Area-Efficient Error Protection for Caches

Soontae Kim
Department of Computer Science and Engineering
University of South Florida, FL 33620
sookim@cse.usf.edu

Abstract

Due to increasing concern about various errors, current processors adopt error protection mechanisms. Especially, protecting L2/L3 caches incur as much as 12.5% area overhead due to error correcting codes. Considering large L2/L3 caches of current processors, the area overhead is very high. This paper proposes an area-efficient error protection scheme for L2/L3 caches. First, it selectively applies ECC (Error Correcting Code) to only dirty cache lines and other clean cache lines are protected using simple parity check codes. Second, the dirty cache lines are periodically cleaned by exploiting the generational behavior of cache lines. Experimental results show that the cleaning technique effectively reduces the number of dirty cache lines per cycle. The ECCs of this reduced number of dirty cache lines can be maintained in a small storage. Our proposed scheme is shown to reduce the area overhead of a 1MB L2 cache for error protection by 59% for SPEC2000 benchmarks running on a typical four-issue superscalar processor.

1. Introduction

With technology scaling, supply voltages are reducing together with threshold voltages for fast transistor switching. At the same time, pipeline is becoming deeper to increase clock frequency. However, reduced supply voltages, high clock frequencies, and low capacitive values of circuits make them more susceptible to soft errors [1, 2, 3]. Soft errors are incurred due to excessive charge carriers mainly by external alpha particles and neutrons. Memory structures are vulnerable to the soft errors since these can change the values stored in them [1, 4, 5]. Especially, caches are good victims for these soft errors since most of modern processors adopt large caches for improving performance.

Consequently, several error detection and correction schemes have been used for improving the reliability of the memory system [4, 5]. Simple one is parity check code that can detect any odd number of bit errors. More powerful one is error correcting code (ECC) that can detect and correct bit errors at the cost of increased complexity and area. For example, in Itanium processor [5], TLBs and L1 caches use parity codes while L2 and L3 unified caches are protected by ECC such as SECDED (Single Error Correction Double Error Detection). Power4 system [4] also uses parity check code for the L1 caches and ECC for the L2 cache. Both systems use write-through L1 caches and write-back L2/L3 caches so that L1 caches can be protected using parity check code. Consequently, ECC protecting L2/L3 caches incurs large area overhead. For example, every 64 bits of data requires 8 bits for ECC in Itanium processor. The area overhead in bits in this case is 12.5%, which corresponds to 128KB in a 1MB L2 cache. This area would be used for other purposes if we could reduce the area overhead for ECC protection.

In this paper, we propose a novel scheme that can provide lower area overhead for error protecting L2 caches. Our scheme consists of three techniques. First, it uses ECC protection for dirty cache lines. Clean (not modified) cache lines are protected using parity check code since non-corrupted data can be found from the next level of the memory hierarchy. This significantly reduces cache lines that require ECC protection. Second, to reduce dirty cache lines further, we employ cleaning of the dirty cache lines by periodically writing them back to the main memory by exploiting generational behavior of cache lines. Our experimental results show that the percentage of dirty cache lines can be reduced by more than 50% on the average without increasing traffic to the main memory much. Finally, the reduced dirty cache lines by our cleaning technique can be maintained in a much smaller storage than the one of the conventional cache architecture. Our scheme is shown to reduce the area overhead due to ECC in a 1MB L2 cache by 59% with less than 1% performance loss when SPEC2000 benchmarks are run on a typical 4-issue superscalar processor.

The rest of the paper is organized as follows. The next section discusses related work and Section 3 explains our non-uniform error protection, dirty cache line cleaning, and our ECC storage architecture in detail. Experimental setup is detailed in Section 4 and experimental results are discussed in Section 5. Finally, Section 6 concludes the paper with future work.

2. Related Work

Kim et al. [9] propose area-efficient error protection techniques for on-chip L1 caches. Their idea is based on the observation that small portion of data cache space is frequently accessed for most of the execution time. They do not provide error protection for all cache lines but only for frequently accessed cache lines using separate error protection circuits, thereby reducing the area overhead for the error protection compared to conventional error check for all cache lines. In contrast, our scheme provides error protection for all cache lines in the context of larger L2/L3 caches.

To enhance the reliability of the data cache, Zhang et al. propose in-cache replication [10]. To replicate data cache blocks, they use a dead block prediction technique proposed for reducing the leakage energy of caches [12]. By replicating active cache blocks in dead cache blocks, they try to enhance reliability of the caches.

Lee et al. propose eager writeback for avoiding performance loss due to clustered bus traffic in a write back cache by writing back dirty cache lines before they are replaced [7]. Li et al. use parity codes for clean cache lines and ECC for dirty cache lines for L1 data cache, and write back the dirty cache lines periodically since parity codes are more energy-efficient than ECC [11]. Their scheme, however, does not provide area reduction.

3. Area-efficient Error Protection

Conventional error protection schemes apply uniform error protection to all cache lines, assuming each cache line has the same probability of errors [4, 5]. Parity check codes are employed in the L1 instruction cache since instructions are not modified. Processors can find non-corrupted instruction data from the next level of the memory hierarchy in cases of errors. In contrast, L1 data cache can employ one of two schemes. In the first scheme, the data cache is protected with ECC since data can be modified by store instructions. In the second scheme, the data cache is protected with parity codes by employing a write-through policy using a write buffer, and L2/L3 caches are protected with ECC as in POWER 4 system and Itanium processor [4, 5]. The write buffer reduces data traffic to L2 cache by combining multiple write backs into single one [6]. We adopt the second scheme as our baseline error protection for caches

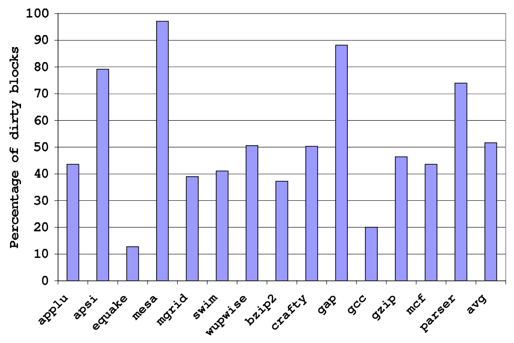

Figure 1. Percentage of dirty cache lines per cycle in a 1MB 4-way L2 cache.

since many modern processors adopt a write-through L1 data cache and write-back L2 cache.

Since the instruction and data caches are protected with simple parity codes, we focus on the L2 cache employing more expensive ECC in this paper.

3.1. Non-uniform Error Protection

We apply non-uniform error protection to cache lines. This is based on the observation that not all cache lines are dirty. Figure 1 shows percentage of dirty cache lines of a unified L2 cache per cycle when SPEC2000 benchmarks are run on a typical 4-issue superscalar processor. Refer to Section 4 for experimental setup. The L2 cache has 4-way set-associative 1MB with 64B cache lines. So it has a total of 16384 cache lines. It is observed from the figure that there are a large percentage of clean cache lines except for four benchmarks. The percentage of dirty cache lines is, on the average, 51.6% across all the benchmarks. Thus, it is inefficient to use the same ECC for all cache lines. Instead, we use parity check codes for clean cache lines and ECC for dirty cache lines. Every 64 bits data requires 1 bit parity check code as in Itanium processor. With our non-uniform error protection, we need 16KB parity check codes for all the L2 cache lines and around 64KB ECC for dirty cache lines, saving 48KB = 128KB - (64KB + 16KB) area. Now, we focus on dirty cache lines constituting about half of the all cache lines.

3.2. Cleaning Dirty Cache Lines

We would reduce dirty cache lines further if we could make them clean. One way is to write dirty cache lines back to the main memory as in the write-through cache. However, this will significantly increase the memory traffic to the main memory if we prematurely write dirty cache lines back to the memory that will be modified soon. Increased memory traffic to the main memory consumes the band-

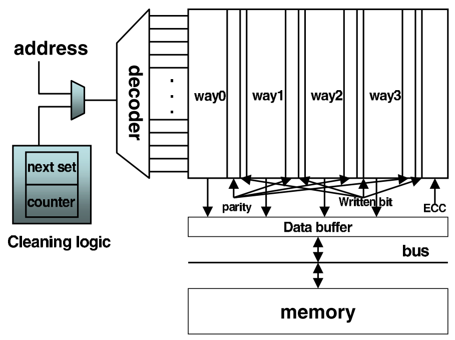

Figure 2. Cleaning logic for the 4-way associative L2 cache. The additional components are shaded.

width of off-chip memory bus, resulting in increased energy consumption and performance degradation. Therefore, we need a technique that can reduce dirty cache lines without increasing writes back to the main memory.

It is reported in [12] that cache lines are frequently accessed after they are brought in from the main memory and then see a period of dead time before evicted. During the dead period, the cache lines are never used, so they can be turned off to reduce leakage power. In a similar way we can think that cache lines will not be modified during the dead period before their eviction. Then, we can make them clean earlier when they are detected to be dead. They will be evicted from the cache after the dead period but this will not incur write backs to the main memory since they are already clean. Therefore, we could reduce dirty cache lines per cycle without increasing memory traffic to the main memory much if the dead time detection would be accurate. We can write back the dirty cache lines before their dead period when it is expected there will be no more write operations to the dirty cache lines, further reducing dirty cache lines per cycle.

Figure 2 shows our implementation of cleaning logic for the L2 cache that has 4-way associative 1MB. Each way in conventional L2 cache architecture has an ECC-bits array while our L2 cache architecture requires a parity-bits array for each cache way and one ECC array for all cache ways. Each cache line is augmented with a *written* bit. The written bit is reset to zero when new data are brought in to the cache line and is set to one when it is modified more than one time, while the dirty bit is set to one when it is modified once. Thus, when the written bit is one, the dirty bit is also one. After some predefined interval has passed since a cache line is brought in from the main memory, it is written back to the main memory if its written bit is zero but its dirty bit is one since this indicates that the corresponding cache line is not likely to be further modified in the near future. Cache line cleaning is performed by the cleaning logic that includes a cycle counter and a latch storing the next cache set number. The cleaning logic checks cache lines belonging to the cache set number stored in the latch after predefined cycles, e.g. 1000 cycles, have passed whether their written bits are zero and dirty bits are one. If so, the cache lines are written back to the main memory and their dirty bits are reset to zero. Note that if this is a premature write back, this increases memory traffic to the main memory but does not results in incorrect memory access. Otherwise, the written bits are reset to zero. The next set number in the latch is increased by one to indicate the next cache set that will be checked after another predefined cycles have passed. When both L1 caches and the cleaning logic request an access to the L2 cache, the L1 caches are given a priority since their requests are more performance-critical. Note that even though we assume our own cycle counter in our implementation, many processors have various cycle counters for operating systems and performance counting [8]. The area overhead due to the written bits is 16K bits and the latch is 12 bits wide since there are 4K cache sets in our 1MB 4-way L2 cache with 64B cache lines. The finite state machine for the cleaning logic is simple; it checks and resets the written and dirty bits of the cache lines in each predefined cycles. So most of area overhead comes from the written bits.

3.3. ECC storage

So far, we have focused on reducing dirty cache lines. Then, how can we exploit these reduced dirty cache lines for low area error protection? All cache lines, regardless of clean or dirty, are protected using parity codes in the L2 cache in our case. The ECCs of the dirty cache lines are stored in an ECC array, which is much smaller than the ECC storage in the conventional cache architecture. Our scheme guarantees that all dirty cache lines find their corresponding ECCs in the ECC array. Figure 2 also shows our ECC array structure. Our scheme has one parity array for each cache way and one ECC array for all cache ways while conventional cache architecture needs an ECC array for each cache way. Therefore, all cache lines belonging to the same set (4 lines in a 4-way associative cache) share an ECC entry, thereby we can reduce the storage required for the ECC protection by four times.

L2 reads are performed as in the conventional error protection. If the cache line accessed is clean, parity code is used. Otherwise, ECC is used for error detection and correction. We focus on L2 writes since reads do not make cache lines dirty. We can categorize memory writes into two cases in the context of L2 cache. First, there is no dirty cache line in the cache set accessed by the current write request. We update the corresponding ECC entry with the new ECC data of the current write. Second, there is a dirty cache line in the accessed cache set before the current write. We need to allo-

Parameter	Configuration
Issue window	64-entry RUU 32-entry LSQ
decode and issue rate	4 instructions per cycle
Functional units	4 INT add, 1 INT mult/div 1 FP add, 1 FP mult/div
L1 instruction cache	32KB 4-way, 32B line, 1-cycle
L1 data cache	32KB 4-way, 32B line, 1-cycle
L2 cache	unified 1MB, 4-way, 64B line, 10-cycle
Main memory	8B-wide, 100-cycle
Branch prediction	2-level , 2K BTB
Instruction TLB	64-entry, 4-way
Data TLB	128-entry, 4-way

Table 1. Baseline processor configuration

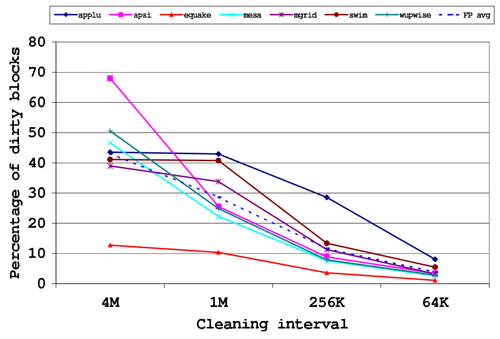

Figure 3. Percentage of dirty cache lines per cycle for different cleaning intervals in floating-point benchmarks.

cate an entry in the ECC array for the current write. This results in an eviction of the ECC data for the dirty cache line already in the cache set, which must be written back to the main memory since we can no longer provide ECC protection for the cache line. Then, how can we identify the corresponding cache line in the L2 cache to the evicted ECC? The cache line with its dirty bit 1 is the corresponding cache line to the evicted ECC since at most one cache line among an L2 cache set can be dirty in our case. In our scheme, write backs from the L2 cache happen in three cases: eviction from the ECC array, dirty line cleaning, and dirty cache line replacement.

4. Experimental Setup

We modified SimpleScalar version 3 tool suite [14] for this study. Our baseline processor models an out-of-order four-issue processor. Table 1 summarizes the simulation parameters of this processor. Since SimpleScalar models write back L1 cache, we modified SimpleScalar to support write-through L1 cache by implementing a write buffer that has

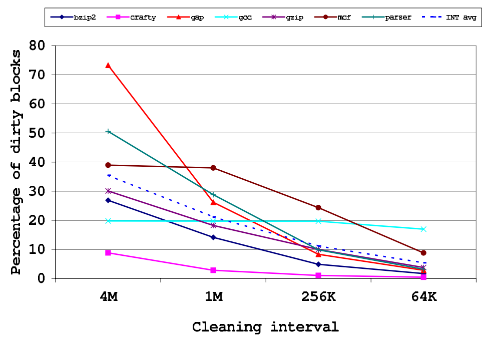

Figure 4. Percentage of dirty cache lines for different cleaning intervals per cycle in integer benchmarks.

fully associative 16 entries. Our simulations have been performed with a subset of SPEC2000 benchmarks using precompiled binaries obtained from [13]. These were compiled with DEC C V5.9-008, Compaq C++ V6.2-024, and Compaq FORTRAN V5.3-915 compilers using high optimization level. Seven programs from each of floating-point and integer benchmarks are randomly chosen for our evaluation. All the benchmarks are fast-forwarded for one billion instructions to avoid initial start-up effects and then simulated for one billion committed instructions. For all simulations, the reference input sets are used.

5. Experimental Results

5.1. Cleaning Results

For dirty cache line cleaning, we have experimented various cleaning intervals from 64K processor cycles to 4M cycles by increasing four times. For example, 1M cleaning interval means that each cache line is checked for every one million processor cycles to see its written bit is zero and dirty bit is one. Intuitively, smaller cleaning interval indicates aggressive write backs of dirty cache lines while larger interval is conservative. We need to find out the best cleaning interval that can minimize dirty cache lines per cycle and, at the same time, additional write backs by prematurely cleaning dirty cache lines that will be modified soon.

Figure 3 and Figure 4 shows percentages of dirty cache lines per cycle for different cleaning intervals in floating-point and integer benchmarks, respectively. As expected, smaller cleaning intervals linearly reduce the percentage of dirty cache lines. The *applu*, *swim*, *mgrid*, *equake*, and *mcf* show little reduction with 4M interval. In other benchmarks, 4M interval shows small reduction in the percentage of dirty cache lines. If we want around 2K dirty cache lines, 256K is the appropriate interval on the average. For 4K dirty cache lines, 1M interval is a good choice.

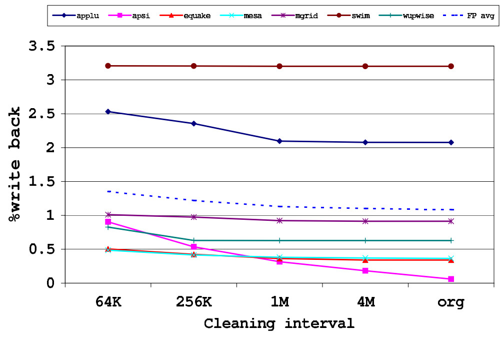

Figure 5. Percentages of write back traffic out of all loads/stores for floating-point benchmarks.

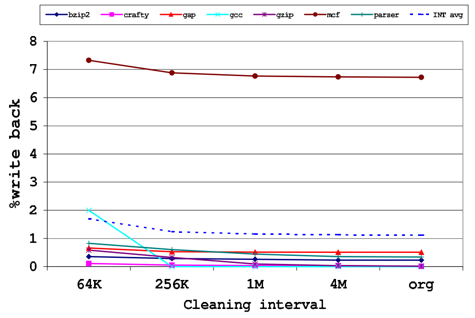

Figure 6. Percentages of write back traffic out of all loads/stores for floating-point integer benchmarks.

However, it should be noted that smaller cleaning interval increases number of write backs due to its aggressiveness. Increased write backs mean more activity in the bus and main memory, which results in more energy consumption and performance loss. Thus, it is important to find out a cleaning interval that provides largest reduction in the number of dirty cache lines and does not increase memory traffic to the main memory as much. Figure 5 and Figure 6 presents results for the memory traffic due to write backs from the L2 cache for floating-point and integer benchmarks, respectively. The %write back in the figures are the percentage of write backs out of all loads/stores. We also show the results without dirty cache line cleaning (*org* in the figures). Overall, 1M cleaning interval approaches the results of the *org*. The percentages of write backs are 1.13% and 1.08% for 1M interval and *org*, respectively, in the floating-point benchmarks. These percentages are 1.16% and 1.12%, respectively, for the integer benchmarks. Thus, our cleaning technique using written bit is effective for detecting dead cache lines that will not be used any more before evicted.

5.2. Performance Results

To experiment our new ECC technique, we determined to use 32KB ECC array from the observations seen before. Since each ECC entry is 8 bytes, there are 4K ECC entries in total, which is the same as the number of sets in the L2 cache of our baseline processor. Each L2 tag and status bits are protected by a 1-bit parity code as in Itanium processor [5]. Thus, our approach has a total of 54KB area overhead for error protection: 16KB for parity codes in the data array, 2KB for written bits, 2KB parity bits for the tag array, 2KB parity bits for the status bits, and 32KB for the ECC array, compared to 132KB in the conventional ECC protected L2 cache: 128KB for the data array and 4KB for the tag array and status bits. This is 59% reduction in area overhead.

Our goal is to have around 4K dirty cache lines per cycle in the L2 cache. We use the profiled information to statically determine the best cleaning interval for all the benchmarks. We use 1M cleaning interval since it gives around 4K dirty cache lines per cycle from Figure 3 and 4 and there is little change in write back traffic beyond 1M cleaning interval. Note that, however, each benchmark will have different cleaning interval for best results.

Figure 7 shows the percentage of dirty cache lines in the L2 cache when we applied dirty line cleaning and a smaller ECC storage. In all the benchmarks, the percentage of dirty cache lines is lower than 25% and in Figure 3 and 4. This means that additional write backs due to ECC entry eviction make more L2 cache lines clean. Let us look at the four benchmarks in detail: *apsi*, *mesa*, *gap*, and *parser*. These benchmarks include a large percentage of dirty cache lines as can be observed from Figure 1. In Figure 7, most of those dirty cache lines are removed. This indicates that our dirty cache line cleaning is effective even in those benchmarks with a large percentage of dirty cache lines.

Figure 8 shows the percentages of write backs out of all loads/stores. Each bar in the figure is partitioned into three portions: *Clean-WB* for write backs due to our dirty cache line cleaning, *WB* for normal write backs due to cache misses, and *ECC-WB* for write backs due to ECC entry eviction. *Clean-WB* and *WB* constitute little portion of the total write backs. *ECC-WB* consists of most of the write back traffic on the average for most of the benchmarks; *ECC-WB* is more dominant due to a large number of ECC entry eviction. The average percentages of write backs are 1.20% and 1.19% for the floating-point and integer benchmarks, respectively. These percentages are 1.08% and 1.12% in the original configuration. Thus, memory traffic increase due to additional write backs is small in our scheme.

We also measured performance loss in terms of IPC due to increased write back traffic from the L2 cache. We modified SimpleScalar to include additional write back traffic due to our scheme. We assumed split transaction bus for the off-chip memory bus. It is observed that the increased memory traffic does not have a big impact on performance. Per-

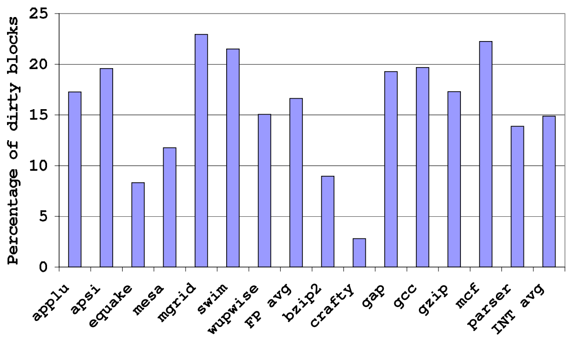

Figure 7. Percentage of dirty cache lines per cycle for our approach.

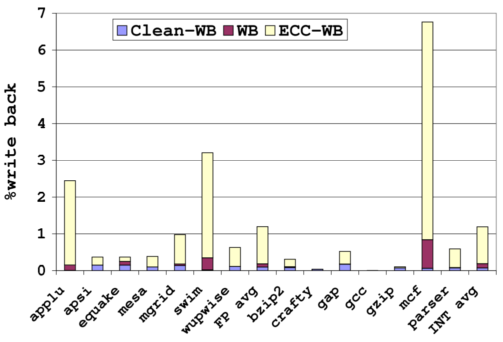

Figure 8. Percentage of write back traffic out of all loads/stores for our approach.

formance loss is 0.14% and 0.65% on the average for the floating-point and integer benchmarks, respectively.

6. Conclusions

Current trends of reduced supply voltage, high frequencies and low capacitive values of circuits make them more vulnerable to soft errors. Especially, cache memories are more susceptible to soft errors due to their large transistor counts. Consequently, modern processors such as Power4 and Itanium processors adopt an error protection mechanism for cache memories. Protecting cache memory using ECC requires as much as 12.5% area overhead. Considering large L2/L3 caches in current processors, the area overhead for error protection is very high.

For area-efficient error protection for caches, we propose a novel scheme that combines dirty cache line cleaning and non-uniform error protection using a new ECC storage architecture. Clean cache lines are protected using less expensive parity codes in terms of bits, while dirty cache lines are ECC protected. To reduce dirty cache lines and thereby area overhead for error protection, our approach employs a dirty cache line cleaning technique that writes dirty cache lines back to the main memory when they are expected to not be modified further in the near future by exploiting the generational behavior of cache lines. The ECCs of reduced dirty cache lines can be maintained in a smaller ECC array than the one in the conventional cache architecture. Experimental results show that our scheme effectively reduces the area overhead by 59% and increase in memory traffic due to write backs is small, resulting in less than 1% performance loss.

References

[1] P. Hazucha and C. Svensson. Impact of CMOS technology scaling on the atmospheric neutron soft error rate. *IEEE Transactions on Nuclear Science*, 47(6), 2000.

[2] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Bokar. Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18um. In *Digest of Technical Papers of Symposium on VLSI Circuits*, 2001.

[3] N. Seifert, D. Moyer, N. Leland, and R. Hokinson. Historical trend in alpha-particle induced soft error rates of the Alpha microprocessor. *IEEE Transactions on VLSI*, 9(1), 2001.

[4] D. C. Bossen, J. M. Tendler, and K. Reick. POWER4 system design for high reliability. *IEEE Micro*, March-April, 2002.

[5] N. Quach. High availability and reliability in the itanium processor. *IEEE Micro*, Sept-Oct, 2000.

[6] K. Skadron and D.W. Clark. Design issues and tradeoffs for write buffers. In *Proc. International Symposium on High-Performance Computer Architecture*, 1997.

[7] H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager writeback - a technique for improving bandwidth utilization. *Proc. International Symposium on Microarchitecture*, 2000.

[8] J. Dean, J. Hicks, et al. Profileme: hardware support for instruction level profiling on out-of-order processors. *Proc. International Symposium on Microarchitecture*, 1997.

[9] S. Kim and A. K. Somani. Area efficient architectures for information integrity in cache memories. In *Proc. International Symposium on Computer Architecture*, 1999.

[10] W. Zhang, S. Gurumurthi, M. Kandemir, and A. Sivasubramaniam. ICR: in-cache replication for enhancing data cache reliability. In *Proc. International Conference on Dependable Systems and Networks*, 2003.

[11] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Soft error and energy consumption interactions: a data cache perspective. In *Proc. International Symposium on Low Power Electronics and Design*, 2004.

[12] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce leakage power. In *Proc. International Symposium on Computer Architecture*, 2001

[13] SPEC2000 Binaries. http://www.eecs.umich.edu/∼chriswea/benchmarks/spec2000.html

[14] D. Burger and T. M. Austin. The Simplescalar tool set. Computer Sciences Department Technical Report. No. 1342. University of Wisconsin, 1997.