Memory Array Protection: Check on Read or Check on Write?

Panagiota Nikolaou*,Yiannakis Sazeides*, Lorena Ndreu*, Emre Özer† and Sachin Idgunji†
*University of Cyprus †ARM

***Abstract*—This work introduces Check-on-Write: a memory array error protection approach that enables a trade-off between a memory array's fault-coverage and energy. The presented approach checks for error in a value stored in an array before it is overwritten rather than, as currently done, when it is read (check-on-read). This aims at reducing the number and energy of error code checks. This lazy protection approach can be used for caches in systems that support failure-atomicity to recover from corrupted state due to a fault.**

The paper proposes and evaluates an adaptive memory protection scheme that is capable of both check-on-read and check-on-write and switches between the two protection modes depending on the energy to be saved and fault coverage requirements. Experimental analysis shows that our technique reduces the average dynamic energy of the L1 instruction cache tag and data arrays by 18.6% and 17.7% respectively. For the L1 data cache, this is 17.2% and 2.9%, and the savings are 13.4% for the L2 tag array. The paper also quantifies the implications of the proposed scheme on fault-coverage by analyzing the mean-time-to-failure as a function of the transient failure rate.

I. Introduction

Technological developments have facilitated the continuous miniaturization of devices on silicon chips. Unfortunately, the scaling of other key design parameters has not followed suit and have elevated power and reliability into prime design constraints across all computing market segments. The soft-error rate per bit decreases at a slower rate than device scaling and consequently the soft error rate per microprocessor has been increasing [4]. Additionally, fixed power envelopes and limited supply voltage scaling may necessitate low voltage operation that exacerbates the probability of parametric and soft-error failures [19], [4]. Clearly, power and reliability are increasingly becoming intertwined.

These distressing trends have created an impetus for the development of cost-effective techniques to address the power and reliability challenges. This is particularly crucial for processor memory arrays, such as caches, that take most of the real-estate in processors and contain numerous, usually minimum sized, vulnerable to faults SRAM cells. Existing techniques used to protect memory arrays in current processors, such as error-correcting-codes (ECC) [6], and redundancy [11], can help mitigate the problem but incur considerable area and energy costs. For instance protecting an array with a 72-64 single-error-correction double-error-detection (SECDED) code requires 12.5% additional cells. Very importantly, accessing the code bits and checking for errors increases energy consumption. Table I shows the normalized energy breakdown for a read hit in three different caches for their respective tag and data arrays. The energy is divided into three components: reading the data, reading the code bits and checking for errors using the ECC logic. Details about the configuration and methodology used to obtain these results are given in Section VI. The data in Table I clearly indicate that error protection entails a significant energy overhead ranging from 11% to 23% depending on the array.

	L1 Dcache		L1 Icache		L2 Cache	
	Tag	Data	Tag	Data	Tag	Data
Data bits	77.1%	88.9%	77.1%	81.9%	77.7%	88%
Code bits	16.9%	11.0%	16.9%	17.9%	17%	11.1%
ECC logic	6%	0.1%	6%	0.2%	5.3%	0.9%

TABLE I. Read Hit Energy Breakdown for different types of Arrays

The goal of this paper is to reduce the energy spent for protecting a memory array from faults with minimal impact on overall performance and fault-coverage. While such trade-off may be of little interest for a high-availability processor unwilling to compromise fault-coverage it may be very desirable for power and energy constrained commodity processors. To this end we introduce Check-on-Write (COW): an error protection approach that checks a value before it is overwritten. This is in contrast to the current practice of checking for error in a value each time is read from an array (check-on-read or COR). COW saves energy when reading a value by not discharging the bitlines used for reading the value's code bits and by not computing the value's code and checking for error. The COW approach can provide energy savings if an array has more reads than writes so that the energy spent for performing a read-check on writes is less than the energy consumed for checking data integrity on reads.

While the energy reductions aimed by COW are significant (Table I) its applicability to caches appears limited. This is a consequence of COW's lazy detection approach that allows a faulty value to be read and used and, therefore, corrupt other locations before COW checks the value for error. The corrupted content remains unrecoverable even if the faulty value that caused the corruption is detected and corrected by COW. This is in contrast to a COR scheme that can detect faults as soon as a value is read and, therefore, prevent the corruption of other locations.

The paper proposes and analyzes the performance of a memory protection scheme that aims to provide the same level of performance and fault-coverage as COR while achieving most of the energy benefits of COW. This is an adaptive scheme capable of operating in either COR or COW mode. It also requires support for failure-atomicity (FAT) and array scrubbing. The FAT is essential to recover from any state corruptions caused by COW. FAT is widely used in existing and proposed platforms and, therefore, there are many opportunities to implement and benefit from the proposed scheme [2], [1], [7], [16], [5]. Scrubbing is needed to check for errors in locations read but not written. The experimental analysis shows the proposed scheme to have a negligible impact on performance and to provide substantial energy savings in caches by giving up minimal fault coverage from transient errors.

II. Background

A memory array is typically protected against faults using data redundancy: include few extra bits per entry that encode the value stored in each array entry. Some of the most popular

COR Error	COW Error	FAT COW Error
NER	NER, DCE, DME, DUE	NER, fDCE,sDCE, DUE,DME
DCE	DCE, DUE, DME, NDE	fDCE,sDCE, DUE,DME,NDE
DUE	DUE, DME, NDE	DUE,DME,NDE
DME	DME, DUE, NDE	DME,DUE,NDE
NDE	NDE, DUE, DME	NDE,DUE,DME

TABLE II. ERROR CLASSIFICATION FROM CACHE SCOPE

coding schemes for memory data are parity and SECDED [6]. When writing an array entry with a value we need to generate the error code of the value and store it together with the value. When reading an array entry both the value and the code are read out and the code of the value is generated and checked against the code read from the array. This corresponds to the current approach of array protection where errors are checked when a value is read. We refer to this approach as Check-on-Read (COR). We assume henceforth, unless indicated otherwise, that the code used to protect the values in an array is an error correction code (ECC) capable of both detection and correction.

Specifically, on a read access when the two codes are the same the value is assumed to be fault free otherwise the codes are different and an error has been detected. Depending on the code strength and the type of error, the error can be corrected-detectable-correctable error (DCE) or it can be non-correctable-detectable-unrecoverable error (DUE). There are some cases of DCE that actually correspond to miscorrections or Detected Miscorrectable (DME). These occur when an error that exceeds the code strength is interpreted as a different correctable error. In the same vein, due to limited code strength a value may appear fault-free but in reality includes error. This case corresponds to a non-detectable error (NDE). If none of the above occurs then the value is error free or No Error (NER).

The above error classification is done by considering the behavior at the cache scope. Another useful error classification considers the implications of errors at the program scope [12]. More specifically, whether the erroneous value does not affect the program output , it causes a program abnormal failure or it does not lead to an abnormal termination but affects program output . In this paper we analyze memory array error behavior at the scope of the cache. The first column of Table II summarizes the different categories of errors when using COR protection.

III. CHECK-ON-WRITE (COW)

In this paper we propose the Check-on-Writes (COW) approach to reduce the energy overhead required for fault protection in memory arrays. COW is a memory array error protection approach that checks for error in a value before it is overwritten. This is unlike COR that does error checking every time a value is read. For the rest of the paper we assume that that when we compare COR and COW they use the same ECC code to protect values.

COW relies on *read-write memory array invariance*: each write to an array location becomes a read-check followed by a write. We denote the read-check before the write henceforth as *RBW*. A RBW is identical to a normal read in COR that reads the code bits and performs error checking. This invariance ensures: (i) that COW checks for faults, at least, all values that COR checks, and (ii) that each value read and checked by COR one or more times before overwritten it only gets checked once by COW. The first condition is important for providing same level of fault-coverage whereas the second can help reduce the energy if the reads are sufficiently more than the writes.

We illustrate in Fig. 1 the COW operation assuming an 11-7 SECDED code for an array with 2 entries initially containing zeros. Each write access is denoted by a $W_i[j]$ with i representing the write entry and j representing the write value. Similarly $R_i[j]$ denotes reading from entry *i* the value *j*. The example assumes a single bit-flip occurs in the array entry 1 between a time the location is written and read. The example shows also when the COR performs the check when the value is read and the correction it performs. The example shows that the faulty value checked by COR is checked also by COW but with a delay. The bit flip in entry 1 is detected and corrected before the value is overwritten, not when it is read.

COW burns less energy on reads as compared to COR by not discharging the bitlines that access the code bits and by not using the ECC logic to compute the code and check for errors. Thus, no energy is spent in bitlines, sense amps and the associated multiplexers. However, it burns more energy on writes than COR since it performs in addition to the write a normal read that includes the code bits and ECC logic. We will assume for now that the reads are sufficiently more than the writes. We will come back to this issue in Section V where we introduce an adaptive scheme to select between COR and COW.

COW has the potential to reduce energy, as compared to COR, but does so at the expense of fault-coverage. This is a consequence of the lazy approach of COW that checks a value not when is read but when it is about to be overwritten and can result in observing different error behavior than COR. For example, COW can render a fault that is detectable by COR to non-detectable, or a correctable error by COR to uncorrectable etc. Fig. 2 illustrates a case where the same word is flipped two separate times by soft errors at the same bit position and the strikes happen between two consecutive writes to the word. A read that occurs between the two faults will detect the first error when using COR and even correct, assuming at least single error correction capability. On other hand, COW will check the value before it is overwritten at which time the error is undetectable: a DCE by COR became NDE by COW.

The various error observed by COW as a function of the corresponding COR errors are summarized in the second column of Table II. When there is no fault between the first read of value and the time is overwritten the error behavior of COW is identical to COR. However, if faults occur then the error behavior can change depending on the number of errors. For example assuming a Hamming based SECDED code [6], when there is no error by COR on a read and by the time the value is overwritten there is: (i) one fault then COW will observe a DCE, (ii) two faults a DUE, (iii) three faults a DUE or DME, (iv) four faults a DUE or NDE, etc. Section VII compares the reliability of COR and COW.

Next we discuss some important COW limitations.

A. Checking Not Overwritten Values

There are several situations that require before execution proceeds to be sure about the integrity of the cache content (for example before we finish a program or a context switch). For such situations COW is proposed to rely on array scrubbing [14] to perform RBW for all array locations that their

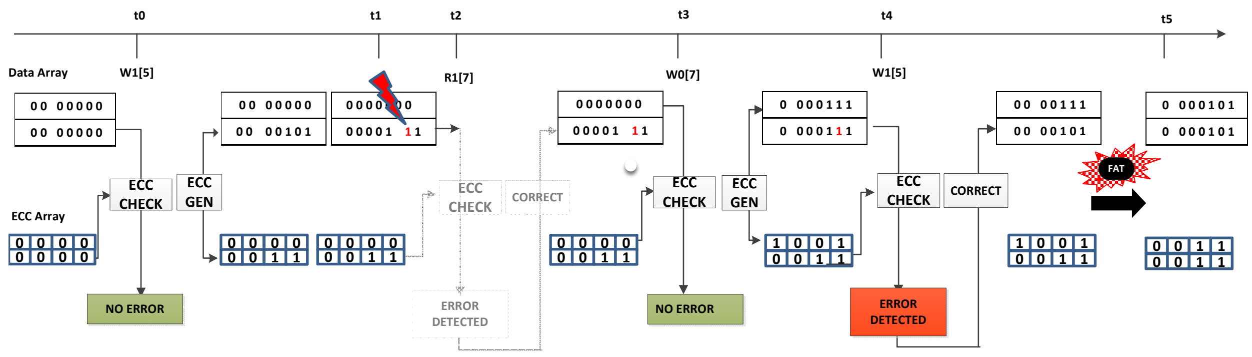

Fig. 1. COW Protection Approach

most recent access is a read. If the cost of tracking such values is an issue or the overhead for checking an entire cache is low, then one can scrub all values in an array to ensure that no read value in an array goes unchecked. Besides the end of the program and context switches, COW can perform scrubbing at regular intervals to reduce error accumulation. The overhead of scrubbing depends on its frequency. In the energy, performance and reliability analysis of the paper we consider the implications of scrubbing.

B. Read-Check before Write Overheads (RBW)

The RBWs are needed before every write and also when we perform array scrubbing. These clearly incur an energy overhead and may also hurt performance as they may delay a normal access. Our analysis for a single thread processor shows that RBWs may cause slight degradation due to access conflicts in L1 D$ but for the other caches, I$ and a lower level cache, the RBWs have negligible performance impact.

The RBW is performed in certain situations already by processors so COW does not add in these cases extra overhead. For example, for data caches protected with error-correction-code when the access datum is smaller than the ECC granularity [10]. In this case each write is preceded by a read that can be used as a RBW. Also, writeback caches need to read a dirty block before inserting a new one, thus the RBW is for free in these cases. We report on the performance impact of RBW in the experimental results.

C. Recovery from Corrupted State

The problem of COW's state corruption is illustrated in Fig. 1. Lets assumes that the faulty value read from entry 1 is written to entry 0 before entry 1 gets overwritten and checked. Even when the error in entry 1 is eventually detected and corrected the corrupted state can not be discarded. This indicates, in general, that even if COW can detect and correct all errors (by RBW or scrubbing) it can not recover from the possible corruption due to the direct or indirect use of faulty values and, therefore, upon error detection execution should be aborted. This essentially suggests that COW is mainly useful for error detection. We discuss subsequently, Section IV, how platforms that support failure-atomicity can help overcome this COW limitation.

D. How to deal with Crashes

COW is more vulnerable than COR to user and system crashes because a faulty value that is detectable and correctable by COR, with COW can corrupt the state and lead to a crash before COW gets to check it. We propose to mitigate

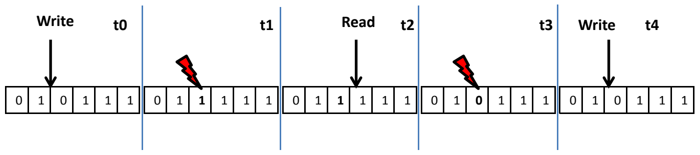

Fig. 2. Error correctable by COR but non-detectable by COW

this by introducing a wrapper for intercepting exceptions raised by an application. Effectively whenever an exception is raised this wrapper will need to first to establish whether the exception is possibly due to COW's delay detection and if so to attempt to correct the error, discard corrupted state and resume execution. The wrapper initiates a scrub of all cache arrays protected by COW. If only correctable errors are found then the errors should be corrected and the corrupted state should be discarded. We discuss in Section IV how failure-atomicity can be used to remove corrupted state. A complete discussion on how the wrapper and the exception handling will interact to assess the system health is an important subject but beyond the scope of this work.

In the case the scrubbing detects an uncorrectable error the corrupted state is discarded and what is propagated to the system is the original exception plus information that indicates the array(s) that contain uncorrectable error(s). If scrubbing does not detect an error then the exception is propagated to the system unmodified.

The proposed wrapper function is similar to a middle layer proposed for intercepting exceptions when using a symptom based fault detection in [9].

IV. COW AND FAILURE ATOMICITY (FAT-COW)

The main limitation of the COW is that it can not recover from corrupted state. In this section we explain how COW combined with failure-atomicity can overcome this problem.

A. Failure Atomicity (FAT)

Several existing and proposed hardware and software platforms provide support for failure-atomicity (FAT). FAT describes the property of having a sequence of operations to either all or none complete. When there is no completion the state of the system remains unaffected. This sequence is usually referred to as atomic.

Examples of such proposals are recovery-blocks and their variations [13], transactional-memory [7], map-reduce programming model [2], thread level speculation [16], and cloud programming models [1].

Failure in the context of FAT covers a range of unexpected events that can cause the sequence to abort, such as hardware failures, mispeculation, concurrency violation etc. When a failure occurs any changes/corruption by the sequence are discarded and execution resumes from the beginning of the sequence. In a FAT system it is possible for a failure to be detected with delay just before the sequence completes. Implementing FAT requires either or both hardware and software support. Depending on the implementation of FAT, the expected granularity of the sequences, and the type of failures detected, the overall overhead for FAT recovery can vary from few cycles to many cycles.

A basic FAT approach is to restart a program or task from the beginning when a failure occurs. But this may be undesirable for long running jobs or jobs that use many resources. A very well known and widely used technique for fault-tolerance that also provides FAT is checkpoint-rollback [5]. Checkpoint-rollback discards corrupted state and can recover previous fault-free state. This, typically, comes at a higher overhead as compared to schemes that only provide FAT.

Overall, there exist several proposals that leverage FAT for fault-tolerance but, as far as we know, none was aimed to trade-off between memory arrays energy and fault-coverage. What we propose next is to leverage FAT to discard corruption when using COW.

B. *FAT-COW*

COW can be combined with FAT (FAT-COW) to discard corrupted state when COW detects a fault. When we are executing an atomic sequence and COW is used for memory protection an error can be detected: (i) during the execution of the sequence before writes, (ii) at the end of the sequence before it completes when a scrub is performed, and (iii) during a scrub that is initiated to response to the wrapper intercepting an exception. In all these cases if an error is detected the corrupted state is discarded using mechanisms provided by FAT. If the error is correctable then the value is corrected and execution resumes from the start of the sequence.

A disadvantage of FAT-COW, as compared to COR, is that it requires longer time to restart execution in the case of a correctable error. In contrast, COR can detect and correct error inline with minimal performance overhead. This performance overhead of FAT-COW may be acceptable if failures happen rarely. We analyze later the implications of mean-time to FAT-COW recovery.

The third column of Table II summarizes the different error types when using FAT-COW protection scheme as compared to COR and COW. The errors are identical COW except that DCE errors are divided into two new categories: fast-DCE (fDCE) and slow-DCE (sDCE). fDCE occurs when COW detects a correctable error in the parity bits. Such error can be corrected inline with no need to recover from corruption since the value used is correct. The sDCE occurs when COW detects a correctable error in the data bits and requires discarding corrupted state. We present a reliability analysis of various FAT-COW errors, including fDCE and sDCE, in Section VII.

Fig. 1 illustrates the FAT-COW operation by showing that after COW detected and corrected the fault in entry 1 the FAT is used to remove the corruption effects in entry 0 and recover it to its previous content.

For read ease henceforth when we use the term COW we actually refer to FAT-COW.

V. Combined COR-COW Scheme

A typical computing system operates in two modes: user and kernel. Applications run in user mode and operating system code and services in kernel mode. Failures in user mode can be recovered in the worst case by aborting a job whereas a failure in kernel mode can lead to system crash. An undetected fault in kernel mode can be exposed to the user or communicate erroneous information to other system entities that can be catastrophic. As argued in [9] kernel recovery is more difficult and, therefore, to minimize system crashes/corruption we propose to use COW only during user mode.

This means that we need to switch between COR and COW depending on the execution mode. The complexity to provide both protections is minimal. An array with dual mode protection uses the same paths to generate and check codes as an array that checks for errors on reads. What is additionally needed, is to selectively clock gate on a read the discharging of code bitlines and of the ECC unit. Clock gating is widely used in processors and we assume that applying as outlined above should be straightforward. Also when in COW mode the array controller needs to introduce reads that check values before their overwritten. We do not discuss these implementation issues further.

Another implication of the dual mode operation is that whenever we enter kernel mode and the user mode is using COW, scrubbing is needed to ensure the integrity of the architectural arrays. This overhead is accounted in our evaluation.

A. *Adaptive Protection Scheme during User Mode*

The benefit from COW depends on the program behavior, specifically the number of reads vs writes and scrubbing overhead. We propose to select between the two protection modes at the granularity of atomic sequences based on their expected energy savings. The adaptive scheme considers at the end of each atomic sequence, depending on the number of different types of accesses, if the COW based scheme has advantage over COR and selects accordingly the protection mode to use for the next sequence.

In the experimental evaluation we will compare the benefits of an adaptive realistic-COW-COR scheme that considers the access during the most recent sequence to decide the next mode vs the baseline scheme that always does COR. We will also evaluate the potential of an adaptive oracle-COW-COR scheme that knows the number of read and write access for the next sequence. The adaptive schemes take a decision for which mode to use according to the number of reads and writes, and the energy to read, write, ecc-check and perform scrubbing as follows:

$$\#Reads \times E_{check} > \#Writes \times (E_{read} + E_{check}) + E_{scrub}$$

VI. Experimental Framework

We extend the validated, cycle accurate, simulator *sim-alpha* [3] to perform measurements using a high performance out-of-order superscalar processor. The key parameters of the processor configuration are: 32KB, 8-way, LRU, 64 per block, 1 cycle access L1 instruction cache, 32KB, 8-way, LRU, 64 per block, write-back, 3 cycle access L1 data cache, and an L2 unified cache 2MB, 16-way, LRU, 64B per block, 12-cycle hit latency, 255 cycles miss latency. A SimPoint-like tool [15] is used to select the regions to simulate from all SPEC2000 benchmarks. Each benchmark is run for 100 million instructions after fast forwarding to the representative regions. All benchmarks are used with their reference input sets.

To compute the MTTF rates for various types of errors (Table II), the intrinsic FIT/bit rates and per-bit AVF numbers are required. Our equations, analysis and inputs are based on the new approximate analytic model introduced in [17] for temporal multi-bit analysis. We use the bit-flip probability of 10^{-25}, which is projected by ITRS [18]. In our experiments we consider the average time between two consecutive reads or writes (Tavg) depending on the protection mechanism. These inputs are taken from the sim-alpha simulator for various benchmarks. The percentage of dirty words in caches and the expected cache AVF are also derived from simulation. We do not present the derived equations and model details due to space limitation.

We evaluate the adaptive scheme COR-COW schemes assuming atomic sequences that correspond to sequence of user mode instructions between consecutive system calls, denoted as syscall-sequences, as well as with sequences of length 10M, 1M and 100K instructions to investigate the trends with changing length.

We evaluate the potential for COW to reduce the energy consumed by the tag and data arrays of the L1 data, L1 instruction and the serially accessed L2. For these experiments we report the performance degradation caused by COW due to RBW and scrubbing operation. The energy results for the analysis are obtained using a CMOS 32nm Low Power cache model based on Artisan Memory Compiler. The code generation and checking logic that uses a Hsiao SECDED code [8] is implemented in Verilog, and the energy numbers for the logic are calculated using Synopsis PrimePower power modeling tool for the 32nm CMOS Low power (LP) library.

VII. Results

A. Energy Consumption

The first four pair of bars in Fig. 3 present the total normalized average dynamic energy savings for all the caches together provided by the realistic and oracle adaptive COR-COW schemes over COR for various interval lengths. The minimum and a maximum across all the benchmarks is shown in this graph as well as the actual average value in the center of each bar.

The results for the realistic adaptive scheme show that the average energy can be reduced by 12.3% per syscall sequences, by 12.2% for 10M, by 12.1% for 1M, and by 11.2% for 100K. The results also reveal that the benefits of the realistic adaptive scheme are close to the optimal. This suggests that the read/write ratio of an array remains relatively the same across program phases. The decrease with shorter interval is due to the increase overhead due to more frequent scrubbing.

A per benchmark analysis of the results per syscall sequence shows that the realistic adaptive schemes can provide substantial energy savings in all arrays except the L2 data array (actual data not shown due to space limitations). The adaptive scheme reduces the average dynamic energy of 32KB L1 instruction cache by 18.6% in the tag array and by 17.6% in the data array, of a 32KB L1 data cache 17.3% in the tag and 3.2% in the data array, and of a 2MB L2 cache only the tag array by 14.9%. The reason that L2 data does not have any benefits is that its ratio of reads to writes is low. We also observe in few cases the realistic scheme for the data array of the L1 data cache to perform slightly worse than the oracle. This is due to large changes in the read/write ratio across consecutive sequences. One other per benchmark observation is that the benefits for L1 instruction data cache remain the same for all the benchmarks due to the fact that the L1 instruction cache accesses are mostly reads.

B. Performance analysis

The last four pairs of bars in Fig. 3 show the relative average performance of the adaptive COR-COW schemes over the COR approach for different interval lengths. The results clearly show that the performance degradation is only notable for interval length of 100K, up to 2%, but for larger intervals and for the syscall sequences the performance overhead of scrub and RBW is negligible, never more than 0.6%.

C. Reliability

We present a transient error reliability analysis for DL1 data cache. The trends for the other caches are similar, so due to space limitations we do not present them. These results are with 10^{-25} bit-flip probability per cycle, however, the observations remain the same for larger bit-flip probabilities.

Figure 4 shows the MTTF for both COW and COR for the fDCE, sDCE, SDC and DUE. The categories NDE and DME, in Table II, are grouped in the category referred to as SDC because both can potentially result in Silent Data Corruption.

We observe that COR fDCE has smaller MTTF (more frequent) than COW sDCE. The reason is that COR fDCE corresponds to single errors that occur in the whole code-word (word and ECC bits), in contrast with COW sDCE that represents the MTTF for errors that occur only in the data. COW fDCE MTTF measures the failures due to errors in the ECC bits only. The combined sDCE and fDCE MTTF for COW (not shown) is still slightly larger (more rare) than the COR fDCE. This occurs because some errors that are correctable by COR turn, mainly, into DUEs for COW.

Fig. 4 shows that COR DUE is larger than COW DUE by 1 order of magnitude. This supports the reasoning for what causes longer combined DCE for COW.

SDC MTTFs are also shown. Notice that the MTTF for the COW approach is 20 orders of magnitude lower. This reduction is mainly due to two bit flips to the same bit in the same word. Nonetheless, the value is still extremely large 10^{24} years. Qualitatively, what we are proposing is to save energy by converting some DUEs to SDC errors.

Finally, we present an availability analysis for sDCE errors for the different caches. We use the MTTF obtained for syscall sequences and observe the trends with changing the mean-time-to repair (MTTR). Availability is given by MTTF/(MTTF+MTTR). The results indicate that the system is available more than 99.9% when MTTR is smaller than 0.01 years. For larger MTTR, the L2 arrays are more sensitive to failures. This indicates that if the MTTFs are in the order of

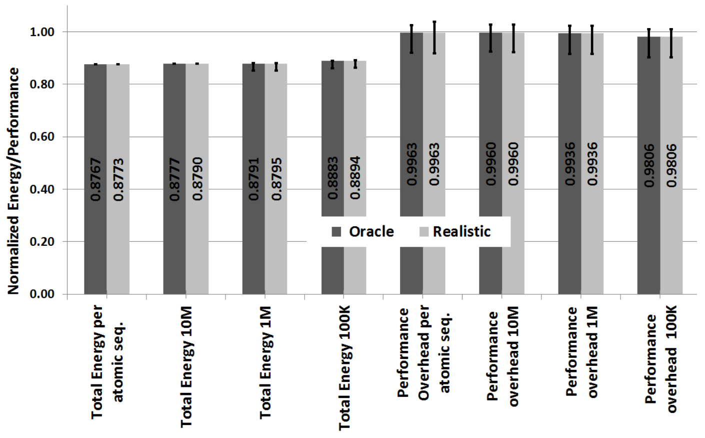

Fig. 3. *Energy and Performance Results*

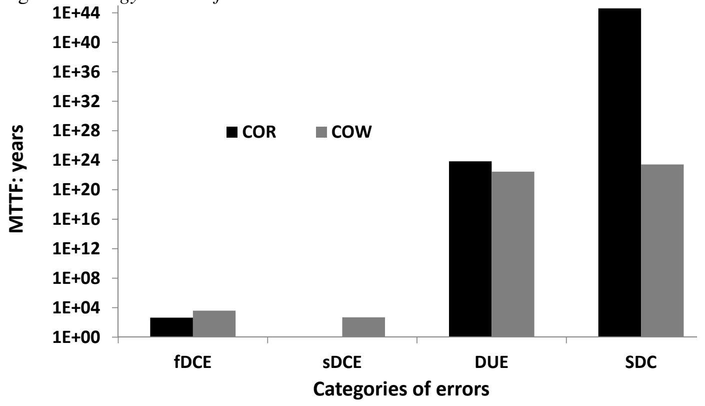

Fig. 4. *MTTF Rates for COR and COW*

what is shown in Fig. 4 the impact of COR-COW to availability is negligible.

VIII. Conclusions

This paper proposes Check-on-Write (COW) a new protection mechanism for cache arrays. COW checks errors on every write access rather than read access as done in the conventional protection schemes (Check-on-Read (COR)). COW can be combined with failure-atomicity to avoid corruptions. An adaptive version of the COW scheme is also proposed where switching between the COW and COR modes is decided dynamically depending on the processor execution mode or energy savings with a scrubbing support needed when switching from COR-COW to COR.

An experimental analysis is performed to compare the energy, performance and reliability results of the adaptive COR-COW scheme to the baseline COR scheme when applied to cache arrays inside a processor. Our results show that cache dynamic energy consumption can be reduced considerably ranging from 3% to 19%. The performance overhead due to scrubbing and recovery is very minimal (i.e. less than 0.5%). The fault coverage of the adaptive COW scheme is less robust as compared to COR. However, this comes with considerable savings in dynamic energy, which is a first-class design parameter in embedded and mobile computing systems.

IX. Acknowledgements

This work is supported by "EuroCloud, Project No 247779" of the European Commission 7th RTD Framework Programme Information and Communication Technologies: Computing Systems, and by the University of Cyprus. The authors would like to thank Marios Kleanthous and Quentin Minster for their help with the paper preparation and also the anonymous reviewers for their constructive critique and suggestions.

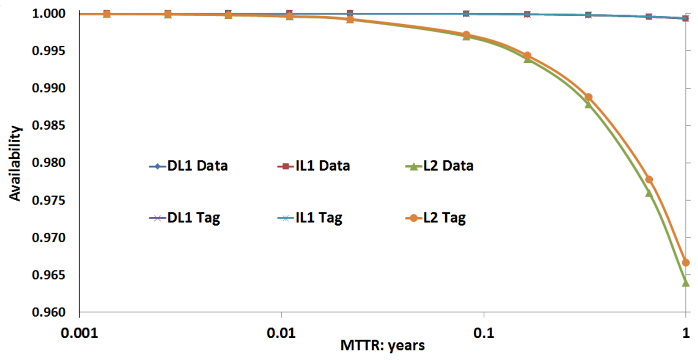

Fig. 5. *Availability for sDCE COW scheme*

References

[1] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud computing for everyone," in *SOCC*, 2011.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[3] R. Desikan, D. Burger, S. Keckler, and T. Austin, "Sim-alpha: a validated execution driven Alpha 21264 simulator," CS Dept., University of Texas at Austin, Tech. Rep. TR-01-23, 2001.

[4] A. Dixit and A. Wood, "The impact of new technology on soft error rates," Mar. 2011.

[5] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[6] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 26, no. 2, pp. 147–160, 1950.

[7] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on computer architecture*, ser. ISCA '93, 1993.

[8] 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.

[9] M. lap Li, P. Ramach, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *In Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems(ASPLOS*, 2008.

[10] K. M. Lepak and M. H. Lipasti, "Silent stores for free," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 33, 2000, pp. 22–31.

[11] P. J. Meaney, S. B. Swaney, P. N. Sanda, and L. Spainhower, "IBM z990 soft error detection and recovery," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 419–427, Sep. 2005.

[12] S. Mukherjee, J. Emer, and S. Reinhardt, "The soft error problem: an architectural perspective," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, feb. 2005, pp. 243 – 247.

[13] B. Randell, "System structure for software fault tolerance," *SIGPLAN Not.*, vol. 10, no. 6, pp. 437–449, Apr. 1975.

[14] A. Saleh, J. Serrano, and J. Patel, "Reliability of scrubbing recovery-techniques for memory systems," *Reliability, IEEE Transactions on*, vol. 39, no. 1, pp. 114 –122, apr 1990.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS10*, Oct. 2002.

[16] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *ISCA*, 1995, pp. 414–425.

[17] J. Suh, M. Manoochehri, M. Annavaram, and M. Dubois, "Soft error benchmarking of l2 caches with parma," in *SIGMETRICS*, 2011, pp. 85–96.

[18] The International Technology Roadmap for Semiconductors, "Edition 2010," ITRS, Tech. Rep., Dec. 2010. [Online]. Available: http://www.itrs.net

[19] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off cache capacity for reliability to enable low voltage operation," in *ISCA35*, Jun. 2008, pp. 203–214.