Guidelines to Design Parity Protected Write-back L1 Data Cache

Yohan Ko*, Reiley Jeyapaul**, Youngbin Kim*,
Kyoungwoo Lee*, and Aviral Shrivastava**
*Department of Computer Science, Yonsei University, Seoul, Korea
**Department of Computer Science and Engineering, Arizona State University, AZ, USA
*{yohan.ko, yb.kim, kyoungwoo.lee}@yonsei.ac.kr
**{reiley, aviral.shrivastava}@asu.edu

ABSTRACT

Several decades of technology scaling has brought the challenge of soft errors to modern computing systems, and caches are most susceptible to soft errors. While it is straightforward to protect L2 and other lower level caches using error correcting coding (ECC), protecting the L1 data caches poses a challenge. Parity-based protection of L1 data cache is a more power-efficient alternative, however, some questions still linger – How effective is parity protection for caches? How can we design a parity-based L1 data cache so as to maximize the protection achieved? The goal of this paper is to perform a quantitative evaluation of the protection afforded by various parity-protected cache design alternatives, and formulate guidelines for the design of power-efficient and reliable L1 data caches. Towards this goal, this paper develops an algorithm to accurately model the vulnerability of data in caches, in the presence of various configurations of parity protection, and validate it against extensive fault injection campaigns. We find that, (i) checking parity at reads only (and not at writes) provides 11% more protection with 30% lesser power overheads as compared to that at both reads and writes; and (ii) when implementing parity at the word-level granularity for 53% improved protection as compared to block-level parity implementation, the dirty-bits in the cache should also be implemented at the same granularity, otherwise, there is no improvement in protection. We find several popular commercial processors – even the ones specifically designed for reliability – not following these design guidelines, and resulting in sub-optimial designs.

1. INTRODUCTION

Soft errors are increasingly becoming a concern in terrestrial and embedded computing systems. Soft errors are transient faults that occur due to electrical noise, external electronic interferences, cross-talk, etc, but majority of soft errors in modern embedded systems happen due to strikes of charge-carrying particles on the processor [5]. With technology scaling, even low-energy neutron particles (10meV - 1eV) will cause soft errors [29]. This coupled with the fact that there are exponentially more low-energy neutrons than those with higher energies [15], soft error rate in electronic systems is going to increase from about once-per-year now, to once-per-day in the near future [11, 17].

In a processor, caches are most susceptible to soft errors. This is not only because caches occupy majority of 60% of chip real-estate but also because they have high transistor density and operate at low voltage swings [24]. Mitra et al. [22] note that soft errors in caches contribute to around 50% of those in processors, and Shazli et al. [27] have shown that 92% of system reboots are triggered by soft errors in the L1 and L2 caches.

Several schemes [6, 18, 26] have been developed to protect the data in caches from soft errors. While the lower level caches (e.g., L2 and L3) are routinely protected by Error Correction Codes (ECC) [19,25], protecting the L1 data cache by SECDED incurs an area overhead of over 20%, and per-access power of over 60%; even if the cycle-time penalty (around 95%) can be avoided. As a result, architects are in search of a power-efficient technique to protect L1 data cache. Although parity protection is quite efficient ($< 1\%$ overhead in power and area), the challenge is that parity-bits can only detect and recover data in clean cache blocks (not written) from single-bit errors. Since parity protection is not complete (i.e., cannot protect the entire cache including dirty blocks), an alternate choice is to apply a write-through cache policy on parity protected caches [16]. This keeps all the cache blocks clean, and provides complete protection, but this results in a huge increase in the number of write-backs to the L2 cache, and consequently will not scale to manycore architectures, where several cores with their own private L1 caches will connect to a single L2 cache. Therefore the question we explore in this paper is: *how effective is parity protection in a write-back L1 data cache, and how do we achieve power-efficient parity-based protection*? The goal is to first quantitatively evaluate the protection achieved by the different implementations of a parity protected cache. Secondly, with the help of analysis results, we derive general guidelines on how to design a parity protected write-back L1 data-cache, that can deliver power-efficient reliability.

We use the metric of **Cache Vulnerability Factor**, or CVF (equal to AVF [8] of cache) to estimate the (un)reliability of a cache configuration. A bit location in the cache is *vulnerable* (un-reliable) at some time during the execution of a program on a processor micraorchitecture, if a fault in the

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA

http://dx.doi.org/10.1145/2744769.2744846

bit at the time will cause the execution or the result to be incorrect. The number of bit-locations in the cache at any time instant is the *instantaneous vulnerability* of the cache, and the total number of vulnerable bits of the cache during the execution of a program is the *cache-vulnerability* of execution. CVF is the fraction of cache bit-cycles that are vulnerable. Clearly larger value of CVF implies a higher failure rate, and that the data in the cache is less protected.

One challenge in performing quantitative evaluation of reliability in different cache configurations is – how accurately can we estimate CVF? Most existing works calculate CVF at a block-level of granularity [4] (which implies that a read/write of any bit in the cache is treated as a read/write access on all the bits in the cache). Researchers have noticed that estimating cache-vulnerability at the block level of granularity is inaccurate, and have proposed techniques to estimate CVF at the word level granularity [8, 10, 33] (of a unprotected cache). On the other hand, there does not exist a method to accurately and quantitatively estimate the CVF of a parity protected cache. In this paper, we build upon previous vulnerability estimation algorithms and develop "gemV-cache" for the accurate and fine-grained estimation of CVF in the presence of parity protection. Modeling vulnerability at a finer granularity (word/byte[1]) is complex, since the vulnerability of a word is now dependent not only on its own access pattern and state (dirty/clean), but also on the access pattern and state (dirty/clean) of the neighboring words in the cache block. Modeling this accurately requires extensive book-keeping and analysis of the cache access time and status bit values of each word in the cache block. Our vulnerability models allow us to estimate vulnerability accurately for different status-bit configurations, e.g., dirty-bit at byte level, but parity at block-level. We validated our vulnerability calculations through extensive fault injection campaigns. We estimate vulnerability of each configuration with more than 99% accuracy.

Our quantitative analysis of various design alternatives in a parity protected L1 data cache yield some not-so-intuitive results (Section 5). These results can be used as simple thumb rules to achieve power-efficient reliability. Two of our key observations are:

(1) Check parity only at reads: Intuitively it seems that checking parity at both reads and writes will provide more "complete" coverage, since we can detect errors before a read, as well as before a write. We have found that several currently popular embedded processors, e.g., ARM1156T2S [1], ARM Cortex A8 [3], and AM3359 [30] implement parity-checking at both reads and writes. Our experiments reveal that this configuration only achieves $< 5\%$ protection to the cache, in addition to adding $> 100\%$ power overhead, over an unprotected cache. *By modifying the parity-checking protocol to only check on cache reads,* $> 15\%$ *cache protection can be achieved, with* 30% *lesser power overheads.*

(2) Parity-bit together with dirty-bit at the word-level achieves higher protection: Parity protection at block level can reduce CVF of L1 data cache by about 15% as compared to no protection. To achieve higher protection designers have implemented parity at finer granularities – e.g., the ARM Cortex R4 [2] - embedded processor designed for high-reliability applications – implements parity-bits at the byte-level. However, in this, the dirty-bit is at the block-level granularity. Our results show that just implementing parity-bit at the byte granularity, but having dirty-bits at the block-level does not provide any better protection. *By modifying the cache status-bits, to hold a dirty-bit and a parity-bit for each byte in the block,* $> 60\%$ *data cache protection can be achieved with acceptable power overheads.*

[1]In our gemV-cache tool, we have designed and implemented the byte-level vulnerability estimation. All the experimental results presented in this paper however contain only word-level accesses as the smallest granularity of cache access.

2. BACKGROUND AND RELATED WORK

Over the years, researchers have used several methods to estimate the soft error reliability of the processor microarchitecture through targeted fault injection campaigns – on the hardware bits [20], assembly code [9], or software variables [32] in a cycle-accurate simulation infrastructure. The failure rate of the system estimated through these methods involve many thousands of runs, and the statistical accuracy of the estimates obtained is dependent on the number of experiment runs, and distribution of the single-bit fault injection campaigns [28].

To quantitatively analyze the soft error exposure of the cache design, in a efficient single simulation run, we develop **CVF (Cache Vulnerability Factor)** to estimate the fraction of time that cache bits are exposed (*vulnerable*) to soft errors. A "datum" in a write-back cache is vulnerable, if it will be read by the processor, or will be written back (e.g., eviction of a dirty cache line) into the memory [8, 23]. If it is overwritten or simply discarded (e.g., eviction of a non-dirty cache line), then it is not vulnerable. CVF is the probability that a soft error in the cache will lead to an architecturally visible error, that will lead to system-level failures; which in turn is the failure rate estimate for the cache microarchitecture block, obtained in a single simulation run. Another distinguishing feature of CVF based quantitative reliability estimation is the fact that the failure rate estimated here is valid for even spatial multi-bit faults on a word/byte in the cache block.

Cache Vulnerability Factor (CVF) is dependent on both the application (the cache data access pattern), and the microarchitecture (exactly how the cache works). Zhang et al. [34] and Asadi et al. [4] estimate the CVF at the *block-level granularity*. In this, every access to a word in the cache block is considered to be an access to the entire block. Though this method is simple and easier to model, it is inherently inaccurate since cache access patterns are based on words not blocks. To enhance the accuracy of previously proposed block-level estimation techniques, finer-level granularity modelings such as *word-level* or *byte-level* of granularity, have been presented [8, 10, 33]. However, all of these models only model the CVF of an unprotected cache. None of them model the vulnerability of parity-protected cache.

3. GEMV-CACHE: IMPLEMENTATION AND VALIDATION

We implement our accurate fine-grained vulnerability estimation model (for a cache with parity protection) in a cycle-accurate simulation environment (gem5 [7]), and develop – *gemV-cache*.

3.1 CVF estimation with parity protection

Vulnerability of a cache block is the sum of vulnerable periods (in cycles) of all the cache bytes/words from *incoming* (data loaded into the cache from memory) through *eviction*

(dirty data written-back into memory from cache). Algorithm 1 presents our implementation to monitor cache accesses at the word-level granularity and estimate the cache vulnerability. In the presence of parity protection at word-level granularity, the mechanism to accurately model vulnerability at the word-level granularity involves anlaysis of the cache status parameters and access patterns of neighboring words in the block. For instance, a word that is written in a previously dirty-block, is deemed *vulnerable*(`word.vulP`) *iff* it is read or evicted from the cache; otherwise it is considered to be uncertain (`word.uc`) and then discarded when overwritten. Vulnerability of a cache computed in gemV-cache is thus the sum of vulnerabilities of all the cache blocks in the processor, during program execution, computed in – byte × cycles.

Algorithm 1 Word-level CVF Estimation with Parity

```
1: procedure estWLCVF_Protected((word_access, word_id, access_time))
2:     wordVulTime ← (time - word.accTime);
3:
4:     switch ACCESS_TYPE do
5:     case INCOMING:
6:     do nothing;
7:     case WRITE:
8:     for all word ∈ {block} do
9:         if word ∈ <accessed_words> then
10:            word.uc ← 0; /* Reset when overwritten */
11:        else
12:            word.uc += word.size × wordVulTime;
13:    block.status = DIRTY;
14:    case READ:
15:    if block.status == DIRTY then
16:        for all word ∈ {block} do
17:            if word ∈ <accessed_words> then
18:                /* Uncertainty removed */
19:                word.vulP += word.uc;
20:                word.uc ← 0;
21:            word.vulP += word.size × wordVulTime;
22:    case EVICTION:
23:    if block.status == DIRTY then
24:        for all word ∈ {block} do
25:            /* Uncertainty removed */
26:            word.vulP += word.uc;
27:            word.vulP += word.size × wordVulTime;
28:    end switch
29:    totalAccTime += wordVulTime;
30:    word.accTime ← time;
```

3.2 Validation through fault injection

To validate our vulnerability models and the implementation of the vulnerability estimations in *gemV-cache*, through fault injection experiments on a cycle-accurate simulation infrastructure. Exhaustive fault injection experiments are infeasible. For example, to exhaustively validate the failure rate of a 256 byte direct-mapped cache with 128bit cache-block, and a benchmark running for 1 million cycles, we will have to perform 128 × 1 million simulation runs. Clearly, since such exhaustive fault injection campaigns for the entire cache is not feasible, we perform exhaustive validation on some randomly selected cache blocks on a few benchmarks from `Livermore Loops` [21] and `matmul`. Assuming a single-bit fault model, we have run over 77 million simulations till now, and compute Failure RateEquation (1):

$$FailureRate = \frac{Num.\ of\ Simulations\ that\ failed}{Total\ Num.\ of\ simulations} \quad (1)$$

For validation, the failure rate should match CVF as defined in Equation (2). Table 1 compares the failure rate (from fault injection) and CVF computed from *gemV-cache* for the respective programs. We can see that the *failure rate* and CVF match perfectly; thus validating our vulnerability models and implementation.

Table 1: The CVF estimated from gemV-cache, is compared with Failure Rate (FR) obtained from our exhaustive fault injection experiments. For each benchmark, we present the CVF and FR numbers in % for one of the several cache-blocks we consider, for our exhaustive FI experiments.

	No Protection		PBDB		PWDB	
Benchmark	FR	CVF	FR	CVF	FR	CVF
Matmul	42.0	42.0	0.00	0.00	0.00	0.00
LL 5	17.4	17.4	16.9	16.9	16.9	16.9
LL 8	46.3	46.3	41.1	41.1	44.4	44.4
LL 12	0.07	0.07	0.00	0.00	0.00	0.00
LL 18	96.2	96.2	96.2	96.2	96.2	96.2

4. EXPERIMENTAL SETUP

We perform extensive experiments over benchmarks from MiBench [12] and SPEC CPU2006 [13] suites. We use the ARM v7a processor architecture with default L1 cache configuration (direct-mapped 4 KB size, with 64 byte block size) for our experiments. We compile our benchmarks using ARM-GCC 4.6.2 cross compiler. The CVF that our experiments report (which is the measure of the probability that a single-bit error in cache will result in a system failure), is calculated as:

$$CVF = \frac{vulnerability\ (byte \times cycles)}{cache\ size\ (byte) \times totalAccTime\ (cycle)} \quad (2)$$

5. GUIDELINES FOR A PARITY-PROTECTED WRITE-BACK L1 DATA CACHE

There are several design considerations in the implementation of parity protection in the L1 data cache – (1) when should we check for parity: at every read, at every write, or at both read and write ?, and (2) what should be the granularity at which one should implement parity-bits ? Naively, it seems like checking parity at both reads and writes must offer more protection, and that implementing parity at a finer granularity should obviously provide more protection. However, these design decisions are not so intuitive and obvious. In this section, we analyze these design decisions through accurate CVF estimation, and quantitative analysis of the reliability across cache configurations.

5.1 Check parity only on reads for power-efficacy

When using parity protection, should we check for parity at every read, or at every write, or at both, every read and write. We perform quantitative evaluation of all of these approaches. Fig. 1 plots the vulnerability of data cache for various benchmarks when parity is checked at every read (P-R), at every write (P-W), and at both read and write (P-RW) normalized to the vulnerability when there is no

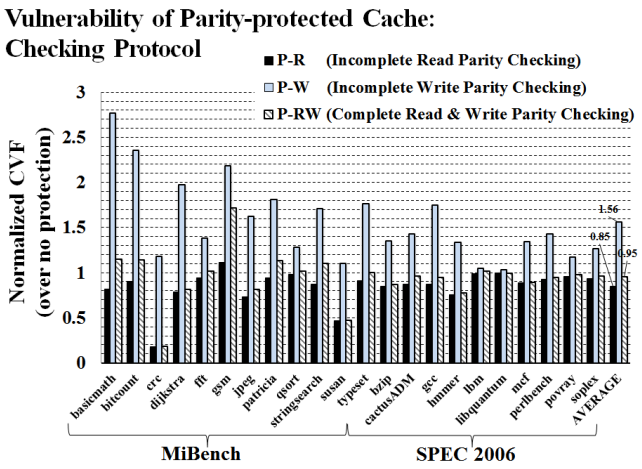

Figure 1: P-R (checking parity only on reads) offers more protection than P-RW (checking parity on reads and writes). This happens because checking parity at write operation makes the period from last access to write *vulnerable*.

protection. This plot shows that P-R, i.e., checking parity at reads only is most effective – it reduces vulnerability by almost 15%, while P-RW, i.e., checking parity at both reads and writes can reduce the vulnerability by only 5%.

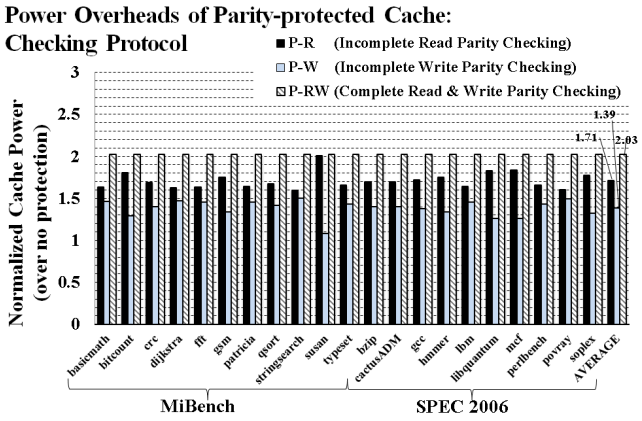

Figure 2: In the design of a parity-protected cache, checking parity-bit on only reads, the power overheads incurred are 30% lower than that when parity is checked on both reads and writes.

To estimate power consumption, we compute the read/write power of this parity-checking protocol implementation in the cache by manipulating CACTI 5.0 [31] for 45 nm technology node. To estimate the power of the parity generation/checking hardware logic, we designed the unit for this cache, synthesized it in 45 nm technology; and obtained power numbers using PowerMill [14]. Fig. 2 plots the power consumption of the cache across three configurations (P-R, P-W, and P-RW) normalized over that of the cache with no protection. We see here that when checking the parity-value on both reads and writes (P-RW), we incur a power overhead of around 103% for only 5% cache protection. On the other hand, an implementation of parity-checking on only reads (P-R), incurs a power overhead of only 71% for around 15% improved cache protection; achieving power-efficient cache protection. It should be noted here that parity is implemented at the block-level granularity.

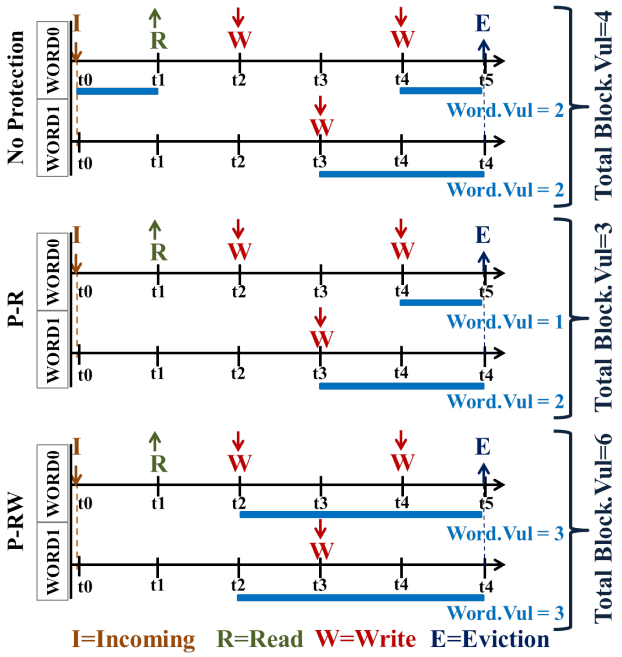

Figure 3: Three cases: (i) No protection. Vulnerability = 4, (ii) P-R reduces CVF since WORD 0 is no longer vulnerable from $t0$ to $t1$. Vulnerability = 3, and (iii) P-RW increases Vulnerability. If an error is detected at at the write at $t3$, then we cannot find out which word has error, therefore both the words are vulnerable from the first write. Thus, Vulnerability = 6.

It is interesting to note that the P-R configuration provides more protection than P-RW. Intuitively, P-RW should achieve more protection than P-R, since we are checking at both reads and writes. The fact is, that checking parity at writes can actually increase the vulnerability. As described in Fig. 3 the parity-check (at the block-level) on a write-access (WORD 0 at time $t2$) also checks for the data integrity on WORD 1 at time $t2$, thereby rendering WORD 1 to be vulnerable from time $t2 \sim t3$; which is not actually vulnerable. The reason why checking parity at reads can reduce the vulnerability in P-R, over that of P-RW is because; if the cache block is clean, and we find an error by checking parity, then the data can be recovered from lower levels of memory – and therefore the data is *not-vulnerable* from last access to this read access. On the other hand, if the cache block is dirty, and we detect an error by checking parity, then the data cannot be recovered – it remains *vulnerable* – and the parity-checking performed on writes (and also power consumed) is in-vain.

5.2 Fine-grained parity implementation: Dirty-bit and parity-bit at word-level

It is intuitive that fine-grained parity implementation (at the word/byte level) can achieve improved cache protection. ARM Cortex R4 [2] processor designed for safety critical applications, allows for a configuration with a parity-bit per

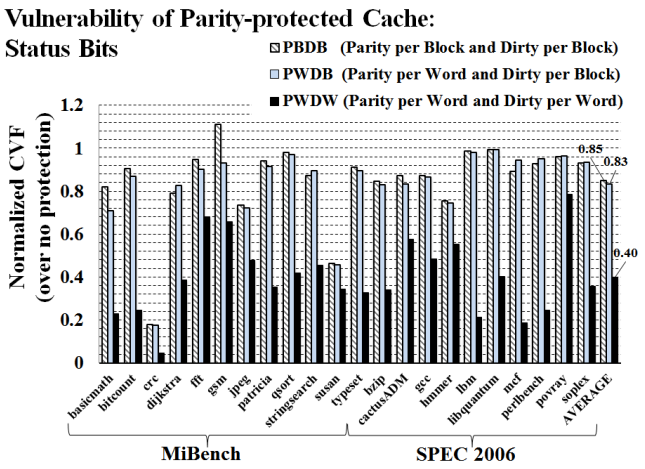

Figure 4: Fine-grain parity with block-level dirty-bit reduces the vulnerability by only 2% as compared to block-level parity and dirty-bits. Fine-grain dirty-bit along with parity-bit per word is the best in terms of vulnerability (53% reduction).

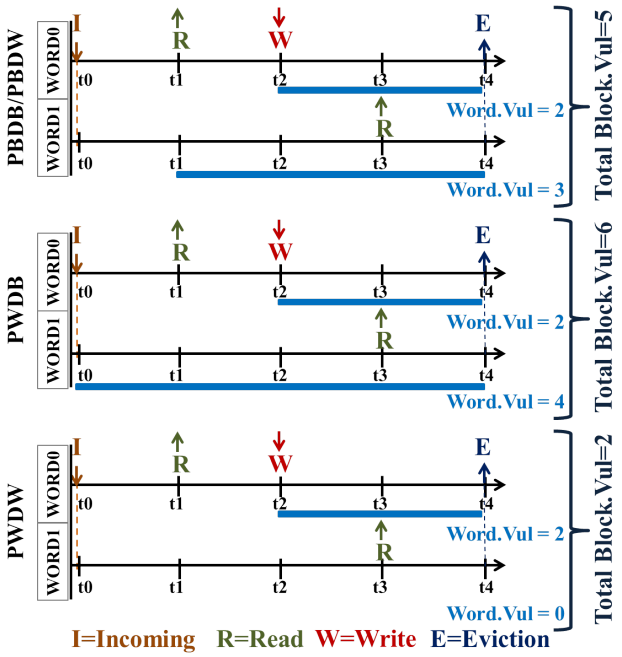

Figure 5: i) PBDB & PBDW. Vulnerability = 5, ii) PWDB may even worse the Vulnerability than PBDB & PBDW. Vulnerability = 6, and iii) PWDW with finer-granularity of status-bits can improve the Vuparity bitlnerability. Thus, Vulnerability = 2.

byte, for improved cache protection. Fig. 4 plots our quantitative analysis results of cache reliability, across cache protection granularity configurations – when the *parity-bit* and *dirty-bit* are implemented on word-level and/or block-level. The most important observation from the graph is that PWDW, i.e., parity-bit at word-level, and dirty-bit at word-level is quite effective in protecting data in the cache; it can improve reliability by an average 60% and up to 96% (`crc`), as compared to a cache with no protection. Interestingly, the PWDB configuration (as in ARM Cortex R4 [2]) achieves only around 17% cache protection. The fine-grained parity-bits enabled here, does not reduce the vulnerability much without a change in the dirty-bit configuration. Through experimental analysis, we observe that the power-overheads incurred for a complete fine-grained PWDW cache, is insignificant when compared to that of the cache in the PWDB configuration. Owing to space limitations, we do not present our experimental results in this paper.

For improved protection of the L1 data cache through parity protection, the design parameter to configure is the status (dirty state of the cache block) and the parity-bit implementation. Whether the duration between two accesses to a datum is vulnerable or not depends on the status-bit configurations. To understand the experimental results observed above, we discuss here the impact of the *status-bit* and *parity-bit* configurations, on the reliability of the data cache with an example:

PBDB: (Parity-bit per Block and Dirty-bit per Block) – Since the entire cache block is configured with one *parity-bit*, a read access on any word (in non-dirty blocks) can trigger recovery of the entire cache block; the single parity-bit cannot identify the exact word that contains a single-bit error. In addition, since the entire cache block is configured with one *dirty-bit*, a write access on any one word makes the entire cache block dirty; thereby rendering every word of the block unrecoverable (vulnerable), on read accesses thereafter. In Fig. 5, the top diagram defines the vulnerability of the cache block under a scenario in this configuration.

PBDW: (Parity-bit per Block and Dirty-bit per Word) – In this configuration, though each word in the block is configured with its respective dirty-bit, its vulnerability does not differ from that of the PBDB configuration as shown in Fig. 5. If any word of a cache block becomes dirty, the entire block will be considered dirty; since the single parity-bit cannot know which word in a block contains a error.

PWDB: (Parity-bit per Word and Dirty-bit per Block) – *Parity-bit* per word can identify single-bit errors at read operations and also trigger the recovery of this erroneous word in case of the clean state. Also it does not affect the vulnerability of the neighboring words at reads. For instance, Fig. 5 shows that the vulnerabilities of WORD 0 and WORD 1 are defined by the read/write accesses on the respective word only, not the other for this configuration. On the other hand, a write on any word makes the entire block dirtdue to *dirty-bit* per block. Therefore, the erroneous word in the block cannot be identified, which makes the entire block vulnerable and affects the recovery mechanism.

PWDW: (Parity-bit per Word and Dirty-bit per Word) – This fine-grained status-bit configuration where every word in the block is associated with its respective dirty-bit and parity-bit can achieve the high reliability in terms of the vulnerability. *Parity-bit* per word can locate the erroneous word at read accesses in case of the clean state. In addition, *dirty-bit* per word can identify the updated word accurately; thus assisting the recovery mechanism. Fig. 5 shows that WORD 1 is non-vulnerable during the entire stay from incoming to eviction in this configuration since the WORD 1 is never updated. Note that only the words that were updated become vulnerable.

In summary, parity-checking only on read accesses improves cache reliability (avg. 11% and max. 35%) to the cache with 30% less power overheads as compared to checking on both reads and writes. Fine-grained status-bit implementations of parity protection only when implemented for both parity and status bits, will achieve improved reliability (avg. 53% and max. 79%).

6. CONCLUSION

Soft errors are becoming a real threat to modern embedded systems. Parity based error detection is popularly used to protect the L1 caches; owing to its power-efficiency and simple design. This paper quantitatively explores the the protection achieved by such techniques at a fine-grained word-level, and also reveals counter-intuitive results that will be instrumental in the development of design guidelines for the power-efficient parity based protection of the L1 data cache: (i) checking the parity-bit on ONLY read accesses provides on an average 11% more protection (for 30% improved power-efficiency) compared to the case when checking on both reads and writes. In fact, checking the parity-bit on write accesses increases the vulnerability of some cache blocks, while also adding to power overheads. (ii) For improved parity-protection, implementing parity-bit protection with status-bits (parity and dirty-bits) at the word level granularity can achieve around 53% improved protection, compared to a traditional block-level implementation.

7. ACKNOWLEDGMENTS

This research was supported by Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Science, ICT & Future Planning (MSIP) (2012-015421), by MSIP under the Research Project on High Performance and Scalable Manycore Operating System (#14-824-09-011), and part by funding from National Science Foundation grants CCF 1055094 (CAREER).

8. REFERENCES

[1] ARM. ARM1156T2-S technical manual, 2007.
[2] ARM. ARM Cortex-R4 processor, 2010.
[3] ARM. Cortex-A8 processor, 2014.
[4] G. H. Asadi, V. S. Mehdi, B. Tahoori, and D. Kaeli. Balancing performance and reliability in the memory hierarchy. In *ISPASS*, 2005.
[5] R. Baumann. Soft errors in advanced computer systems. *Design Test of Computers, IEEE*, 22(3), 2005.
[6] M. P. Baze, S. P. Buchner, and D. McMorrow. A digital CMOS design technique for SEU hardening. *Nuclear Science, IEEE Trans on*, 47(6), 2000.
[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.
[8] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *ISCA*, 2005.
[9] W. Chao, F. Zhongchuan, C. Hongsong, B. Wei, L. Bin, C. Lin, Z. Zexu, W. Yuying, and C. Gang. CFCSS without aliasing for SPARC architecture. In *CIT*, 2010.
[10] X. Fu, T. Li, and J. Fortes. Sim-soda: A unified framework for architectural level software reliability analysis. In *PMBS*, 2006.
[11] T. Granlund, B. Granbom, and N. Olsson. Soft error rate increase for new generations of SRAMs. *Nuclear Science, IEEE Transactions on*, 50(6), 2003.
[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC*, 2001.
[13] J. L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4), 2006.
[14] C. X. Huang, B. Zhang, A.-C. Deng, and B. Swirski. The design and implementation of PowerMill. In *ISLPED*, 1995.
[15] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba. Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule. *Electron Devices, IEEE Trans on*, 57(7), 2010.
[16] Intel. Intel pentium 4 processor on 90 nm process datasheet. In *Intel Corporation*, April 2004.
[17] ITRS. The international technology roadmap for semiconductors, 2007.
[18] R. Jeyapaul and A. Shrivastava. Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors. In *CASES*, 2011.
[19] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Soft error and energy consumption interactions: a data cache perspective. In *ISLPED*, 2004.
[20] H. Madeira and J. Silva. On-line signature learning and checking: experimental evaluation. In *CompEuro*, 1991.
[21] F. H. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical report, Lawrence National Lab., 1986.
[22] S. Mitra, M. Zhang, N. Seifert, T. M. Mak, and K. S. Kim. Built-in soft error resilience for robust system design. In *ICICDT*, 2007.
[23] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Micro*, 2003.
[24] R. Naseer, Y. Boulghassoul, J. Draper, S. DasGupta, and A. Witulski. Critical charge characterization for soft error rate modeling in 90nm SRAM. In *ISCAS*, 2007.
[25] N. Sadler and D. Sorin. Choosing an error protection scheme for a microprocessor's L1 data cache. In *ICCD*, 2006.
[26] A. Seyedi, G. Yalcin, O. S. Unsal, and A. Cristal. Circuit design of a novel adaptable and reliable L1 data cache. In *GLSVLSI*, 2013.
[27] S. Shazli, M. Abdul-Aziz, M. Tahoori, and D. Kaeli. A field analysis of system-level effects of soft errors occurring in microprocessors used in information systems. In *ITC*, 2008.
[28] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu. Quantitative analysis of control flow checking mechanisms for soft errors. In *DAC*, 2014.
[29] C. Slayman. Alpha particle or neutron SER-what will dominate in future IC technology. 2010.
[30] Texas Instruments. AM3359 sitara processor, 2011.
[31] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. *HP Laboratories, April*, 2, 2008.
[32] R. Vemu and J. Abraham. CEDA: Control-flow error detection using assertions. *IEEE Transactions on Computers*, 60(9), 2011.
[33] S. Wang, J. Hu, and S. Ziavras. On the characterization of data cache vulnerability in high-performance embedded microprocessors. In *IC-SAMOS*, 2006.
[34] W. Zhang. Computing cache vulnerability to transient errors and its implication. In *DFTVS*, 2005.