SMARTag: Error Correction in Cache Tag Array by Exploiting Address Locality

Seyedeh Golsana Ghaemi*, Iman Ahmadpour*, Mehdi Ardebili†, and Hamed Farbeh‡
*Department of Computer Engineering, Sharif University of Technology
Tehran 11155-11365, Iran
Email: gghaemi@ce.sharif.edu, ahmadpour@ce.sharif.edu
†College of Engineering, Tehran University
Tehran 14174-66191, Iran
Email: m.ardebili@ut.ac.ir
‡School of Computer Science, Institute for Research in Fundamental Sciences (IPM)
Tehran 19538-33511, Iran
Email: farbeh@ipm.ir

***Abstract*—Soft errors in on-chip caches are the major cause of processors failure. Partitioning the cache into data and tag arrays, recent reports show that the vulnerability of the latter is as high as or even higher than that of the former. Although Error-Correcting Codes (ECCs) are widely used to protect the data array, their overheads are not affordable in the tag array and its protection is conventionally limited to parity code. In this paper, we propose *Similarity-Managed Robust Tag* (SMARTag) technique to provide the error correction capability in parity-protected tags. SMARTag exploits the inherent similarity between the upper parts of the tags in a cache set to share these parts between addresses and ECCs. Using SMARTag, the cache access time is intact since the ECC part is bypassed in normal cache operation and no extra memory is required since ECCs are stored in available tag space. The simulation results show that SMARTag is capable of correcting more than 98% of errors in the tag array, on average, and its energy consumption, area, and performance overhead is less than 0.2%.**

***Index Terms*—Address locality, error-correcting code, on-chip caches, soft errors, tag array.**

1. Introduction

Soft errors due to energetic particles strike are known as the main reliability concern in computer systems [1], [2], [3], [4], [5]. On-chip cache memories, as the largest processor components [6], [7], are the most vulnerable part to soft errors [2], [6], [7], [8], [9]. Vulnerability of cache memories can result in incorrect outputs and system failures. Hence, protecting the contents of the cache memories is of decisive importance.

Both data and tag arrays in the caches need error protection capability for a reliable operation. However, most existing studies have focused on protecting data array. Employing Error-Correcting Codes (ECCs) are the dominant approach for protecting data array [4], [6], [7], [10], [11]. However, the energy consumption, performance, and area overhead of ECCs in tag array are significantly higher than that in data array. Because of unaffordable ECC overheads, Error-Detecting Codes (EDCs), e.g., parity check, are considered for tag protection in most of the commercial processors as well as research studies [12], [13], [14], [15].

Several studies addressed the vulnerability of tag array and demonstrated that it can be even more vulnerable than data array [16], [17], [18]. To provide error correction capability, recent techniques have tried to replicate the content of parity-protected tag array [12], [13], [15]. However, a large amount of tags in these techniques remain unreplicated and uncorrectable or a significant overhead is imposed to provide a full-replication of the tags.

In this paper, we propose *SMARTag (Similarity-MAnaged Robust Tag)* technique to provide ECC protection in a parity-protected tag array. *SMARTag* exploits the inherent redundancy in the tags of a cache set to share the upper parts of the tags between the addresses and ECCs. Based on the address locality in memory accesses, it is highly probable that upper bits of the tags in a set be similar. *SMARTag* takes the advantages of this similarity and allocates the upper parts of the tags to ECCs for protecting these tags. To this aim, the upper part of one tag remains unchanged while the upper parts of other tags are allocated to ECC. The ECC(s) of a set is updated for each change in the tag content, which happens on a cache miss. For a tag access, the ECC is bypassed and the tags containing the ECC retrieve their upper part from the unchanged tag. ECC part is activated only when the parity decoding logic detects an error in a tag. By storing ECCs in the available upper parts of the tags and bypassing it, *SMARTag* requires no dedicated memory for storing ECCs and imposes no ECC decoding latency.

SMARTag is evaluated using gem5 cycle-accurate simulator [19] running workloads from SPEC CPU2006 benchmark suite [20] for both single- and multi-core systems. The simulation results show that a negligible percentage of

978-3-9819263-0-9/DATE18/©2018 EDAA

tag remains parity-protected and SMARTag is capable of correcting more than 98% of errors in the tag array. This is achieved by imposing only 0.03% performance overhead.

The rest of this paper is organized as follows. Related work is explored in Section 2. In Section 3, we explain our motivation and observations about the similarity in tags and introduce the proposed SMARTag technique. Section 4 presents simulation system setup and results. Finally, Section 5 concludes the paper.

2. Related Work

In [13], a small fully-associative cache, so-called *Tag Replication Buffer (TRB)*, is inserted besides the cache to keep a replica for the recently accessed tags. Since the tags of clean cache lines can be recovered by invalidating the line and re-fetching the line from lower memory level, *TRB* is allocated to only dirty lines to increase the replication probability. However, the evaluation results indicated that a large fraction of tag still remain unreplicated. To guarantee a full protection, the authors suggested writing back the dirty lines without replica.

SimTag technique considers the similar tags in adjacent cache sets as the replicas of each other. Although the overhead of this technique is not considerable, a large fraction of tags remain unprotected due to high probability of the lack of similar tags in adjacent set or the eviction of the replicas [12].

Replicated in Altered Tag (RAW-Tag) technique [15] tries to overcome the drawbacks of the SimTag technique. In RAW-Tag, for a cache miss in a set, the adjacent sets are searched to find a replica for the incoming tag. If no similar tag is found, a prefetching operation is initiated on the first write access to the unreplicated tag for fetching a replica to the cache. On the other hand, both adjacent sets are searched on the eviction of a line to check whether the evicting line is a replica for another line. If so, an early writeback is conducted for the lines whose replica is evicting while no other replica is found for it in other adjacent set.

As observed, previous studies tried to provide error correction in a parity-protected tag array by replicating the tag entries. The studies that exploited the inherent redundancy in the tag contents focused only on the similarity of the tags in adjacent cache sets [12], [15]. To the best of our knowledge, our proposed SMARTag is the first technique that exploits the similarity of the tags inside a set to improving the tag reliability and the first technique that provides ECC protection for a parity-protected tag array.

3. Proposed SMARTag Technique

3.1. Observation and Motivation

The special locality in accessing the memory and limited size of the programs working set are well-known facts. Because of that, it can be expected to observe a locality in addresses requesting to access the memory. We interpret this locality as a high probability of the existence of a similarity in higher significant bits (upper part) of the address field. As the tag array contains an upper portion of the address, the mentioned similarity means that the upper parts of the tags in a set contain identical values. If this is the case, we can exploit this inherent redundancy for reliability improvement purpose. Our propose *SMARTag* technique make use of available space to store the ECC of the two tags with similar upper part.

To confirm the hypothesis of existing similar upper parts, we conduct a set of simulations using gem5 cycle-accurate simulator [19] running SPEC CPU2006 benchmark suite [20] as the workload. We consider a 32-Kbyte 4-way set-associative data-cache (D-cache) and a 32-bit address field. Considering a byte-addressable memory and 64-byte cache lines, the width of tag array in the cache is 19-bit. The probability of finding tags with similar X upper bits increases by smaller values of X. To protect the tags using SEC-DED code, X should be 7 based on the configuration used in this paper. If 7 upper bits of two tags are similar, *SMARTag* keep one tag intact and stores a SEC-DED(38,31) code (which is the same as conventional SEC-DED(39,32) code in encoding and decoding logic) on the upper 7 bits of the other one. This SEC-DED code is generated by the unchanged tag and the 12 lower bits of the tag that contains the code. Hereafter, by upper part we are referring to 7-bit upper part of the tag.

Considering the similarity in upper part, a cache set can be in one the following five states:

- *State S40*: upper parts of all tags in the set are the same.
- *State S30*: three out of four upper parts are the same.
- *State S22*: two out of four upper parts are the same and the other two upper parts are also the same.
- *State S20*: two out of four upper parts are the same and the other two upper parts are different.
- *State S00*: none of the upper parts are the same.

Fig. 1 shows the contribution of each state in cache sets for different workloads. The results show that an average of 82.41% of sets is in state S40. The contribution of state S30 is 5.48% and this value for state S22 is 5.16%. On average, 1.90% of set are in states S20 and the remaining 5.05% are in state S00. For 12 out of 17 workloads, i.e., *Perlbench*, *bzip2*, *gcc*, *mcf*, *namd*, *dealII*, *soplex*, *hmmer*, *libquantum*, *h264ref*, *omnetpp* and *xalancbmk*, more than 99.5% of the sets are in state S40 and the upper part of all tags in a set are the same. Fig. 1 implies on the fact that in about 95% of cases, cache sets have at least two tags with similar upper parts.

For a parity-protected tag, error correction in dirty lines is the main reliability threat, because there is no copy for these lines. However, on erroneous clean line can be corrected by just invalidating the line and recovering the error-free copy from lower memory level. Therefore, we need to find the probability of similarity in upper parts for dirty lines and providing ECCs only for these lines in *SMARTag*.

Fig. 2 shows the probability of finding a similar upper part for dirty tags in cache sets for different workloads. The results show that for 93.28% of dirty tag lines, on average, there is at least one similar upper part.This value for 14

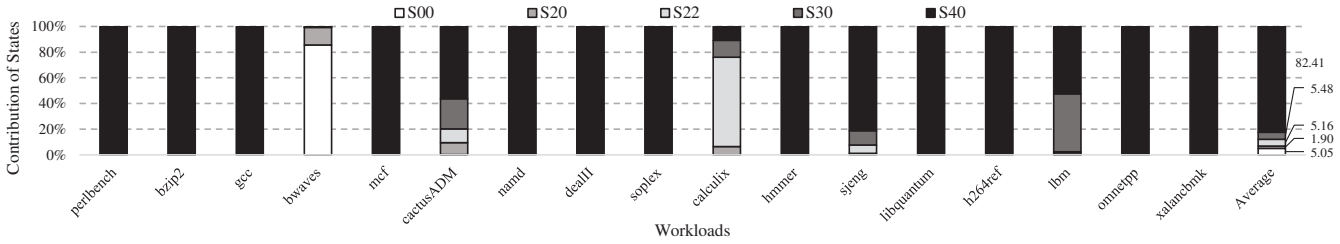

Figure 1. Contribution of each state of sets.

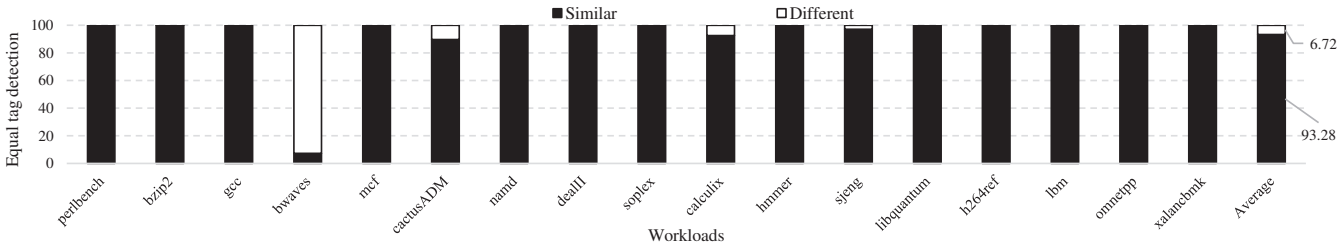

Figure 2. Probability of detecting equal upper tag bits in a set.

out of 17 workloads is even more than 99.5% and is only one workload, i.e., *bwaves*, is less than 90%. Based on this observation, using the inherent upper part similarity for dirty tag lines are completely justifiable.

3.2. SMARTag in Detail

As observed, it is highly probable to have a similar value in the upper parts (upper 7 bits in our configuration) of tags in a set. We propose *Similarity-Managed Robust Tag (SMARTag)* technique that exploits these similarities to protect the tags in a cache set. The key idea in *SMARTag* for two tags with similar upper parts is to keep one upper part and share it between the two and utilize the other upper part for storing a SEC-DED code to protect these tags. Since the content of the tags change only on a cache miss occurrence, in which a data block in the cache is replaced by a new one, *SMARTag* checks the similarity between the tags and updates the ECCs on each cache miss. *SMARTag* adds a 2-bit flag to a tag to determine whether it is only parity-protected or is ECC-protected, and for an ECC-protected tag, to indicate the location of its upper part and its ECC. This flag is named UPL (Upper Part Location). On a tag access, each tag can simply find its upper part in a set by multiplexing the upper parts controlled by the UPL.

As mentioned, there are five states for a cache set based on upper part similarities and a set state may change only on a cache miss. For state S00, no similarity is found and *SMARTag* keeps the tags unchanged. For other four states, i.e., S20, S22, S30, S40, the ECC is embedded inside the upper parts of the tags according to Fig. 3. The operations of SMARTag for each of the four mentioned states are as follows:

- **State S20:** In this state as depicted in Fig. 3 (a), one of the two tags with similar upper part remains unchanged (tag of block 0) and a SEC-DED (38,31) code is generated using this unchanged tag and the

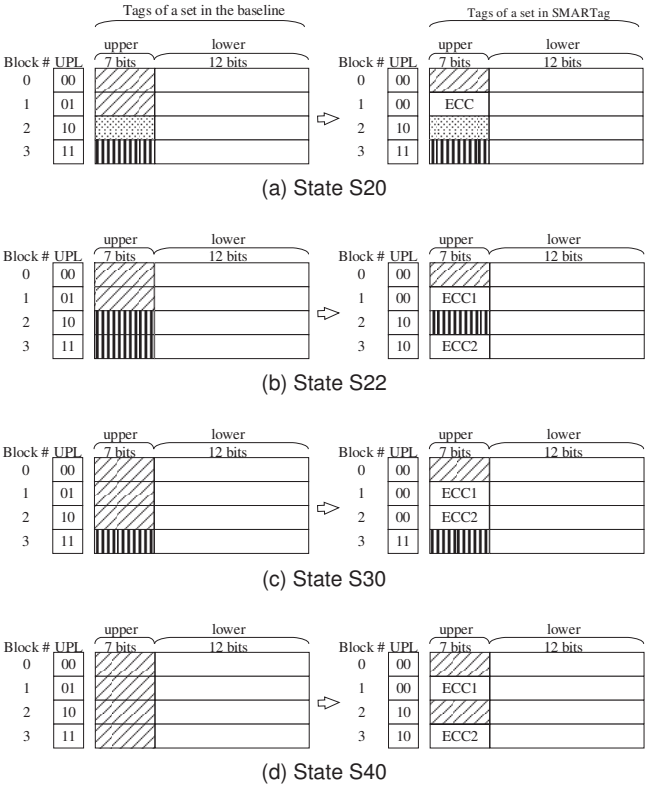

Figure 3. Examples of applying proposed method on sets.

12-bit lower part of the tag 1 (or tag of block 1). The code is stored in the upper part of tag 1. The tags with no similarity, i.e., tag 2 and tag 3, remain unchanged and without ECC protection. The content of UPL is updated accordingly to indicate the correct location of the upper parts. For tags that contain their own upper part, the UPL is equal to their block number in the set. The UPL of a tag containing the ECC is equal to

a block number whose tag is being protected by this ECC. In state S20 shown in Fig. 3 (a), UPL of block 1 is 00 and the UPL of other blocks are the same as their block number. In this state, two out of four tags are ECC-protected.

- **State S22:** In this state as depicted in Fig. 3 (b), two pairs of tags are similar in their upper part, i.e., tag 0 and tag 1 as one pair and tag 2 and tag 3 as the other pair. In this case, ECC1 protects tag 0 and tag 1 and ECC2 protects the two other tags. Accordingly, UPL of tag 1, which the upper part of its tag contains ECC1, is updated to 00 and UPL of tag 3 is set to 10. In this case, all four tags are ECC-protected.
- **State S30:** In this state as depicted in Fig. 3 (c), there are three tags with similar upper part. *SMARTag* makes two pairs of similar tags using these three tags. One pair is tag 0 and tag 1 and the other pair is tag 0 and tag 2. The upper part of tag 0 remains unchanged and is used for tag 0, tag 1, and tag 2 for a cache access. ECC1 is generated using tag 0 and the 12-bit lower part of tag 1. For ECC2, tag 0 and the 12-bit lower part of tag 2 is used. In this case, three out of four tags are ECC-protected.
- **State S40:** In this state as depicted in Fig. 3 (d), the upper parts of all four tags are the same. *SMARTag* makes two pairs of tags, tag 0 and tag 1 as the first pair and tag 2 and tag 3 as the second pair. ECC1 protects tag 0 and tag 1 and ECC3 protects tag 2 and tag 3. In this case, all tags are ECC-protected.

On a cache miss occurrence in a cache set, the state of the set is changed if the upper part of the incoming tag is different from that of the evicting tag. Fig. 4 illustrates the state machine of a cache set in *SMARTag*. When the state of a set remains unchanged, no *SMARTag* operation is required. This situation can be distinguished by comparing the upper parts of the evicting and incoming tags. No state transition occurs when the upper part of the incoming tag is the same as the upper part of the evicting tag. The state is not also changed if the upper part of both evicting and incoming tags

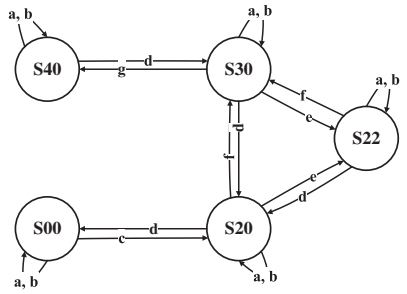

Figure 4. State mechine for a set and the possible transitions in SMARTag.

TABLE 1. System configuration.

Cores	1 GHz, Out-of-Order, 4-issue Superscalar
L1 I-Cache	32-Kbyte, 64-byte cache line, LRU, 4-way associative, 2-cycle read, 2-cycle write, per-core, SRAM
L1 D-Cache	32-Kbyte, 64-byte cache line, LRU, 4-way associative, write-back, 2-cycle read, 2-cycle write, per-core, SRAM
L2 Cache	256-Kbyte, 64-byte cache line, LRU, 8-way associative, write-back, 10-cycle read, 10-cycle write, per-core, SRAM

are different from that of the existing tags. The state of a set changes situations c, d, e, f and g as defined in Fig. 4.

SMARTag technique provides ECC-protection for a parity-protected tag array. The error correction procedure in *SMARTag* is activated whenever a parity check detects an error in a tag. If a tag is found whose UPL is similar to that of the erroneous tag, there is an ECC for these tags and the error is correctable. Otherwise, if the erroneous tag belongs to a clean block the error is again correctable by invalidating the block. The error is uncorrectable only if the erroneous tag belongs to a dirty block and no ECC is found for it, which is a rare event.

4. Simulation Setup and Results

The proposed *SMARTag* technique is evaluated on both single- and quad-core processors. We use gem5 cycle-accurate simulator [19] for these evaluations. A single-core system with an I-cache and a writeback D-cache as well as an L2 cache shared between data and instruction has been considered. The configuration of each core for quad-core system is the same as the single-core system, and the L2 cache is shared between all cores. Table 1 shows the system configurations in detail. SPEC CPU2006 benchmark suite [20] is used as our workloads. For quad-core system, a set of multi-programmed workloads are selected from the benchmark and their combinations are depicted in Table 2. The warm-up phase for evaluations is the first one billion instructions and the results are extracted based on the next 10 billion instructions. We compare the proposed *SMARTag* technique with a parity-protected cache considered as the baseline.

We divide the simulation results into two subsections. First, the error correction capability in the baseline and *SMARTag* is compared. This value in the baseline is the fraction of clean tags in total tags of the cache over the simulation time and for *SMARTag*, the fraction of ECC-protected dirty tags is also included. In the second subsection, the performance overhead of *SMARTag* is evaluated using Clock per Instruction (CPI) metric. *SMARTag* is applied to D-cache for all configurations.

4.1. Error Correction Capability

SMARTag is capable of correcting errors in ECC-protected tags as well as clean tags, while only clean tags are

TABLE 2. SPEC CPU2006 benchmarks combinations as multi-programmed workloads in the evaluation of a quad-core processor.

Workload	Benchmarks			
	Core 0	Core 1	Core 2	Core 3
mix1	perlbench	bzip2	gcc	bwaves
mix2	bzip2	gcc	bwaves	mcf
mix3	gcc	bwaves	mcf	cactusADM
mix4	bwaves	mcf	cactusADM	namd
mix5	mcf	cactusADM	namd	dealII
mix6	cactusADM	namd	dealII	soplex
mix7	namd	dealII	soplex	calculix
mix8	dealII	soplex	calculix	hmmer
mix9	soplex	calculix	hmmer	sjeng
mix10	calculix	hmmer	sjeng	libquantum
mix11	hmmer	sjeng	libquantum	h264ref
mix12	sjeng	libquantum	h264ref	lbm
mix13	libquantum	h264ref	lbm	omnetpp
mix14	h264ref	lbm	omnetpp	astar
mix15	lbm	omnetpp	astar	xalancbmk
mix16	omnetpp	astar	xalancbmk	perlbench
mix17	astar	xalancbmk	perlbench	bzip2
mix18	xalancbmk	perlbench	bzip2	gcc

correctable in the baseline. Fig. 5 shows the error correction capability in the baseline and *SMARTag* for the single-core system. On average, 47.49% of errors are uncorrectable in the baseline. This value is even more than 80% in some workloads, e.g., *dealII* and *libquantum*. Due to this high probability of occurring uncorrectable errors, only the detection capability of a parity-protected tag can be relied on. On the other hand, *SMARTag* provides an average of 98.06% error correction capability. This means that only 1.94% of errors in the tags are detectable but not correctable. The error correction capability in the majority of the workloads, i.e., 14 out of 17 workloads, is even higher than 99.5% and only 0.5% of tags remain parity-protected.

The error correction capability of the baseline and *SMARTag* in the quad-core system is depicted in Fig. 6. On average, the baseline is capable of correcting only 57.50% of errors, while *SMARTag* provides 96.56% correction capability. This high error correction capability in *SMARTag* is achieved by exploiting the already available hardware resources, i.e., upper part of the tags, and without a dedicated ECC memory.

4.2. Performance Overhead

In each cache miss, *SMARTag* checks whether to change the state of the cache set and if so, two operations are required according to the new set state, i.e., generating new ECC(s) and updating the corresponding ECC(s) and UPL(s). These operations can be overlapped with the normal cache operation on a miss occurrence or be conducted at background in cache idle clocks. We pessimistically added one extra clock cycle to the cache delay in replacing a block for a cache miss and compare the performance with the baseline without any overhead.

Fig. 7 illustrates the CPI in *SMARTag* normalized to the baseline for the single-core system. The results show that, on average, *SMARTag* increases the CPI by only 0.03%. This value in the worst case is less than 0.15% in *cactusADM* workload. The performance overhead in the quad-core system is depicted in Fig. 8. *SMARTag* increases the CPI by only 0.06%, on average. In the worst case, this overhead is less than 0.2% in *mix18*. As observed, the performance overhead of *SMARTag* is extremely low while a high level of error correction capability is provided.

5. Conclusion

Tag array in on-chip cache is among the most vulnerable components to soft errors. Protecting tags using ECCs imposes significant overhead and the widely used parity-protected tags are unable to correct the errors. This paper

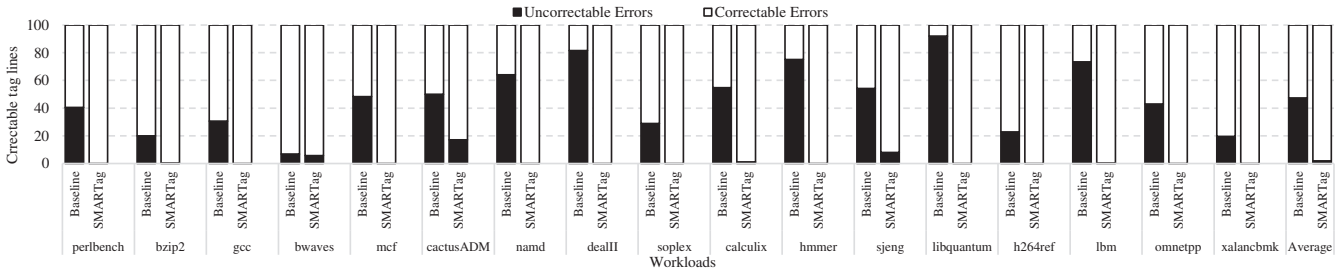

Figure 5. Error correction capability for single-core system.

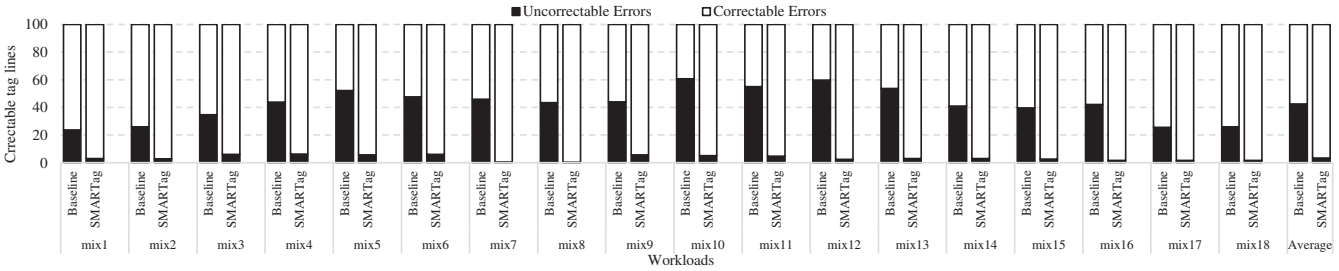

Figure 6. Error correction capability for multicore system.

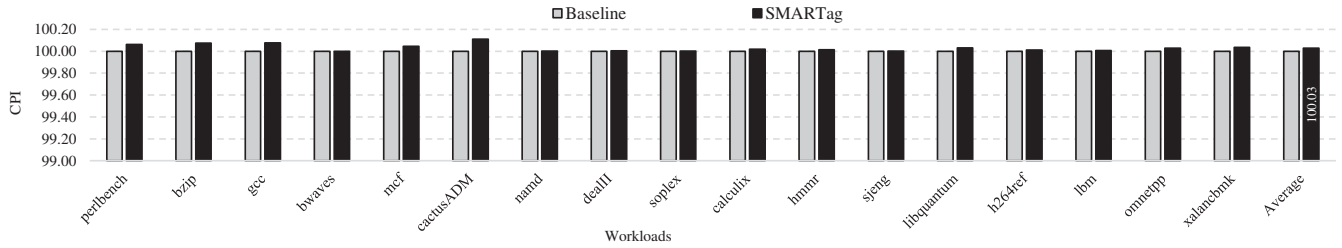

Figure 7. Clock Per Instruction (CPI) of the single-core system. All values are normalized to CPI of the baseline.

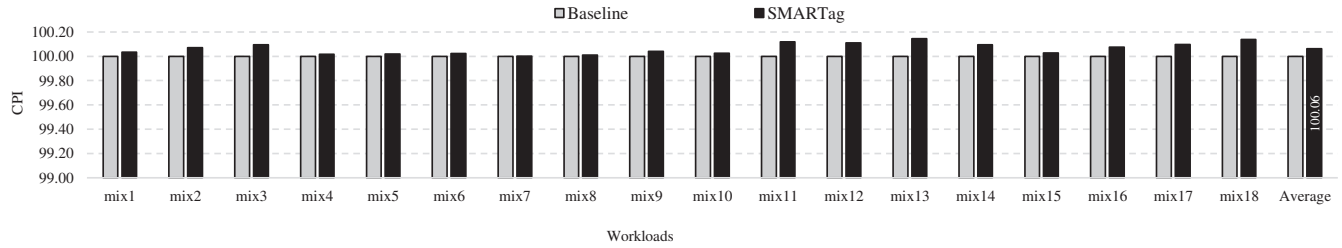

Figure 8. Clock Per Instruction (CPI) of the multicore system. All values are normalized to CPI of the baseline.

proposed *SMARTag* technique to provide ECC-protection in a parity-protected tag array. Based on the similarity in upper parts of the tags in a cache set, *SMARTag* share these spaces between address bits and ECC bits and activates the ECCs only on occurrence of an error detected by the parity checking. The evaluations show that *SMARTag* is capable of correcting 98.06% and 96.56% of errors in the tags of a single- and multi-core system with negligible overhead.

References

[1] R. Jeyapaul and A. Shrivastava, "Enabling Energy Efficient Reliability in Embedded Systems through Smart Cache Cleaning," *ACM Trans. Des. Autom. Electron Syst.*, vol. 18, no. 4, pp. 53:1–53:25, 2013.

[2] Z. Ming, X. L. Yi, L. Chang, and Z. J. Wei, "Reliability of Memories Protected by Multibit Error Correction Codes against MBUs," *IEEE Trans. Nucl. Sci.*, vol. 58, no. 1, pp. 289–295, 2011.

[3] H. Farbeh, N. S. Mirzadeh, N. F. Ghalaty, S. G. Miremadi, M. Fazeli, and H. Asadi, "A Cache-assisted Scratchpad Memory for Multiple-bit-error Correction," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 24, no. 11, pp. 3296–3309, 2016.

[4] W. Zhang, S. Gurumurthi, M. Kandemir, and A. Sivasubramaniam, "ICR: In-Cache Replication for Enhancing Data Cache Reliability," in *Proc. Int. Conf. Dependable Syst. Net.*, 2003, pp. 291–300.

[5] H. Sayadi, H. Farbeh, A. M. H. Monazzah, and S. G. Miremadi, "A Data Recomputation Approach for Reliability Improvement of Scratchpad Memory in Embedded Systems," in *Proc. Int. Symp. Defect and Fault Tolerance VLSI Nanotech. Syst.*, 2014, pp. 228–233.

[6] M. Manoochehri, M. Annavaram, and M. Dubois, "Extremely Low Cost Error Protection with Correctable Parity Protected Cache," *IEEE Trans. Comput.*, vol. 63, no. 10, pp. 2431–2444, 2014.

[7] M. Maghsoudloo and H. R. Zarandi, "Design Space Exploration of Non-Uniform Cache Access for Soft-Error Vulnerability Mitigation," *Elsevier Microelectron. Rel.*, vol. 55, no. 11, pp. 2439–2452, 2015.

[8] A. Timor, A. Mendelson, Y. Birk, and N. Suri, "Using Underutilized CPU Resources to Enhance its Reliability," *IEEE Trans. Dependable Secure Comput.*, vol. 7, no. 1, pp. 94–109, 2010.

[9] C. A. Argyrides, P. Reviriego, D. K. Pradhan, and J. A. Maestro, "Matrix-Based Codes for Adjacent Error Correction," *IEEE Trans. Nucl. Sci.*, vol. 57, no. 4, pp. 2106–2111, 2010.

[10] H. Farbeh and S. G. Miremadi, "PSP-cache: A Low-cost Fault-tolerant Cache Memory Architecture," in *Proc. Conf. Des., Autom. & Test in Eur.*, 2014, pp. 1–4.

[11] L. Delshadtehrani, H. Farbeh, and S. G. Miremadi, "In-Scratchpad Memory Replication: Protecting Scratchpad Memories in Multicore Embedded Systems Against Soft Errors," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, no. 4, pp. 61:1–61:28, 2015.

[12] J. Hong, J. Kim, and S. Kim, "Exploiting Same Tag Bits to Improve the Reliability of the Cache Memories," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 23, no. 2, pp. 254–265, 2015.

[13] S. Wang, J. Hu, and S. G. Ziavras, "Replicating Tag Entries for Reliability Enhancement in Cache Tag Arrays," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 20, no. 4, pp. 643–654, 2012.

[14] P. Reviriego, S. Pontarelli, M. Ottavi, and J. A. Maestro, "FastTag: A Technique to Protect Cache Tags against Soft Errors," *IEEE Trans. Device Mater. Rel.*, vol. 14, no. 3, pp. 935–937, 2014.

[15] H. Farbeh, f. Mozafari, M. Zabihi, and S. G. Miremadi, "RAW-Tag: Replicating in Altered Cache Ways for Correcting Multiple-Bit Errors in Tag Array," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–14, 2017.

[16] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures," in *Proc. Int. Symp. on Comput. Archit.*, 2005, pp. 532–543.

[17] H. Asadi, V. Sridharan, M. B. Tahoori, and D. Kaeli, "Vulnerability Analysis of L2 Cache Elements to Single Event Upsets," in *Proc. Conf. Des., Autom. & Test in Eur.*, 2006, pp. 1–6.

[18] S. Wang, J. Hu, and S. G. Ziavras, "On the Characterization and Optimization of On-Chip Cache Reliability against Soft Errors," *IEEE Trans. Comput.*, vol. 58, no. 9, pp. 1171–1184, 2009.

[19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[20] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.