.82 %
5	56.23 %	43.77 %	0.00 %
6	0.00 %	99.22 %	0.78 %
7	56.25 %	43.75 %	0.00 %
8	0.00 %	99.22 %	0.78 %

detector, as it has an almost 100 % detection rate for any MBU. However, when single-bit flips occur, ECC can *correct* them. A detection can either mean we have a correctable single-bit flip or an MBU that would be miscorrected. Since single-bit flips are much more common, it is desirable to preserve the correction ability for them. In this case, SEC-DED ECC cannot be used on its own for high-probability MBU detection.

III. ECC AS LOW COST HARDWARE-BASED DETECTOR

Consequently, the goal of our approach is to build upon the strong error detection capability of ECC memory hardware and to compensate its high MBU miscorrection rate. To this end, we apply a software mechanism to detect such miscorrections.

We take advantage of the fact that ECC can *detect* an error in almost all cases. This also applies to MBUs with a size of three bits and more, whereby with an odd numbered MBU, around 56 % were detected but miscorrected (Tab. I). With an even numbered MBU, almost all errors are recognized as uncorrectable, with less than 1 % not being detected at all. We focus in particular on MBUs with an odd number, and aim at minimizing the number of errors that are *detected but miscorrected* by recategorizing them as *uncorrectable*.

To achieve this, we use the ability of the memory controller to signal that an error has been detected and corrected in the ECC memory. We propose the following procedure to detect MBU miscorrections: 1) A single/multi-bit error occurs in the ECC memory. 2) The memory controller detects the error and attempts to correct it, which might result in a miscorrection. 3) The memory controller triggers an interrupt in either case. 4) The interrupt initiates a further check of the data, which is performed by a software method.

However, software methods are typically associated with a high overhead, both in terms of memory and execution time. To achieve a significant efficiency improvement by using it in combination with ECC interrupts, we assume that the following requirements are met: 1) The software method shall be optimized for the error-free case. That is, the execution-time overhead shall be minimal when there is no error. 2) The actual detection of MBUs needs not to be particularly fast, because it is only triggered by an ECC interrupt, which shall be a rare event. 3) The memory overhead of the software method shall be minimal to avoid polluting the cache with redundant data.

A. MBU Detection by Differential In-Memory Checksums

In this paper, we evaluate the multi-bit error detection capability of software-implemented in-memory checksums on top of ECC RAM. A checksum in general is calculated as a function of the data it is associated with. For example, a checksum for an array of values can be computed by the arithmetic sum of all elements in that array. Software can store such a checksum next to the data in memory. The data can be checked for errors by recomputing the checksum and testing whether the stored and recomputed checksums match. On match, there is a high probability that there are no errors in the data.

In-memory checksums require that when a piece of data is modified (e.g., a program *variable*), the checksum needs to be updated. A fast approach is to use *differential checksum* computations [7], that is, to patch the checksum without full recomputation. For instance, we can patch the aforementioned arithmetic checksum efficiently by subtracting an old data value from the checksum and by adding in a new data value.

In previous work [7], we developed a compiler-based approach that automatically inserts differential checksums into C/C++ data structures to avoid a manual and otherwise tedious implementation by the software developer, because each write access to a program variable needs to be considered. However, without knowledge about the temporal occurrence of memory errors, the compiler conservatively inserts at least one checksum check into each C/C++ function before reading the respective data. Even though we instruct the compiler to minimize repeated checks, the previous approach results in high execution-time overhead of more than 100 %.

B. Implementation of a Hybrid Hardware/Software Approach

A large share of the execution-time overhead of software-implemented in-memory checksums originates from recomputing the checksum to detect errors proactively. Our approach is to combine in-memory checksums with hardware ECC memory and to only use the hardware for error detection. Thus, we never recompute the checksum unless ECC has detected an error. In other words, we apply the compiler-implemented in-memory checksums to automatically maintain an always up-to-date checksum for each data structure by differential checksum patches on each memory write access. In addition, we occasionally test whether the ECC memory hardware signals the occurrence of an error. Only if an error has been signaled, we check the data for multi-bit errors by recomputing the checksums. As a heuristic, we use our compiler to insert a test for a global ECC interrupt flag at every C/C++ function return.

The presented heuristic makes little demands on the hardware architecture, because neither the ECC interrupt is required to trigger immediately on data access nor the exact addresses of memory errors need to be exposed by the hardware. On the software side, however, this means that we do not know exactly where and when a memory error has occurred, so we need to check *all* checksum-protected data structures when the global ECC interrupt flag is set. Thus, we extend the aforementioned compiler-based approach to automatically set up a linked list of

all checksum-protected data structures at runtime. Once each data structure in that list has been checked successfully, we reset the ECC interrupt flag to avoid further software checks.

Since the global ECC interrupt flag is only checked at every C/C++ function return, there is usually a latency between the occurrence of an error and its detection by the checksum. However, an error is always detected, even if the error has already been overwritten before the checksum is checked. This property is achieved by the differential checksums, as these are updated relative to the previous checksum. Note that we intentionally do not check the data structures directly in the ECC interrupt routine, as the data structures and the associated checksums might not be in a consistent state on interrupt.

In summary, the hybrid hardware/software approach to MBU detection aims at reducing the execution-time overhead of the software-only approach by avoiding checksum recomputation when there is no memory error to be expected.

IV. Evaluation

We evaluate our approach from Sec. III using FAIL* [11], a simulation-based fault injector, which we extended by an ECC-memory simulation. To perform the fault injections we used a two-way AMD EPYC 7451 system with 384 GiB of main memory.

We apply fault injection to simulate MBUs in DRAM with a uniform random distribution in space and time. Specifically, an MBU is injected at once after a random instruction and locally into an ECC word, whereby any combination is possible within this word. We use the extrapolated absolute failure count (EAFC) [12] as a metric to determine the fault tolerance of a program. This means that we scale the number of failures based on the fault-space size, since a longer running program and more memory used results in a higher probability that a memory fault occurs.

A. TacleBench

We use seven benchmarks from the TACLe suite [8] as examples for the evaluation. We consider three different variants per benchmark, all of which use simulated hardware ECC. We consider a *baseline* variant without SIHFT, the *checksum* variant that uses the original differential checksums as described in Sec. III-A, and the *IRQ-checksum* variant that uses our new approach from Sec. III-B with ECC interrupts.

We injected 1,000,000 three-bit MBUs for each variant. Since we inject three-bit MBUs, the SEC-DED ECC causes a miscorrection in about 56 % of all cases while about 44 % are recognized as uncorrectable. At the end of the program execution, we record how the MBU has affected the program. This means whether it has become a silent data corruption (SDC) or caused another failure (resulting in a timeout or a CPU exception). We primarily want to minimize the number of miscorrected MBUs that lead to an SDC (miscorrected SDC), since neither ECC nor the program can detect this error. All other failure types (timeout, CPU exception, SDC recognized as uncorrectable by the ECC) can be detected.

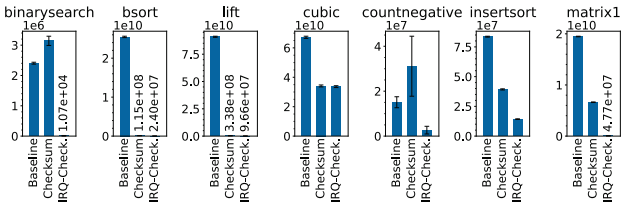

Fig. 1. Comparison of different benchmark variants in regard to their number of miscorrected SDCs. On average, the IRQ-checksum approach reduces significantly the number of miscorrected SDCs compared to the other variants. Error bars show the 99 percent confidence interval. (Lower values are better).

TABLE II
Number of dynamic instructions for different benchmarks variants. Over all benchmarks, the IRQ-checksum approach induces 29 % more instructions over the baseline, but 27 % less than the original differential checksum variant.

Benchmark	Baseline	Checksum	IRQ-Checksum
binarysearch	549	1,105	982
bsort	73,325	82,682	77,996
lift	537,276	947,832	670,504
cubic	612,810	660,024	662,975
countnegative	9,861	68,989	14,362
insertsort	732	1,144	1,066
matrix1	8,233	9,700	9,198
Geom. mean	18,659	33,124	24,152

We can see in Fig. 1 that the number of miscorrected SDCs is massively reduced with the IRQ checksum compared to the baseline and also to the original differential checksums. If we aggregate the benchmarks with the geometric mean, then our IRQ-checksum approach reduces the number of miscorrected SDCs by 98.5 % compared to the baseline and by 91.1 % compared to the original variant of differential checksums. In the case of *binarysearch* and *countnegative*, we can see that the checksum variant is worse than the baseline. This could be due to the fact that the regular checking of the checksums increases the program execution time and therefore makes the occurrence of faults more likely. The IRQ-checksum variant reduces this problem, as the checksums are only checked during an ECC interrupt, which results in much less miscorrected SDCs.

We also examine how the program execution time changes as a result of our approach. To do this, we record the number of simulated dynamic instructions for the benchmark variants. The results are shown in Tab. II. If we aggregate all benchmarks over the geometric mean, then the IRQ-checksum approach requires 29 % more instructions than the baseline, but 27 % less than the original differential checksum variant.

In summary, we find that our hybrid approach to MBU detection reduces the rate of miscorrected SDCs by 98.5 % at an execution-time overhead of 29 % on average.

B. Threats to Validity

The current implementation of the IRQ-checksum approach checks the global ECC interrupt flag at every function return. In the current implementation, this might increase the detection

latency and programs that frequently call small functions can suffer a significant overhead.

Furthermore, the behavior of the hardware ECC is only simulated in this paper. An evaluation using real hardware ECC is subject to future work.

V. Related Work

Other work has already provided approaches for hybrid hardware/software fault tolerance. Kuvaiskii et al. present an approach to hardware-assisted fault tolerance (HAFT) [13], which can detect and recover faults as a compiler-based hardening mechanism through instruction level redundancy and hardware transactional memory. HAFT is on average 2x slower compared to a native version.

Other work deals with the detection of MBUs in general. Neuberger et al. [14] have developed a method that can correct single and double-bit faults and a large number of multiple faults. To do this, they combined Hamming and Reed-Solomon codes. They implemented their approach on an FPGA and found a performance penalty of about 50 %. Lee et al. [15] use a hardware-implemented multi-bit correctable BCH code. The aim is to minimize the memory footprint by applying the BCH code only to 32-bit data words and storing the check bits of the BCH code in the remaining 32 bits. In the case of a 64-bit data word, a SEC-DED ECC is used. This can reduce system failures by 47.9 %, with a performance overhead of 0.9 %. However, since SEC-DED ECC is used for 64 bit data words, this approach should not have a detection rate for MBUs comparable to our approach. Narasimham and Luk [16] have dealt with the limitation of SEC-DED ECC memory to detect larger MBUs. They implement an exclusively hardware-based approach that uses Berger codes to calculate a ckecksum for the data. This method, however, can only detect unidirectional upsets. Additionally, specialized hardware is required for this method. Our approach can also be implemented after deploying and run on commercially available hardware.

To the best of our knowledge, we are not aware of any work that can exploit commercially available hardware ECC in such a way that high error detection probability for MBUs can be realized with low overhead.

VI. Conclusions

In this paper, we have presented an approach that combines the efficiency of hardware ECC to detect and correct memory errors with the safety of software to avoid miscorrections of MBUs. Our approach significantly reduces the number of miscorrected MBUs that lead to an SDC by 98.5 % on average compared to hardware-only ECC. At the same time, the software execution-time overhead is moderate with about 29 % more dynamic instructions compared to the baseline.

In future work, we aim at further reducing the software execution-time overhead by checking the data directly in the ECC interrupt routine to avoid the frequent testing of the global ECC interrupt flag. Moreover, we also want to consider additional error correction in software to further increase the system availability in case of MBUs.

References

[1] I. Chatterjee, *From MOSFETs to FinFETs – the soft error scaling trends*, https://radnext.web.cern.ch/blog/from-mosfets-to-finfets/, Accessed: 2023-10-20, Nov. 2020.

[2] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: A large-scale study of DRAM raw error rate on a supercomputer," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 645–655. DOI: 10.1109/SC.2016.54.

[3] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*, Salt Lake City, Utah: IEEE Press, Nov. 2012, 76:1–76:11, ISBN: 978-1-4673-0804-5. DOI: 10.1109/SC.2012.13.

[4] V. Sridharan, N. DeBardeleben, S. Blanchard, *et al.*, "Memory errors in modern systems: The good, the bad, and the ugly," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey: ACM Press, Mar. 2015, pp. 297–310, ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694348.

[5] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," *IBM Microelectronics division*, 1997.

[6] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," *SIGPLAN Not.*, vol. 45, no. 3, pp. 397–408, 2010. DOI: 10.1145/1735971.1736064.

[7] C. Borchert, H. Schirmeier, and O. Spinczyk, "Compiler-implemented differential checksums: Effective detection and correction of transient and permanent memory errors," in *Proceedings of the 53rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '23)*, (Porto, Portugal), Piscataway, NJ, USA: IEEE Press, Jun. 2023. DOI: 10.1109/DSN58367.2023.00021.

[8] H. Falk, S. Altmeyer, P. Hellinckx, *et al.*, "TACLeBench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, M. Schoeberl, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 55, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, 2:1–2:10. DOI: 10.4230/OASIcs.WCET.2016.2.

[9] M. Y. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970. DOI: 10.1147/rd.144.0395.

[10] C.-Y. Chen and C.-W. Wu, "An adaptive code rate edac scheme for random access memory," in *2010 Design, Automation and Test in Europe Conference and Exhibition (DATE 2010)*, 2010, pp. 735–740. DOI: 10.1109/DATE.2010.5456955.

[11] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," in *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*, (Paris, France), Piscataway, NJ, USA: IEEE Press, Sep. 2015. DOI: 10.1109/EDCC.2015.28.

[12] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, (Rio de Janeiro, Brazil), Piscataway, NJ, USA: IEEE Press, Jun. 2015, pp. 319–330. DOI: 10.1109/DSN.2015.44.

[13] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "HAFT: Hardware-assisted fault tolerance," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16, Association for Computing Machinery, 2016. DOI: 10.1145/2901318.2901339.

[14] G. Neuberger, F. de Lima, L. Carro, and R. Reis, "A multiple bit upset tolerant SRAM memory," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, no. 4, Oct. 2003. DOI: 10.1145/944027.944038.

[15] Y. S. Lee, G. Koo, Y.-H. Gong, and S. W. Chung, "Stealth ECC: A data-width aware adaptive ECC scheme for DRAM error resilience," in *2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2022, pp. 382–387. DOI: 10.23919/DATE54114.2022.9774775.

[16] B. Narasimham and W. K. Luk, "A multi-bit error detection scheme for DRAM using partial sums with parallel counters," in *2008 IEEE International Reliability Physics Symposium*, 2008, pp. 202–205. DOI: 10.1109/RELPHY.2008.4558886.