Reducing Data Cache Susceptibility to Soft Errors

Hossein Asadi, Vilas Sridharan, Mehdi B. Tahoori, David Kaeli

Northeastern University, Dept. of ECE, Boston MA 02115

Email: {gasadi, vilas, mtahoori, kaeli}@ece.neu.edu

Abstract

Data caches are a fundamental component of most modern microprocessors. They provide for efficient read/write access to data memory. Errors occurring in the data cache can corrupt data values or state, and can easily propagate throughout the memory hierarchy. One of the main threats to data cache reliability is soft (transient, non-reproducible) errors. These errors can occur more often than hard (permanent) errors, and most often arise from Single Event Upsets (SEUs) caused by strikes from energetic particles such as neutrons and alpha particles.

Many protection techniques exist for data caches; the most common are ECC (Error Correcting Codes) and parity. These protection techniques detect all single bit errors and, in the case of ECC, correct them. To make proper design decisions about which protection technique to use, accurate design-time modeling of cache reliability is crucial. In addition, as caches increase in storage capacity, another important goal is to reduce the failure rate of a cache, to limit disruption to normal system operation.

In this paper, we present our modeling approach for assessing the impact of soft errors using architectural simulators. We also describe a new technique for reducing the vulnerability of data caches: *refetching*. By selectively refetching cache lines from the ECC-protected L2 cache, we can significantly reduce the vulnerability of the L1 data cache. We discuss and present results for two different algorithms that perform selective refetch. Experimental results show that we can obtain an 85% decrease in vulnerability when running the SPEC2K benchmark suite while only experiencing a slight decrease in performance. Our results demonstrate that selective refetch can cost-effectively decrease the error rate of an L1 data cache.

Index Terms

fault tolerance, reliability, soft errors, error modeling, cache memories, refresh, refetch

I. Introduction

Cache memory is a fundamental component used to enhance the performance of modern-day microprocessors. Data caches, typically implemented in static random access memory (SRAM) technology, store frequently accessed data close to the processor pipeline in order to reduce the latencies associated with accessing off-chip data memory. Data values are especially susceptible to errors, since data can be both read (e.g., as input to a computation) and written (e.g., as result of a computation). Errors occurring in the data cache can propagate to main memory, and can easily lead to data integrity issues [21].

The 2004 update of the International Technology Roadmap Survey (ITRS) predicts that we will see an exponential growth in integration levels in computer systems (memory, processors, and random logic) [14]. When this growth is coupled together with trends such as reductions in voltage levels and noise margins, we should expect to see an exponential growth in the vulnerability of computer systems to radiation-induced soft errors and noise-induced bit flips. Soft errors, which generally occur more frequently than hard (permanent) errors [18], arise from *Single Event Upsets* (SEU) caused by particle strikes.

Soft errors due to cosmic rays have already had an impact on the microelectronics industry. A number of commercial computer manufacturers have reported that cosmic rays have become a major cause of disruptions at customer sites [31] [6]. In reaction to these events, several major vendors have announced aggressive reliability and protection targets for current and future processors [13] [7]. Achieving these goals in a cost-effective way, however, is extremely difficult.

Previous studies have concluded that in current systems, unprotected memory elements are the most vulnerable system component to soft errors [10] [25]. Estimated soft error rate of typical designs such as microprocessors, network processors, and network storage controllers shows that sequential elements and unprotected SRAMs contribute to 49% and 40% of the overall soft error rate, respectively [27]. Moreover, it is observed that the nominal soft error rate of sequential elements increases with technology scaling [6]. These errors are both hard to reproduce and challenging to remedy [30].

In addition, to enhance microprocessor performance, computer designers have steadily increased the

Processor	DL1	IL1	L2
Pentium IV [12]	8K/16K	8K/16K	256K/512K/1M
POWER4 [16]	32K per core	64K per core	1.41M
POWER5 [17]	32K per core	64K per core	1.875M
AMD Athlon64 [2]	64K	64K	1M
Itanium2 [34]	16K	16K	256K
IBM 390 [37]	256K	256K	4M

TABLE I

SURVEY OF SOME CACHE ORGANIZATIONS IN RECENT MICROPROCESSORS.

size of on-chip caches. Since reliability of cache memory is projected to remain constant for the next several years (the positive impact of smaller bit areas will be offset by the negative impact of storing less charge per bit), the cache error rate will increase linearly with cache size [11] [19]. As can be seen in Table I, the typical size of first-level data and instruction caches (DL1 and IL1) and the typical size of second-level caches (L2) in current high-performance processors are on the order of 64KB and 1MB, respectively. In this paper, we consider only single-bit errors; most double-bit errors can be avoided by layout techniques such as interleaving and the error rate of multi-bit errors is extremely low.

Many commonly used protection techniques such as byte- and line-based *parity*, and *Single Error Correction-Double Error Detection Error Correcting Codes* (SEC-DED ECC) use spatial redundancy to protect memory elements [21]. These techniques all detect single-bit errors, and ECC corrects single-bit errors and detects double-bit errors. These techniques vary in their impact on processor design: for example, line-based parity consumes less area but introduces more performance overhead than byte-based parity. The higher the error rate, the more expensive protection needed; thus, many L1 caches have no protection or line parity, while L2 caches and main memory are almost always ECC protected [12] [16] [2].

While these error protection methods are well-understood, methods specific to reducing the number of failures in a cache have been less studied. We define a failure as an error that leads to a micro-architecturally visible result such as a parity mismatch. Decreasing the failure rate of caches is an increasingly important goal. As caches grow in storage capacity, a larger fraction of die area is occupied by cache arrays, resulting in a greater percentage of all processor errors coming from caches. Thus, to meet FIT goals, the cache failure rate must be reduced. In addition, if the failure rate can be reduced, less expensive error protection

techniques can be used (for instance, using line-based parity instead of byte-based parity or ECC in a large cache).

One suggested technique to enhance the reliability of cache memory is to reuse dead cache lines (lines that have not been accessed for a long time) to hold replicas of hot lines (those that are used frequently) [42]. This approach depends heavily on how often dead lines are available. Moreover, replicating can sometimes increase the cache miss rate, which will directly impact overall performance.

Another microarchitectural design choice that can reduce the vulnerability of level 1 data caches is to use a write-thru policy instead of a write-back policy. Experiments on SPEC2K benchmarks show that for a 64KB 4-way set associative DL1 cache, vulnerability to soft errors is reduced by more than 85% by using a write-thru cache [3]. This reduction in vulnerability is obtained because when using a write-back cache with allocate-on-write-miss, a single dirty word in a cache line causes the data and tag/status bits of the entire line to become vulnerable until the line is written back to memory. In a write-thru cache, this line is not vulnerable since the data is immediately written to the memory with a very short vulnerable time during the residency period in the store buffer. In our work, we model a write-thru L1 data cache.

In this paper, we present a technique that can be applied to a write-thru cache to signficantly decrease its vulnerability to soft errors. This technique selectively refetches cache lines from the next level in the memory hierarchy to *refresh* the cache line values, in effect reducing the residency time of these lines. In our work, we consider the reliability of the entire data cache, including tag and status bits. We introduce the concepts of critical words/time and present a complete soft error modeling methodology that captures the different types of soft errors that can occur in data caches (data errors, pseudo-hits, and multi-hits). A *Critical Word* (CW) is a word in a cache that is either read by the CPU or written back to lower memory. The *Critical Time* (CT) of this word is the time period during which the context of this CW is important.

We have built our modeling infrastructure upon the Simplescalar framework. Experiments on SPECint2K and SPECfp2K benchmarks show that our proposed techniques enhance the reliability of cache memory by up to 85%, while increasing power by less than 3.5% for the L1 data cache, and only introducing

minimal performance impact.

The rest of this paper is organized as follows: Section II describes the error rate and reliability background; Section III describes our reliability model and computation; Section IV describes our refetching algorithms. Section V presents our experimental setup. Section VI presents our reliability profiling results, while Section VII presents simulation results and discusses our findings, and Section VIII concludes the paper.

II. Background

When a particle strikes a sensitive region of an SRAM cell, the charge that accumulates can flip the value stored in the cell, resulting in a soft error. Soft errors are often classified as *Detected/Unrecoverable Errors* (DUE) or *Undetected Errors* (UE) (which are included in a more general class of errors called *Silent Data Corruption* (SDC)) [29]. The *Soft Error Rate* (SER) for a device is defined as the error rate due to SEUs.

A system's error budget for repairable components is commonly expressed in terms of the *Mean Time Between Failures* (MTBF) and the *Mean-Time-To-Repair* (MTTR), whereas for non-repairable components, *Mean-Time-To-Failure* (MTTF) is normally used. *Failures-in-Time* (FIT) is another commonly used error rate metric. FIT error rates are inversely proportional to MTBFs if the reliability function obeys the exponential failure law [15]. One FIT is equal to one failure in a billion hours; a 1-year MTBF equals 114,000 FIT. Current predictions show that typical FIT rates for latches and SRAM cells (measured at sea level) vary between 0.001-0.01 FIT/bit [11], [19], [31], but this amount at an elevation of 10km is 100x higher [44].

The overall FIT rate of a chip is calculated by adding the effective FIT rates of all the individual components. The FIT rate of each component is the product of its *raw FIT rate* and its associated *Vulnerability Factor*. The *Vulnerability Factor* (VF) is defined as the fraction of faults that become

errors [28]. Therefore, the FIT rate of the entire system can be computed as follows:

$$FIT_{Chip} = \sum_{i} raw\ FIT_{Element(i)} \times VF_{Element(i)} \quad (1)$$

The reliability of a chip during the period $[0, t]$ is defined as the probability that the chip operates correctly throughout this period [15]. The reliability of a chip at time t can be computed as follows:

$$Reliability_{Chip}(t) = e^{-FIT_{Chip} \times t} = e^{\frac{-t}{MTTF_{Chip}}} \quad (2)$$

III. Reliability Model and Computation

In this paper, we use the reliability/performance model presented in [3]. For data RAM, we define *Critical Words* (CWs) as those words in the cache that are either eventually consumed by the CPU or committed to memory by a write. In other words, if the CPU reads a word from the cache or the word is written to the memory, it is a CW. The *Critical Time* (CT) associated with a CW is defined as the time period in which the context of that CW is important. If an error in a CW is encountered during its critical time, this will result in an erroneous value being propagated. All words that are not CWs are called *Non-Critical Words* (NCWs); any failure to NCWs will not affect the correct operation of the system.

For tag addresses, we define four types of errors:

Pseudo-miss: The tag address of a line incorrectly does not match the requested address tag.

Pseudo-hit: The tag address of a line erroneously matches the requested address tag.

Replacement error: The tag address of a line is changed after the the line has been modified but before it is written to lower memory.

Multi-hit: The tag address of a line is changed to match another tag address in the same cache set.

Only pseudo-hits, replacement errors, and multi-hits cause data corruption. In write-thru caches, which we consider in this paper, the replacement error is zero by definition.

Status bits typically consist of a valid bit and, in a write-back caches, a dirty bit. An error in the valid bit will corrupt data only if the line is re-referenced, or, in a write-back cache, if the dirty bit is updated.

Similarly, an error in the dirty bit (if present) will corrupt data only on a 1 to 0 flip, when new data will be lost.

Since we model a write-thru L1 cache, there is no dirty bit, and the status vulnerability is a tiny fraction of the total cache vulnerability. We approximate it as:

$$Vulnerability_{Status} = (CacheBlocks) \times (PercentageofInvalidBlocks) \tag{3}$$

In a write-thru cache, this equation always over-estimates the vulnerability contribution of the valid bit.

Note that we model all errors in both data RAM and tag addresses, but we only approximate errors in status bits. As our results will show, the vulnerability of status bits is less than 0.2% of the total cache vulnerability. Therefore, our estimation method is 99.8% accurate.

Since our methodology is focused on computing the reliability of the cache and not the full system, we count as vulnerable any word that propagates from the cache to another part of the system (either the processor or lower memory). Therefore, results that are never re-referenced (results of dynamically dead instructions [28]) contribute to the vulnerability only if they are written to lower memory. Results which are referenced by dynamically-dead instructions will contribute to the vulnerability. The vulnerability of silent stores [24] is computed properly: assuming no intervening read, the first store will not contribute to the cache vulnerability.

Since the reliability of the cache system only depends on the correctness of the CW words, the vulnerability factor of a memory system can be defined as [28]:

$$VF_{system} = \frac{\sum residency\ time\ of\ all\ critical\ words}{Total\ Execution\ Time \times M} \qquad (M = system\ size\ in\ words) \tag{4}$$

If a CT is assigned to every CW, and we assume the raw FIT rates for all words are the same, then the *vulnerability* of the cache can be computed as:

$$Vulnerability_{Cache} = \frac{B_{pw} \times \sum_{i=1}^{N} CT_i}{TT} \tag{5}$$

The effective FIT rate for the cache can then be computed as:

$$FIT_{Cache} = raw\ FIT\ per\ bit \times Vulnerability_{Cache} \tag{6}$$

Note that $Vulnerability_{Cache}$ is a dimensionless quantity. As CTs increase, the vulnerability of the cache system increases as well. In other words, the longer that critical data stays in the cache, the greater the probability that an error in the cache will be propagated to the outside.

IV. ALGORITHMS

```
1  Algorithm:Vulnerability computation for data.
2  w_i: word
3  ws: word size in bits
4  begin
5      if FILL then
6          for each w_i in block do
7              AccessTime[w_i] = now
8          end
9      end
10     if WRITE HIT then
11         AccessTime[w_i] = now
12     end
13     if READ HIT then
14         CacheDataVulTime += (now - AccessTime[w_i]) * ws
15         AccessTime[w_i] = now
16     end
17     if REPLACE OR FLUSH b_i (WriteBack) then
18         if b_i is dirty then
19             for each w_i in block do
20                 CacheDataVulTime += (now - AccessTime[w_i]) * ws
21             end
22         end
23     end
24     if REPLACE OR FLUSH (WriteThru) then
25         No action needed
26     end
27     if End of simulation then
28         Flush L2 cache
29         CacheDataVul = CacheDataVulTime / TotalSimulationCycle
30     end
31 end
   algocf
```

Algorithm 1: Vulnerability computation for data RAM.

In this section, we present a methodology to compute the vulnerability of data, tag-addresses, and status

bits. Algorithm 1 shows how we compute the vulnerability of data RAMs. We associate one variable ($AccessTime$) with each word in the cache. As shown in lines 5-12, $AccessTime$ is reset on a FILL or WRITE HIT event. On a READ HIT (lines 13-16), we first update the total vulnerable time and then reset $AccessTime$.

Note that we multiply the length of the vulnerable interval by the word size in bits (ws). For a line REPLACEMENT or FLUSH in a writeback cache (lines 17-23), we update the cache vulnerable time only if the line is dirty. No action is needed when we do a REPLACEMENT or FLUSH for a writethru cache since there is no dirty bit in the cache.

Generally, the execution of most programs takes a very long time. To overcome this difficulty, researchers simulate just a portion of the program which captures a representative sample of the whole program [32]. In our simulations, we utilize *SimPoint,Perelman03*, only executing a limited number of program instructions rather than the whole program execution (see Table III). However, to account for the vulnerability of those dirty lines that are still in the cache, we flush those lines at the end of simulations (line 28). Finally, we divide the computed data vulnerability by the number of simulated cycles according to equation 5 (line 29).

The second algorithm (Algorithm 2) shows how we compute the vulnerability of tag-addresses and status bits. As we described before in Section II, tag vulnerability consists of pseudo-hit vulnerability, multi-hit vulnerability, and replacement vulnerability. We associate three factors with each block to compute these vulnerabilities ($BlockFillTime$, $BlockMultihitTime$, and $BlockWriteTime$).

To compute the pseudo-hit vulnerability, we compare the tag of the requested address (tag_r) to all tag addresses inside the target set (tag_{b_i}) on a miss (lines 11-17). If tag_r and tag_{b_i} differ by one bit, tag_{b_i} was vulnerable between the time tag_{b_i} was brought into the cache and the time tag_r was requested ($now - BlockFillTime[b_i]$). $BlockFillTime$ keeps the fill time of the block (line 8).

To compute the multi-hit vulnerability on a hit, we compare the requested tag (tag_r) to all tag addresses inside the target set (tag_{b_i}) excluding the target block (lines 18-25). If there is exactly one bit difference

```
1  Algorithm:Vulnerability computation for tag addresses and status bits.
2  w_i: word
3  b_i: block
4  r: requested block
5  begin
6      Update BlockWriteTime[r] on the first WRITE HIT to the block
7      if FILL then
8          BlockFillTime[r] = now
9          BlockMultihitTime[r] = now
10     end
11     if MISS then
12         for each block b_i in target set do
13             if tag_{b_i} and tag_r differ by one bit then
14                 CachePseudoVulTime += (now - BlockFillTime[b_i])
15             end
16         end
17     end
18     if HIT then
19         for each block b_i in target set except r do
20             if tag_{b_i} and tag_r differ by one bit then
21                 CacheMultiHitVulTime += (now - BlockMultihitTime[b_i])
22                 BlockMultihitTime[b_i] = now
23             end
24         end
25     end
26     if REPLACE OR FLUSH r (WriteBack) then
27         if r is dirty then
28             CacheReplaceVulTime += (now - BlockWriteTime[r])×(tag size)
29             CacheStatusVulTime += (now - BlockWriteTime[r])×(status size)
30         end
31     end
32     if End of simulation then
33         Flush L2 cache
34         CachePseudoVul = CachePseudoVulTime / TotalSimulationCycle
35         CacheMultiHitVul = CacheMultiHitVulTime / TotalSimulationCycle
36         CacheReplaceVul = CacheReplaceVulTime / TotalSimulationCycle
37         CacheStatusVul = CacheStatusVulTime / TotalSimulationCycle
38     end
39 end
   algocf
```

Algorithm 2: Vulnerability computation for tag addresses and status bits.

between tag_r and tag_{b_i}, tag_{b_i} was vulnerable between the time tag_{b_i} was fetched into the cache and the time tag_r was requested ($now - BlockMultihitTime[b_i]$). To avoid double counting of the multi-hit vulnerability, we associate $BlockMultihitTime[b_i]$ with each block. Initially, we set $BlockMultihitTime[b_i]$ to the fill time (line 9). Thereafter, during every time interval we update the multi-hit vulnerability, and we reset $BlockMultihitTime[b_i]$ to the current cycle (now) (line 22).

To compute the replacement vulnerability, we associate one variable with each block ($BlockWriteTime$).

Configuration Parameter	Value
Processor	
Functional Units	4 integer ALUs, 1 integer multiplier/divider 4 FP ALUs, 1 FP multiplier/divider
LSQ Size / RUU Size	8 Instructions / 16 Instructions
Fetch / Decode / Issue / Commit Width	4 / 4 / 4 / 4 instructions/cycle
Fetch Queue Size	4 instructions
Cycle Time	1 ns
Cache and Memory Hierarchy	
L1 Instruction Cache (IL1)	64KB, 1-way, 64 byte lines 1 cycle latency, 18Kbit tag array
L1 Data Cache (DL1)	64KB, 4-way, 64 byte lines Write-thru, no allocate on write miss 1 cycle latency, 20Kbit tag array
L2	1MB unified, 8-way 128 byte lines, 6 cycle latency
Memory	100 cycle latency
Branch Logic	
Predictor	Combined, bimodal 2KB table two-level 1KB table, 8 bit history
BTB	512 entry, 4-way
Mis-prediction Penalty	3 cycles

TABLE II

DEFAULT CONFIGURATION PARAMETERS USED IN OUR SIMULATIONS.

Benchmarks	SimPoint	Benchmarks	SimPoint
art-110	33,500 M	wupwise	58,500 M
bzip2-source	59,300 M	swim	600 M
gcc-166	10,000 M	mgrid	700 M
gzip-source	31,700 M	applu	1,900 M
mcf	97,800 M	galgel	315,100 M
mesa	9,000 M	equake	19,500 M
vpr-place	6,800 M	ammp	213,100 M
crafty	100 M	lucas	3,600 M
parser	1,700 M	fma3d	29,900 M
twolf	3,200 M	apsi	4,700 M

TABLE III

SPEC2K BENCHMARKS USED IN OUR WORK.

This variable is set on the first write to the block. On a replacement or flush of block r, we calculate the total time the block was dirty, and add that to the replacement vulnerability ($now - BlockWriteTime[r]$) (see lines 26-31). To compute the status-bit vulnerability, we compute the interval time that replacement vulnerability matters (line 29). As described earlier, to account for the vulnerability of the dirty lines that are still in the cache, we flush those lines at the end of the simulations (line 33). Finally, we divide the computed vulnerabilities by the number of simulated cycles ($TotalSimulationCycle$) according to equation 5 (lines 34-37).

V. Experimental Setup

For these experiments, we used the *sim-outorder* processor simulator provided in *SimpleScalar 3.0* [8]. To evaluate the reliability of DL1 caches, we have extended the SimpleScalar source code to integrate our reliability estimation method. We have also incorporated our refetch algorithm in the SimpleScalar tool set.

Our evaluation uses the SPECint2K and SPECfp2K benchmark suite [38] compiled for the Alpha ISA [20]. We utilize the SimPoint early simulation points in all of our results [32], as shown in Table III. We chose *Instructions Per Cycle* (IPC) as our metric to measure the performance impact of our methods against the baseline results. The default system parameters (cache size, associativity, etc.) are detailed in Table II, and were chosen to be representative of modern state-of-the-art processors. Since most modern microprocessors utilize ECC protection on their L2 caches and main memory, we assume that these lower levels of memory are free of single-bit errors.

VI. Reliability Profiling Results

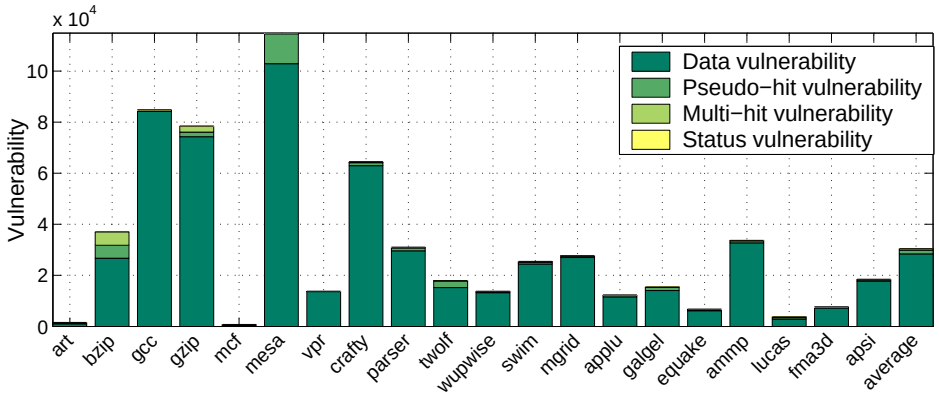

Fig. 1. Vulnerability of DL1 cache

Figure 1 shows the L1 data cache vulnerability for twenty SPEC2K benchmark programs. From this data, one can determine the likelihood of having no failures (*reliability*) based on the target workload, duration, and raw environmental FIT rate.

For example, assume that the target application is *mcf* ($vulnerability_{mcf} = 750$), the execution duration

is 6 months (4320 hours), and the raw error rate is 0.002 FIT/bit. The reliability of the DL1 cache can be calculated according to expressions 6 and 2 as follows:

$FIT_{DL1} = 0.002 FIT_{bit} \times 750 = 1.5 FIT$

$Reliability_{DL1} = e^{-4320 \times 1.5 \times 10^{-9}} = 0.99999935$

Now consider running *mesa* ($vulnerability_{mesa} = 114,920$) in a noisy environment (0.05 FIT/bit) for 2 years (17,520 hours). The reliability of the DL1 cache can be computed as follows:

$FIT_{DL1} = 0.05 FIT_{bit} \times 114,920 = 5,746 FIT$

$Reliability_{DL1} = e^{-17,520 \times 5,746 \times 10^{-9}} = 0.904$

When running *mesa* for 2 years in a noisy environment, the probability of a failure is almost 10%. This level of reliability is not acceptable for critical applications, and should drive a decision to implement some form of error detection or prevention that will reduce the vunerability.

Another interesting result shown in Figure 1 is that the reliability of the DL1 is highly application-dependent. For example, the vulnerability of the DL1 when running *mesa* or *gcc* is at least 10 times greater than the vulnerability for *mcf* or *art*. Finally, as the results show, data vulnerability constitutes more than 93% of the total vulnerability in the DL1 cache.

VII. Analysis and Results

Next, we present some observations on vulnerability and memory access patterns in the SPEC2000 benchmark suite which led to the development of our refetching algorithms.

The major motivation for developing our refetch technique was the observed bus utilization of the L2 data bus for the SPEC benchmarks, as shown in Figure 2. The L2 bus utilization is defined as:

$$BusUtil_{L2} = \frac{LoadMisses_{IL1} + LoadMisses_{DL1} + Writes_{DL1}}{Total\ Execution\ Cycles} \tag{7}$$

On average, the L2 bus was idle more than 85% of the time. These spare cycles can be used to refetch lines from the L2 cache and reload them into L1, thereby reducing their vulnerability.

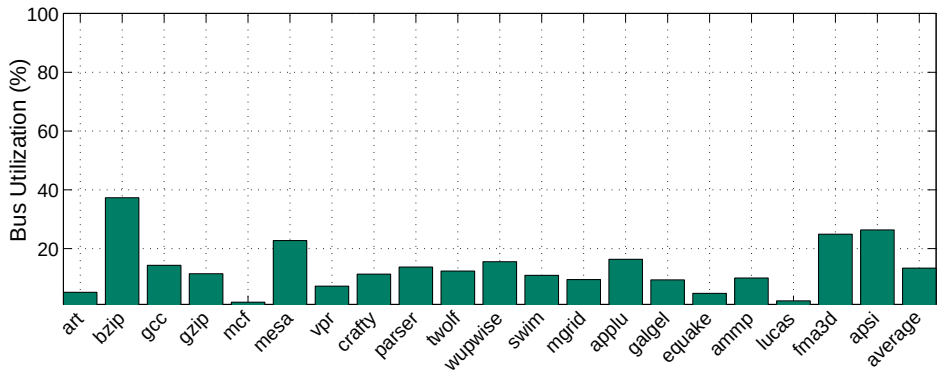

Fig. 2. Bus utilization for SPEC2K benchmarks.

Next, we investigate and present results for two different refetching schemes. Our *static* method refetches a selected set of cache blocks at a pre-defined frequency. Our *event-driven* method uses L1 read hits as a trigger to initiate refetches to selected cache lines that would decrease vulnerability the most.

A. Static Method

The static method selects cache blocks in order, issuing refetches to each subsequent cache block at after a pre-defined interval (similar to a DRAM refresh controller). Using this method, we refetch all cache blocks, but at different frequencies based on their relative position in the *Most Recently Used* (MRU) stack. This has been shown to work well for timely prefetching in a data cache [1], but also serves as an effective method of determining vulnerability contribution. As an added benefit, the MRU predictor is already implemented in many caches that use *Least Recently Used* (LRU) for block replacements. As can be seen from Figure 3, blocks residing at the top of the MRU stack (the most recently used blocks) account for more than 78% of the total cache data vulnerability, while LRU blocks account for only 6% of the total cache data vulnerability. However, since accesses to the LRU blocks are rare, the vulnerability per access of these blocks is high. Thus, frequent refetches to MRU blocks can be coupled with infrequent refetches to the LRU blocks to significantly decrease vulnerability.

We next examine the problem of selecting an optimal refetch frequency. An analysis of the *inter-reference gap* (IRG), as defined in [33], indicates that the vast majority of re-references to blocks at the top of the MRU stack occur within 200 cycles, as shown in Figures 4 and 5. Note that unlike the approach

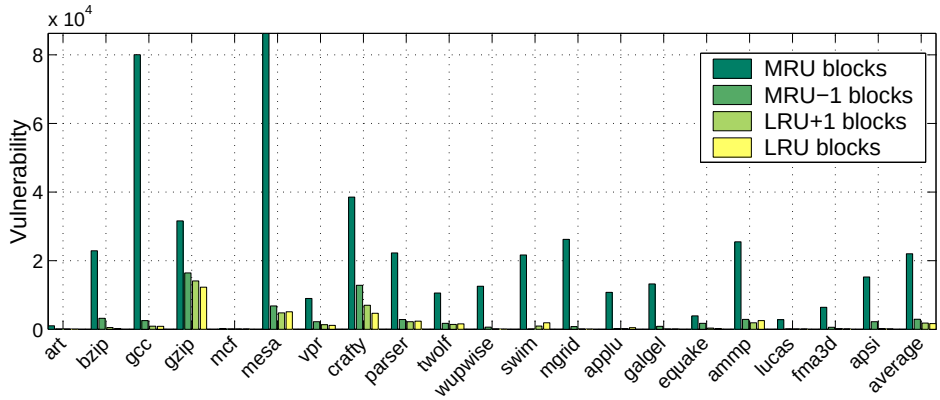

Fig. 3. Vulnerability vs. MRU slot

Benchmark	Original IRG	Adjusted IRG	Benchmark	Original IRG	Adjusted IRG
art	781	1275	wupwise	1014	3126
bzip2	363	3495	swim	4104	7369
gcc	4259	6414	mgrid	1721	2539
gzip	1974	1088	applu	1408	2928
mcf	380	2034	galgel	1002	2171
mesa	1243	3661	equake	751	2973
vpr	1523	3648	ammp	1965	6698
crafty	1447	3764	lucas	444	1082
parser	1662	8249	fma3d	482	1198
twolf	1024	3738	apsi	1068	1700

Original IRG average	1431	Adjusted IRG average	3958

TABLE IV
ORIGINAL MEAN GAP VERSUS ADJUSTED MEAN GAP (IGNORING IRG VALUES < 200).

in [33], we measure the inter-reference gap in units of clock cycles between consecutive accesses to the same word. The mean inter-reference gap for blocks at the top of the MRU stack is shown in columns 2 and 5 of Table IV. In order to have an impact on vulnerability, each line would get at least one refetch between adjacent accesses; this means refetching every MRU slot in the cache in fewer than "mean gap" cycles on average.

Unfortunately the mean gap is too small to make this a practical proposition for most of the benchmarks, due to the large proportion of small inter-reference gaps. If we examine the contribution of these small inter-reference gaps to the overall vulnerability, however, we notice that the vulnerability contribution of blocks with an IRG less than 200 is quite small. As shown in Figure 6, accesses with IRG < 200 contribute less than 3% of the total MRU vulnerability. Ignoring these accesses and re-computing the mean IRG yields the data shown in columns 3 and 6 of Table IV. Note the almost 3 times increase in the

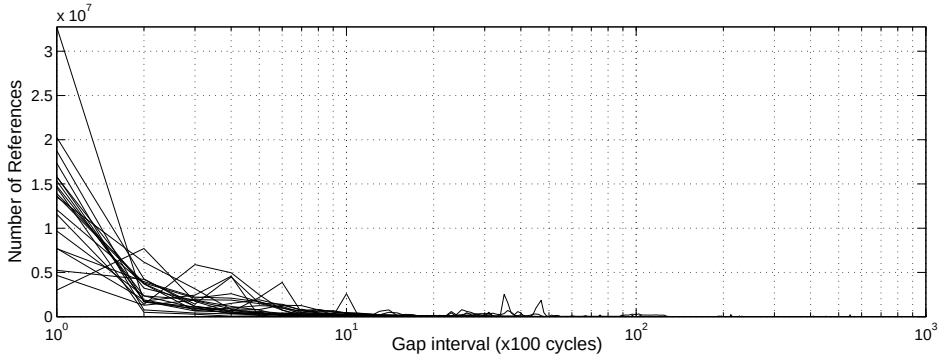

Fig. 4. IRG vs. number of references - per benchmark.

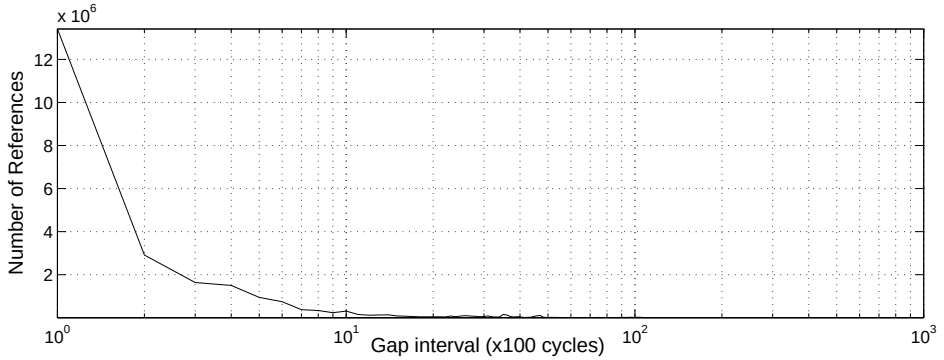

Fig. 5. IRG vs. number of references - SPEC average.

mean IRG value from the original. Due to this dramatic difference between the non-adjusted and adjusted mean IRG and the relatively small vulnerability contribution of the blocks with small IRGs, using the adjusted mean to set our refresh frequency greatly reduces the IPC impact of refetching while maintaining its effectiveness.

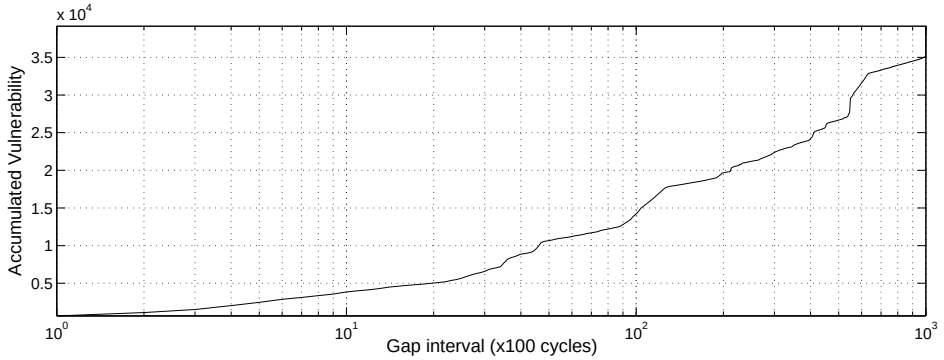

Fig. 6. Vulnerability (cumulative) vs. IRG - SPEC average.

The static approach refetches all cache blocks, but at different distances based on their position in the MRU stack. Table V shows the refetch period used for each position of the MRU stack. These represent

MRU Position	Refetch period (cycles)
MRU	3958
MRU-1	7969
LRU+1	20346
LRU	35853

TABLE V
REFETCH PERIOD PER MRU POSITION

the mean (adjusted) IRGs over all the SPEC benchmarks, and the same refetch frequency is used for all benchmarks.

B. Event-Driven Method

Unlike the static method, the event-driven method does not issue refetches at a pre-defined frequency. It instead uses L1 read hits as a trigger to issue refetches to selected cache blocks. The method uses the address of the current access to decide which cache blocks to refetch. Using this workload-specific information has the benefit of taking into account workload access patterns for which the static method cannot obtain. The problem then becomes determining which blocks to refetch. To provide the most benefit, refetches should be issued in a way to minimize critical time. This implies that refetches should be issued to blocks likely to be accessed in the near future.

To find these blocks, we examine the address stream of the SPEC benchmarks. In the following analysis, on each access we record the next 40 accessed addresses and map these addresses to cache blocks. These results are shown in Figure 7. The x-axis is the distance (in cache blocks) of the subsequent accesses (e.g., 0 indicates the same cache block, +1 indicates the next cache block), and the y-axis is the number of accesses to that block.

As expected, the reference patterns clearly illustrate the principles of temporal and spatial locality; on average, 46% of the accesses were to an address in the same cache block as the current address, while 20% were to an address that mapped to the next cache block (ie, the subsequent cache index but with the same cache tag). Therefore, on a hit, we choose to refetch the “current” block (the block containing the current address) and the “next” block (the next encounterd cache block when linearly traversing the

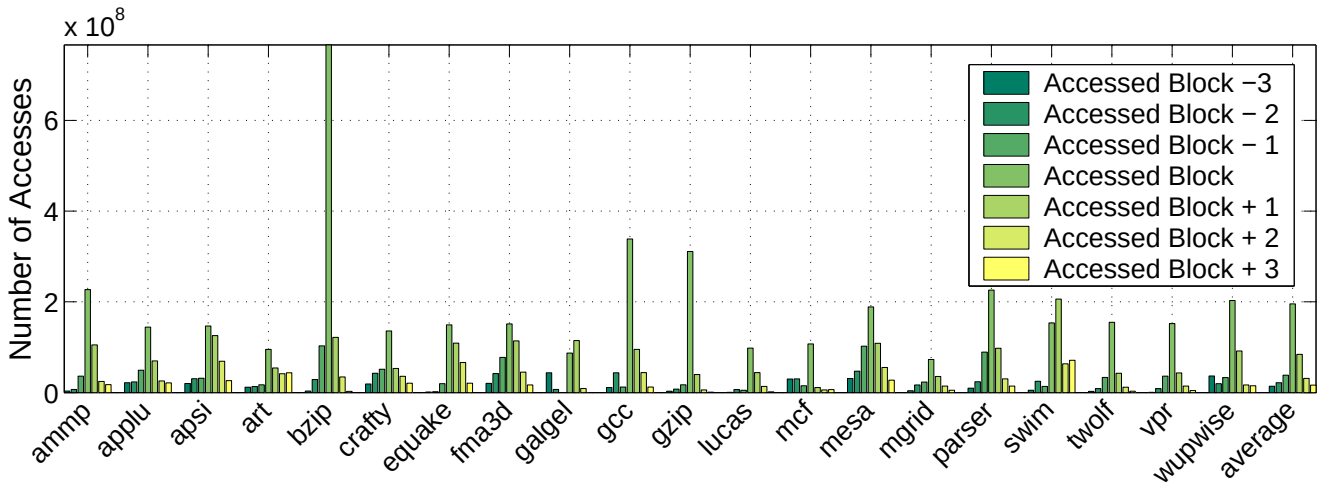

Fig. 7. Number of Accesses vs. Distance in cache blocks.

address space). Note that we only issue a refetch to a block if the block is already present in cache.

Unfortunately, this scheme by itself would result in too many refetches to be practical. However, we can reduce this in practice if we only issue refetches to blocks that do not already have a refetch in process. Thus, loads to addresses within the same cache block on nearby clock cycles would cause only a single refetch to be issued. In our experiments, this optimization alone reduced the number of potential refetches by 57%, more than halving the bus utilization due to refetches.

We also extended this optimization to track recently-completed refetches in addition to in-process refetches. We then squash a refetch if a refetch to the block has completed within the past K cycles. Setting K to 0 cycles yields the Event-Driven method described above, which we term ED-0. Increasing K yields a performance-reliability tradeoff: larger values for K issue fewer refetches, decreasing the impact on performance but also increasing the vulnerability. In our results, we examine three different values for K: 0 cycles (ED-0); 100 cycles (ED-100); and 200 cycles (ED-200). As stated, ED-0 reduces the number of refetches by 57% due to the cancelling of in-process refetches, while ED-100 reduces the number of refetches by 84% and ED-200 by 89.5%.

Clearly, ED-100 and ED-200 dramatically lower the bus utilization of refetching compared to ED-0. In the previous section, we noted that blocks with an IRG of less than 200 cycles were accessed often but contributed at most 3% of the overall vulnerability. Since these are exactly the blocks not refetched

by ED-100 and ED-200, we expect the increase in vulnerability of these methods over ED-0 to be slight despite the lower number of refetches.

C. Results

Figure 8 shows the vulnerability of the five cases we tested: the baseline (no refetch), the static method, the ED-0 method, the ED-100 method, and the ED-200 method. On average, the static method decreased the vulnerability by 71%. Note that not only did static refetching decrease the average vulnerability by 71%, but the vulnerability of mesa (the most vulnerable application) decreased by almost 95%. The ED-0 method performs even better, reducing average vulnerability by 85%. As expected, ED-100 and ED-200 perform almost as well as ED-0, showing average decreases in vulnerability of 84.5% and 84% respectively. A cache access in ED-200 cancels all refetches to that block during the next $(200 + Lat_{L2})$ cycles. Since accesses during this time only contribute slightly to the overall vulnerability, the reduction in effectiveness of ED-200 is slight.

Figure 9 shows the IPC impact of these selective refetching techniques. The reduction in IPC of the static refetch mechanism is 0.7%, and is no more than 3.5% in the worst case. The ED-0 method, by contrast, reduces IPC by 4% on average, while the ED-100 and ED-200 methods, which issue fewer refetches, show less of an impact on IPC: approximately 3%.

D. Power

One of the potential problems of reliability techniques is their associated power consumption. The importance of power has been increasing rapidly over the past few years [5], [9], [40], [41].

In our experiments, to compute the power overhead of our refetching techniques, we use the power consumption parameters given in [43] (technology=0.07 micron, supply voltage=1.0 V). As shown in Figure 10, the static approach adds less than 2.5% extra power dissipation (including both dynamic and leakage power) to the L1 data cache, while ED-0, ED-100, ED-200, on average, add less than 7%, 2.5%, 1.5% extra power dissipation, respectively.

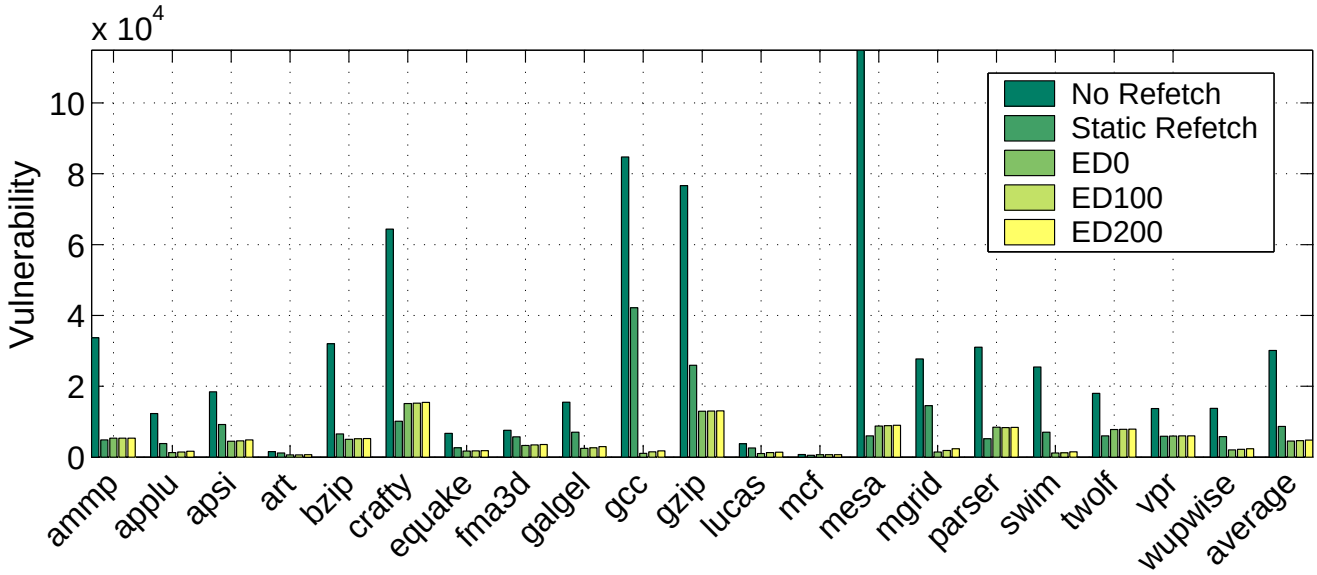

Fig. 8. Data Cache Vulnerability

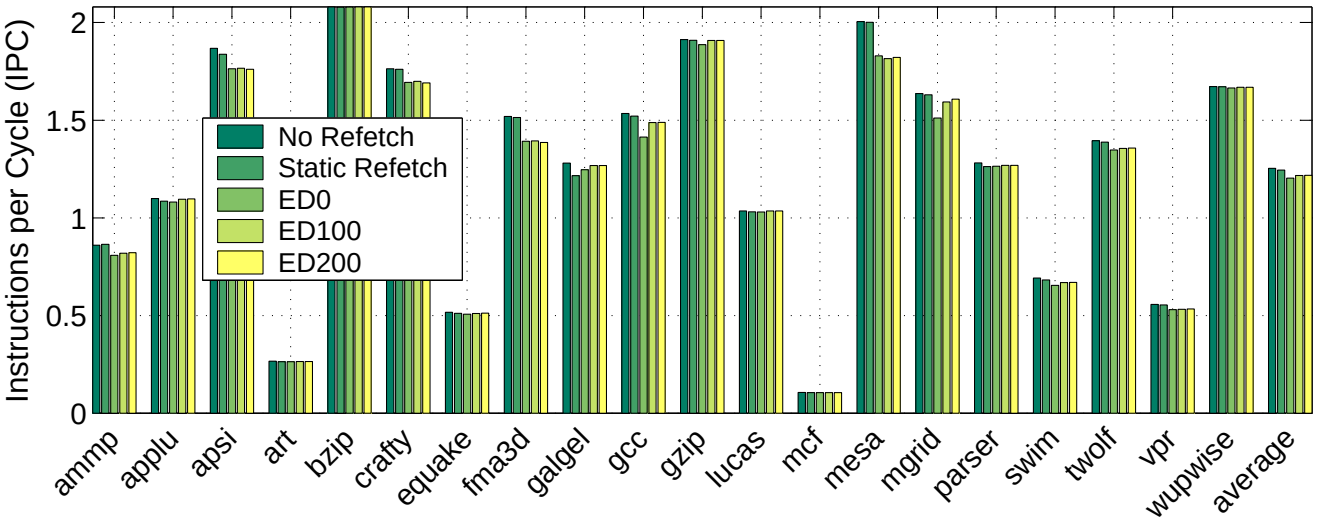

Fig. 9. IPC

We expect the increase in L2 power dissipation to be quite small as well, since dynamic power is dominated by leakage power, as was reported in [23]. The reason for the low power overhead associated with refetching is that it does not use information redundancy; the only extra power overhead comes from the additional accesses to the L1 and L2 caches.

E. Discussion

The high vulnerability reduction, low performance impact, and relatively low power consumption of selective refetch lend themselves to an interesting application: using selective refetch in conjunction with parity on a cache instead of using ECC. For instance, line parity provides single-bit error detection, but offers no correction. In some situations ECC, which provides detection and correction for 100% of single-bit errors, may be infeasible due to the large area/power impact [4]. Instead, using line-parity (1 to 15% area impact [4]) in combination with selective refetch provides 100% detection of single-bit errors and correction for 85% of these errors. In applications where die area or power are at a premium, this may be an acceptable tradeoff. In addition, unlike SEC-DED ECC, selective refetch will reduce the failure rate from double- and multi-bit errors in addition to single-bit errors. Thus, a cache using parity and refetch may not require special layout to avoid these types of errors. Lastly, selective refetch corrects errors before they cause any visible system events (such as an ECC or parity error reported to the operating system). Thus, it can also reduce the DUE (Detected Unrecoverable Error) [29] rate of the cache, reducing the impact of the cache on the DUE budget for the processor.

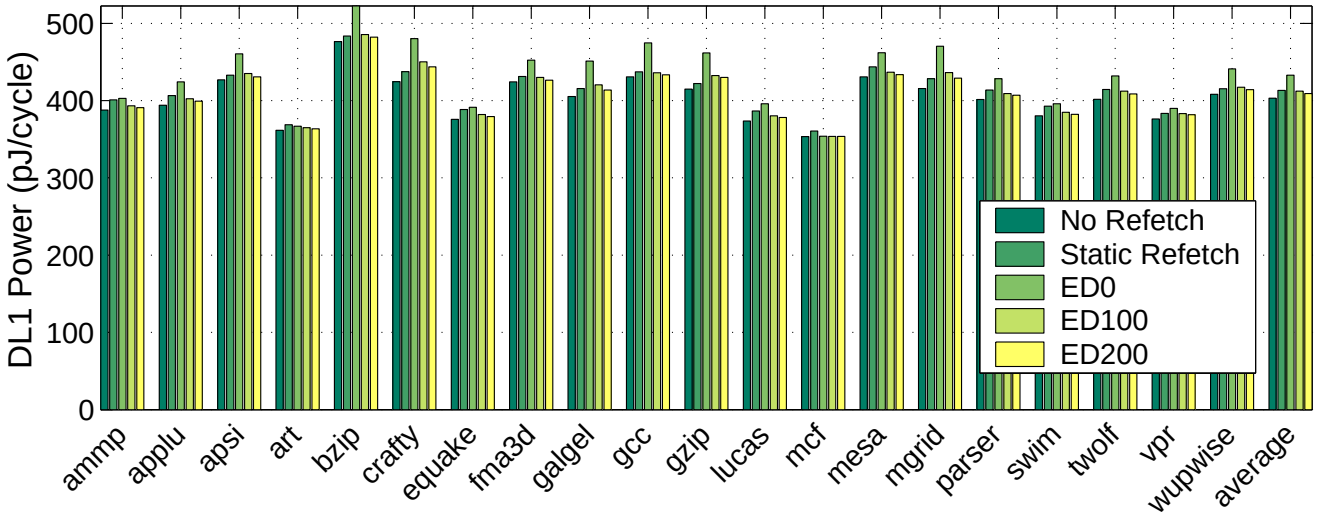

Fig. 10. Power overhead

VIII. Conclusion and Further Work

This paper has presented a new method for calculating cache vulnerability, along with a complete modeling methodology that captures the different types of possible soft errors that can occur in data caches. We introduced concepts such as Critical Time and Critical Words, and used these as the basis for suggesting two novel methods for selective cache line refetch to increase cache reliability. These method reduce the vulnerability of an L1 data cache by up to 85%, while having a very small impact on performance. Although a real system may see a slightly greater impact on IPC, we believe that the small performance penalty makes refetching a promising area of study to decrease vulnerability.

In the future, investigating a range of refetch algorithms could certainly provide further reliability improvements. In addition, developing a controller that assigns refetches the lowest priority on the bus would likely limit the IPC impact even more.

We will also look for more algorithms to select which blocks are most useful to refetch. An improved algorithm could result in even larger reliability improvements with even smaller performance impacts. In addition to dynamic hardware prediction, we also plan to explore some compiler-based prediction schemes based on data flow.

Lastly, our work so far has only focused on the L1 data caches. For the benchmarks we examined, the L1 instruction cache miss rate is relatively low, thus we believe issuing refetches to the instruction cache could also be of benefit in reducing the overall system vulnerability. Refetching may also be useful for smaller structures (such as TLBs) where ECC would be too costly in either area or access time. We believe our results show the significant benefits of selective refetch as a method to reduce cache vulnerability, and highlight its potential application towards other processor structures.

References

[1] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, “Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite,” Proc. of the 42nd ACM Southeast Regional Conference, 2004.

[2] AMD Athlon(TM) 64 Processor, http://www.amd.com.

[3] G. Asadi, V. Sridharan, M. B. Tahoori, and David Kaeli, “Balancing Performance and Reliability in the Memory Hierarchy,” Proc. of the IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS05), pp. 269-279, March 2005.

[4] G. Asadi, V. Sridharan, M. B. Tahoori, and David Kaeli, “Reliability Tradeoffs in Design of Cache Memories,” Proc. of the First Workshop on Architectural Reliability (WAR-1), in conjunction with the 38th International Symposium on Microarchitecture (MICRO-38), November 2005.

[5] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, “Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures,” Proc. of the 33^{rd} Intl. Symp. on Microarchitecture, pp. 245-257, Dec. 2000.

[6] R. Baumann, “Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends,” Proc. of IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals, pp. 121.01.1–121.01.14, April 2002.

[7] D. Bossen, Workshop Talk, International Reliability Physics Symposium (IRPS), 2002.

[8] D. Burger and T. M. Austin, “The SimpleScalar Tool Set, Version 2.0,” University of Wisconsin-Madison, Computer Science Dept., Technical Report No. 1342, June 1997.

[9] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M.L. Scott, “Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power,” Proc. of the 11^{th} Intl. Conference on Parallel Architectures and Compilation Techniques, pp. 141-152, Sep. 2000.

[10] J. Gaisler, “Evaluation of a 32-bit Microprocessor with Built-in Concurrent Error-Detection,” Proc. of 27^{th} Intl. Symp. on Fault-Tolerant Computing(FTCS-27), pp. 42-46, June 1997.

[11] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walstra, and C. Dai, “Impact of CMOS Scaling and SOI on soft error rates of logic processes,” Symposium on VLSI Technology, Digest of Technical Papers, PP. 73-74, June 2001.

[12] Intel Pentium IV Processor, http://www.intel.com.

[13] FUJITSU Corporation, International Solid-State Circuits Conference (ISSC), 2003.

[14] International Technology Roadmap for Semiconductors, http://www.itrs.net/, 2004.

[15] B. W. Johnson, “Design & analysis of fault tolerant digital systems,” A&W Longman Publishing, ISBN:0-201-07570-9, Boston, MA, 1988.

[16] S. Behling, R. Bell, P. Farrell, H. Holthoff, F.O. Connell, and W. Weir, “The POWER4 Processor Introduction and Tuning Guide,” IBM redbooks, www.redbooks.ibm.com/pubs/pdfs/redbooks/sg247041.pdf, Nov. 2001.

[17] R. Kalla, S. Balaram, J.M Tendler, “IBM Power5 Chip: a Dual-Core Multithreaded Processor,” IEEE Micro, pp. 40-47, Vol. 24 , Issue 2, Mar-Apr 2004.

[18] J. Karlsson, P. Ledan, P. Dahlgren, and R. Johansson, “Using Heavy-Ion Radiation to Validate Fault Handling Mechanisms,” IEEE Micro, 14(1), pp. 8-23, Feb. 1994.

[19] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar, “Scaling Trends of Cosmic Rays Induced Soft Errors in Static Latches Beyond 0.18μ,” Symposium on VLSI Circuits, Digest of Technical Papers, pp. 61-62, June 2001.

[20] R. Kessler, “The Alpha 21264 Microprocessor,” IEEE Micro, 19(2):24–36, March 1999

[21] S. Kim and A. K. Somani, “Area Efficient Architectures for Information Integrity in Cache Memories,” Proc. of the Intl. Symp. on Computer Architecture (ISCA’99), pp. 246-255, Atlanta, Georgia, May 1999.

[22] S. Kim and A. K. Somani, “Soft Error Sensitivity Characterization for Microprocessor Dependability Enhancement Strategy,” Proc. of the Intl. Conference on Dependable Systems and Networks (DSN), pp. 416-425, June 2002.

[23] N. S. Kim, D. Blaauw, and T. Mudge, “Leakage Power Optimization Techniques for Ultra Deep Sub-Micron Multi-Level Caches,” Proc. of the Intl. Conference on Computer-Aided Design, pp. 627-632, San Jose, California, November 2003.

[24] K. M. Lepak and M. H. Lipasti, “Silent Stores for Free,” Proc. of the Intl. Symp. on Microarchitecture (MICRO-33), pp. 22-31, Dec. 2000.

[25] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson, “On Latching Probability of Particle Induced Transients in Combinational Networks,” Proc. of the 24^{th} Symp. on Fault-Tolerant Computing (FTCS-24), pp. 340-349, June 1994.

[26] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir and M. J. Irwin, “Soft Error and Energy Consumption Interactions: A Data Cache Perspective”, in the Proc. of Intl. Symp. for Low Power Electronics and Design, Newport May 2004.

[27] S. Mitra, N. Seifert, M. Zhang, Q. Shi and K. Kim, ”Robust System Design with Built-In Soft-Error Resilience,” IEEE Computer, vol. 38, pp. 43-52, Feb.2005.

[28] 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,” Proc. of the Intl. Symp. on Micro-architecture (MICRO-36), pp. 29-40, 2003.

[29] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, “Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor” Proc. of the Intl. Symp. on Computer Architecture (ISCA’04), pp. 264-275, June 2004.

[30] H. T. Nguyen and Y. Yagil, “A Systematic Approach to SER Estimation and Solutions,” Proceedings of the Intl. Reliability Physical Symp., pp. 60-70, Dallas, Texas, 2003.

[31] E. Normand, “Single Event Upset at Ground Level,”IEEE Trans. on Nuclear Science, Vol. 43, No. 6, pp. 2742-2750, Dec. 1996.

[32] E. Perelman, G. Hamerly, and B. Calder “Picking Statistically Valid and Early Simulation Points,” Proc. of the Intl.

Conference on Parallel Architectures and Compilation Techniques, September 2003.

[33] V. Phalke and B. Gopinath, “An Inter-Reference Gap Model for Temporal Locality in Program Behavior,” Proc. of the 1995 ACM SIGMETRICS Joint Intl. Conf. on Measurement and Modeling of Computer Systems, Ottowa, May 1995.

[34] S. Rusu, H. Muljono, and B. Cherkauer, “Itanium 2 processor 6M: higher frequency and larger L3 cache,” IEEE Micro, pp. 10-18, Vol. 24, Issue 2 , Mar-Apr 2004.

[35] A. M. Saleh, J. J. Serrano, and J. H. Patel, “Reliability of Scrubbing Recovery-Techniques for Memory Systems,” IEEE Trans. on Reliability, Vol. 39, No. 1, pp. 114-122, April 1990.

[36] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi, “Modeling the Effect of Technology Trends on the Soft Error Rate of Combinatorial Logic,” Proc. of the Intl. Conference on Dependable Systems and Networks (DSN), pp. 389-398, June 2002.

[37] T.J. Slegel, E. Pfeffer, and J.A. MaGee, “The IBM eServer z990 microprocessor,”, IBM Journal of Research and Development, Vol. 48, No. 3/4, pp. 295-310, April 2004.

[38] SPEC CPU2000 Benchmarks, http://www.specbench.org/osg/cpu2000.

[39] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. “Guided Region Prefetching: A Cooperative Hardware/Software Approach,” Proc. of the Intl. Symp. on Computer Architecture (ISCA’03), pp. 388-398, June 2003.

[40] C. Zhang, F. Vahid, and W. Najjar, “A Highly Configurable Cache Architecture for Embedded Systems,” Proc. of the Intl. Symp. on Computer Architecture (ISCA’03), pp. 136-146, June 2003.

[41] C. Zhang, F. Vahid, and R. Lysecky, “A Self-Tuning Cache Architecture for Embedded Systems,” Proc. of the Design, Automation and Test in Europe Conference (DATE’04), pp. 142-147, Feb. 2004.

[42] W. Zhang, S. Gurumurthi, M. Kandemir, and A. Siavasubramaniam, “ICR: In-Cache Replication for Enhancing Data Cache Reliability,” Proc. of the Intl. Conference on Dependable Systems and Networks (DSN), pp. 291-300, June 2003.

[43] W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, “Reducing Instruction Cache Energy Consumption Using a Compiler-Based Strategy ”, ACM Trans. on Architecture and Code Optimization (TACO), pp. 3-33, Vol. 1, Issue 1, March 2004.

[44] J. F. Ziegler, “Terrestrial Cosmic Rays,” IBM Journal of Research and Development, pp. 19-39, Vol.40, No.1, Jan. 1996.