# Protecting Caches from Soft Errors: A Microarchitect's Perspective

YOHAN KO, Yonsei University
REILEY JEYAPAUL, ARM Research
YOUNGBIN KIM and KYOUNGWOO LEE, Yonsei University
AVIRAL SHRIVASTAVA, Arizona State University



Soft error is one of the most important design concerns in modern embedded systems with aggressive technology scaling. Among various microarchitectural components in a processor, cache is the most susceptible component to soft errors. Error detection and correction codes are common protection techniques for cache memory due to their design simplicity. In order to design effective protection techniques for caches, it is important to quantitatively estimate the susceptibility of caches without and even with protections. At the architectural level, vulnerability is the metric to quantify the susceptibility of data in caches. However, existing tools and techniques calculate the vulnerability of data in caches through coarse-grained block-level estimation. Further, they ignore common cache protection techniques such as error detection and correction codes. In this article, we demonstrate that our word-level vulnerability estimation is accurate through intensive fault injection campaigns as compared to block-level one. Further, our extensive experiments over benchmark suites reveal several counter-intuitive and interesting results. Parity checking when performed over just reads provides reliable and power-efficient protection than that when performed over both reads and writes. On the other hand, checking error correcting codes only at reads alone can be vulnerable even for single-bit soft errors, while that at both reads and writes provides the perfect reliability.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; C.4 [**Computer Systems Organization**]: Performance of Systems—*Fault tolerance*

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Cache, soft error, reliability, vulnerability, transient fault, error correction code, parity code, simulation

**ACM Reference Format:**
Yohan Ko, Reiley Jeyapaul, Youngbin Kim, Kyoungwoo Lee, and Aviral Shrivastava. 2017. Protecting caches from soft errors: A microarchitect's perspective. ACM Trans. Embed. Comput. Syst. 16, 4, Article 93 (May 2017), 28 pages.
DOI: http://dx.doi.org/10.1145/3063180

This research was supported in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, and future Planning (NRF-2015R1A2A1A15053435); by Next-Generation Information Computing Development Program through the NRF funded by the Ministry of Science, ICT, and Future Planning (NRF-2015M3C4A7065522); by MSIP under the Research Project on High Performance and Scalable Manycore Operating System (#14-824-09-011); and by funding from National Science Foundation grants CCF 1055094 (CAREER), CCF-0916652, and CNS 1525855.
Authors' addresses: Y. Ko, Y. Kim, and K. Lee, Department of Computer Science, Yonsei University, Korea; emails: {yohan.ko, yb.kim, kyoungwoo.lee}@yonsei.ac.kr; R. Jeyapaul, ARM Research, UK; email: reiley.jeyapaul@arm.com; A. Shrivastava, Department of Computer Science and Engineering, Arizona State University, USA; email: aviral.shrivastava@asu.edu.

DOI: http://dx.doi.org/10.1145/3063180

## 1. INTRODUCTION

Soft errors are increasingly becoming a critical design concern in embedded computing systems [Ferreira et al. 2016]. Soft errors are transient bit flips due to the electrical noise, external interference, cross-talk, radiation, and the like. The majority of soft errors in modern embedded systems occur due to charge-carrying particles on the processor. With technology scaling, even low-energy neutron particles (10meV–1eV) can cause soft errors [Slayman 2010]. Critical charge is generally defined as the minimum amount of charge to change the bit value in a semiconductor device. Critical charge also decreases significantly due to shrinking feature size and decreasing supply voltage [Dixit and Wood 2011]. Thus, the soft error rate is going to increase especially in low-power and tiny embedded systems [Ko et al. 2016].

In a processor, the cache is one of the most susceptible microarchitectural components to soft errors [Mittal and Vetter 2016]. Mitra et al. [2005] note that soft errors in caches (unprotected static random access memory (SRAM)) contribute to around 40% in processors, and Shazli et al. [2008] have shown that 92% of machine checks are triggered by soft errors at the level 1 and 2 caches. This is not only because caches occupy the majority of the chip area, but also because they have high transistor density and operate at low-voltage swings [Naseer et al. 2007]. Since data in caches are frequently accessed by central processing unit (CPU) and written back to lower-level memory in case of write-back caches, some of erroneous bits can be propagated to the lower-level memory or used by CPU. However, not all the soft errors in the cache memory can cause system failures (i.e., vulnerable) during all the execution time mainly due to several masking effects. Thus, there is a necessity to quantify the susceptibility of caches in order to know how many bits and how long cache data can be vulnerable.

In order to accurately calculate susceptibility of microarchitectural components, the particle beam testing [Kudva et al. 2007] and emulated fault injection campaigns [Entrena et al. 2012] have been proposed. However, they are not only expensive but also challenging to set up correctly. Architectural vulnerability factor (AVF) is an alternative metric to estimate the susceptibility of data in microarchitectural components [Mukherjee et al. 2003]. AVF is essentially the average number of bits in a microarchitectural component that are susceptible to soft errors over time. Existing methods to estimate the AVF of cache, which we call cache vulnerability factor (CVF), have two main problems. Firstly, existing vulnerability modeling is based on block-level cache behaviors [Zhang 2005a], although the basic unit of cache access is a word, not a block. Secondly, there is no method to estimate cache vulnerability in the presence of protection techniques. Thus, they cannot provide the accurate vulnerability modeling with and without protection techniques.

Architectural Vulnerability generally used to denote the reliability of a single architecture component, while Vulnerability is used to denote that of the entire processor. In this article, we use the term Vulnerability to denote the architectural vulnerability of the cache since we only analyze the cache reliability. Cache vulnerability estimation at a block-level granularity is quite inaccurate since the basic unit size of data accesses in caches is a word, not a block. For instance, the specific cache word is vulnerable when just a single word of a block is read by CPU. However, block-level vulnerability estimation defines the whole block as vulnerable, not just the specific word. The average inaccuracy of block-level estimation is 37% as compared to our more accurate word-level one. Note that our word-level vulnerability estimation includes byte-level granularity since we analyze word-level cache behaviors for vulnerability estimation. The average inaccuracy is not significant, but the actual wrong decision based on block-level behaviors can be more severe. It is because the difference is aggregated statistics of entire cache blocks during the entire execution time. First off, block-level analysis

can underestimate or overestimate vulnerability as compared to the word-level one, but the inaccuracy only can show the difference between the underestimation and overestimation. Secondly, the error of each block can be much larger than the average error of all the blocks. For example, the error of a specific block is up to 5,700%, while the average error of all the blocks is only 121% for the same benchmark, *basicmath*.

Existing cache vulnerability estimation schemes ignore protection techniques even though several techniques have been presented for reliable cache memory. These techniques span across the design spectrum from the circuit, microarchitecture, software, and even hybrid level. In practice, parity and error correction code (ECC) are the most popular cache protection techniques due to their design simplicity. Parity-based methods allow the error recovery by bringing data from lower-level memory as long as cache data is not updated by the processor (i.e., clean state). ECC-based techniques provide the error recovery regardless of the clean or dirty state. However, it can incur up to an additional 50% hardware area, more than five times power consumption, and about 115% runtime overheads as compared to unprotected cache [Sadler and Sorin 2006]. Parity protection is preferred for higher-level (e.g., level 1) caches while lower-level caches (e.g., level 2 or other lower-level caches) are protected by ECC in common processors. There are several design choices when we implement parity and ECC protection, for example: When should we check for parity-bit and ECC-bits—at read, write, or both read and write? At what granularity should we have parity-bit and ECC-bits? At what granularity should we have dirty-bit?

In order to correctly answer these questions, we definitely need techniques to quantitatively and accurately estimate the susceptibility of cache data to soft errors with or without protection techniques. We have validated the accuracy of our word-level estimation by comprehensive fault injection experiments. The logic to estimate vulnerability at a word-level granularity with the presence of protection techniques is much more involved than the logic to estimate vulnerability at a block-level granularity without considering protections. The main source of complexity comes from the fact that (i) the access time to each word should be logged for word-level estimation while the access time for a block is needed for block-level estimation; and, (ii) vulnerability estimation at a word-level granularity may not be independent of the accesses of the other words in the same block.

The main contributions of this work are accurate word-level modeling and awareness of protection techniques. First off, we have modeled more accurate word-level vulnerability modeling than the previous block-level one since the basic unit of cache accesses is a word, not a block. And, we have also validated our vulnerability modeling against exhaustive fault injection campaigns. Secondly, we have modeled cache vulnerability estimation without and with common protection techniques such as error detection codes (parity) and error correction codes. We explore the design space of parity and ECC protection with various protection configurations based on accurate word-level vulnerability estimation. Our analysis reveals several interesting and counter-intuitive results.

—Checking parity at reads provides better level of protection than checking parity at both read and write. This is surprising, since it is more intuitive to believe that checking parity at both occasions will provide better protection mainly due to more redundancy. The implication is that better protection can be achieved by simpler hardware and less overhead of parity-checking power.
—In order to achieve higher levels of protection, both parity-bit and dirty bit should be implemented at word-level of granularity. It can reduce the vulnerability by 60% as compared to the vulnerability without protections. However, only either parity-bit or dirty-bit at a word-level granularity does not protect caches effectively, i.e., it can

reduce the vulnerability by just 15% on average as compared to unprotected caches in spite of additional hardware overheads.
—Checking block-level ECC-bits only at reads can still be vulnerable because of other words' behaviors in the same block. About 10% of vulnerability comes from unprotected caches' remains with checking at reads, while checking at both reads and writes provides zero vulnerability. If the perfect reliability is required for caches, ECC should be checked at both reads and writes or ECC-bits should be implemented at a word-level granularity.

## 2. BACKGROUND AND RELATED WORK

Soft errors are transient faults in the semiconductor devices caused by external energy such as alpha particles, cosmic ray, thermal neutron, and inductive cross-talk. Caches are one of the most vulnerable microarchitectural components in processors against soft errors. It is not only because caches occupy lots of area in processors, but also because that cache data is frequently accessed by CPU and quickly propagated to lower-level memory. In order to improve the reliability of cache memory without area cost, Li et al. [2004] proposed early write-back policy. Early write-back policy combines the performance efficiency of write-back with the reliability of write-through policy by exploiting the least recently used algorithm or dead-time–based approaches. Manoochehri et al. [2011] proposed the correctable parity protected cache (CPPC) to correct errors, which can be detected by parity. CPPC corrects soft errors including spatial multi-bit errors at the dirty state by multi-dimensional parity without the severe overhead in terms of hardware area and performance. However, they can still be vulnerable for temporal multi-bit upsets and errors in the cache tag array and status-bits such as dirty bits.

Soft errors on variables do not induce system failures due to the software masking effects, e.g., errors on multimedia data in a program can degrade the quality of service, but they do not result in system failures. Partially protected cache (PPC) [Lee et al. 2006] improved the reliability with the comparable performance overheads by enhancing the software masking. PPC only protects failure-critical data such as control variables based on data profiling at the compile time. On the other hand, they do not protect multimedia data since errors on multimedia data cause loss in quality of service instead of system failures. Smart cache cleaning [Jeyapaul and Shrivastava 2011] protects specific cache blocks at specific periods by applying the hardware-software hybrid methodology. At the software level, we can protect data efficiently by software-based or hybrid-based selective protection, but the decision of importance in data is an extremely complex task.

In order to mitigate the reliability analysis overheads of cache memory and to provide the accurate reliability reflecting various masking effects, CVF is proposed based on cache access patterns [Zhang 2005b; Asadi et al. 2005]. Data 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. If it is overwritten or simply discarded (e.g., eviction of a non-dirty cache line), then it is not vulnerable. In a system, the reliability metric – *vulnerability*, is a measure of the probability of soft errors during the time period when data is exposed in the cache which is predominantly dependent on the data access pattern of the program. Vulnerability estimation of a cache block can be implemented at two granularity levels: **(a) block-level**—when every access to a word in the cache-block is considered to be an access to the whole block or every word in the cache-block has the same data access; **(b) word-level**—when every access to a word in the cache-block is considered as an access to each respective word in the block. In a cache-block composed of multiple words, the total vulnerability of the block is an

accumulation of the vulnerabilities of the individual words in the block, which is based on the data access patterns of the words in the cache-block.

However, how can we accurately measure the reliability of caches without protections? How much do these protection techniques afford as compared to the reliability without protections? Thus, there is a necessity to quantify the susceptibility of caches against soft errors without protection or even with protection techniques.

## 3. ACCURATE VULNERABILITY ESTIMATION AT A WORD-LEVEL GRANULARITY

In order to accurately estimate the cache vulnerability, we have developed vulnerability estimation tools with protections for caches, named gemV-cache [Ko et al. 2015] by modifying the gem5 simulator [Binkert et al. 2011]. We have named our cache vulnerability modeling frameworks "gemV-cache" due to two following reasons. First off, "V" of gemV-cache stands for both vulnerability and Roman numeral 5 (5 from gem5). Secondly, "cache" from gemV-cache is named since we have modeled cache vulnerability, not all the microarchitectural components. In modeling of gemV-cache, we consider single-bit soft errors throughout a program execution in caches for simplicity. Vulnerability of a cache block is the sum of vulnerable periods in cycles of all cache words from incoming through eviction. Thus, the vulnerability of a cache is the sum of vulnerabilities of all cache blocks during a program execution and the unit of cache vulnerability is byte $\times$ cycle. We have performed extensive experiments with gemV-cache over benchmarks from MiBench [Guthaus et al. 2001] and SPEC CPU2006 [Henning 2006] suites. We use the ARM v7a processor architecture with default L1 cache configuration as direct-mapped 4KB with 64 byte block size. This is just one set of parameters for our simulation studies. Our gemV-cache is configurable as is gem5 simulator. For instance, we can also configure ISAs and number of cores, which are not directly related with cache configurations. We have compiled our suite of benchmarks using gcc cross-compiler for ARM (ver. 4.6.2), run them on gemV-cache in system emulation mode, and gathered vulnerability statistics in just one simulation.

In a system, the reliability metric, *vulnerability*, is a measure of the probability of soft errors during the time period that data is exposed in the cache, which is predominantly dependent on the data access patterns of the program as shown in Figure 1. First off, data is brought into cache memory (incoming). If data is written by write operations after incoming, it is not vulnerable as shown in Figure 1(a) since it does not affect system behaviors. If cache data is overwritten by write operation after write or read operations, it is not vulnerable since the soft error induced data can be overwritten. Based on this vulnerability definition, write operations always make periods non-vulnerable from the last behavior to the current write operation as shown in Figure 1(b) and (c), since it overwrites the corrupted data (no impact on system behaviors and propagation) if a soft error occurred. If cache data is simply discarded (e.g., eviction of a non-dirty cache line), it is also not vulnerable as shown in Figure 1(d). Since cache data is identical to the data in lower-level memory, it does not update lower-level memory.

On the other hand, cache data is vulnerable if it will be read by a processor since it can affect system behaviors. If cache data is read after incoming, it is vulnerable since reading corrupted data affects system behaviors as shown in Figure 1(e). Read operations always make periods vulnerable from the last behavior to the current read operation as shown in Figure 1(f) and (g), since the corrupted data is read by processor execution, affecting the system behavior, i.e., inducing the high possibility to change the original system behaviors and to result in incorrect outputs or even system crashes. Data in a write-back cache can be also vulnerable if it will be written back into the lower-level memory since it propagates corrupted data to system memory. If cache data is written back at the dirty state, it is defined as vulnerable as shown in Figure 1(h).

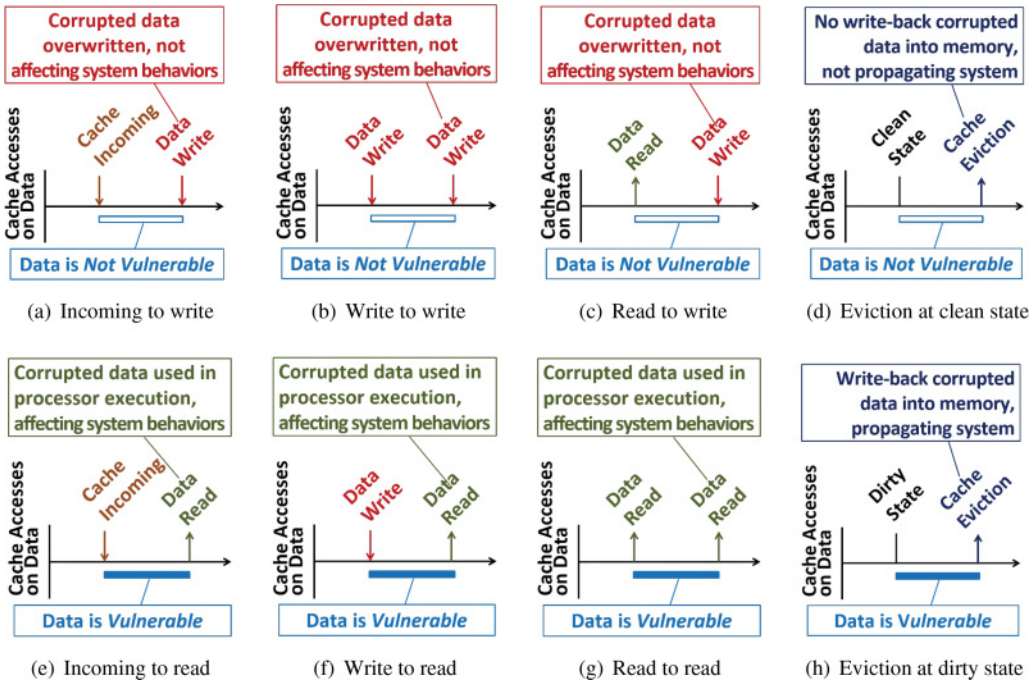



Fig. 1. Example demonstrating the vulnerability of a *data*, over different data accesses.

CVF is the probability that a single-bit error in cache will result in a system fault or failure. CVF is calculated as vulnerable bytes and their periods, vulnerability in byte $\times$ cycles, over the total cache size, and access time as described in Eqation (1). The denominator of the CVF equation is the same for block-level and word-level vulnerability estimation. The denominator of CVF is the product of cache size in bytes and total execution time in cycles, and it is not different for vulnerability modeling between block-level and word-level. Indeed, the difference between block-level and word-level vulnerability modeling is numerator and vulnerability, as shown in Figure 2.

$$CVF = \frac{vulnerability\ (byte \times cycles)}{cache\ size\ (byte) \times total\ execution\ time\ (cycle)} \tag{1}$$

### 3.1. Vulnerability Estimation at a Block-Level Granularity Is Quite Inaccurate

We have estimated the vulnerability at a word-level granularity since the basic unit of data access in cache is a word, not a block, in order to achieve the vulnerability accurately. Figure 2 clearly shows differences of vulnerabilities (shaded region) between word-level and block-level estimations under a simple scenario where a block (Block) containing two words (WORD0 and WORD1) is brought at $t_0$, and evicted at $t_4$. We consider two cases. The data stored in WORD0 is read at $t_1$, $t_2$, and $t_3$ in Case 1, while they are written in Case 2. A single-bit soft error for the entire scenario is assumed in write-back cache, and each period $(t_i, t_{i+1})$ is considered as one cycle for brevity's sake. We also assume that each word contains one byte data.

It is much more complex to estimate the vulnerability based on word-level vulnerability estimation than the block-level vulnerability modeling. The access information per word in a block is required and analyzed for word-level modeling, while the only

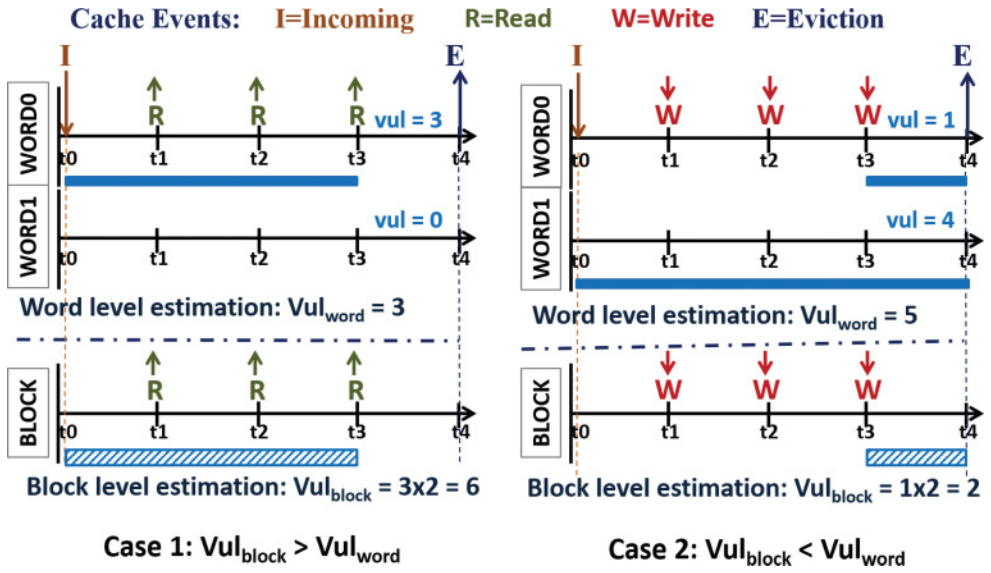



Fig. 2. Block-level and word-level vulnerability estimation examples without protection techniques.

access information per block for block-level modeling is required. Figure 2 shows two examples where previous block-level vulnerability modeling cannot provide the accurate modeling as compared to our word-level one. At both cases, one BLOCK contains two words such as WORD0 and WORD1, and read and write operations occur at the word-level while incoming and eviction happen at the block-level. Also, note that each time interval between $t_n$ and $t_{n+1}$ is assumed to one cycle and the word size to one byte for simplicity. Case 1 consists of three consecutive read operations of WORD0 at $t_1$, $t_2$, and $t_3$ after the incoming at $t_0$, and the eviction at $t_4$. Case 2 consists of three consecutive write operations of WORD0 at $t_1$, $t_2$, and $t_3$ after the incoming at $t_0$, and the eviction at $t_4$.

In Case 1, the read operations at $t_1$, $t_2$, and $t_3$ make the period from $t_0$ to $t_3$ of WORD0 vulnerable according to the vulnerability definition. Note that the read operations of corrupted data can affect the system behaviors, i.e., vulnerable. The interval ($t_3$, $t_4$) of WORD0 and ($t_0$, $t_4$) of WORD1 are not vulnerable due to the eviction at the clean state at $t_4$. Note that the cache data is not written back to the lower memory, i.e., no propagation of corrupted data and non-vulnerable, if it is clean under the write-back policy. Thus, our accurate word-level modeling estimates 3 as the vulnerability of this BLOCK under Case 1 where vulnerabilities of WORD0 and WORD1 are 3 and 0 byte $\times$ cycles, respectively. However, block-level estimation models these word access behaviors, i.e., read operations of WORD0 at $t_1$, $t_2$, and $t_3$ as block access ones (two bytes of block for three cycles) and it estimates the vulnerability as 6 byte $\times$ cycles ($=$ 3 cycles $\times$ 2 bytes) as shown in Figure 2 (left one). Thus, block-level vulnerability overestimates, i.e., $Vul_{block}$ (6 byte $\times$ cycles) is larger than $Vul_{word}$ (3 byte $\times$ cycles) which is correct in Case 1.

In Case 2, the write operations at $t_1$, $t_2$, and $t_3$ make the period from $t_0$ to $t_3$ of WORD0 non-vulnerable according to the vulnerability definition. Note that the write operations onto the corrupted data can erase the impact of induced soft errors, i.e., non-vulnerable. However, the eviction at dirty state at $t_4$ makes ($t_3$, $t_4$) of WORD0 and ($t_0$, $t_4$) of WORD1 vulnerable. Note that the corrupted data will be propagated to lower-level memory at the eviction if it is dirty under the write-back policy. Thus,

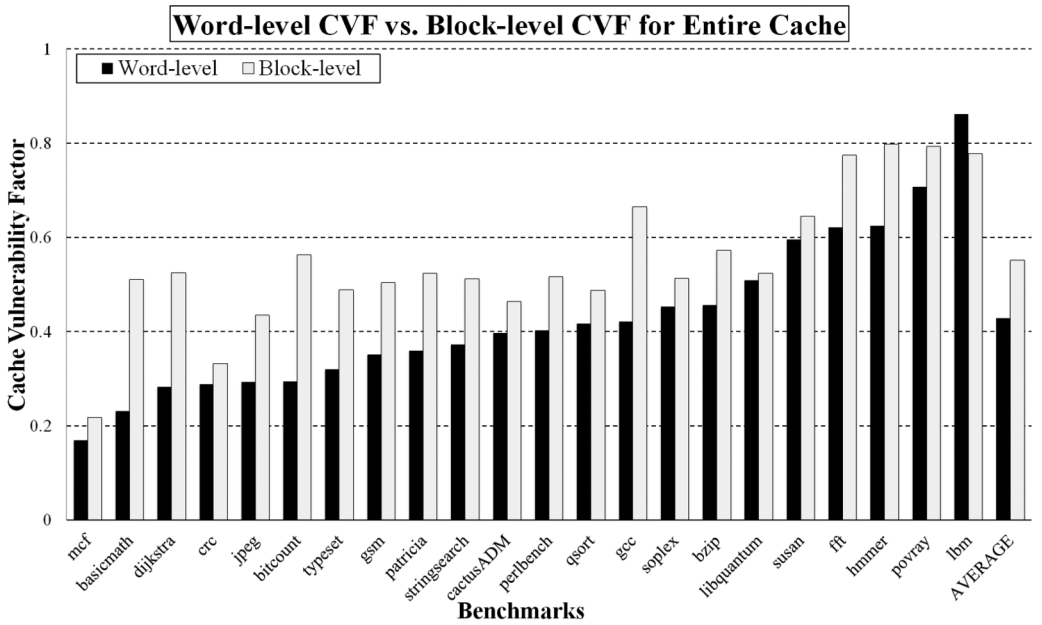



Fig. 3. Inaccuracy of block-level CVF estimations. Block-level vulnerability estimation is up to 121% inaccurate for the benchmark *basicmath*.

our accurate word-level modeling estimates five as the vulnerability of this BLOCK under Case 2 where vulnerabilities of WORD0 and WORD1 are 1 and 4 byte $\times$ cycles, respectively. However, block-level estimation models these word access behaviors, i.e., write operations of WORD0 at $t_1$, $t_2$, and $t_3$ as block access ones (two bytes of block for three cycles) and they are all non-vulnerable. Thus, it just estimates the vulnerability as 2 byte $\times$ cycles (= 1 cycle $\times$ 2 bytes) as shown in Figure 2 (right one). Thus, block-level vulnerability underestimates, i.e., $Vul_{block}$ (2 byte $\times$ cycles) is smaller than $Vul_{word}$ (5 byte $\times$ cycles) which is correct in Case 2.

Figure 3 plots L1 data CVF without protection over benchmarks with our gemV-cache. In Figure 3, the x-axis represents benchmarks sorted in the ascending order of CVF and the y-axis represents CVF. The dark bars show the CVF at our word-level granularity, while the light bars for each benchmark show the CVF when the vulnerability is estimated at a block-level granularity. We make several interesting observations from this graph. The difference between CVF estimations at word-level and block-level granularity varies with benchmarks. The maximum inaccuracy is 121% for *basicmath* benchmark, while the average is 37%. Even though the average inaccuracy is not much, the maximum inaccuracy is quite significant. Therefore, only block-level estimation can be significantly erroneous although block-level estimation is easier to understand and implement. More detailed analysis and breakdown of the differences will be in Section 3.2.

We also find that the block-level CVF is almost always (except for the *lbm* benchmark) larger than word-level CVF in Figure 3. This is because of two important reasons. The first reason is that a read to a word is considered as a read to the whole block with block-level vulnerability estimation. The accesses to a cache block are not evenly distributed among words in the cache block. CPU will read some specific words more often than other words in common. For example, in *basicmath* benchmark, only 4 bytes out of 64 bytes in a cache block are read by the CPU for about 98% of the whole time. Moreover,

80% of cache operations are read operation. In this case, block-level estimation always updates the vulnerability of the whole block (which is inaccurate) while word-level estimation only updates the vulnerabilities of the specific words (which are accurate). The second reason comes into play when a clean cache block is evicted. For example, about 77% of cache blocks that are evicted are at the clean state in the *basicmath* benchmark. Block-level estimation calculates that the interval from the last behavior of a block to its eviction is non-vulnerable. However, word-level estimation calculates that each interval from the last behavior of each word to the block's eviction is non-vulnerable. The sum of the non-vulnerable periods of each word in word-level CVF is clearly larger than the non-vulnerable period of the block in block-level CVF, which is why block-level CVF can be larger than word-level CVF in most cases. However, word-level CVF is larger than block-level CVF for the benchmark *lbm*. The main reason is that data evicts at the dirty state rather than the clean state. In case of eviction at the dirty state, block-level estimation decides the interval between the last behavior of that block and the block's eviction as vulnerable. However, accurate word-level estimation decides the interval between the last behavior of each word to the block's eviction in that case. Thus, the sum of vulnerable periods in each word can be larger. Indeed, 81% of evictions occur at the dirty state for the *lbm* benchmark, while 34% on average occur at the dirty state for other benchmarks.

Another interesting observation from Figure 3 is that CVF varies quite significantly among the benchmarks. Benchmark *mcf* has CVF of just 0.17, but for *lbm* benchmark, the CVF is about 0.86. Namely, only 17% of lifetime is vulnerable in the benchmark *mcf*, while almost 90% of lifetime is vulnerable for *lbm*. CVF depends on several factors, including temporal locality of accesses. For instance, if a block is read several times in quick succession and not accessed after that, then it will be less vulnerable, than a block that is read intermittently over a long duration of times. In fact, CVF calculation is quite complex, and therefore, we need a detailed algorithm to estimate it. If several factors have strong co-relations with CVF, then we can exploit this information to estimate the vulnerability for further purposes, e.g., dynamic protection at runtime.

The dirty state duration of cache blocks is one of the main factors that affect CVF over benchmarks. Dirty state duration is defined as the sum of dirty states' time over the execution time in percentage. Thus, the dirty state duration of each cache block is the interval between the first write operation to the eviction. When a cache block is dirty, it needs to be written back to the lower-level memory at its eviction, which can propagate the errors if they occurred. Thus, the larger portion of dirty state duration in the lifetime of cache blocks can increase the vulnerability as shown in Figure 4. In Figure 4, benchmarks are in an ascending order of CVF, and the dirty state duration follows this pattern in general. Therefore, cache vulnerability can be reduced if we can minimize the dirty state duration by protection techniques such as write-through or early write-back [Li et al. 2004] policies. Some benchmarks, however, such as *crc*, *gsm*, and *susan* do not follow this trend. In order to analyze the reason, we classify two CVFs: (i) Read CVF and (ii) Eviction CVF. Read CVF is defined as CVF when reads make the interval vulnerable at either the clean or dirty state, and Eviction CVF is when an eviction makes the interval vulnerable at the dirty state.

Note that only reads and evictions at the dirty state can make the interval vulnerable without protections. Figure 4 depicts Read CVF and Eviction CVF over benchmarks, and Eviction CVF takes up most portions of CVFs in benchmarks that follow the trend of dirty state durations in general. Interestingly, benchmarks *crc*, *gsm*, and *susan* present relatively larger portions of Read CVF such as 86%, 63%, and 56%, respectively, in CVFs. In *crc* and *susan*, the dirty state duration is much smaller than CVF and it means that these benchmarks have large portions of Read CVF at the clean state. Indeed, 96% and 97% of read CVFs occur at the clean state in benchmarks

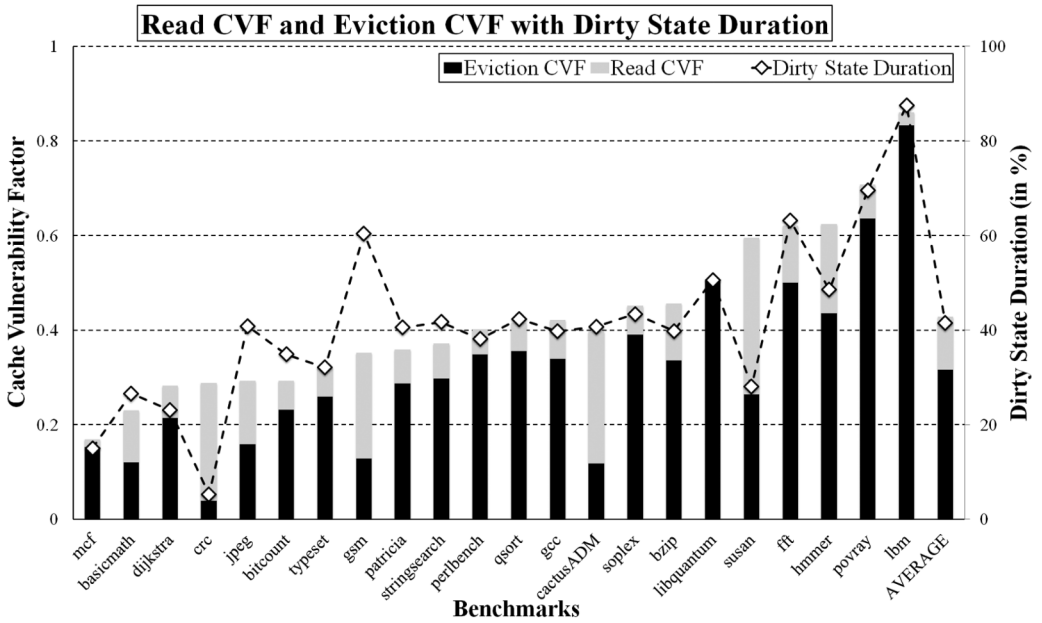



Fig. 4. CVF based on word-level modeling is proportional to the dirty state in general since vulnerability mainly comes from eviction at the dirty state.

*crc* and *susan*, respectively. However, Read CVF also takes up many portions of CVF in *gsm*, but the dirty state duration is much larger than CVF in that case. On the contrary to other benchmarks, *gsm* shows that relatively small portions of dirty state durations are vulnerable (e.g., writes at the dirty state happen often to make long time periods non-vulnerable). Indeed, only 55% of dirty state durations are vulnerable in *gsm*, while more than 90% of dirty state durations are vulnerable on average for the other benchmarks.

### 3.2. Dig Deeper into Inaccuracy of Block-Level CVF

From Figure 3, we note that CVF estimation at block-level granularity can be inaccurate by 37% on average and by up to 121%. However, this is just the tip of the iceberg – this is just the inaccuracy in the aggregate vulnerability statistics over all the data cache blocks. When we consider the vulnerability of a particular block or inaccuracy of vulnerabilities over specific time, the inaccuracy is even more dramatic. Figure 5 shows significant inaccuracy of block-level CVF when the CVF of each cache block is evaluated. The light bars show block-level CVF and the dark bars show word-level CVF for all 64 blocks in the 4KB direct-mapped cache with a benchmark, *basicmath*. A block (block number 1) shows up to 0.92 difference, and the average difference is about 0.36, which is even higher than the difference of the aggregate CVF (0.28). It is important since researchers have proposed partial protection techniques [Jeyapaul and Shrivastava 2011; Lee et al. 2006] to protect selected blocks, rather than all the blocks in caches, against soft errors due to high overheads. If they implement block-level estimation to select blocks for partial protections, they would select wrong blocks causing no improvement of reliability. Assume that the most vulnerable three blocks in data cache are completely protected by ECC or other protection techniques for the benchmark *basicmath* in Figure 5. Blocks 18, 29, and 35 are selected and protected by block-level estimation since they have the highest block-level CVF (0.97, 0.98, and

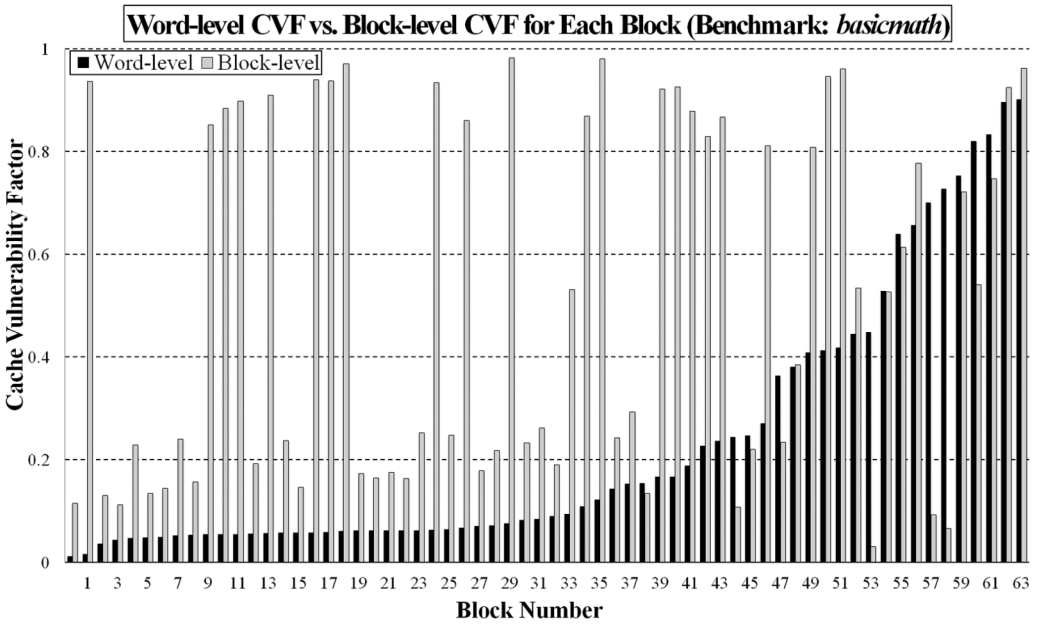
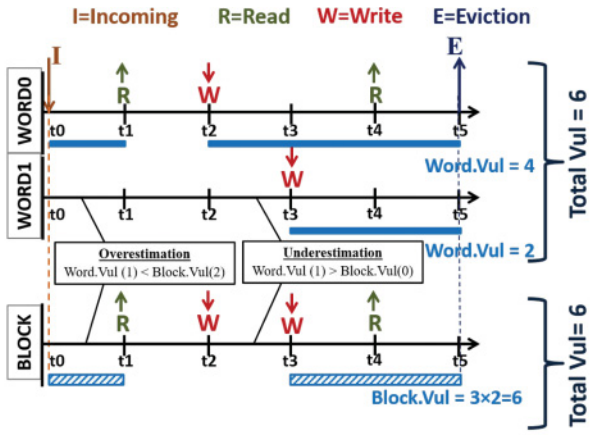


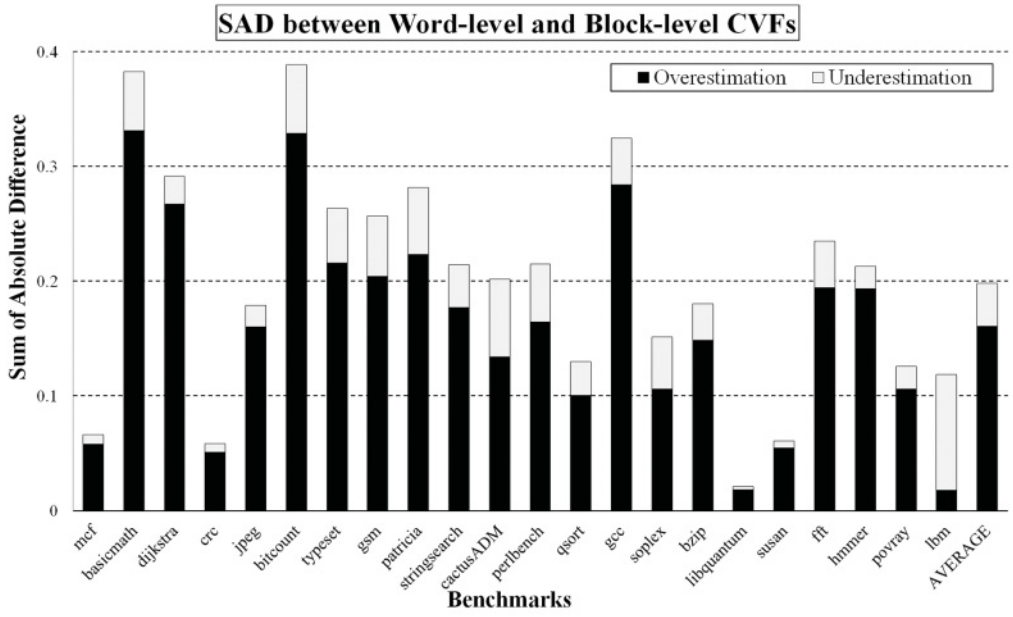

Fig. 5. Dramatic difference of block-level and word-level CVF for each block. If the vulnerability of a cache block is estimated based on block-level modeling, it can be 5,700% inaccurate as compared to the accurate word-level one.

0.98). However, it only reduces the vulnerability by just 1.8% as compared to the vulnerability without protection since word-level vulnerabilities of block 18, 29, and 35 are just 0.06, 0.08, and 0.09, respectively. Blocks 61, 62, and 63 are selected for protection based on our word-level estimation since their word-level vulnerabilities are 0.83, 0.90, and 0.90. In addition, vulnerability can be reduced by 18% as compared to no protection by protecting just 3 blocks out of 64 blocks.

Figure 6 shows the realistic accuracy of our word-level estimation as compared to block-level estimation. In Figure 6(a), we have estimated cache vulnerability based on word-level and block-level modeling under a simple scenario where a block (BLOCK) containing two words (WORD0 and WORD1) is brought at $t_0$ (incoming) and evicted at $t_5$. The data in WORD0 is read at $t_1$ and $t_4$, and they are written at $t_2$. The data in WORD1 is written at $t_3$. A single-bit soft error for the entire scenario is assumed in write-back cache, and each period $(t_i, t_{i+1})$ is considered as one cycle for brevity's sake. We also assume that each word contains one byte data. Read operations at $t_1$ and $t_4$ make periods $(t_0, t_1)$ and $(t_2, t_4)$ of WORD0 vulnerable according to the vulnerability definition. Note that the read operations of corrupted data can affect the system behaviors, i.e., vulnerable. The interval $(t_1, t_2)$ of WORD0 is not vulnerable due to write operation $t_2$, and $(t_0, t_3)$ of WORD1 is also not vulnerable due to write operation $t_3$. Note that the write operations onto the corrupted data can eliminate the impact of induced soft errors, i.e., non-vulnerable. However, the eviction at dirty state at $t_5$ makes $(t_4, t_5)$ of WORD0 and $(t_3, t_5)$ of WORD1 vulnerable. Note that the corrupted data will be propagated to lower-level memory at the eviction if it is dirty. Thus, our accurate word-level modeling estimates six as the vulnerability of this BLOCK where vulnerabilities of WORD0 and WORD1 are 4 and 2 byte $\times$ cycles, respectively. On the other hand, block-level estimation models these word access behaviors, i.e., read operations at $t_1$ and $t_4$ of WORD0 as block operations, and $(t_0, t_1)$ and $(t_3, t_4)$ are all vulnerable. In case of write operations at $t_2$ of WORD0 and at $t_3$ of WORD1, make the whole block non-vulnerable.




(a) Even though word-level and block-level vulnerability is the exactly same, block-level modeling is inaccurate due to overestimation and underestimation.




(b) The relative difference between word-level and block-level CVFs is just 0.28 as shown in Figure 3, but sum of absolute difference between them can be reach up to 0.39.

Fig. 6. SAD (Sum of Absolute Difference) is the sum of overestimation and underestimation of inaccurate block-level estimation as compared to the accurate word-level estimation. SAD can show the realistic inaccuracy of block-level estimation.

Eviction at $t_5$ makes this block vulnerable because of the dirty state. Thus, block-level modeling also estimates the vulnerability as 6 byte $\times$ cycles (=3 cycle $\times$ 2 bytes).

Interestingly, both block-level and word-level modeling results in the same vulnerability (6 byte $\times$ cycle) as shown in Figure 6(a). However, block-level vulnerability estimation is still inaccurate due to overestimation and underestimation even though aggravated vulnerability is the exactly same. For the interval $(t_0, t_1)$, block-level

modeling (2 byte × cycle) overestimates the vulnerability as compared to word-level one (1 byte × cycle). On the other hand, block-level modeling (0 byte × cycle) underestimates the vulnerability as compared to word-level one (1 byte × cycle) for $(t_2, t_3)$. Thus, we can compute word-level vulnerability by using overestimation and underestimation ($vul_{word} = vul_{block}$ - *overestimation* + *underestimation*). For instance, word-level vulnerability can be calculated as 6 ($=6 - 1 + 1$) in Figure 6(a).

Inaccurate block-level estimation can overestimate or underestimate cache vulnerability as shown in Figure 6(a), but the entire CVF can just show the relative difference between overestimation and underestimation. The Sum of Absolute Difference (SAD) is the sum of incorrectly estimated CVF between block-level and word-level estimation. In Figure 6(a), the CVF of block-level and word-level modeling is 0.6, so their relative difference is zero. However, SAD between block-level and word-level vulnerability estimation is 0.2, and it shows the realistic accuracy of block-level modeling. In Figure 6(b), the x-axis represents a set of benchmarks, and the y-axis represents the sum of overestimation and underestimation between block-level and word-level modeling. The upper light one at each bar in Figure 6(b) represents the portion that is vulnerable at word-level estimation even though block-level estimation considers it non-vulnerable (underestimation), and the lower dark one is opposite (overestimation). Hence, the upper one can cause the under-protection, and the lower one can cause the over-protection if hardware architects implement the protection technique for caches based on block-level estimation. On an average, the SAD (overestimation + underestimation) is about 0.2 and it can reach up to 0.39 for a benchmark, *bitcount*. Interestingly, relative difference for the benchmark *cactusADM* is just 0.07 as shown in Figure 3, but their SAD is 0.20 as shown in Figure 6(b).

### 3.3. Validation with Fault Injections

In order to validate our vulnerability models and the implementation of the vulnerability estimations in gemV-cache, we have performed 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` [McMahon 1986] and *Matrix Multiplication*. We have implemented an in-house *Matrix Multiplication* benchmark for exhaustive fault injection. We have chosen simple benchmark suites in order to exclude software-level masking effects. We have injected single-bit faults at a specific block during a specific interval and compared its output to the correct one that the benchmark returns without faults. It is declared as a failure if they are different or a system crashes. Otherwise, it is a success. Assuming a single-bit fault model, we have run millions of simulations, and compute the Failure Rate Equation (2):

$$FailureRate = \frac{Num.\ of\ Simulations\ that\ failed}{Total\ Num.\ of\ simulations} \tag{2}$$

For validation, the failure rate should match *CVF* as defined in Equation (1). Table I 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 word-level *CVF* match perfectly, thus validating our vulnerability models and implementation. For the block-level vulnerability estimation, it can be up to 300% inaccurate for block number 8 for the benchmark *Livermore Loops 12*. On average, block-level vulnerability modeling is

Table I. Validation of Our Models and Implementation of Word-level Vulnerability Estimation

| Benchmark | Matrix Multiplication | | |
|---|---|---|---|
| Block Number | 4 | 9 | 10 |
| Number of Simulations | 1,084,544 | 107,136 | 914,048 |
| Failure Rate | 97.54 | 96.35 | 94.83 |
| Word-level CVF | 97.54 | 96.35 | 94.83 |
| Block-level CVF | 97.51 | 95.82 | 87.06 |
| Inaccuracy | 0.03 | 0.56 | 8.19 |
| Benchmark | Livermore Loop 5 | | |
| Block Number | 1 | 11 | 12 |
| Number of Simulations | 44,672 | 53,504 | 115,968 |
| Failure Rate | 99.64 | 96.76 | 3.07 |
| Word-level CVF | 99.64 | 96.76 | 3.07 |
| Block-level CVF | 99.43 | 94.79 | 9.69 |
| Inaccuracy | 0.22 | 2.04 | 215.61 |
| Benchmark | Livermore Loop 8 | | |
| Block Number | 0 | 1 | 3 |
| Number of Simulations | 12,895,872 | 2,977,152 | 700,672 |
| Failure Rate | 46.27 | 93.84 | 4.80 |
| Word-level CVF | 46.27 | 93.84 | 4.80 |
| Block-level CVF | 47.72 | 91.86 | 8.79 |
| Inaccuracy | 3.13 | 2.10 | 83.06 |
| Benchmark | Livermore Loop 12 | | |
| Block Number | 0 | 8 | 15 |
| Number of Simulations | 452,096 | 118,912 | 53,376 |
| Failure Rate | 23.28 | 0.70 | 97.54 |
| Word-level CVF | 23.28 | 0.70 | 97.54 |
| Block-level CVF | 24.58 | 2.80 | 96.40 |
| Inaccuracy | 5.56 | 300.00 | 1.17 |
| Benchmark | Livermore Loop 18 | | |
| Block Number | 2 | 3 | 5 |
| Number of Simulations | 19,280,640 | 1,448,704 | 1,534,592 |
| Failure Rate | 96.20 | 55.61 | 52.15 |
| Word-level CVF | 96.20 | 55.61 | 52.15 |
| Block-level CVF | 92.38 | 92.07 | 8.66 |
| Inaccuracy | 3.97 | 65.55 | 83.40 |

*Note*: For all the selected words, we can get the perfectly matched vulnerability as compared to the exhaustive fault injection campaigns.

52% inaccurate as compared to *failure rate* from fault injection campaigns even for these simple set of benchmarks.

### 3.4. gemV-cache Implementation

In order to implement gemV-cache [Ko et al. 2015], we have developed algorithms that consider every behavior at the word-level in cache blocks and calculate the vulnerability accordingly. *algoECV* describes how to calculate vulnerability of cache without protection ($Vul_{np}$) and with block-level parity protection ($Vul_{bp}$). In this algorithm, *curTick* and *recentTick* represent the current tick and the tick of the most recent access to a block, respectively. *History* is a data structure which stores the last behavior such as incoming, read, write, and eviction and the most recent tick. In *history*, we use *uncertain* tick variable to postpone the decision of its vulnerability to being recorded. *Accessed*

**ALGORITHM 1:** algoECV (Estimate Cache Vulnerability)

*algoECV* returns vulnerabilities of cache without protections ($Vul_{np}$) and with block-level parity protection ($Vul_{bp}$). Based on our algorithm, we can get vulnerabilities with and without protections by just one simulation.

```
01: curtick ←current tick;
02: recentTick ←tick of the most recent access of block;
03: switch Operation do
04:    case INCOMING:
05:       clear dirty-bit of the block
06:       for all history h in the block do
07:          h.tick ← curTick
08:       end for
09:    case WRITE:
10:       set dirty-bit of the block;
11:       for all history h in the block do
12:          if h ∈ Accessed then
13:             h.tick ← curTick;
14:             h.uncertain ← 0;
15:          else
16:             h.uncertain += curTick − recentTick;
17:          end if
18:       end for
19:    case READ:
20:       for all history h in the block do
21:          if block is dirty then
22:             Vul_bp += curTick − recentTick;
23:             if h ∈ Accessed then
24:                Vul_bp += h.uncertain;
25:                h.uncertain ← 0;
26:                Vul_np += curTick − h.tick;
27:                h.tick ← curTick;
28:             end if
29:          else if h ∈ Accessed then
30:             Vul_np += curTick − h.tick;
31:             h.tick ← curTick;
32:          end if
33:       end for
34:    case EVICTION:
35:       for all history h in the block do
36:          if block is dirty then
37:             Vul_bp += curTick − recentTick + h.uncertain;
38:             Vul_np += curTick − h.tick;
39:          end if
40:       end for
41: end switch
42: return Vul_bp, Vul_np;
```

is another data structure containing all the *History* information of CPU requested words.

According to each operation such as Incoming, Write, Read, and Eviction, *algoECV* handles these tick values and associated data structures, and estimates the vulnerability of caches as shown in Algorithm 1. In case of Incoming, it clears the dirty-bit of the block (line 05) and saves the current tick to each word's *h* for every *History* in the block (line 07). In case of Write operation, it sets the dirty-bit of the block (line

10) and stores the current tick to *h* and resets the *uncertain* (line 13 and 14) if it is accessed. Otherwise, the difference between the current tick and the most recent tick (the uncertain duration for the other words due to the write operation to the word in the same block) is added to the *uncertain* tick (line 16). A write access to word(s) in the block can affect the other words' vulnerability estimation and this time period from the last behavior for the other words needs to be kept *uncertain* since the next behavior is going to make it vulnerable or not. Indeed, *uncertain* becomes non-vulnerable if the next behavior to this word is a write operation (line 14) while it becomes vulnerable if it is a read operation (line 24) or eviction (line 37) when this block is dirty. It is why we accumulate these intervals in *history* and reflect these effects in vulnerability calculation to deal with *uncertain* periods according to the neighbor word's behaviors. In case of Read operation, no protection can simply accumulate the time period from the last behavior to the vulnerability, $Vul_{np}$, and saves the current tick in *History* (lines 26, 27, 30, and 31).

To the contrary, the block-level parity protection can recover the corrupted data if it is clean, which means non-vulnerable. However, if it is dirty, the difference between the current tick and the most recent tick needs to be added to the vulnerability, $Vul_{bp}$, (line 22) and further *uncertain* ticks need to be added (line 24) if it is in *Accessed*. In case of Eviction, the difference between the current tick and the tick in *History* is added to the vulnerability for no protection (line 38) while their difference and *uncertain* in *History* need to be added to the vulnerability for parity protection (line 37), similar to the Read operation. Of course, these ticks are ignored to add up the vulnerability if it is clean. At last, *algoECV* returns $Vul_{np}$ and $Vul_{bp}$ for a block and the sum of all vulnerabilities of all the blocks in a cache is the vulnerability for a program. Similarly, we have developed algorithms to estimate the vulnerabilities for word-level parity protections by modifying configurations of status-bits and cache parameters. For instance, the word-level parity protection returns the same vulnerability as that of no protection except for the impacts of read operations at the clean state (line 30) since the word-level parity clears all the vulnerabilities at reads if cache is in the clean state.

In gemV-cache, we have implemented vulnerability estimation with various cache configurations such as protections (ECC or parity) and the granularity (dirty-bits and parity- and ECC-bits). Further, all of these different protection schemes can be returned at once, i.e., just one simulation with our gemV-cache. Our word-level vulnerability estimation on gemV-cache is much more complex than the traditional block-level vulnerability modeling due to the following reasons. First off, many more state variables are needed for bookkeeping at the word-level modeling, as opposed to that at the block-level. The access information per word, in a cache block needs to be recorded and is analyzed for word-level estimation. For instance, block-level modeling considers each word-access (read and write) as a block-level operation in case of unprotected caches and makes the whole block vulnerable when read. However, word-level modeling makes each word vulnerable or non-vulnerable according to their respective word-level accesses, and thereby estimates cache vulnerability accurately.

Secondly, the behavior of one word, affects the perceived vulnerability of the other words in the same block, which becomes a more important and complex factor when cache protection modeling is involved at the fine-grained granularity. The vulnerability of one word in a cache-block can depend on the read and write accesses on a neighboring word. For instance, if one word in the cache-block is written, then all the words in the same block become dirty (since they share the same dirty-bit on a block). Even if the other words are not accessed, they become vulnerable. It is because they will be written back into the memory and not discarded during eviction in a write-back cache.

Lastly, the protection granularity makes it more complicated to estimate cache reliability through word-level vulnerability modeling. If we assume that the parity is encoded at every write operation (no decoding), it is decoded to check whether an error occurred or not both at every read operation and at eviction. For instance, parity protection can be implemented at the word-level (a parity-bit per word) or block-level (a parity-bit per block), and our algorithm needs to log every word's time. On the other hand, different granularities of protections always return the same vulnerability statistics based on block-level vulnerability modeling.

## 4. TRICKY PROTECTION TECHNIQUES

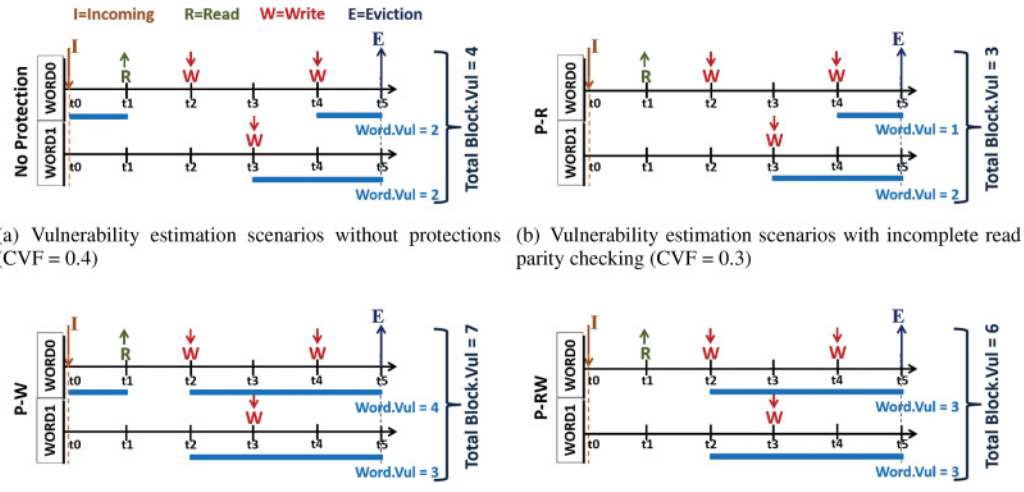
Most existing vulnerability estimation toolsets only allow modeling the vulnerability of unprotected caches. However, we need a vulnerability estimation toolset when protections are introduced to caches since soft errors are becoming a real threat. Our gemV-cache is able to estimate the vulnerability of caches with parity and ECC protections as described in Section 3.4. One very simple and power-efficient cache protection technique, widely implemented in the cache architecture of most existing commodity processors (e.g., ARM [Phelan 2003], Intel [Demshki and Shiveley 2010]), available in the market today, is parity-bit based protection against single-bit errors. And, another effective method to protect the cache memory against soft errors is applying ECC to the entire cache memory. In this article, we model data cache vulnerability of a parity-protected and ECC-protected cache at the word-level granularity in our gemV-cache toolset. And, we also study the impact of the design parameters on the protection achieved.

### 4.1. Incomplete Parity Checking Achieves Efficient Protection

The key hardware component involved in the design of parity-bit based protection in the cache, is the *Parity-Generator/Checker*, which generates the 1-bit parity for the data stored in a cache block and also updates the parity-bit when any data update occurs on the respective cache block. The same hardware component will also be used to detect errors by comparing the current parity-bit value with the stored parity-bit value in the stored data. For instance, (i) on only read access (P-R), the parity-checker is accessed to verify the parity-bit (PowerQUICC III [Semiconductor 2007] and ARM1156T2S [ARM 2007]); (ii) on only write access (P-W), the parity-checker is accessed to verify the parity-bit; and, (iii) on both read and write access (P-RW), the parity-checker is accessed to verify the parity-bit (ARM Cortex A8 [ARM 2014] and AM3359 [Texas Instruments 2011]). Note that parity-bit is checked (decoded) before read or write operations.

Figure 7 depicts the vulnerability estimation according the parity checking configurations. We assume that a block, which is composed of two words (WORD0 and WORD1) in this scenario for Figure 7, has one parity-bit. And, parity-bit can be checked at read operations (P-R/incomplete read parity checking), at write operations (P-W/incomplete write parity checking), and at both read and write operations (P-RW/complete read and write parity checking). Data is brought at $t_0$ and evicted $t_5$ in this block. Data stored in WORD0 is read at $t_1$ and written at $t_2$ and $t_4$, and data stored in WORD1 is written at $t_3$. Without protections, $(t_0, t_1)$ of WORD0 is vulnerable due to read operation, and $(t_4, t_5)$ of WORD0 and $(t_3, t_5)$ of WORD1 is vulnerable as shown in Figure 7(a).

With *complete read and write parity checking*, it has zero vulnerability in case of clean state as shown in Figure 7(d). If errors are detected at clean state, clean data in the lower-level memory can be brought to the cache to correct errors. However, it is always vulnerable after first write operation at $t_2$ since soft errors can be detected but there is no identical data in the lower-level memory in case of the dirty state. In P-RW, parity check during the first write at $t_2$ can detect and recover an error while it cannot recover after then (such as at $t_3$, $t_4$, and $t_5$). We can correct soft errors if we




(a) Vulnerability estimation scenarios without protections (CVF = 0.4)

(b) Vulnerability estimation scenarios with incomplete read parity checking (CVF = 0.3)

(c) Vulnerability estimation scenarios with incomplete write parity checking (CVF = 0.7)

(d) Vulnerability estimation scenarios with complete read and write parity checking(CVF = 0.6)

Fig. 7. Vulnerability estimation scenarios with diverse parity checking protocols.

have additional recovery mechanism such as checkpoint and rollback [Wang and Patel 2006]. However, we do not consider the additional protection techniques except parity and ECC in this article. On the other hand, with *incomplete read parity checking*, it has the least vulnerability among these checking configurations as shown in Figure 7(b). The periods $(t_2, t_4)$ of WORD0 and $(t_2, t_3)$ of WORD1 are not vulnerable since errors in these periods can be overwritten due to the write operations at $t_3$ and $t_4$. Note that parity-bit is decoded before read (P-R) and before read and write (P-RW) operations. Interestingly, P-W is more vulnerable than unprotected cache even with the additional redundancy as shown in Figure 7(b). In P-W, read at the clean state makes vulnerable periods since it does not check the parity-bit.

Figure 8 clearly shows the efficacy of parity protections with *incomplete read parity checking* with benchmarks. In Figure 8, the x-axis represents benchmarks and the y-axis represents the normalized vulnerability of each parity checking configuration to that of no protection. *Complete parity checking* reduces the vulnerability by only 5%, while *incomplete read parity checking* reduces it by 15% on average. However, the vulnerability is worsened by 56% with *incomplete write parity checking* as shown in Figure 7(b). It is interesting that *incomplete read parity checking* is the most effective way to reduce the vulnerability among parity checking protocols in spite of the lesser checking overhead in terms of hardware and power consumption, as shown in Figure 9. The effectiveness of parity techniques with *incomplete read parity checking* depends on the characteristics of benchmarks such as cache access patterns. For instance, parity protection with *incomplete read parity checking* for *crc* can decrease the vulnerability by 82% as compared to that of no protection. We have observed that the efficacy of parity protection is affected by the amount of vulnerable periods at the clean state. Note that parity protections cannot bring data from the lower-level memory at the dirty state. In our experiments, more than 80% of vulnerable periods occur at the clean state in the benchmark *crc* in case of no protection, as stated in Section 3.2, and these intervals can be effectively corrected by parity protections. With the same reason (high portion of the clean state), parity is also effective for the benchmark, *susan*. On the

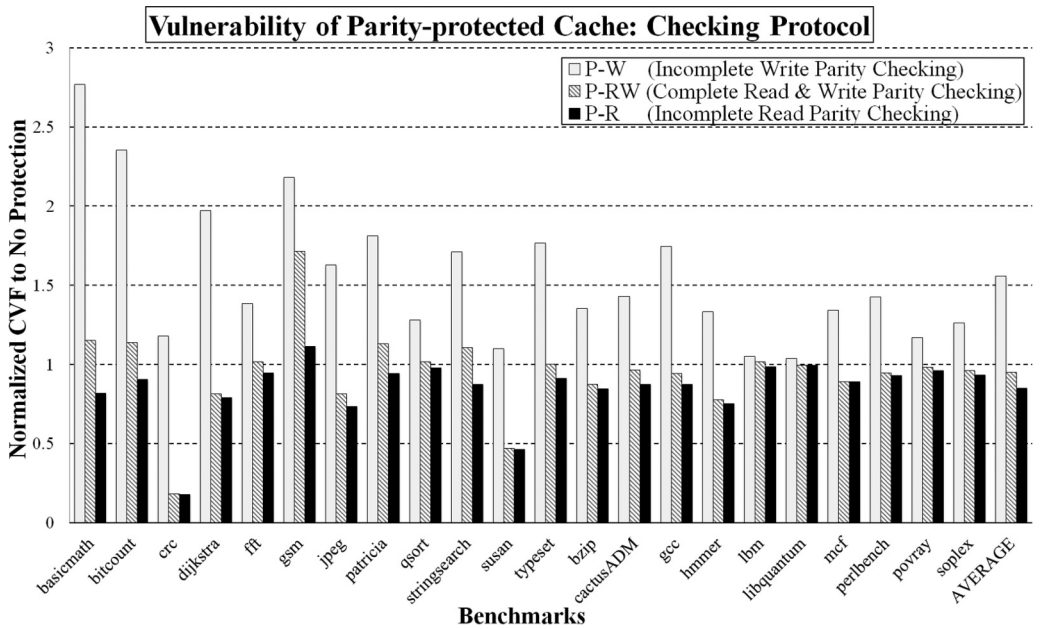



Fig. 8. Incomplete read parity checking (checking only at reads) achieves the highest reliability among various parity checking protocols. Complete parity checking is more vulnerable than incomplete read parity checking even with the additional redundancy.

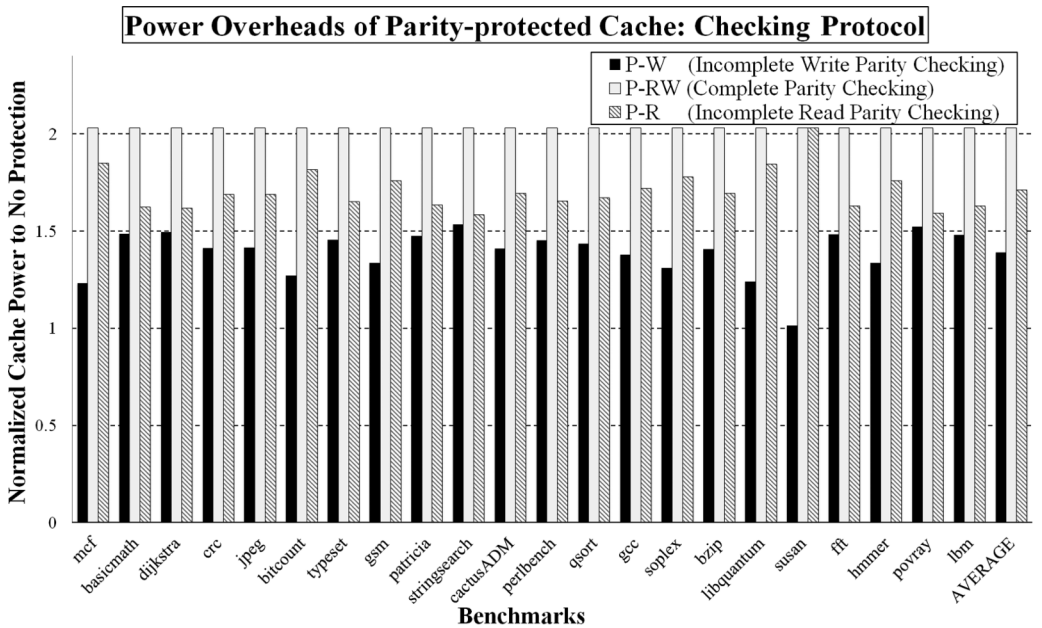



Fig. 9. In the design of a parity-protected cache, the power overheads caused by parity checking at reads are 30% lower than that when parity is checked on both reads and writes.

other hand, vulnerable periods at the clean state are only 7% in the benchmark *gsm* and thus it is less effective and even up to 71% worse, as shown in Figure 8.

Figure 9 plots the relative power overheads incurred by the parity-checking configurations for 4KB cache architecture across benchmarks. The y-axis in Figure 9 represents the normalized power consumption to that of unprotected cache. To estimate power consumption, we compute the read/write power of this parity-checking protocol implementation in the cache by manipulating CACTI 5.0 [Thoziyoor et al. 2008] for 45nm technology node. We have designed the unit for this cache, synthesized it in 45nm technology, and obtained power numbers using PowerMill[Huang et al. 1995] in order to estimate the power of the parity generation/checking hardware logic. We can observe that when checking the parity-value on both reads and writes (P-RW), P-RW incurs 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 that parity is implemented at the block-level granularity. The power overhead that comes from P-W is the least (39%) among parity checking protocols, but it increases the vulnerability by 56% on average as compared to no protection.

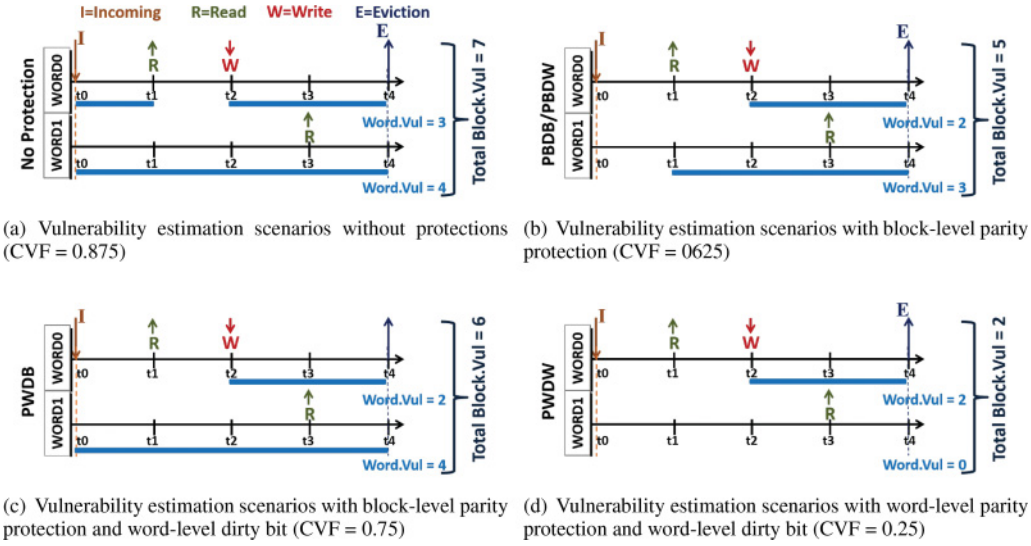
In short, parity-checking when performed over read accesses alone provides the better level of parity protection (avg. 15% and max. 82%) to the cache at 30% lower-power overheads compared with parity-checking when performed over both read and write accesses. We have run several simulations varying in cache size and cache associativity and observe that the power-efficient protection achieved through the P-R parity-checking protocol is consistent across cache configurations.

### 4.2. Fine-Grained Status-Bits Maximize the Achieved Parity Protection

Another key design parameter involved in the design of a parity-protected cache is the configuration of the status-bits (parity-bit and dirty-bit), which adds to the hardware overhead. In addition, the vulnerability definition of the parity-protected cache is dependent on the status-bit configurations. In a parity-protected cache, a parity-bit can be implemented at the block-level (Itanium 2 [McNairy and Soltis 2003]) and word-level (PowerQUICC III [Semiconductor 2007], Cortex R4 [ARM 2010], and CPPC [Manoochehri et al. 2011]), and the dirty-bit can be also implemented for block-level (Cortex R4 [ARM 2010]) and word-level (CPPC [Manoochehri et al. 2011]). Figure 10 demonstrates the vulnerability definition of a cache block composed of two words (WORD0 and WORD1). In Figure 10, data is brought in the cache at $t_0$ and evicted at $t_4$. Data in WORD0 is read at $t_1$ and written at $t_2$, and WORD1 data is read at $t_3$. In case of unprotected cache, $(t_0, t_1)$ of WORD0 and $(t_0, t_3)$ of WORD1 is vulnerable due to read operations, and $(t_2, t_4)$ of WORD0 and $(t_3, t_4)$ of WORD1 are also vulnerable because of eviction at dirty state, as shown in Figure 10(a).

We have implemented with the P-R (parity-check on reads only) protocol for the following status-bit configurations:

**Parity Per Block and Dirty Per Block (PBDB):** *Coarse-grained* status-bits – Since the entire cache-block is configured with one parity-bit, a read access on any one word (in non-dirty blocks) can trigger recovery of the entire cache-block, since the single parity-bit cannot identify the exact word that is erroneous as shown in Figure 10(b). 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, as described in *algoECV*.




(a) Vulnerability estimation scenarios without protections (CVF = 0.875)

(b) Vulnerability estimation scenarios with block-level parity protection (CVF = 0625)

(c) Vulnerability estimation scenarios with block-level parity protection and word-level dirty bit (CVF = 0.75)

(d) Vulnerability estimation scenarios with word-level parity protection and word-level dirty bit (CVF = 0.25)

Fig. 10. Vulnerability estimation examples with diverse status-bit configurations. Note that the granularity of dirty-bit does not affect the vulnerability if a parity-bit is implemented on block-level.

**Parity Per Block and Dirty Per Word (PBDW):** *Medium-grained* status-bits – In this configuration, though each word in the block is configured with its respective dirty-bit, the vulnerability definition does not differ from that of the PBDB configuration as shown in Figure 10(b). If any one word of a cache-block is dirty (based on its respective dirty-bit), the entire cache-block will have to be considered dirty, because the single parity-bit cannot know which word in a cache block has corrupted values.

**Parity Per Word and Dirty Per Block (PWDB):** *Medium-grained* status-bits – If a parity-bit is associated with every word in the cache-block, parity-checks on read accesses can identify single-bit errors and also trigger targeted recovery of the erroneous word in case of clean state. Owing to this targeted recovery mechanism, vulnerability of the neighboring words during read accesses are not affected. For instance, we see that the vulnerability of WORD0 and WORD1 are defined by the read/write accesses on the respective words only as shown in Figure 10(c). Since the entire cache block is configured with one dirty-bit, a write on any one word renders the entire cache block dirty, and therefore an updated word in the cache block cannot be identified. This affects the recovery mechanism and therefore renders the entire cache block vulnerable.

**Parity Per Word and Dirty Per Word (PWDW):** *Fine-grained* status-bits – If every word in the cache block is associated with its respective dirty-bit and parity-bit, the fine-grained status-bit configuration helps achieve increased parity-based protection. Since each word has its respective parity-bit, targeted recovery is possible during read accesses in case of clean state. In addition, since each word has its respective dirty-bit, the updated words can be identified accurately, thus assisting in the targeted recovery mechanism. In Figure 10(d), we see that WORD1 is non-vulnerable from incoming to eviction, because the WORD1 has never been updated by the program, and only the words that have been updated are vulnerable.

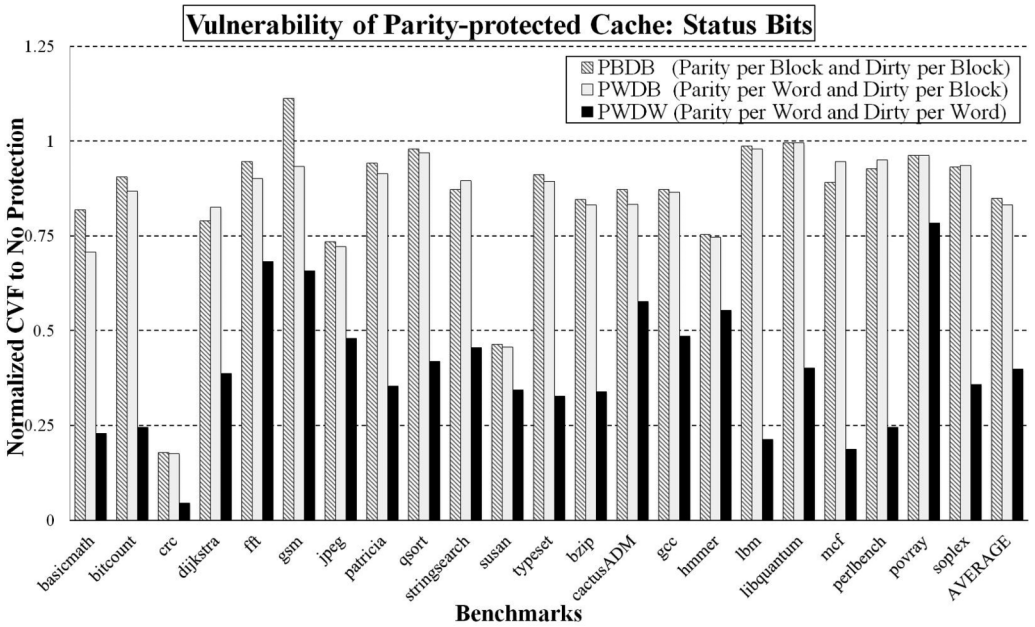


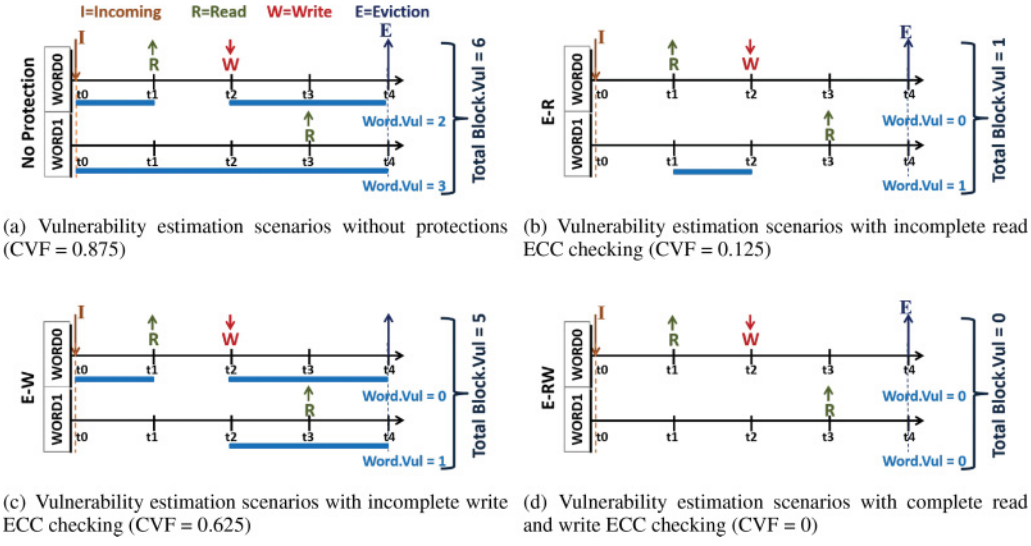

Fig. 11. Fine-grained parity with block-level dirty-bit reduces the vulnerability by only 2% as compared to block-level parity and dirty-bits. Fine-grained dirty-bit along with parity-bit per word is the best in terms of vulnerability (60% reduction).

Figure 11 shows the effectiveness of parity protections by varying the granularity and configuration of the status-bits (parity-bit and dirty-bit) which can induce hardware overhead in a parity-protected cache implementation. In Figure 11, the x-axis represents benchmarks and the y-axis represents normalized vulnerability of each status-bits granularity and configuration by the vulnerability without protections. The coarse-grained PBDB configuration reduces the vulnerability by 15% on average. Interestingly, medium-grained PBDW configuration (as in ARM Cortex R4 [ARM 2010]) reduces only 17% on average even though it needs parity-bit per every word in cache blocks. The fine-grained PWDW reduces the vulnerability 60% and it achieves the maximum level of protection.

In short, it is interesting that the granularity of parity-bits does not affect the vulnerability much without fine-grained dirty-bits. It is mainly because it cannot locate which word is dirty or clean. Hence, hardware architects have to change the granularity of both parity-bits and dirty-bits in order to reduce the vulnerability effectively.

### 4.3. ECC Protection Can Be Vulnerable for Single-Bit Flips

We need to consider two kinds of key hardware components of ECC-based protection in the cache—ECC checking protocol and the granularity of status-bits, especially ECC-bits. In ECC protections, dirty-bit does not affect the protection efficacy since it can correct soft errors regardless of dirty status. First off, Figure 12 depicts the vulnerability estimation according the ECC checking configurations. In Figure 12, we assume there are ECC-bits implemented per single block, i.e., block-level ECC protection. In Figure 12, a block is composed of two words (WORD0 and WORD1), and cache data is brought at $t_0$ and evicted at $t_4$. And, cache behavior is exactly the same with Figure 10.




(a) Vulnerability estimation scenarios without protections (CVF = 0.875)

(b) Vulnerability estimation scenarios with incomplete read ECC checking (CVF = 0.125)

(c) Vulnerability estimation scenarios with incomplete write ECC checking (CVF = 0.625)

(d) Vulnerability estimation scenarios with complete read and write ECC checking (CVF = 0)

Fig. 12. Vulnerability estimation examples with diverse status-bit configurations. Note that checking ECC-bits at read operations is more vulnerable than that at both read and write operations.

ECC checking at both read and write operations (complete read and write ECC checking or E-RW) provides the complete reliability, which means zero vulnerability. In case of ECC-protection, it can correct soft errors regardless of dirty status if detected. Since ECC-bits are checked at every behavior in E-RW, it can detect and correct all the single-bit soft errors as shown in Figure 12(d). In incomplete write ECC-checking or E-W, read operations always make vulnerable periods as shown in Figure 12(c) since it does not check ECC-bits at read operations.

Interestingly, ECC checking at reads (incomplete read ECC checking or E-R) is still vulnerable even for single-bit flips as shown in Figure 12(b), while incomplete read checking provides better vulnerability than complete read and write checking in case of parity protection. ECC checking at read operations can be vulnerable due to the behaviors of other words in the same block. As depicted in Figure 12(b), the period $(t_1, t_2)$ of WORD1 is vulnerable since a write operation of WORD0 at $t_2$ generates new check bits for the whole block. In addition, the erroneous data could be included if an error occurred $(t_1, t_2)$ at WORD1; and, this erroneous data can be propagated to CPU due to the read operation at $t_3$. It is interesting that incomplete ECC protection such as E-R and E-W cannot guarantee the perfect fault coverage (zero vulnerability) even for single-bit flips according to the ECC checking protocols.

Figure 13 shows CVF with complete and incomplete read ECC protections as compared to CVF of no protection. ECC protection may provide the perfect error recovery against soft errors regardless of the cache state. However, it is interesting that incomplete block-level ECC protections do not completely remove the vulnerability. On average, incomplete read ECC and incomplete write ECC reduce the vulnerability by 90% and 25%, respectively, over benchmarks. In case of ECC protections, the frequency of write accesses affects the effectiveness of vulnerability reductions. For instance, only 1% of total accesses in *susan* are the write operation and its vulnerability can be effectively reduced into almost zero by the block-level ECC protection with the incomplete read checking protocol.

Another key design parameter to the design of ECC-protected cache is the configuration of status-bits, especially ECC-bits. Figure 14 shows the vulnerability estimation

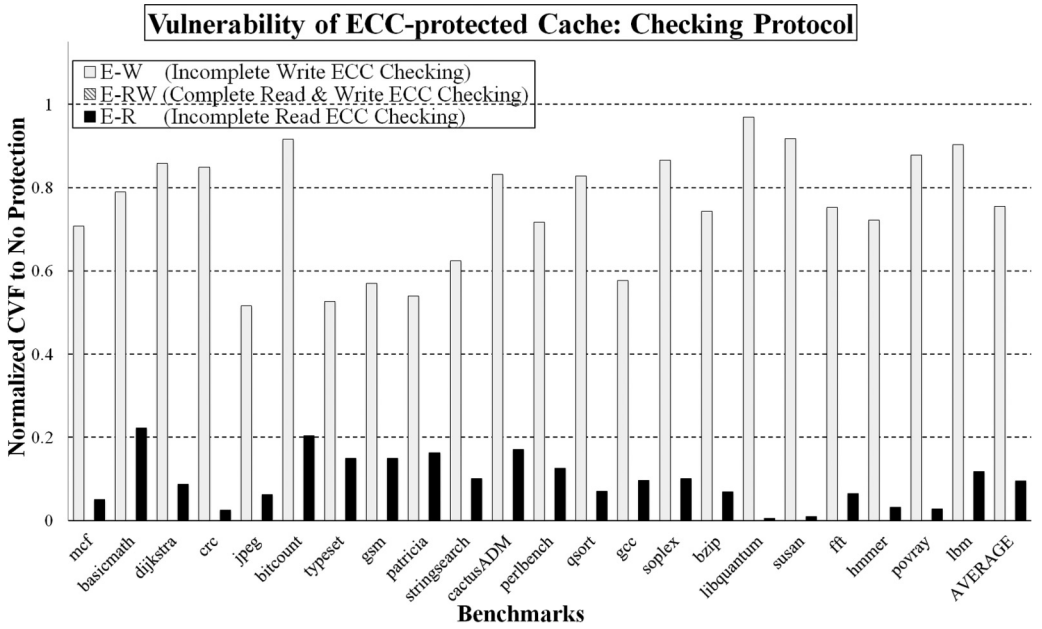
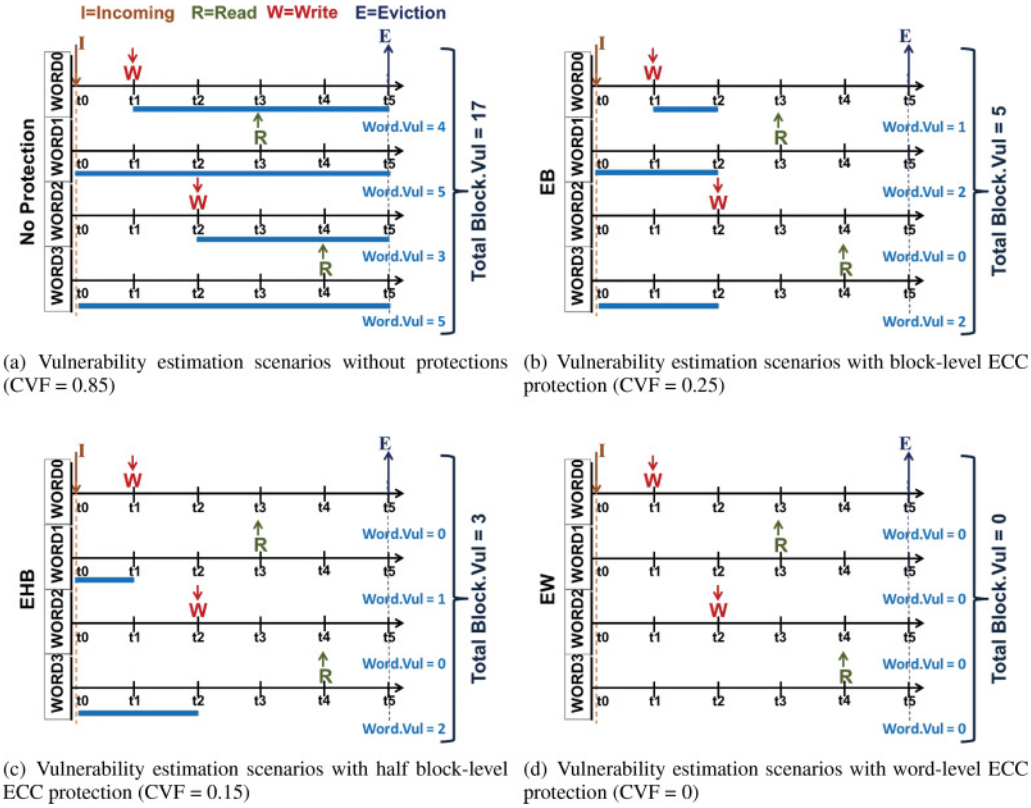



Fig. 13. Incomplete read ECC checking does not remove the vulnerability completely, while complete ECC checking provides zero vulnerability.

with varying the granularity of ECC-bits under a sample scenario. Note that ECC-bits are checked at only read operations since ECC-checking at both read and write operations makes vulnerability zero. In Figure 14, a block is composed of 4 words (WORD0, WORD1, WORD2, and WORD3) in order to separate ECC-bits per half-block and ECC-bits per word. Cache data is brought into this block at $t_0$ and evicted at $t_5$. Data stored in WORD0 and WORD2 is written at $t_1$ and $t_3$, respectively. Data in WORD1 and WORD3 is read at $t_2$ and $t_4$, respectively. Without protections, $(t_1, t_5)$ of WORD0, $(t_0, t_5)$ of WORD1, $(t_2, t_5)$ of WORD2, and $(t_0, t_5)$ of WORD3 are vulnerable as shown in Figure 14(a).

With block-level ECC protection or EB, write operation of other words in the same block can make vulnerable periods as shown in Figure 14(b). For instance, $(t_1, t_2)$ of WORD0 and $(t_0, t_2)$ of WORD1 are vulnerable due to the write operation of WORD2 at $t_2$. However, $(t_1, t_2)$ of WORD0 and WORD1 are not vulnerable with half block-level ECC protection or EHB as shown in Figure 14(c). In EHB, WORD0 and WORD1 are protected by ECC protection, and their vulnerability estimation is not affected by other words such as WORD2 and WORD3. However, $(t_0, t_1)$ of WORD1 is still vulnerable even with half block-level ECC protections due to write operation of WORD0 at $t_1$. In case of word-level ECC protection or EW, there are no vulnerable periods as shown in Figure 14(d).

Figure 15 shows CVF according to the granularity of ECC-bits with the incomplete read ECC checking protocols. As we described before, 10% of lifetime is still vulnerable by the block-level ECC protection when we check ECC-bits only at reads. CVF can be decreased by 38% by two sets of ECC-bits per block as compared to ECC per one block. We have used 64 bytes as a single block, so we apply ECC-bits per 32 bytes as ECC-bits half block (EHB) implementation for our environments. ECC protection with ECC-bits per word can eliminate the entire vulnerable periods, but it requires ECC-bits per each word. Thus, there are two methodologies in order to protect cache




(a) Vulnerability estimation scenarios without protections (CVF = 0.85)

(b) Vulnerability estimation scenarios with block-level ECC protection (CVF = 0.25)

(c) Vulnerability estimation scenarios with half block-level ECC protection (CVF = 0.15)

(d) Vulnerability estimation scenarios with word-level ECC protection (CVF = 0)

Fig. 14. Vulnerability estimation examples with diverse status-bit configurations on ECC protection. Note that ECC-bits are checked at just read operations.

memory perfectly by ECC protection—complete read and write ECC checking and fine-grained ECC-bits. The former checks ECC-bits more frequently than the latter due to the additional checking at write operations. And, the later can incur larger area overhead than the former since the later requires ECC-bits per each word, not a block. interAptiv [Imagination 2012] processor has ECC-bits per word, and it checks ECC-bits at both read and write operations. However, we do not have to check ECC-bits at both reads and writes if there are ECC-bits per word for the single-bit flips.

In short, we need to be careful in the implementation of parity and ECC protections on caches. We can think that parity should be checked for write and read operations to improve the reliability of cache memories. It is, however, interesting that parity implementations of additional checks at writes (decoded and encoded as well) can even increase the vulnerability. Over benchmarks, fine-grained protections (parity-bit and dirty-bit at a word-level granularity) with checks at read operations (incomplete read parity protection) can decrease the vulnerability by about 15%, while those with checks at both read and write operations (complete parity protection) can only decrease the vulnerability by about 5%. It is also interesting that block-level ECC checks at both read and write operations (complete ECC protection) can make the vulnerability zero, while ECC checks at read operations only (incomplete read ECC protection) do not bring the vulnerability down to zero.

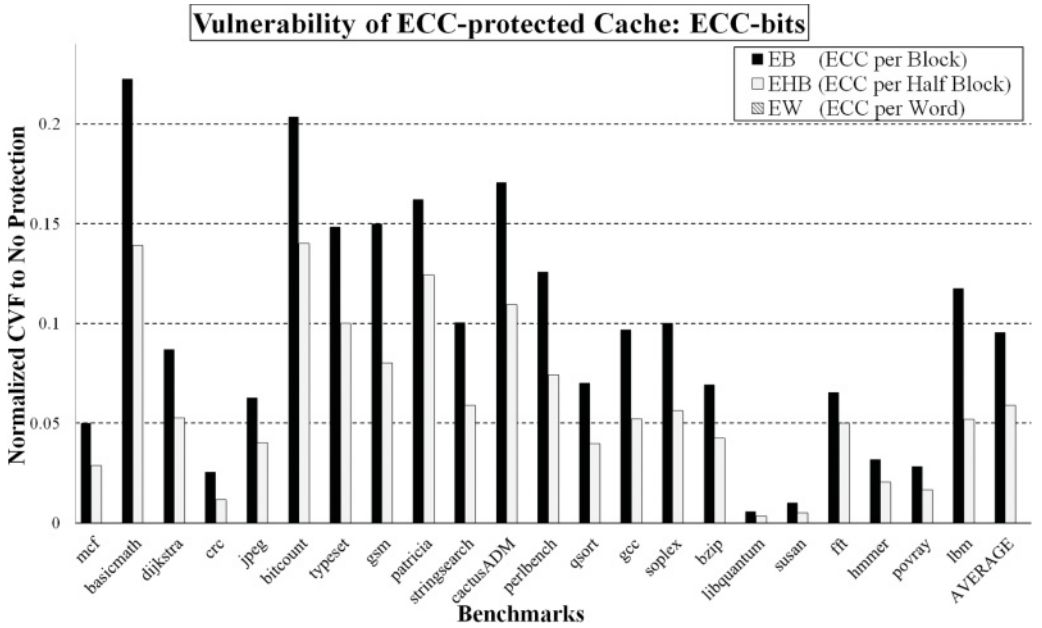



Fig. 15. CVF with ECC protections are affected by the granularity of ECC-bits.

## 5. CONCLUSION

Soft errors are becoming a real threat to modern embedded systems. Caches are the most susceptible to soft errors and several protection techniques based on parity and ECC have been presented. However, no existing scheme can accurately estimate how effective these protection techniques are in terms of vulnerability. To this end, we propose a protection-aware vulnerability estimation by gemV-cache at the fine-grained word-level modeling. Our experiments with gemV-cache find out several interesting results: (i) block-level modeling and estimation are quite inaccurate as compared to word-level one, (ii) parity protection is not a good option in case of the early dirtiness, (iii) parity checking at only read operations is more effective in terms of vulnerability and power consumption than that at both read and write operations, (iv) the granularity of either only parity-bit or dirty-bit does not affect the vulnerability mainly, (v) introduction of both parity-bit and dirty-bit per word can significantly improve the efficacy of parity protections, and (vi) ECC protection can be vulnerable if block-level ECC-bits are checked only at read operations, not at both read and writes.

Our future works will consider multiple bit soft errors in cache memory since the error rate of multi-bit soft errors is becoming larger for modern embedded systems. Further, our future works also include tradeoff study of various protection techniques in terms of power and performance with vulnerability, fine-grained modeling, and estimation of vulnerability for other hardware components in embedded processors, and further investigation of vulnerability reduction techniques.

## REFERENCES

ARM. 2007. ARM1156T2-S Technical Manual. (2007). http://infocenter.arm.com/help/topic/com.arm.doc.ddi0338g/index.html.

ARM. 2010. ARM Cortex-R4 and Cortex-R4F Technical Reference Manual. (2010). http://infocenter.arm.com/help/topic/com.arm.doc.ddi0363e/index.html.

ARM. 2014. Cortex-A8 Technical Reference Manual. (2014). http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344h/index.html.

G.-H. Asadi, V. S. Mehdi, B. Tahoori, and D. Kaeli. 2005. Balancing performance and reliability in the memory hierarchy. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*. IEEE Computer Society, Washington, D.C., 269–279.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, and others. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

Michael Demshki and Robert Shiveley. 2010. Advanced reliability for Intel Xeon processor-based servers. *Intel Corporation*.

A. Dixit and A. Wood. 2011. The impact of new technology on soft error rates. In *IEEE International Reliability Physics Symposium*. 5B.4.1–5B.4.7.

L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, and C. Lopez-Ongil. 2012. Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. *IEEE Trans. Comput.* 61, 3 (March 2012), 313–322.

Ronaldo R. Ferreira, Gabriel L. Nazar, Jean Da Rolt, Álvaro F. Moreira, and Luigi Carro. 2016. Live-out register fencing: Interrupt-triggered soft error correction based on the elimination of register-to-register communication. *ACM Transactions on Embedded Computing Systems* 15, 3, Article 60 (May 2016), 25 pages.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization (WWC-4)*. IEEE Computer Society, 3–14.

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17.

Charlie X. Huang, Bill Zhang, An-Chang Deng, and Burkhard Swirski. 1995. The design and implementation of PowerMill. In *International Symposium on Low Power Design (ISLPED'95)*. ACM, 105–110.

Imagination. 2012. interAptiv Multiprocessing System Datasheet. (2012).

R. Jeyapaul and A. Shrivastava. 2011. Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 105–114.

Yohan Ko, Reiley Jeyapaul, Youngbin Kim, Kyoungwoo Lee, and Aviral Shrivastava. 2015. Guidelines to design parity protected write-back L1 data cache. In *Design Automation Conference (DAC'15)*. ACM, Article 24, 6 pages.

Yohan Ko, Jihoon Kang, Jongwon Lee, Yongjoo Kim, Joonhyun Kim, Hwisoo So, Kyoungwoo Lee, and Yunheung Paek. 2016. Software-based selective validation techniques for robust CGRAs against soft errors. *ACM Transactions on Embedded Computing Systems* 15, 1, Article 20 (Jan. 2016), 26 pages.

PaKJW Kudva, J. Kellington, P. Sanda, Ryan McBeth, John Schumann, and Ron Kalla. 2007. Fault injection verification of IBM POWER6 soft error resilience. In *Architectural Support for Gigascale Integration Workshop*. Citeseer.

Kyoungwoo Lee, Aviral Shrivastava, Ilya Issenin, Nikil Dutt, and Nalini Venkatasubramanian. 2006. Mitigating soft error failures for multimedia applications by selective data protection. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. ACM, 411–420.

Lin Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. 2004. Soft error and energy consumption interactions: A data cache perspective. In *International Symposium on Low Power Electronics and Design*. 132–137.

Mehrtash Manoochehri, Murali Annavaram, and Michel Dubois. 2011. CPPC: Correctable parity protected cache. In *International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 223–234.

Frank H. McMahon. 1986. *The Livermore Fortran Kernels: A computer test of the numerical performance range*. Technical Report. Lawrence Livermore National Lab., CA.

C. McNairy and D. Soltis. 2003. Itanium 2 processor microarchitecture. *Micro, IEEE* 23, 2 (2003), 44–55.

Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. 2005. Robust system design with built-in soft-error resilience. *Computer* 38, 2 (2005), 43–52.

Sparsh Mittal and Jeffrey S. Vetter. 2016. Reducing soft-error vulnerability of caches using data compression. In *Great Lakes Symposium on VLSI (GLSVLSI'16)*. ACM, 197–202.

Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *IEEE/ACM International Symposium on Microarchitecture*. 29–40.

R. Naseer, Y. Boulghassoul, J. Draper, S. DasGupta, and A. Witulski. 2007. Critical charge characterization for soft error rate modeling in 90nm SRAM. In *IEEE International Symposium on Circuits and Systems*. 1879–1882.

Richard Phelan. 2003. Addressing soft errors in ARM core-based designs. *White Paper*, ARM Ltd. (Dec. 2003).

N. N. Sadler and D. J. Sorin. 2006. Choosing an error protection scheme for a microprocessor's L1 data cache. In *International Conference on Computer Design*. 499–505.

Freescale Semiconductor Application Note. 2007. Error Correction and Error Handling on PowerQUICC III Processors. (2007). http://application-notes.digchip.com/314/314-66495.pdf.

S. Z. Shazli, M. Abdul-Aziz, M. B. Tahoori, and D. R. Kaeli. 2008. A field analysis of system-level effects of soft errors occurring in microprocessors used in information systems. In *IEEE International Test Conference*. 1–10.

C. Slayman. 2010. Alpha particle or neutron SER-What will dominate in future IC technology. (2010).

Texas Instruments. 2011. AM3359 Sitara Processor. (2011). http://www.ti.com/lit/ds/symlink/am3351.pdf.

Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. 2008. CACTI 5.1. *HP Laboratories, April* 2 (2008).

Nicholas J. Wang and Sanjay J. Patel. 2006. ReStore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Trans on* 3, 3 (2006), 188–201.

Wei Zhang. 2005a. Computing cache vulnerability to transient errors and its implication. In *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)*. 427–435.

Wei Zhang. 2005b. Computing cache vulnerability to transient errors and its implication. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. 427–435.

Received May 2016; revised November 2016; accepted February 2017