

# Reducing Soft Error Vulnerability of Data Caches

X. Vera, J. Abella, A. González

Intel Barcelona Research Center

Intel Labs - Universitat Politècnica de Catalunya, Barcelona, Spain

{xavier.vera, jaumex.abella, antonio.gonzalez}@intel.com

R. Ronen

Microprocessor Technology Lab, Intel Corporation

Israel Development Center, Haifa, Israel

ronny.ronen@intel.com

**Abstract**—Current microprocessors are becoming more vulnerable to cosmic particle strikes that may cause soft (transient) errors. Multi-bit upsets (MBUs) on memory such as SRAMs and caches are upsets in which a single radiation event causes several bits to flip, as opposed to single-bit upsets (SBUs) in which only one bit is affected. While SBUs may be detected by parity, MBUs require error correction algorithms like ECC.

This paper proposes a simple scheme for reducing the vulnerability of data in the L1 cache. We propose implementations for reducing cache vulnerability by invalidating cache lines when they hold data not likely to be reused in the short term. As a result, we achieve high coverage (lines which are reused after a long time are not vulnerable any more since they are invalidated) with low performance degradation; on average, for a set of more than 300 traces representing all kind of benchmarks and a processor configuration similar to Intel®Core™Micro-Architecture, we decrease cache vulnerability by 80% with a small performance degradation of 3.7%.

## I. INTRODUCTION

Alpha particles released by radioactive impurities and neutrons coming from outer space are known to cause transient errors in contemporary microprocessors [1]. Single and multi-bit upsets may arise when these particles hit intermediate capacitive nodes of processor storage components such as SRAM bitcells and latches. Since these transient errors occur due to an incorrect charge or discharge of an intermediate capacitive node, they do not cause permanent failure in the hardware and hence are termed soft errors in the literature.

Cache memory reliability is essential to ensure dependable computing. Errors in cache memories can corrupt data values, and can easily propagate through the system to cause data integrity issues. Whereas many techniques have been proposed and are used for protecting L2 caches (like ECC [2], parity and cache scrubbing [3], [4]), many difficulties arise when applied to L1 caches: (i) complex logic like ECC can add extra cycles to the access time, (ii) scrubbing the L1 cache can reduce significantly the L1 bandwidth, (iii) multi-bit errors, which are expected to be common due to technology shrinkage, may be neither detected nor corrected, (iv) power increases due to increased logic and storage, and (v) area overhead is important (adding parity at a byte level for a 32KB cache requires 4KB).

Soft error vulnerability of a processor (or a specific component in a processor) is proportional to:

- The percentage of bits that hold valid data (which are needed for architecturally correct execution) that can be affected by a soft error.

- Time the data spend inside the processor or a specific processor component.

These items can be all summarized and expressed as:

$$Vulnerability = OccupiedBitArea \times TimeSpent \quad (1)$$

Therefore, one may reduce the vulnerability of caches against soft errors by either reducing the number of bits that hold valid data (for instance, one may identify narrow values [5]) or reducing the time data is left unprotected. Currently, L1 caches typically use parity as a method to detect possible soft errors, since employing ECC is too expensive. Thus, L1 caches can detect single-bit upsets, and if write-through policies are used, recovery is possible by fetching data from upper levels. However, many cache lines in L1 stay long time in the cache, and both temporal double-bit errors and spatial multi-bit errors are possible, which would leave the cache lines unprotected leading to data corruption and system crashes.

Our proposal consists of reducing the vulnerability of data stored in the cache by invalidating the cache lines that are reused in long intervals; this way, by sacrificing performance slightly (invalidations will cause some extra cache misses), we reduce the probability of that cache line having a bit flipped. Simply stated, if a cache line is not accessed in a given number of cycles, the cache line is invalidated.

As a result, we can reduce the number of errors detected by parity (that can only be corrected if write-through caches are used) and the number of multi-bit (spatial or temporal) errors that would not be detected, which are expected to be very common.

The rest of the paper is organized as follows. Section II discusses the related work. More details and implementation issues of our proposal are given in Section III. We evaluate the performance cost and vulnerability reduction in Section IV. Finally, we conclude in Section V.

## II. RELATED WORK

Cache scrubbing [3], [4] is one possible solution for temporal double-bit errors. It has also been widely used in the past for main memories, which are typically protected with SECDED ECC. Scrubbing involves reading the bits from the cache, correcting any latent single-bit error, recomputing the ECC (or parity), and writing the bits back. If the scrubbing interval is short enough, the probability for a temporal double-bit error to happen is practically eliminated. However, it causes

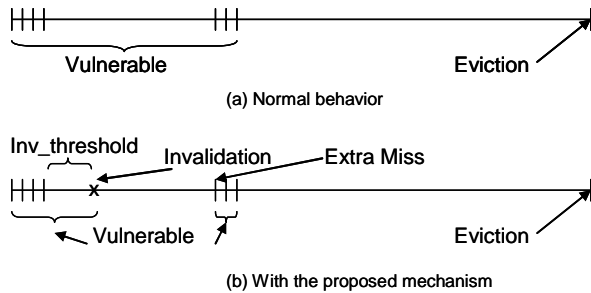


Fig. 1. Example of the invalidation mechanism.

an important overhead since it consumes cache bandwidth and extra power. Interleaving the error detection codes may avoid spatial multi-bit errors, since consecutive bits will be protected by different parity bits, although it increases the complexity of the logic to read/write data. Eager writeback has been studied for speeding up cache lines evictions and reducing performance loss due to clustered traffic [6]. Although it reduces vulnerability for writeback caches, it does not cover for spatial MBUs. Similarly, an early-write-back scheme [7] that enhances the ability to use a less powerful error protection scheme for a longer time without sacrificing reliability and performance has been proposed. Compared to our proposal, it only protects SBUs and only works for writeback caches.

Although attacking a different structure, flushing and restarting the pipeline [8] upon an L2 miss is a similar technique to ours; in order to avoid particle strikes on data waiting in the issue queue, the pipeline is flushed.

Another similar work is cache decay [9]: in order to decrease power consumption, the authors try to switch off the cache line right after the last hit (if inter-hit time is high). Note that the objective of our technique is invalidating before the last hit to reduce the vulnerability: after the last hit, write-through caches are no longer vulnerable since the data they hold will not ever be used.

### III. CACHE LINE INVALIDATION

Unlike parity and ECC that target error detection and correction, our target is reducing cache vulnerability. Soft error vulnerability of data in L1 cache is proportional to the time it spends in the cache. Figure 1(a) represents the accesses and vulnerability to particle strikes for a cache line of a write-through cache; the data hold in the cache line is vulnerable from the time data is brought to cache until the last hit. If data were protected with parity, a single-bit upset (SBU) would be detected and corrected with data held in upper levels of cache. On the other hand, ECC would detect SBUs and most multi-bit upsets (MBUs). Once data is last accessed, data is not vulnerable anymore, since any particle strike would not have any effect on program output. In the case of a writeback cache, a dirty cache line would be vulnerable until eviction time, since upper levels of cache would not have a correct copy.

Our approach is based on the nature of cache line usage: cache lines typically have a flurry of frequent use when brought into the cache, and then have a period of "sleep time" before they are reused again or evicted [9]. We propose implementations for reducing cache vulnerability by invalidating cache lines when they hold data not likely to be reused in the short term. As shown in Figure 1(b), once a line has not been touched for some time ( $inv\_threshold$ ), the line is invalidated. This way, we reduce the probability of that cache line of having a bit (or multiple bits) flipped. However, this may cause some extra misses and performance degradation.

A key aspect of our proposal is to balance the potential for decreasing MBUs vulnerability against the potential for incurring extra level-two cache accesses (when we introduce extra misses by invalidating lines). Invalidating cache lines has a net effect of a cache size reduction. In order to decrease the performance loss due to the reduced cache size (i.e., increased number of misses), we propose and evaluate two alternative options:

- 1) *LocalInvalidation*. This mechanism tries to minimize the number of extra misses per cache set by making sure that at least, some degree of associativity is available by limiting the total number of invalidated lines of each set.
- 2) *GlobalInvalidation*. Instead of limiting the number of invalidated lines per set, we limit the total number of extra misses. We do so by dynamically adapting the length of  $inv\_threshold$ . The idea is as follows: we set two different thresholds ( $min\_th$ : minimum and  $max\_th$ : maximum number of extra misses); then, we keep track of the total number of extra misses in the whole cache for a given interval (usually 10 thousand cycles). If the total number of extra misses is larger than  $max\_th$ , we are being too aggressive and thus we increase the  $inv\_threshold$ ; if the total number of extra misses is smaller than  $min\_th$  (we are not being aggressive enough), we decrease  $inv\_threshold$ .

#### A. Implementation Issues

The extra hardware required to know whether a cache line has not been touched in  $inv\_threshold$  cycles is minimal: for instance, we can add a 3 bits counter to each cache line. The counter is reset to "000" when the line is brought into cache or whenever a word is read/written (i.e., the error detection code is checked or generated). It is incremented by 1 every  $1/8$  of  $inv\_threshold$ . Once all bits are 1, it means that the cache line has not been touched in  $inv\_threshold$  cycles, and thus, it has to be invalidated. Another alternative is employing only one bit and scan every cache line in a round-robin fashion. When the line is scanned, the bit is set to "1". Reading and writing the cache line resets the bit. If the bit is already "1" at scan time, it means that the cache line has to be invalidated (it has not been touched since the last scan). If the cache is write-through, the invalidation can be done in the next access to the cache line. If it is copy back, it is better to invalidate immediately so the dirty data is protected from possible future strikes.

Parameter	Value
UL2	4 MB, 16-way, 12 cycle hit, 1 R/W port, mem lat 45ns
ROB/LSQ	128/96
DL1	32KB, 8-way, 3 cycle hit, 1 read + 1 write port
Execution Unit	32 entries scheduler, 6 issue, up to 3 Ints, up to 2 FP

TABLE I  
SIMULATION PARAMETERS.

Benchmark suite	#traces	Desc./Examples
Encoder	62	Audio/video encoding
SPECfp	41	Spec Fp 2K
SPECint	35	Spec Int 2K
Kernels	52	VectorAdd, FIRs
Multimedia	85	WMedia, photoshop
Office	75	Excel, word, powerpoint
Productivity	45	Internet contents creation
Server	53	TPC-C
Workstation	49	CAD, rendering

TABLE II  
WORKLOADS.

*LocalInvalidation* consists of limiting the number of invalidations to each cache set to a small percentage of the total accesses, and therefore decrease the performance loss. We do so by keeping a small bit-vector (our experiments show that 10 bits is enough) for each cache set that indicates whether the latest accesses resulted in invalidations. Every time the cache set is accessed, the oldest bit is dropped and a "1" is inserted if the line is invalidated, or a "0" otherwise. Every time a line is considered for invalidation, this history register is first checked, and if the number of invalidations in the latest accesses has reached a given threshold, the invalidation is disregarded (for instance, one "1" represents 10% invalidations).

For implementing *GlobalInvalidation*, we need a small register that keeps track of the total number of extra misses and a couple of comparators for checking the stored values against *min\_th* and *max\_th*. Extra misses can be identified simply keeping the tags of the invalidated cache lines. In order to increase and decrease *min\_th* and *max\_th* we opt to multiply and divide by 2, which is straightforward to do with a shifter.

Note that strikes on the extra logic only produce early or late invalidations, but do not affect the correctness of the data.

#### IV. EVALUATION

We have evaluated different configurations and compared both *LocalInvalidation* and *GlobalInvalidation*. The simulated L1 cache is write-through (thus, data is only vulnerable if a later load will hit, otherwise it is not since it is already stored in L2). Vulnerability is measured following equation 1; a cache line is considered vulnerable from the time it is fetched until the last load. The processor configuration is described in Table I and resembles Intel®Core™Micro-Architecture, a modern processor. Our set of benchmarks are described in Table II.

##### A. Cost of invalidations

We start analyzing the impact of invalidating cache lines on performance. Figure 2 details the slowdown (*sdown* on the legend) and the vulnerability reduction for different configurations of the proposed *LocalInvalidation* for the SPEC benchmark suite (SPECfp and SPECint in Table II). Figure 2(a) represents the *LocalInvalidation* mechanism without limiting the number of invalidation per set, whereas Figure 2 limits the number of invalidations to 10% for the total number of accesses. Labels *RedInt* and *RedFP* represent the vulnerability reduction, labels *5%* and *10%* represent the total number of programs from the SPEC benchmarks that experience more than 5% (10%) performance loss, and labels *SdownInt* and *SdownFP* show the average performance loss for the SPECint and SPECfp respectively.

The results show that if we invalidate all cache lines that spend more than 500 cycles without being touched, the vulnerability reduction is very large (over 90% for SPECfp); however, the cost in performance is too high, with 7 programs out of the 26 in the SPEC benchmarks losing more than 10%. When we increase *inv\_threshold* to 1500, performance loss and coverage decrease. Performance loss decreases to 4.4% (1.1%) for SPECfp (SPECint), with only 2 programs with slowdowns higher than 10%. On the other hand, coverage goes down only slightly, which makes this configuration pretty attractive.

In order to reduce the performance loss, we run more experiments where *inv\_threshold* was increased. The results showed that increasing *inv\_threshold* hardly decreased the number of programs with performance loss higher than 10%. Therefore, we tried to minimize the number of cache misses per set. Results are shown in Figure 2(b). As one can see, if we choose an *inv\_threshold* of 1000, by limiting the number of invalidations per cache set to 10% we can achieve similar performance loss and coverage to that of *inv\_1000*, but only 5 programs have a performance loss higher than 5%, and only 1 program losses more than 10%. In general, this second scheme is less aggressive and harms performance of fewer programs than the previous one, while achieving similar average coverage and performance loss.

##### B. LocalInvalidation

As per the previous discussion, we choose *inv\_1000* with 10% of invalidations per cache set as our configuration. Figure 3(a) shows the s-curve distribution for the different traces. Grey<sup>1</sup> bullets show the vulnerability reduction and the black<sup>1</sup> points the performance loss. The Y axis represents percentages. Our results for a large set of traces confirm what we have shown for the SPEC suite: invalidating cache lines that have not been touched (i.e., have not been read or written and thus have not been checked for errors) for the last 1000 cycles, and whose set has less than 10% invalidations, gives very good results. It shows that we can invalidate cache lines that are reused infrequently with small performance degradation only. We run more than 500 different traces and

<sup>1</sup>Grey is red in color, black is blue.

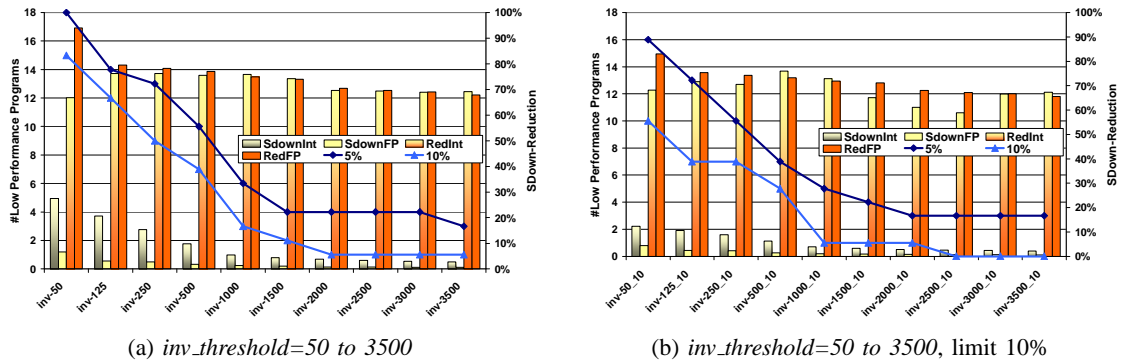


Fig. 2. Detailed evaluation of *LocalInvalidation* for SPEC benchmarks

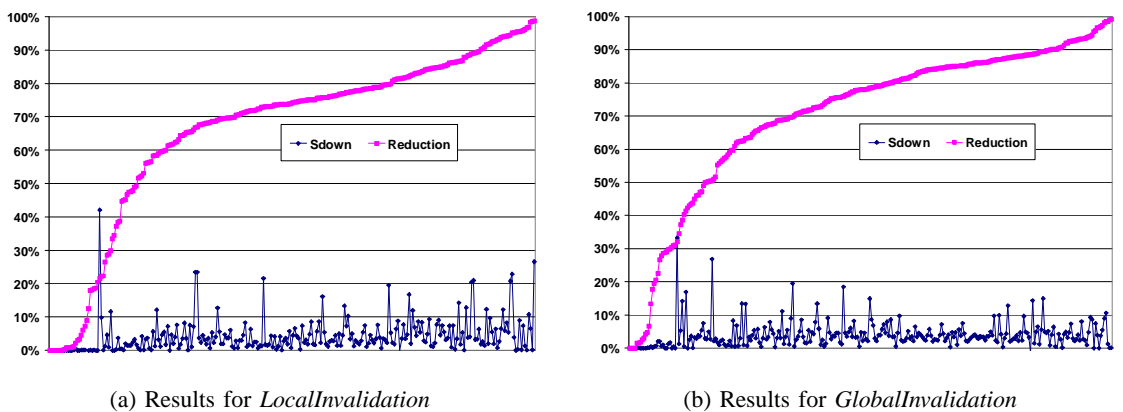


Fig. 3. Performance and coverage numbers

achieved an average decrease in vulnerability of 80% in the cache with an average performance loss of 3.9% ( $\sigma=6\%$ ).

### C. GlobalInvalidation

We also wanted to evaluate *GlobalInvalidation*, our proposal that considers global history rather than local history. We have evaluated a configuration where every 10 thousand cycles, the total number of invalidated cache lines (i.e., lines that have not been touched in *inv\_threshold* cycles) is checked. We use a configuration with *max\_th*=256 and *min\_th*=128; *inv\_threshold* starts with 1000. According to the algorithm described before, if in one interval the total number of extra misses is larger than 256, *inv\_threshold* doubles; if it is smaller than 128, it halves.

We have run the same traces as for the *LocalInvalidation* mechanism and the results are slightly better. Results are summarized in Figure 3(b). Compared to the local approach, the average decrease in vulnerability is roughly the same (80%). However, one may observe some differences in terms of performance loss: as the chart shows, performance loss is roughly the same ( $\mu=3.7\%$ ), but it has less glass jaws (there are less variations  $\sigma=4\%$ ).

## V. CONCLUSIONS

With each generation of microprocessor manufacturing technology, both single-bit upsets and multi-bit errors will increase. We propose a new technique to reduce the vulnerability of the cache by invalidating those lines that will not

be reused for a long period. Our results show that for our set of benchmarks is possible to achieve an 80% vulnerability reduction with a small 3.7% slowdown. Experiments also show that a global approach is more consistent and has less variability than a local approach.

## REFERENCES

- [1] R. Baumann, "Soft errors in advanced computer systems," in *Proceedings of IEEE Design and Test of Computers*, 2005.
- [2] D. Pradhan, *Fault-tolerant computer system design*. Computer Science Press, 2003.
- [3] S. Mukherjee, J. Emer, T. Fossom, and S. Reinhardt, "Cache scrubbing in microprocessor," in *Proceedings of the 10th International Symposium on Pacific Rim Dependable Computing (PRDC)*, 2004.
- [4] A. Saleh, J. Serrano, and J. Patel, "Reliability of scrubbing recovery techniques for memory systems," *IEEE Transactions on Reliability*, vol. 39, no. 1, 1990.
- [5] O. Ergin, O. Unsal, X. Vera, and A. Gonzalez, "Exploiting narrow values for soft error tolerance," *IEEE Computer Architecture Letters*, 2006.
- [6] H.-H. Lee, G. Tyson, and M. Farrens, "Improving bandwidth utilization using eager writeback," *Journal of Instruction Level Parallelism*, 2001.
- [7] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. Irwin, "Soft error and energy consumption interactions: a data cache perspective," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.
- [8] C. Weaver, J. Emer, S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft errors rate in a high-performance microprocessor," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2004.
- [9] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2001.