A Vulnerability Factor for ECC-protected Memory

Luc Jaulmes, Miquel Moretó, Mateo Valero and Marc Casas
Barcelona Supercomputing Center (BSC)
Universidad Politecnica de Catalunya (UPC)
name.surname@bsc.es

***Abstract*—Fault injection studies and vulnerability analyses have been used to estimate the reliability of data structures in memory. We survey these metrics and look at their adequacy to describe the data stored in ECC-protected memory. We also introduce FEA, a new metric improving on the memory derating factor by ignoring a class of false errors. We measure all metrics using simulations and compare them to the outcomes of injecting errors in real runs. This in-depth study reveals that FEA provides more accurate results than any state-of-the-art vulnerability metric. Furthermore, FEA gives an upper bound on the failure probability due to an error in memory, making this metric a tool of choice to quantify memory vulnerability. Finally, we show that ignoring these false errors reduces the failure rate on average by 12.75% and up to over 45%.**

I. Introduction

Faults in DRAM cells are becoming ever more prevalent with new process technologies, causing the protection using Error Correcting Codes (ECC) to increase from strong ChipKill protection [7] to include a second layer of ECC on-chip [13]. This trend includes even mobile devices [21] and efforts to gain in energy efficiency could further multiply the soft error rates [2]. To apply more granular protection techniques than uniformly increasing ECC strength, such as dynamically adaptable ECC [27], it is necessary to quantify the risk associated with any data stored in memory.

In particular, we want to find out if there is a good indicator to identify which memory to protect in priority, given the current and future ECC-protected memory. Some metrics have been proposed, and applied to exploit heterogeneous memory with different reliability characteristics [6, 10, 18, 20]. Experimental approaches focus on injecting errors, to identify which data structures to protect with software fault-tolerance techniques [4, 5, 9]. However, most of these approaches suppose an error model based on flipped bits, which is not realistic for ECC-protected memory.

In this paper, we survey metrics from the literature, including derating factors [22, 28], and metrics based on memory accesses [10, 18]. We also identify a subset of Detected and Uncorrected Errors (DUE) in memory that are reported but would have no impact on the running program if ignored. We extend the appropriate factor to take these false DUE into account and call the resulting metric the False Error Aware (FEA) vulnerability metric. Furthermore, we compare all the vulnerability metrics against the effects of injecting errors in real systems, evaluating not only the effects of flipping bits in memory, but also of injecting DUE. These experiments demonstrate that FEA is the metric that correlates best with the probability of program failure due to a DUE, while giving a consistent upper bound on this probability. FEA also shows higher variability than the metric it extends, allowing for more opportunities to identify regions with different vulnerability.

The paper is organised as follows: in Section II we review the state-of-the-art memory vulnerability metrics, while in Section III we define FEA. Section IV presents our experimental setup and Section V the comparison between metrics and error rates, as well as the reduction in failure rates from ignoring false errors. We present our concluding remarks in Section VI.

II. Existing Vulnerability Metrics for Memory

A. Access-Based Metrics

Yu *et al.* [28] define the *Data Vulnerability Factor (DVF)* per data structure d, defined as the multiplication of the structure's size S_d, the program execution time T, the number of *hardware accesses* to this structure in memory N_{ha}, and the overall fault rate λ: $DVF_d = \lambda \cdot T \cdot S_d \cdot N_{ha}$. As λ and T are constant respectively per system and program, they do not help in differentiating vulnerability between data structures in the same program. For this reason, and due to the similarity with the formulation for expected number of errors using the derating factors (see Section II-C), we look simply at $S_d \cdot N_{ha}$. Gupta *et al.* [10] also use a heuristic, which is the ratio of stores (ST) to loads (LD), thus ST/LD.

These two metrics do not take into account the relative timing of the memory accesses, and DVF does not consider whether errors are masked, only considering if they are accessed. The DVF is computed using mathematical models based on memory access patterns, while the store-to-load ratio is used as a proxy for a timing-based metric.

B. Timing-Based Metrics (also known as Derating Factors)

The most widely used metric for architectural components is the Architectural Vulnerability Factor (AVF) [22]. The AVF of a bit is the percentage of cycles during which it is required for Architecturally Correct Execution (ACE). A bit is said to be ACE when it affects the final program output. That is, the AVF of a bit is the fraction of time it contains a value that will affect program outcome.

A similar timing analysis can be performed by defining errors as a wrong value returned from memory, instead of considering program output. Then, the masking of faults simply takes into account whether a memory location is subsequently loaded. In that case, any faults are returned and cause an error. On the other hand, if a memory location is subsequently

DOI 10.1109/IOLTS.2019.8854397

overwritten or not used until the end of the program, the fault is masked. Gupta *et al.* [10] reuse the name AVF, having defined an error as an erroneous value from memory rather than an incorrect program output. For clarity, we call this timing vulnerability factor the *Memory Vulnerability Factor (MVF)*. The relevant granularity here is that of memory accesses, thus cache lines, and the MVF of a cache line in memory is the fraction of time it contains a value that will be accessed.

Luo et al. [18] use the *safe ratio*, which is the fraction of time that data resides in memory before being overwritten. This quantifies the same effects as MVF, as the safe ratio sr and MVF are related by $MVF + sr = 1$. Hence it is only necessary to look at MVF for the purpose of this work.

C. Linking Derating Factors to Program Outcome

We categorise the outcome of a program as correct when its execution is indistinguishable from an execution without errors. We also call incorrect outcomes *failures*. The average program failure rate can be derived from timing vulnerability metrics under common assumptions for soft errors [17]. In particular, for a set of components c with vulnerability factors F_c, a duration T, and a memoryless distribution of faults with rate λ such that $\lambda \cdot T \ll 1$, then the expected number of errors per component is $\lambda \cdot T \cdot F_c$ and the overall failure rate $\sum_c \lambda \cdot F_c$.

However, care must be taken in considering which error model is used. AVF is computed per bit, thus $\sum_{\text{bits}} (\lambda_{\text{bit flip}} \cdot AVF)$ is the expected failure rate due to single bit faults. However, the use of ECC in memory to correct faults (thus using codes stronger than parity) complicates looking at the impact of individual bits in two ways: any bit taken individually has no impact on the memory or program outcome, and many multi-bit faults that can not be corrected are still detected, causing DUEs.

Using the AVF then requires to compute the occurrence rates and derating factors of multiple bit flips observed simultaneously [26], for every multi-bit fault pattern that can not be corrected by the ECC. Furthermore, those patterns are different for every ECC code.

A more appropriate error model is to consider DUE in memory instead of individual bit flips, since any multi-bit fault pattern that causes a DUE will have the same final impact, regardless of which ECC is used. Indeed, the operating system kills any process attempting to access corrupted data [15], which is the main cause of hardware failures in supercomputers [16]. Furthermore, all the most frequent multi-bit faults cause a DUE by design. For example, for a Single Error Correct Double Error Detect (SECDED) code, all double bit flips in a single word cause a DUE. The *program failure rate* is then $\sum_{\text{memory cache lines}} (\lambda_{\text{DUE}} \cdot MVF)$, with λ_{DUE} the arrival rate of DUE.

Some of the uncorrectable errors in memory may have no impact on the program outcome if left uncorrected, due to being ignored or having a negligible impact on the program. These errors are called false DUE and can be quantified, by the difference between the program failure rate due to DUE and the failure rate due to the same uncorrected multi-bit faults.

III. FEA Metric Definition

A. Accounting for False DUE

In this section, we identify one specific cause of false DUE and introduce a metric to quantify it. Let us consider a write or a set of writes that spans a full ECC word (thus 8B for SECDED or 16B for ChipKill [7]). In a cache with a write-allocate policy, if these writes cause a cache miss, data will be fetched from memory. MVF quantifies this data as vulnerable because it is accessed, while any errors in this data are masked as they are immediately overwritten.

Therefore, we introduce the False Error Aware MVF (FEA), which we define as *the fraction of time a memory location contains data that will be consumed.* That is, we consider a memory location as safe not only when it is next accessed by a store, but also when it is next accessed by a load whose contents will be overwritten without being used. The data is considered vulnerable in memory only before a load whose contents will be used. As DUE is reported per ECC word, that is the granularity at which FEA must be computed. The advantage of the FEA metric is that it is still easily quantifiable with simple timing metrics, while slightly more complex than MVF as it includes the cache hierarchy in its analysis instead of only memory. It is worth noting that FEA does not take into account errors that happen in caches, only whether errors that happen in memory are immediately overwritten in the cache hierarchy, or if they may be propagated.

B. FEA is linked to Program Outcome with Deferred Errors

In modern processors, the reporting of errors in various caches and register files is delayed until this error is actually consumed [1, chap. 9.4] [11, chap. 15.5] [3, chap. A8.3]. When data in a cache is found to be inconsistent with its ECC, the error is reported in a machine check register that can be later polled. If the data can not be corrected, the cache hierarchy tracks this data marked as *known bad* (or *poisoned*) and the error is reported as an uncorrected error. An interrupt requiring a software recovery is only triggered when the data is consumed.

It is unclear however whether this *deferred error* capability is already implemented in hardware for DRAM errors. DUE in memory discovered during scrubbing and during a load are explicitly listed as recoverable uncorrected errors, however there is no mention of DUE in memory discovered through indirect accesses (e.g. a store, or data fetched as part of a cache line where only another ECC word is accessed). In any case, as modern microarchitectures already track incorrect data from other causes, this tracking can be reused or very simply expanded to mark lines that contain errors originating in DRAM. For example, marking such data in cache as both dirty and poisoned would cause an exception only if accessed.

Supposing that the reporting of DUE is delayed until the erroneous data is consumed, the false errors due to fetched but overwritten ECC words will be ignored. Then the program failure rate then becomes $\sum_{\text{memory cache lines}} (\lambda_{\text{DUE}} \cdot FEA)$ instead of $\sum_{\text{memory cache lines}} (\lambda_{\text{DUE}} \cdot MVF)$.

TABLE I: Benchmarks used for evaluation

Name (shortened)	Benchmark description	Category	Input size	Built-in verification
BlackScholes (BlkSch)	Option pricing	Partial Differential Equation	400M options	✓
Cholesky (Chol.)	Cholesky factorization	Dense linear algebra	8192×8192 matrix	✓
CG	Conjugate Gradient [12]	Sparse linear algebra	16Mi×16Mi matrix	✓
DGEMM (MM)	Matrix multiplication	Dense linear algebra	5120×5120 matrix	✓
FFT	Stockham Fast Fourier Transform	Spectral method	1 dimension, 2Mi points	✗
Gauss-Seidel (G-S)	Heat diffusion, Gauss-Seidel solver	Structured grid	4500×4500 grid	✗
Jacobi	Heat diffusion, Jacobi solver	Structured grid	4500×4500 grid	✗
KNN	K-nearest neighbours classification	Machine learning	500K points training, 5k testing sets	✗
PRK2 Stencil (Stencil)	Parallel Research Kernels stencil [25]	Stencil operation	16Ki×16Ki grid, 130 iterations	✓
Red-Black (R-B)	Heat diffusion, red-black solver	Structured grid	4500×4500 grid	✗
SMI	Symmetric matrix inverse	Dense linear algebra	4608×4608 matrix	✓
Stream	Stream Triad [19]	Memory bandwidth benchmark	192MB	✓

IV. Experimental Methodology

To compare FEA and the state-of-the-art vulnerability metrics with the program failure rates, we examine outcomes when injecting errors in native runs of 12 parallel benchmarks, and measure vulnerability ratings precisely using a cycle-accurate simulation infrastructure. We first explain the simulator setup, then detail the error injection experiments.

We use parallel benchmarks from various origins, selected because they represent a varied set of application types, and because they can all have the validity of their output verified. These parallel benchmarks are listed in Table I and are all written for a shared-memory environment using the OmpSs programming model [8], with tasks that use real data-flow dependencies.

A. Measuring the Memory Vulnerability

We extend TaskSim [23, 24], a cycle-accurate task-trace based multicore simulator, to compute the exact memory vulnerability ratings of data. Its infrastructure relies on task-based execution models to generate detailed traces for each task, including the basic blocks that are executed and memory addresses that are accessed. TaskSim's multicore architecture simulator then simulates parallel runs in detail by fetching and executing all instructions, using a simple core model and a full memory hierarchy. The simulator also relies on a real runtime system, to schedule the tasks across the simulated hardware. The memory is simulated using Ramulator [14].

To compute the various vulnerability metrics, we capture all loads and stores and the time at which they reach main memory. We then update at each access the necessary counters per memory location: time before stores, time before loads whose contents will be overwritten (FEA safe time), time before remaining loads, and time before the end of the program. We also count the number of loads and stores. From this data, we compute the fraction of time that each location is vulnerable, the DVF and the store-to-load ratio. For MVF and FEA the time until the end of the program is counted as vulnerable only for the program's output. We only update these counters during the Region Of Interest (ROI), thus excluding the setup and clean up parts of the benchmarks. We compute all metrics at a 64 bit granularity, which is the granularity used for SECDED and a subset of the granularity commonly used in ChipKill. Finally, we also report for each memory page of a benchmark the average fraction of time it resides in cache.

TABLE II: TaskSim cache parameters

cache	shared	assoc.	size	latency	MSHRs
L1D	private	8-way	32kB	4 cycles	32
L2	private	8-way	256kB	12 cycles	32
L3	shared	16-way	20MB	28 cycles	128

We trace applications on an Intel x86_64 Xeon E5-2670 and simulate a multicore architecture whose configuration mirrors the Xeon E5-2670's characteristics. It consists of 8 cores running at a frequency of 2.6GHz, each with a reorder buffer of 192 entries, and one thread per core. The cache hierarchy's parameters are summed up in Table II. All cache levels have 64B lines, write-back and write-allocate policies, are non inclusive, and track outstanding misses in Miss Status Handling Registers (MSHRs). Ramulator simulates 4GB of DDR4 DRAM memory, organised in one rank of 4Gb x8 chips clocked at 2400MHz.

B. Measuring the Program Outcome

To analyse and validate the memory vulnerability metrics, they need to be compared against the outcomes of injecting errors in the memory of the benchmark. Thus, we inject errors in native parallel runs on a real system, using the same Xeon E5-2670 as modelled by the simulation infrastructure.

Errors are injected using a separate thread, at a uniformly random point in the targeted application-level data, and at a uniformly random time during the ROI. The injector thread sleeps for the selected amount of time, then injects either a number of bit flips (1, 2 or 3), where random bits in the selected 64 bit word are flipped, or a DUE. A DUE consists in poisoning the data, and always causes an incorrect program outcome if and only if the data is consumed. In practice, it consists of inserting a `NaN` in floating-point data or a very high value for integer data, causing a crash if this data is used to index other arrays, or an incorrect result otherwise. In systems with any level of ECC, DUE is the more likely type of error to be encountered. However, looking at both bit flips and DUE allows to both gauge the false DUE rate, and to compare the metrics with non-ECC failure rates as well.

Each experiment consists of a single error injection in a single data structure. The program runs until it finishes abnormally or until completion, in which case the validity of its solution is verified, which is always done against the unmodified input data. If the program performs more work (e.g., more iterations) than the baseline, we classify its execution as incorrect. There are between 3 and 768 data structures

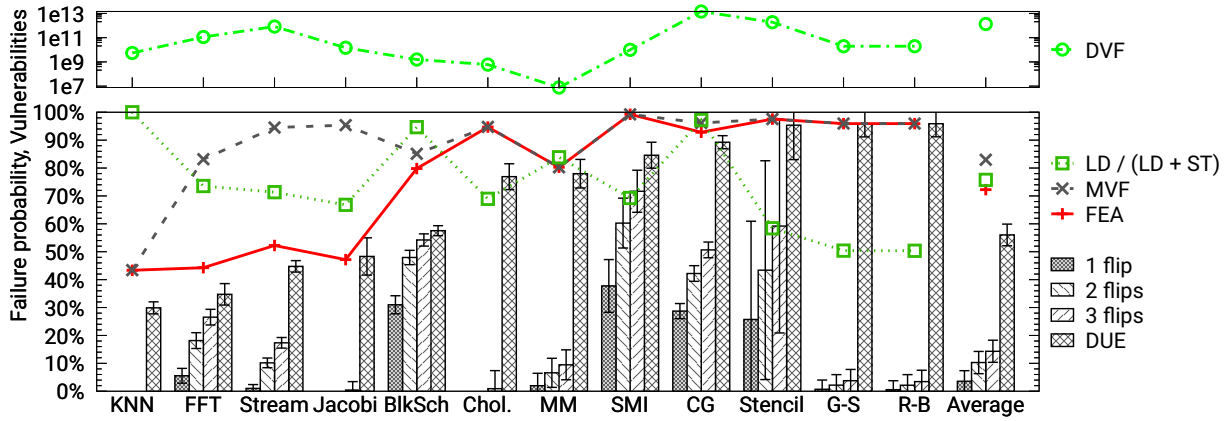

Fig. 1: Incorrect outcome frequencies from fault injections and vulnerability metrics, averaged per benchmark.
Bars represent the average failure rate when injecting faults in real runs, with 95% confidence intervals as whiskers.
Simulation-based vulnerability ratings are displayed as lines. Vulnerability metrics have values between 0 and 1, except DVF which is unbounded.

per benchmark, and we run at least 100 experiments per data structure. We also ensure a minimum of 1500 experiments per benchmark, which guarantees a 95% confidence interval of at most ± 2.5 percentage points for a discrete probability distribution with 2 possible outcomes (program success or failure).

Since the error is injected at the CPU level, the program consumes errors that would be hidden by caching behaviour. Indeed, an error in data that is cached would not be seen by the program as it would not be fetched from memory. We account for this masking by correcting the outcome frequency, using the fraction of time that data is cached obtained from simulations.

V. Comparing Vulnerability Metrics

In this section, we compare the failure probability when injecting errors against MVF, FEA, and other related metrics.

A. Outcomes from Fault Injections

The results of error injections in real runs are presented as bars in Figure 1, while the various vulnerability ratings obtained from cycle-accurate simulations are presented as lines. The bars represent the average number of incorrect program outcomes obtained from error injections, with whiskers on top of the bars representing the 95% confidence interval for the failure probability. Benchmarks are sorted in increasing failure probability for DUE.

The failure probabilities are varied per benchmark and per data structure within each benchmark. First, we should note that the probability of failure due to a DUE is always bigger than for any number of bit flips, sometimes by several orders of magnitude. Similarly, failures are always increasingly probable with the number of bits flipped.

Some benchmarks are very tolerant to faults, such as KNN, which uses training points that are assigned a class, and classifies a different set of points based on the classes of their nearest neighbours. Indeed, a bit flip in the biggest data structure, which is the set of points used for training, modifies at most one point, which in itself does not significantly alter the outcome of the classifications. Benchmarks such as Cholesky, Jacobi, Gauss-Seidel and Red-black are more resilient to bit flips due to the nature of their data. Modifying a single of many floating point values is likely to only have a small impact on the final outcome. The outcome can however be significantly perturbed if this error is significant for Red-black and Gauss-Seidel, as indicated by the high failure probabilities for DUE. On the more vulnerable side of the spectrum, modifying any data in the precise computations of BlackScholes or the numerically unstable SMI directly impacts the output computations. Similarly, the biggest data structures in CG are the matrix values, columns, and rows. The two latter of those data structures are integer data, and used to index the values, thus even a single bit flip can cause a benchmark crash.

The difference between bit flip and DUE failure probability constitutes the amount of false errors caused by the ECC. Depending on the ECC level, different bars must be considered. For example for SECDED, double bit flips cause DUE while single bit flips are corrected silently, thus the false DUE probability is the difference between the DUE and double bit flip failure probabilities. For more complex codes such as ChipKill, additional assumptions must be made as to the physical distribution of errors, thus the exact probability of false DUE is harder to derive.

B. Comparing Metrics and Fault Injections

The various vulnerability metrics are presented as lines in Figure 1. They are MVF (dashed line), defined in Section II-B and similar to the *safe ratio* [18], FEA (unbroken line), defined in Section III-A, the store-to-load ratio [10] (dotted line), and finally DVF [28] in the separate graph above. MVF consistently gives an upper bound of all failure probabilities, both due to bit flips and DUE. However the MVF ratings of most benchmarks are between 80% and 100%, and thus not very useful to distinguish vulnerable benchmarks or data structures from safe ones.

FEA often has the same value as MVF, and also always provides an upper bound on all failure probabilities. FEA gives a much tighter bound for the Jacobi, FFT and Stream benchmarks, and a slightly tighter bound for BlackScholes and

CG. This is due to these benchmarks having data structures that are overwritten and not updated, thus false errors that can be ignored. Overall, FEA correlates very well with the failure probability for DUE injections, and noticeably overestimates the vulnerability only of KNN and BlackScholes. This is due to these benchmarks having a relatively small memory footprint, thus with an important error masking effect from the cache. The only two other benchmarks where FEA differs from the DUE failure probability are Cholesky and SMI, which are two benchmarks respectively factorising and inverting a symmetric matrix. In these benchmarks, the diagonal blocks are fetched entirely from memory, however only half of this data is used and thus affects the program outcome. The unused data that is fetched is not overwritten however, thus FEA does not identify it as safe. The reason why unimportant data is accessed is unclear, as the accesses are part of Lapack library calls. All remaining benchmarks have an average FEA value very close to the average failure frequency when injecting DUE. As such, it also gives a reasonable good proxy and consistent upper bound for the bit flip error injections.

As the store-to-load ratio ST/LD [10] is unbounded and inversely correlated to the vulnerability, we normalize it as $LD/(LD+ST)$, and present this value as the dotted line in Figure 1. This transformation maintains the relative ordering of the ratings' values (in opposite direction), while bringing them back in the interval $[0, 1]$. We see several problems with this metric: KNN, which is a very safe benchmark, is rated with maximal vulnerability, and the vulnerability of several other benchmarks is lower than their failure probability. This metric can thus not be safely used as an upper bound on failure probability. On Cholesky, Gauss-Seidel and Red-black, the load-to-store ratio severely underestimates the probability of failure due to a DUE. This metric correlates better with the bit flip failure rates than the failure rate due to DUE, with Gauss-Seidel and Red-black rated lower than FFT and Stream, and those lower than BlackScholes and CG. However many other benchmarks contradict this correlation, such as DGEMM, Cholesky, Jacobi, KNN, and in particular PRK2 Stencil. While this metric might be a satisfying heuristic for online optimisations, due to the fact it is simple to compute, it is easy to see that it lacks the timing information to satisfyingly inform on vulnerability. One example is the iterate of CG, which is an iterative method that updates this vector at every iteration to get closer to the solution. The load-to-store ratio sees as many loads as stores, and thus rates it with a value of 50%, the lowest data structure of the whole benchmark. However the store immediately follows the load, as the iterate is simply updated, and a whole iteration subsides before the next update. Thus the iterate is in fact one of the most vulnerable data structures of CG, with 70% of incorrect outcomes for double bit flips and 96% for DUE.

The DVF metric [28] is not bounded either and its values are high and very spread out, hence we display them on a log scale at the top of Figure 1. The main factor impacting the DVF of a data structure is its size, which causes benchmarks such as CG and PRK2 Stencil to be rated with high DVF, however

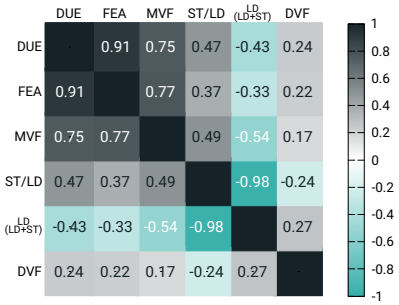

Fig. 2: Correlations between the failure rate from DUE injections and the various vulnerability metrics.

BlackScholes has low DVF while being the second most vulnerable benchmark for bit flips. Similarly, DVF correlates poorly with the probability of failures due to DUE, as Red-black, Gauss-Seidel and DGEMM are rated with middle to low DVF values when compared to other benchmarks. Finally, the fact that the metric is not bounded makes it harder to use at runtime, as the metric only has meaning when comparing values relative to each other.

C. Quantifying the Correlation Between Metric and DUE

We report in Figure 2 the Pearson correlation coefficients between all the vulnerability metrics and the DUE failure rate. For each pair of metrics, the correlation is computed using all of the data structure values from every benchmark, normalised for data structure size within each benchmark. Metrics are sorted using the magnitude of correlation with DUE. In addition to the metrics used so far, we display the original ST/LD metric from Gupta et al. [10], which allows to verify that the normalised version, $LD/(LD+ST)$, gives the same information. Indeed, both metrics correlate with a factor of -0.98, meaning a very strong negative linear correlation.

Confirming our previous observations, the DVF is the metric correlating the worst with DUE fault rates. Furthermore, it correlates weakly (≤ 0.27) with all other metrics. While the DVF provides some information on vulnerability, it seems to be the least suitable metric. The ST/LD and $LD/(LD+ST)$ metrics correlate with the DUE failure rate with coefficients of 0.47 and -0.43 respectively. These two metrics use the same information, however it seems that the distribution of the non-normalised version is slightly better at informing on vulnerability. Interestingly, Gupta et al. verify the relevance of the ST/LD metric by comparing it against a MVF value obtained from simulation, and report a correlation of -0.32. This is contrary to our findings, even when computing the correlation with MVF per memory page instead of per data structure (respectively 0.37 for ST/LD and -0.40 for $LD/(LD+ST)$). It is worth noting however that in their work, Gupta et al. use benchmarks from different suites, for which they report much lower MVF (ranging from 1.7% to 22.5%) than what we measure for our benchmarks. This difference could be the cause for the correlation discrepancies.

The MVF metric correlates rather well with FEA (0.77) and with the DUE failure rate (0.75), while FEA correlates really

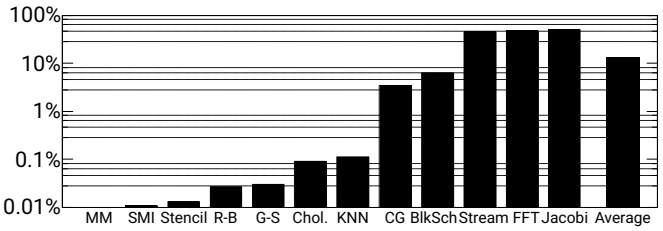

Fig. 3: Reduction in failure rate from ignoring false errors

well with DUE (0.91). This confirms that FEA improves the precision from MVF, and that the timing information which is exploited by these two metrics only is key in providing useful information on memory reliability.

D. Failure Rate Reduction from Delaying Error Reporting

The metrics evaluation reveals that there are a number of benchmarks where the impact of false errors significantly causes the failure probability to be overestimated. We quantify here the effects of ignoring false errors by delaying the reporting of DUE until they are actually consumed. As explained in Section III-A, the program failure rate due to DUE becomes $\lambda_{DUE} FEA$ instead of $\lambda_{DUE} MVF$. Thus, we present in Figure 3 the reduction in the failure rate from deferring the reporting of errors and ignoring false DUE, that is, $(MVF - FEA)/MVF$.

A number of benchmarks have their average failure rate reduced by over 45%: Stream with 45.4%, FFT with 48.1%, and Jacobi with 50.5%. FFT and Jacobi are benchmarks that can not perform in-place computations. Instead, these algorithms use additional memory to store results of intermediate computations, which creates memory accesses that overwrite data. Other benchmarks only overwrite one of their data structures, such as CG and BlackScholes. The remaining benchmarks have no or negligible amounts of data (less than 0.1%) that is accessed without being consumed. The overall average reduction in failure rate is of 12.75%.

VI. Conclusion

A number of metrics aim at quantifying the risk associated with encountering an error in data in memory. Comparing these metrics with the likelihood of incorrect program outcomes due to an error in memory indicates that the metric we introduce in this paper, FEA, is the most accurate one. This is especially true for DUE, which is the most common type of error for ECC-protected memory. Simultaneously, FEA also consistently provides an upper bound on the failure probability. This can be explained by the fact that it takes into account timing effects, as opposed to metrics based on access counts, and by the fact that it takes into account false errors that are overlooked by MVF. This work opens the door to runtime-level optimizations that can now accurately model the risk associated with any given data by providing a more precise metric that allows to identify a wider range of memory vulnerability profiles across memory.

Acknowledgements

This work has been supported by the RoMoL ERC Advanced Grant (GA 321253), by the European HiPEAC Network of Excellence, by the Spanish Ministry of Economy and Competitiveness (contract TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328), by the Spanish Government (Severo Ochoa grant SEV-2015-0493) and by the European Union's Horizon 2020 research and innovation programme (grant agreements 671697 and 779877). L. Jaulmes has been partially supported by the Spanish Ministry of Education, Culture and Sports under grant FPU2013/06982. M. Moretó and M. Casas have been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramón y Cajal fellowships RYC-2016-21104 and RYC-2017-23269.

The authors would like to thank Francesc Martínez Palau for his precious help and support with the TaskSim infrastructure.

References

[1] Advanced Micro Devices, Inc., "AMD64 Architecture Programmer's Manual Vol. 2: System Programming," Publication # 24593, 2018, rev. 2.30.
[2] J. H. Ahn *et al.*, "Future scaling of processor-memory interfaces," in SC'09, 2009. DOI: 10.1145/1654059.1654102.
[3] ARM, "ARM Cortex-A55 Core," Technical Reference Manual 100442_0100_00_en, 2017, Revision: r1p0.
[4] G. Bronevetsky *et al.*, "Soft error vulnerability of iterative linear algebra methods," in ICS'08, 2008. DOI: 10.1145/1375527.1375552.
[5] M. Casas *et al.*, "Fault resilience of the algebraic multi-grid solver," in ICS'12, ACM, 2012, DOI: 10.1145/2304576.2304590.
[6] G. Chen *et al.*, "Compiler-directed Selective Data Protection Against Soft Errors," in ASP-DAC'05, 2005, DOI: 10.1145/1120725.1121000.
[7] T. J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," IBM Microelectronics, white paper, 1997.
[8] A. Duran *et al.*, "OmpSs," Parallel Process. Lett., vol. 21, no. 2, pp. 173–193, 2011, DOI: 10.1142/S0129626411000151.
[9] J. Elliott *et al.*, "Evaluating the Impact of SDC on the GMRES Iterative Solver," in IPDPS, 2014, DOI: 10.1109/IPDPS.2014.123.
[10] M. Gupta *et al.*, "Reliability-Aware Data Placement for Heterogeneous Memory Architecture," in HPCA, 2018. DOI: 10.1109/HPCA.2018.00056.
[11] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer Manual Vol. 3: System Programming Guide," 325384, 2017, version 052.
[12] L. Jaulmes *et al.*, "Exploiting Asynchrony from Exact Forward Recovery for DUE," in SC'15, 2015, DOI: 10.1145/2807591.2807599.
[13] U. Kang *et al.*, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.
[14] Y. Kim *et al.*, "Ramulator," IEEE Comput. Archit. Lett., vol. 15, no. 1, pp. 45–49, 2016, DOI: 10.1109/LCA.2015.2414456.
[15] A. Kleen, "mcelog," presented at the Linux Kongress, 2010, pp. 159–166,
[16] S. Levy *et al.*, "Lessons Learned from Memory Errors Observed over the Lifetime of Cielo," in SC '18, 2018.
[17] X. Li *et al.*, "Architecture-Level Soft Error Analysis: Examining the Limits of Common Assumptions," in DSN, 2007. DOI: 10.1109/DSN.2007.15.
[18] Y. Luo *et al.*, "Characterizing Application Memory Error Vulnerability," in DSN, 2014, DOI: 10.1109/DSN.2014.50.
[19] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current HPC," IEEE Comp. Soc. TCCA Newsletter, pp. 19–25, 1995.
[20] M. Mehrara *et al.*, "Exploiting Selective Placement for Low-cost Memory Protection," Trans. Archit. Code Optim., vol. 5, no. 3, 14:1–14:24, 2008, DOI: 10.1145/1455650.1455653.
[21] Micron Technology, Inc., "ECC Brings Reliability and Power Efficiency to Mobile Devices," white paper, 2017.
[22] S. S. Mukherjee *et al.*, "A Systematic Methodology to Compute the Architectural Vulnerability Factors," in MICRO 36, 2003, DOI: 10.1109/MICRO.2003.1253181.
[23] A. Rico *et al.*, "Trace-driven simulation of multithreaded applications," in ISPASS, 2011. DOI: 10.1109/ISPASS.2011.5762718.
[24] A. Rico *et al.*, "On the Simulation of Large-scale Architectures Using Multiple Application Abstraction Levels," ACM Trans. Architec. Code Optim., vol. 8, no. 4, 36:1–36:20, 2012, DOI: 10.1145/2086696.2086715.
[25] R. F. V. d. Wijngaart *et al.*, "The Parallel Research Kernels," in HPEC, 2014. DOI: 10.1109/HPEC.2014.7040972.
[26] M. Wilkening *et al.*, "Calculating AVF for Spatial Multi-Bit Transient Faults," in MICRO 47, 2014. DOI: 10.1109/MICRO.2014.15.
[27] D. H. Yoon *et al.*, "Virtualized and Flexible ECC for Main Memory," in ASPLOS XV, 2010, DOI: 10.1145/1736020.1736064.
[28] L. Yu *et al.*, "Quantitatively Modeling Application Resilience with the Data Vulnerability Factor," in SC'14, 2014, DOI: 10.1109/SC.2014.62.