Area Efficient Architectures for Information Integrity in Cache Memories

Seongwoo Kim and Arun K. Somani
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011, USA
{skim, arun}@iastate.edu

Abstract

Information integrity in cache memories is a fundamental requirement for dependable computing. Conventional architectures for enhancing cache reliability using check codes make it difficult to trade between the level of data integrity and the chip area requirement. We focus on transient fault tolerance in primary cache memories and develop new architectural solutions to maximize fault coverage when the budgeted silicon area is not sufficient for the conventional configuration of an error checking code. The underlying idea is to exploit the corollary of reference locality in the organization and management of the code. A higher protection priority is dynamically assigned to the portions of the cache that are more error-prone and have a higher probability of access. The error-prone likelihood prediction is based on the access frequency. We evaluate the effectiveness of the proposed schemes using a trace-driven simulation combined with software error injection using four different fault manifestation models. From the simulation results, we show that for most benchmarks the proposed architectures are effective and area efficient for increasing the cache integrity under all four models.

1. Introduction

Memory hierarchy is one of the most important elements in modern computer systems. The reliability of the memory significantly affects the overall system dependability. The purposes of integrating an error checking scheme in the memory system are to prevent any error that has occurred in the memory from propagating to other components and to overcome the effects of errors locally, contributing to the overall goal of achieving failure-free computation.

Transient faults, which occur more often than permanent faults [6], [14], can corrupt information in the memory, i.e., instruction and data errors. These soft errors may result in erroneous computation. In particular, errors in cache memory, which is the closest data storage to the CPU, can easily propagate into the processor registers and other memory elements, and eventually cause computation failures. Although the cache memory quality has improved tremendously due to advances in VLSI technology, it is not possible to completely avoid transient fault occurrence. As a result, data integrity checking, i.e., detecting and correcting soft errors, is frequently used in cache memories.

The primary technique for ensuring data integrity is the addition of information redundancy to the original data. Whenever a data item is written into the cache, a check (or protection) code such as parity or error-correcting code (ECC) is also included. We denote a pair of data and check code by a *parity group*. When an item is requested, the corresponding parity group is read and an error syndrome is generated to check and correct the error if there is one. The capability of the protection code needs to be determined properly depending on the degree of required data integrity, expected error rate based on harshness of the operating environment, and design and test cost.

Despite the fact that predicting the exact rate and behavior of transient faults in a system is not possible, current data integrity checking schemes for caches are generally selected on a single-bit failure model basis. Thus, byte-parity scheme (one bit parity per 8-bit data) [15] and single error correcting-double error detecting (SEC-DED) code [5] are widespread. Many higher capability codes for byte or burst error control have also been studied [3], [8].

Check codes employed for increased reliability in the caches are constructed in a *uniform structure*, i.e., every unit of data is protected by a check code of the chosen capability. This conventional method is reasonable under the assumption that each cache item has the same probability of error occurrence. However, it has the following deficiencies.

- Check code in the uniform structure is an expensive way to enhance cache reliability. Therefore, it is overkill under extremely low error rates.
- It is not flexible in terms of chip area requirement, as the area occupied by the check code is directly pro-

portional to the cache size. If the budgeted area is not sufficient for the uniform structure, no intermediate architectures are currently available. The high overhead may result in sacrificing the integrity checking.

The uniform structure enables every item to be checked. However, error checking is necessary only for those items that are likely to be corrupted. If it is possible to predict such cache items, a higher data integrity can be achieved with a smaller amount of chip area for the check code. In practice, there are several reasons that soft error occurrence tends to concentrate in a few locations. Information in the cache can be altered during read/write operations due to low noise margins, and thus cache lines that are frequently accessed may have a higher probability of corruption. Cross-coupling effects may also induce errors in neighboring locations of a line being accessed. On the other hand, global random disturbances commonly affect any location. More importantly, errors in unused lines are no concern.

In this paper, we take these factors into account to develop area efficient architectural solutions for improving cache integrity. The underlying idea is that more error-prone and more likely used cache lines must be protected first. The random faults are not biased to a specific location or time. However, if a fault occurs during the access of a line, it is more likely to affect the data being accessed. As a result, access frequency makes a difference in the probability of error occurrence between *active* (more access) and *inactive* (less access) lines. With large caches, the majority of cache accesses are usually localized in a small portion of the cache. This frequently accessed part is considered more error-prone. The corrupted items in the most frequently used (MFU) lines are likely to be used as instructions or operands, quickly affecting the computation. On the other hand, errors in inactive lines have a higher probability of being replaced or overwritten with new, correct data [13]. Data errors are harmful only if they are used for operation, suggesting that not providing check codes for inactive lines may not affect the integrity of the computation.

We present three new architectures, *parity caching*, *shadow checking*, and *selective checking*, to protect primary caches. These schemes allow flexible trade-offs between silicon area and level of data integrity so that both the reliability and area requirements can be met. The new schemes can achieve an acceptably high level of fault coverage with much less area than the uniform structure, realizing area efficient enhancement of cache system reliability.

This paper is organized as follows. Section 2 explains what affects the transient fault rate, how the cache data status changes, and how soft errors affect cache operations. Section 3 describes our new architectures and corresponding management policies. We also provide an area comparison with the conventional organization. To evaluate the effectiveness of the proposed schemes, we conducted error injection experiments on a simulation model of the system. The error models and evaluation methodology are presented in Section 4. The performance of the proposed schemes is discussed in Section 5. In Section 6, we make some concluding remarks.

2. Errors in Cache and Their Effects

To reduce CPU-memory bandwidth gap, up to 60% of the area of recent microprocessors is dedicated to caches and other memory latency-hiding hardwares [9]. The cache memory stores instructions or data in data RAM along with address tags in tag RAM. The primary caches are required to operate at the processor's clock speed. Use of lower voltage levels, high speed data read/write operations, and extremely dense circuitry increase the probability of transient fault occurrence, resulting in more bit errors in cache memories. Moreover, external disturbances such as noise, power jitter, local heat densities, and ionization due to alpha-particle hits [10] can also corrupt the information.

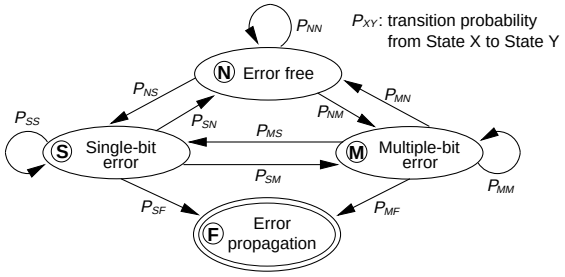


Figure 1. Cache data state transition.

Figure 1 shows how the state of a cache data block is affected by error occurrence and recovery. Initially, the block is error-free (State N). Single- and multiple-bit errors due to some transient faults lead to State S and State M, respectively. The state of a corrupted block changes back to State N if the error is overwritten by a new correct item or is corrected by a recovery mechanism. The absorbing state, F, represents the situation where an error propagates outside the cache boundary through a normal access. If the cache memory always operates in State N ($P_{\mathsf{NN}} = 1$) or erroneous items are never used, then no fault tolerance schemes are necessary. However, in practice, this is not likely to be the case. The check codes help keep P_{SN} and P_{MN} nearly equal to 1 so that the cache block rarely reaches State F (i.e., $P_{\mathsf{SF}} \simeq P_{\mathsf{MF}} \simeq 0$).

One may suspect that extremely infrequent error propagation ($0 \neq P_{\mathsf{SF}} << 1$, $0 \neq P_{\mathsf{MF}} << 1$) may not have any notable effects. However, even a single-bit error can bring a complete system failure. Through program execution, corrupted data items propagate to the processor's registers and

produce an erroneous outcome that can eventually propagate to the external world. A single error can also spread to other registers, cache lines, and memory locations as the processor continues to use the corrupted data recursively. The erroneous contents of registers can also cause page or segmentation faults and incorrect control flow changes. It is shown in [13] that the probability of a single bit leading to a failure is about 50%. However, the actual probability heavily depends on the cache hit rate.

Bit changes in the tag RAM also cause the following improper cache hit and miss decisions that make the processor's memory references chaotic.

- Pseudo-hit: the tag portion of the incoming reference address matches with the wrong cache line's tag field.
- Pseudo-miss: the tag associated with the desired data item does not match with the reference address.
- Multi-hit: the tag portion of the reference address matches with the tags of multiple lines in a set.

In the case of a pseudo-hit, the processor gets wrong data on a read and updates the data in the wrong location on a write. A pseudo-miss generates an unnecessary main memory access. The multi-hit may be detected by the cache controller without check code support for the tags, but handling is not simple. The controller cannot distinguish between the multiple hit lines to service the processor's request. Moreover, invalidating all of those lines and treating the access as a miss is not a solution if any of the lines are dirty, i.e., valid data may exist only in the cache. Writing the data back to main memory should precede the invalidation of a dirty line. However, this cannot be done without resolving the line selection problem. Thus, a resolution may not be possible or may lead to data consistency failure.

Due to an error in the cache status field, a valid line can be unintentionally invalidated if its valid bit is changed erroneously. In the case of a dirty bit error, a dirty line is considered to be clean and may be replaced without a write-back. Therefore, the most recent data can be lost. Line replacement based on access history can also be performed improperly if faults flip the corresponding history bits.

3. New Architectural Approaches

In conventional systems, data integrity checking schemes are implemented in a uniform structure. In this section, we describe three alternative architectures that have flexible chip area requirements. The parity caching scheme described in Section 3.1 is proposed as a substitute for uniformly organized check code under extremely low error rates. Section 3.2 describes shadow checking, which is an inexpensive variant of replication architecture under very noisy environments. Selective checking is presented in Section 3.3 as a simpler alternative to the first two when a cache has multi-way set associativity. In Section 3.4, we also discuss integration of cache scrubbing [11] into the proposed architectures to enhance their capabilities.

A data read/write involves accessing cache cell arrays including data, tag, and status bits. Errors can appear in any field. Therefore, integrity checking is required for all three fields. For brevity in presentation, we do not always address the proposed schemes separately for each field. However, the operation and management mechanisms apply to all three fields in an identical manner.

3.1. Parity caching

One of the widely known program properties is that only 10% of program instructions are responsible for 90% of instructions executed [4]. For some programs, a similar observation can be made in the data segment of main memory. Cache accesses are also often localized. Under considerably low error environments, it can be expected that most soft errors of any significance will occur in these most commonly used portions of instructions and data.

In a low error rate environment, when the budgeted area is not sufficient for the check codes of the uniform structure, the number of check codes needs not be continuously increased with the primary cache size to maintain high data integrity. Based on the assumption that the MFU lines are most error-prone and errors in those lines easily propagate unless checked, we organize a *parity cache*, whose entries contain the check codes for the MFU lines. This scheme is called *parity caching*: the caching of check codes.

The organization and operation of the parity cache are similar to general cache memories, but it provides integrity checking for the main cache. It covers the most error-prone main cache locations using $log_2 l$ index bits, where l is the number of lines in the main cache. The number of parity cache entries, n, is smaller than l. The main cache lines for which check codes are held in the parity cache are selected dynamically such that the MFU portion of the cache can be protected first. This is accomplished by employing least recently used (LRU) replacement policy for the parity cache, where the entry that has not been used for the longest time is replaced with a new item.

Figure 2 shows the logical organization of a 16KB direct-mapped data cache or D-cache (left half) protected by a parity cache of 16 entries (right half) in conjunction with an ECC unit. The 16 parity cache entries are organized in a 4-way set associative manner and store check codes for 16 lines selected from the main cache. The ECC unit performs error checking. The parity cache type and the check code capability can be flexibly determined. The main data line consists of 32 bytes, and 32 parity bits (1 per byte) are used for its protection. The tag is protected with a SEC-DEC code. For the status bits, one even parity bit is used.

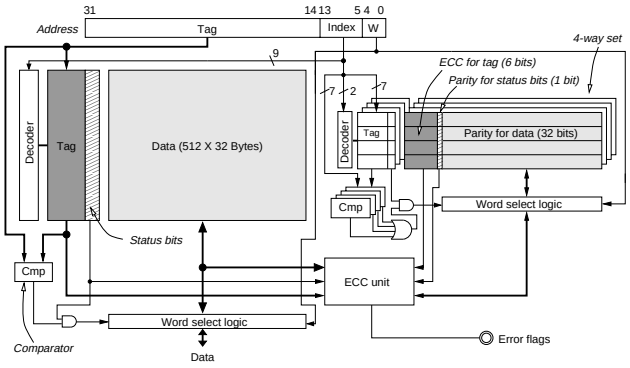


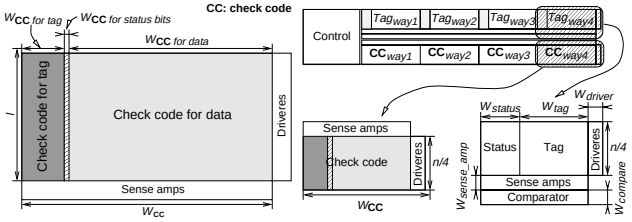
Figure 2. A 16KB D-cache and a parity cache.

For the mapping between the parity and main caches, each entry in the parity cache is tagged. In the case of direct-mapped main caches, the index field of a reference address is used for the parity cache tag as it exactly corresponds to one line in the main cache. In the example above, the first seven bits of the index are stored as a tag and the last two bits are used in selecting a set in the parity cache. Note that the number of parity tag bits is small in comparison to the main cache tag, resulting in simpler tag comparators. If the parity cache uses direct-mapping, the tag is reduced to five bits. If the main cache is k-way ($k > 1$) set associative, multiple lines of the same index value can coexist in the main cache and one parity entry may erroneously be mapped to all of those lines. This problem is solved by storing additional $log_2 k$ bits in the parity tag to distinguish the correct line from k ways. In this section, we apply the parity caching only to direct-mapped main caches. We present an alternate method (selective checking) for set associative main caches in Section 3.3.

While the main cache serves the processor's request, the parity cache synchronously monitors the integrity of the main cache and updates the check codes. This is the same as existing systems with the uniform array of check codes, and so, no additional delay in cache access time is needed. When a miss occurs in the main cache, new data items are fetched from the lower level memory with check codes, and they are written into the parity cache in a free entry, if any, or replace the LRU entry of the mapped set. Whenever a read hit occurs in both caches, error checking on all three fields of the line is performed using the ECC unit. Handling errors is the same as in the conventional systems. Whenever a main cache line is replaced, the check codes in the corresponding parity entry are also updated for the new line. Thereby, a hit in only the parity cache never occurs.

If the parity cache misses the entry for a data block being requested by the processor, the integrity checking cannot be processed. In this case, the check codes are computed from the data in the main cache and stored in a selected parity entry for future protection. If the accessed line has a corrupted item, error propagation is possible. Check codes generated using erroneous data do not help error checking. This event is denoted by *misconstruction*. Although the error does not always propagate, we assume the worst case and treat it as error propagation in the evaluation of the scheme.

The area estimate of the parity cache is obtained from an on-chip cache area model presented by Mulder et al. in [7]. They have used the technology independent notion of a register-bit equivalent or *rbe*. One *rbe* equals the area of a one-bit register storage cell that has the highest bandwidth. The static storage cell of medium bandwidth that we use here has been empirically determined to be 0.6 *rbe*. Thereby, an area represented in *rbe* for register cells is converted to static cell area by multiplying by a factor of 0.6.



(a) Conventional array (b) 4-way parity cache

Figure 3. Check code area model.

Figure 3 depicts the check code array of a conventional cache in the uniform structure and a k-way set associative parity cache ($k = 4$). We denote the total area of the two models by $\mathcal{S}_c$ and $\mathcal{S}_p$, respectively. Let W_{E} be the width of an element E and let CC represent the check code. The area is the sum of areas of all memory elements and is given by

$$\begin{aligned} \mathcal{S}_c &= \mathcal{A}(\mathsf{CC}\text{ for tag}) + \mathcal{A}(\mathsf{CC}\text{ for status bits}) + \mathcal{A}(\mathsf{CC}\text{ for data}) \\ &\quad + \mathcal{A}(\text{drivers}) + \mathcal{A}(\text{bitline sense amplifiers}), \qquad (1) \\ \mathcal{S}_p &= \mathcal{A}(\mathsf{CC} + ovhd_{\mathsf{CC}}) + \mathcal{A}(\text{tag} + ovhd_{tag} + \text{status} \\ &\quad + ovhd_{status}) + \mathcal{A}(control), \qquad (2) \end{aligned}$$

where $\mathcal{A}(\mathsf{M})$ and $ovhd_{\mathsf{E}}$ denote the area of module M and the overhead for element E, respectively. The $ovhd_{\mathsf{E}}$ includes comparators if any, drivers, and sense amps. In the implementation of MIPS-X [1], $W_{compare}$, W_{driver}, and $W_{sense\text{-}amp}$ are approximately 6 *rbe* each. For a static cell array of $l \times W_{\mathsf{CC}}$ bits (Figure 3a), the total size (Eqn. (1)) is

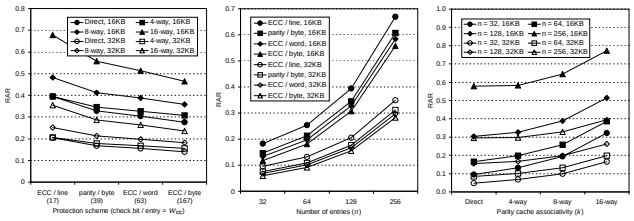
$$\begin{aligned} \mathcal{S}_c &= 0.6(W_{\mathsf{CC}} + W_{driver})(l + W_{sense\text{-}amp}) \\ &= 0.6(W_{\mathsf{CC}} + 6)(l + 6), \end{aligned}$$

where $W_{\mathsf{CC}} = W_{\mathsf{CC}\ for\ tag} + W_{\mathsf{CC}\ for\ status\ bits} + W_{\mathsf{CC}\ for\ data}$.

The control logic for the parity cache can be implemented in a programmable logic array (PLA) as a part of the main cache controller. A PLA of 130 *rbe* is presumed according to [7]. From Eqn. (2), the area of a k-way parity cache of n entries for $k \neq n$ (Figure 3b) is obtained as

$$\begin{aligned} \mathcal{S}_p &= 0.6(W_{\mathsf{CC}} \cdot k + 6)(\frac{n}{k} + 6) + 0.6\{(W_{status} + W_{tag}) \cdot k \\ &\quad + 6\} \cdot (\frac{n}{k} + 6 + 6) + 130 \\ &= 0.6n\left(\Delta_{\mathsf{CC}} \cdot W_{\mathsf{CC}} + \Delta_{tag}(W_{status} + W_{tag})\right) + 195, \qquad (3) \end{aligned}$$

where W_{status} = LRU bits + valid bit + parity for the tag = $\log_2 k + 1 + 1$, $W_{tag} = \log_2 l - \log_2 n + \log_2 k = \log_2 \frac{l \cdot k}{n}$, $\Delta_{CC} = 1 + \frac{6 \cdot k}{n} + \frac{6}{W_{CC} \cdot k}$, and $\Delta_{tag} = 1 + \frac{12 \cdot k}{n} + \frac{6}{(W_{tag} + W_{status}) \cdot k}$.



(a) n = 128 (b) k = 4 (c) W_{CC} = 63

Figure 4. Relative area requirement.

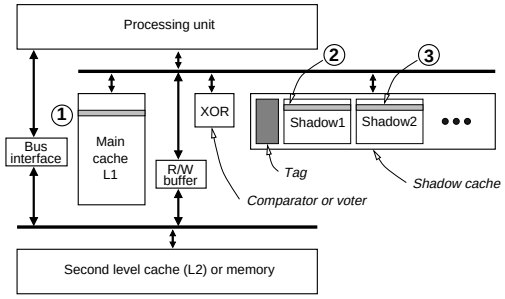
To compare the areas of the two organizations, we compute relative area ratio[1] (RAR). The RAR for parity caching equals $\frac{\mathcal{S}_p}{\mathcal{S}_c}$. Figure 4 plots the RARs for various sets of configuration parameters: check code type, data unit size, number of parity entry, and associativity. Considering current microprocessors, two main cache sizes, 16KB for instruction cache (I-cache) and 32KB for D-cache, are compared. However, some microprocessors employ larger caches. In that case, the RARs become even better for parity caching. The number of check bits per entry corresponds to W_{CC}. Four values are compared, representing different protection capabilities. Several conclusions can be drawn. An increase in check code width results in a decrease in the RAR. Obviously, more overhead is required for higher associativity and the RAR is proportional to the number of entries. The parity cache with an RAR of less than 1 is of interest to us. The corresponding protection coverages for these organizations are given in Section 5.

3.2. Shadow checking

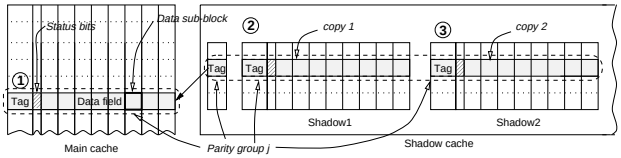
For applications that require very high data integrity or operate under highly noisy environments, parity- and ECC-based protection cannot be satisfactory. One general approach in this case is to use replicated architecture such as N modular redundancy (NMR) with majority voting, but it is very expensive. Instead of full replications, we present an alternative approach, called *shadow checking*, where multiple copies are partially supported to meet a budgeted area which is not adequate for a complete NMR. The copies of the MFU lines are stored in *shadow cache*. The underlying idea is the same as parity caching, but the shadow cache performs error checking by means of comparison using the copies of data rather than check codes. The goal is to obtain a high reliability enhancement even in the presence of multiple-bit errors with smaller chip area overhead.

Figure 5a shows the diagram of shadow checking architecture. Depending on space availability, N identical additional cache modules, called *shadow* i for $1 \leq i \leq N$, are included in the shadow cache. We adopt the same address mapping mechanism used in parity caching. Figure 5b depicts a parity group, j, consisting of a shadow cache tag and

[1]We also use this metric for shadow checking and selective checking.



(a) Basic organization



(b) Components of parity group

Figure 5. Shadow checking architecture.

N copies of information, each of which contains tag, status, and data bits. The shadow cache operates like a shadow of the main cache. Data written into the main cache is also written into the shadow cache along with the corresponding tag and status bits in parallel. Error checking is performed on read hits in both caches, and its effect depends on the number of erroneous *shadow* modules and their error patterns. This is equivalent to known reliability gain in an NMR system [12]. Thus, we do not discuss it here.

We show the advantages of shadow checking over full replication with respect to chip area requirement and resultant reliability enhancement. In the comparison, we ignore the common factors of the two architectures such as comparator/voter reliability and delay, and synchronization cost. Figure 6 shows the RARs of a shadow cache of two shadow modules in comparison with a triple modular redundant (TMR) cache using the same area model presented in Section 3.1. The RARs are smaller than those of a parity cache for the same parameters due to the higher overhead of a conventional TMR system.

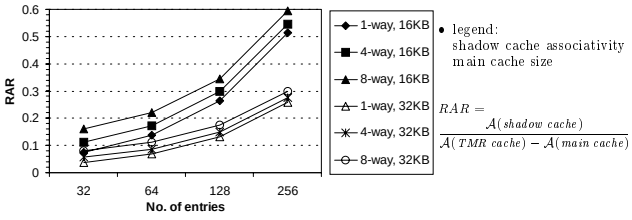


Figure 6. Relative area ratio.

3.3. Selective checking

Parity caching and shadow checking have been proposed as alternative architectures to the uniform check code structure and replication method, respectively. If the main cache has k-way ($k \geq 2$) set associativity, we can also configure

redundancy in a simpler manner. Out of k lines per set, only s lines ($1 \leq s < k$) are selected to assign the check codes. Similar to the previous two schemes, line selection is based on the access frequency. We simply choose the most recently used (MRU) lines of each set for error checking with the expectation that those lines are MFU, and thus error-prone and likely to be accessed in near future. We call this scheme *selective checking*. It is obvious that the RAR for selective checking is approximately $\frac{s}{k}$.

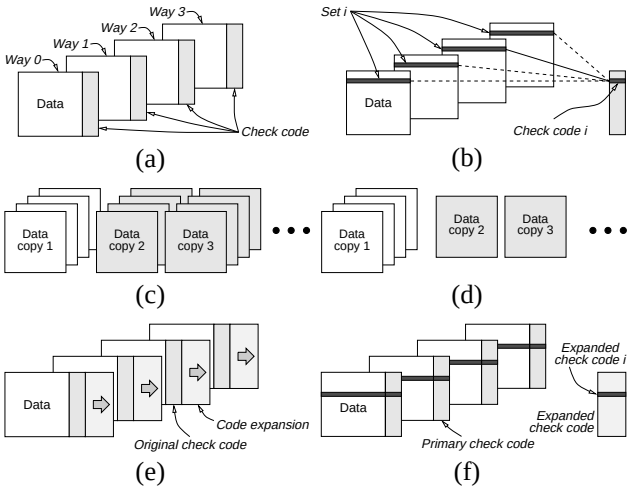


Figure 7. Uniform vs. selective organization.

Figure 7 depicts a comparison of redundant code organizations between conventional approaches (left column) and selective checking (right column) for $k = 4$ and $s = 1$. Many commercial microprocessors use byte-parity or SEC-DED codes in the uniform structure as shown in Figure 7a. Alternatively, in selective checking with $s = 1$ (Figure 7b), for Set i, only the MRU line is protected by a check code. (If $s \geq 2$, s MRU lines are guarded.) Each check code entry is independently assigned to any line of a set while keeping track of the MRU. Similar concepts can be applied to the NMR cache (Figure 7c) for a selective NMR (Figure 7d) where N copies are maintained for only the MRU portions. In the case of a miss, a line is fetched from memory with check code and it becomes the MRU line. Whenever the MRU line is requested, the cache controller recognizes that the current check code belongs to the MRU and performs error correction. As a result, no tag and resultant overhead are necessary to achieve dynamic mapping.

The reliability of a cache that is already equipped with a check code can be further enhanced in many ways. One approach is to expand the check code with more check bits as shown in Figure 7e. The new wider check code has a higher checking capability. In case the full expansion is not affordable, we can adopt the selective structure here. Only s additional code entries for each set are provided to enhance the protection of the MRU lines. Figure 7f shows the selective expansion for $s = 1$. The combination of primary and expanded check codes is called *enhanced check code*. The primary check code is separable from the enhanced check code. The expanded code is built in such a way that the chosen line is protected by more intensive checking in conjunction with primary check code. Any non-MRU line of a set turns into a new MRU line whenever it is accessed. In this case, only the checking by the primary code is valid. The expanded code is ignored and is replaced by the code for a new MRU line. Other basic operations of enhanced check codes are the same as for the previous cases.

In selective checking, the redundant code entry is constructed for only s ($< k$) lines per set. Thus, the redundant codes are always evenly distributed over different sets irrespective of their usage frequency. On the other hand, the parity and shadow caches maintain the redundant code entries for the MFU lines in the range of the entire main cache. This results in differences in cost and protection coverage of the selective checking compared to the other two schemes.

3.4. Cache scrubbing

For most programs, less than 30% of instructions are memory references. The D-cache is occupied during the executions of those instructions. Depending on the processor architecture, some D-cache cycles may be idle. To further enhance the data integrity, we can scrub off the latent errors in the D-cache whenever possible. Soft error scrubbing is accomplished by reading out the data and check bits, verifying their correctness, and writing back the corrected data [11]. Scrubbing is more advantageous to caches protected by a low capability check code.

Since in our proposed schemes we shrink the check code array, we suggest the use of the cache scrubbing technique to increase the protection coverage. On every idle cycle, the cache controller executes a single scrubbing cycle using an entry from the check code array and its corresponding line in the cache. One question that arises here is how to pick a line for scrubbing. Random selection is the easiest method to implement but performance may be poor. Intuitively, it could be beneficial to check the lines whose check codes are expected to be discarded soon to make room for new codes. These can be the LRU lines in consideration of temporal locality. By also taking spatial locality into account, lines away from the MRU line and their neighboring lines can also be selected.

4. Error Model and Evaluation Methodology

For the evaluation of the schemes, we employed a trace-driven simulation combined with software error injection. An error injection process inverts a single or multiple bits in any field of a selected line. To reflect diverse possible fault manifestation patterns, we conducted error injection based on the following four error models.

1. For a cache item access, the mapped line/set in the RAM is activated and probed. Any fault during the access can result in errors in the line being accessed. Thus, a higher error probability is expected in more frequently accessed lines. Under this error model, the error injection is executed in the target line right after the access. We call this *direct injection*.

2. Cross-coupling effects of faults can generate errors in the adjoining lines of the currently accessed line. During a line access, an error is injected in a neighboring line on either side. We call this *adjacent injection*.
3. Independent of line access, external disturbances and single-event upsets can generate soft errors in any location at any time. To simulate this occasion, an error is injected in a random location at a random time. This is called *random injection*.
4. Unlike previous models, faults can cause errors in a group such as column, row, or cell cluster. Only column errors can induce a performance difference between the conventional and proposed architectures. Thus, we include a model, called *column injection*, where an error is injected in every row of a selected column. This model is added to examine the schemes' performance in the worst situation.

All error models apply to both the main and protection cache or array. The accuracy of error models depends on the nature of the physical faults. Fault behavior and the distribution of different types of faults are likely to vary depending on the operational environment. Therefore, it is very difficult to judge which error model is dominant and realistic in a general situation. For this reason, we carried out a set of simulation experiments for each model separately.

Parameters	Main cache	Parity/Shadow cache
Size	I-cache: 16KB, D-cache: 32KB	256 check code entries
Associativity	Direct-mapped	4-way set associative
Replacement	Least recently used (LRU) line first	
Write policy	Write-through, write around	
Line size	32 Bytes	32 Bytes (shadow)

Table 1. Base cache parameters

The simulations were performed on-the-fly and every operation was handled on a clock-by-clock basis, assuming that a single instruction is issued and finished in every clock cycle on a perfectly pipelined processor. The same protection scheme was applied to both the I-cache and D-cache on each simulation run. Table 1 lists the base cache parameters used for the simulations unless specified otherwise. All programs of SPEC95 suite [16] were instrumented on the Sun Ultra1 model using Sun's Shade tool V5.32C [2]. Table 2 shows input files and memory access rates of the programs and hit rates on the base caches. These benchmarks provide a range of computation and memory access patterns, and their instruction and data sets are much larger than the simulated cache sizes. All benchmarks' executable files were built using Sun WorkShop Compilers, `cc` and `f77`, with the optimization flags `-fast`, `-xO4`, and `-xdepend`.

The moment at which the decision of an error injection is made is defined as an *injection decision point* (IDP). For direct and adjacent injection models, the end of each read/write access cycle was considered as an IDP, while the trailing edge of every CPU clock cycle was used for random and column injections. At each IDP, an error (multiple errors for the column injection) is injected with a constant probability, which is set to 10^{-6} for direct and adjacent, 0.2×10^{-6} for random, and 0.5×10^{-8} for column injection. These are accelerated rates for the rare events. If an item selected for the error injection already has an error, no additional error is injected. The number of error injections, I, at N IDPs is a binomial random variable and the error injection rate is I/N. To ignore initial warm up routines, the error injection was started after the first 10 million instructions while the caches operated under normal conditions. Errors were injected independently for the next 500 million instruction executions and no injection was performed afterward. In consideration of latent errors, the simulations were terminated after the following 100 million instructions.

Benchmark	Input file	Load (%)	Store (%)	I-cache (%)	D-cache (%)
compress	`bigtest.in`	7.20	2.16	99.99	84.68
gcc	`cp-decl.i`	19.34	5.52	96.40	93.83
go	`9stone21.in`	18.79	6.77	97.74	93.60
ijpeg	`vigo.ppm`	17.03	6.60	99.91	89.74
li	`*.lsp`	20.89	9.89	98.61	93.66
m88ksim	`ctl.in`	17.17	9.95	97.47	98.33
perl	`primes.in`	26.03	12.41	96.08	95.76
vortex	`vortex.in`	18.74	8.47	95.10	91.23
SPECint95		**18.15**	**7.72**	**97.66**	**92.60**
applu	`applu.in`	25.43	11.40	99.99	86.25
apsi	`apsi.in`	28.81	12.83	99.76	88.57
fpppp	`natoms.in`	38.21	11.04	93.66	96.69
hydro2d	`hydro2d.in`	21.65	9.28	99.52	75.03
mgrid	`mgrid.in`	38.29	19.84	99.99	95.68
su2cor	`su2cor.in`	22.24	7.23	97.11	91.52
swim	`swim.in`	24.34	10.35	99.99	79.19
tomcatv	`tomcatv.in`	22.20	7.74	97.47	92.57
turb3d	`turb3d.in`	17.16	12.24	99.90	93.25
wave5	`wave5.in`	19.33	10.26	97.95	84.01
SPECfp95		**25.77**	**11.22**	**98.53**	**88.28**

Table 2. Summary of benchmarks

The performance comparison targets for the proposed schemes are uniformly organized check code and replication. The number of error bits per injection is not an important factor in the comparison because the same capability of the unit protection code is assumed. Only the number of code entries is different. The parity cache was implemented with a SEC-DED code, and single-bit errors were injected. D-cache scrubbing was tested along with parity caching. A shadow cache of two shadow modules was compared with a general TMR cache. To simulate a harsher environment for shadow checking, multiple-bit errors were used. Although we proposed three organizations for selective checking, the main idea of the three is common. Therefore, we investigated only a simple case where only s entries of SEC-DED codes are maintained for each main cache set (Figure 7b) with single-bit error injection.

Our main interest is how many injected errors propagate to other components under the proposed schemes. For a quantitative performance measurement, we use error propagation rate (EPR), defined by

$$EPR = \frac{\textit{total number of errors propagated}}{\textit{total number of errors injected}} \times 100.$$

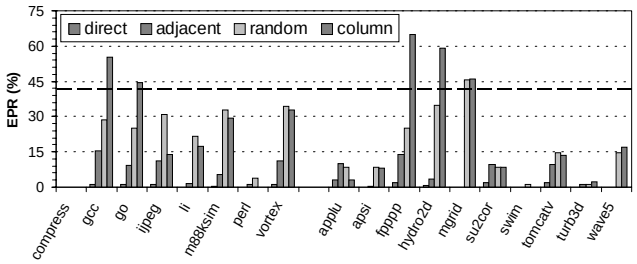
5. Results and Analysis

This section reports the performance of the three proposed architectures. Instead of presenting the simulation

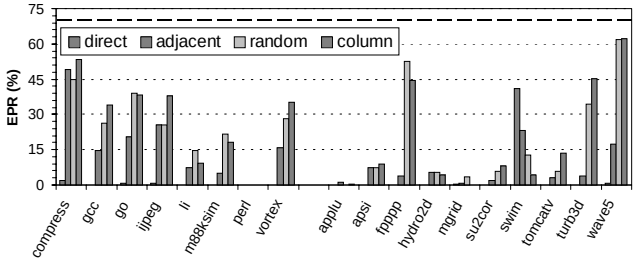
results in an exhaustive manner for all parameters, we focus only on the salient features to gain insight into how the parameters affect the protection capabilities of the schemes.

5.1. The performance of parity caching

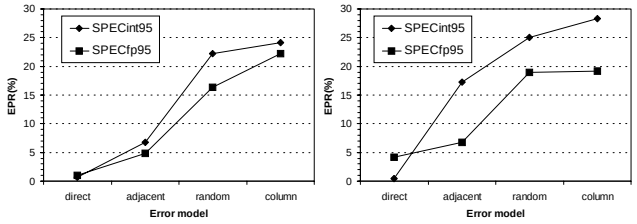
Recall that single-bit errors in cache protected by SEC-DED code in the conventional uniform manner cannot propagate (i.e., EPR = 0%). Figure 8 shows the EPRs under the protection of two independent parity caches whose RARs are 0.58 and 0.30 for checking the I-cache and D-cache, respectively. The results of the four error models are compared. If we assume that error propagation is equally likely to occur in all locations, the expected EPRs with the check codes of these areas would be 42% and 70% (identified by thick dashed-lines in the figures), respectively. However, in most cases, the EPRs are much lower than these values because the distribution of error propagation is not uniform. Organizing the check codes in a cache makes the most of the budgeted area to prevent errors from propagating.



(a) On I-cache



(b) On D-cache



(c) Average EPR on I-cache (d) Average EPR on D-cache

Figure 8. Error propagation rate (EPR).

Under the direct error injection model, for all benchmarks except *swim*, the small parity cache allows less than about 3% of injected errors to propagate. This is because this error model and the applications match well with the premise on which parity caching is developed. Due to spatial locality, the parity cache also provides relatively high protection coverage in the adjacent error model. However, we observe larger EPRs on the D-cache for *compress*, *ijpeg* and *swim*. One common attribute of these benchmarks is that their data access does not show good locality, as can be ascertained from their low hit rates in the D-cache given in Table 2. The hit rate in the parity cache is even lower. Thus, more items in the D-cache, whose check codes are not present in the parity cache, can be requested. In this case, error propagation takes place unless the items are error-free.

Even if error occurrence is evenly distributed (i.e., random error model), localized error propagation makes the parity cache area efficient. In the column error model where every line gets an error injection, the results are not very promising. Once a column error injection is executed, any read miss in the parity cache after that results in an error propagation. The column error injection is the worst case test model. Nevertheless, if we consider the area occupancy, on average parity caching provides more protection coverage with a given area as shown in Figures 8c and 8d.

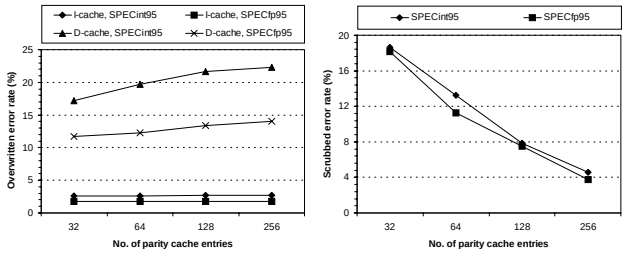
Cache type		I-cache				D-cache			
	No. of entries	32	64	128	256	32	64	128	256
	RAR	**0.13**	**0.20**	**0.33**	**0.58**	**0.07**	**0.10**	**0.17**	**0.30**
Benchmark	compress	0.00	0.00	0.00	0.00	8.33	2.08	0.00	2.08
	gcc	6.84	5.34	2.99	1.28	1.85	1.85	0.00	0.00
	go	7.59	6.29	3.04	1.08	3.91	0.78	0.00	0.78
	ijpeg	6.35	3.94	3.94	1.31	5.30	5.30	6.06	0.76
	li	5.38	1.79	0.22	0.00	0.00	0.00	0.00	0.00
	m88ksim	4.57	4.35	3.48	0.22	0.00	0.00	0.00	0.00
	perl	13.51	8.50	3.27	0.00	3.28	1.09	0.00	0.00
	vortex	12.33	7.71	2.86	1.32	2.36	0.00	0.00	0.00
	SPECint95	**7.07**	**4.74**	**2.47**	**0.65**	**3.13**	**1.39**	**0.76**	**0.45**
	applu	10.75	8.55	3.51	3.07	2.65	1.06	0.00	0.00
	apsi	5.42	3.65	0.65	0.00	14.16	12.79	4.11	0.00
	fpppp	6.64	5.72	4.35	1.83	13.71	5.65	1.61	0.00
	hydro2d	1.36	1.36	1.36	0.91	1.94	1.29	0.00	0.00
	mgrid	0.00	0.00	0.00	0.00	1.52	1.89	0.38	0.38
	su2cor	10.11	9.67	8.35	1.76	0.67	1.34	0.00	0.00
	swim	2.25	0.00	0.00	0.00	29.14	37.09	39.74	41.06
	tomcatv	8.22	8.22	7.31	2.05	2.40	1.80	0.00	0.00
	turb3d	4.55	1.73	1.08	0.00	17.53	5.19	0.65	0.00
	wave5	9.75	4.08	0.00	0.00	3.25	1.30	0.65	0.65
	SPECfp95	**5.90**	**4.30**	**2.66**	**0.96**	**8.69**	**6.94**	**4.71**	**4.21**

Table 3. EPR (%) vs. the number of parity entries

Table 3 gives the EPRs on the direct error model for an increasing number of check code entries along with RARs. Only 32 entries, which occupy 13% and 7% of the area needed for the uniform structure for the I-cache and D-cache, respectively, bring significantly high coverages. Again, this results from the fact that for many applications the cache access is localized to a very small region. Interestingly, *swim* exhibits a different attribute: as more entries are added, the EPR increases. For *swim*, it turns out that increasing the parity cache associativity is more beneficial than increasing the size. The EPR is reduced to 0.66% on the 16-way set associative parity cache of 256 entries.

Some errors can be removed by normal write operations. Figure 9a depicts the portion of overwritten errors out of total injected errors. In the I-cache, only an instruction miss generates a write, while a store request also causes a write in the D-cache. This is why the overwritten error rate is higher for the D-cache than for the I-cache. The results presented so far are collected from parity caching in combination with error scrubbing (D-cache only). In the case of fewer entries, errors eliminated by scrubbing account for a large portion of total eliminated errors as shown in Figure 9b. This is

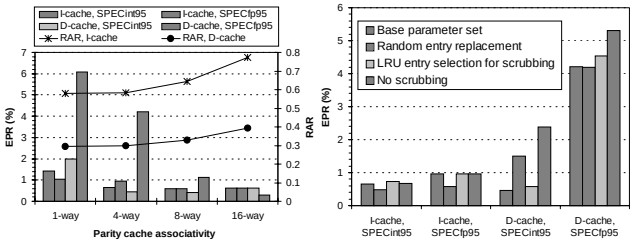
because the number of scrubbing cycles executed per check code entry is larger in a small parity cache. As the parity cache includes more entries, error removal at read access time becomes dominant for most applications.



(a) By normal operation (b) By scrubbing

Figure 9. Error removal.

We also present the average EPRs for other parameters. From Figure 10a we note that higher parity cache associativity enhances the error checking capability. However, the increase becomes insignificant with more than 8-way associativity. On the other hand, area requirement grows rapidly (RARs are plotted on the right vertical axis).



(a) Parity cache associativity (b) Base parameters

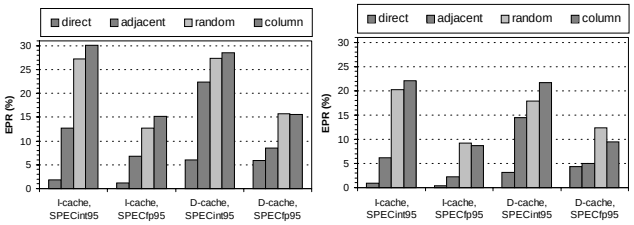
Figure 10. Effects of other parameters.

In Figure 10b, *base parameter set* consists of LRU policy for entry replacement, error scrubbing, and random entry selection for scrubbing. For performance comparison, three additional simulations were performed with only one parameter variation at a time. Due to the small number of check code entries, one may question if a simpler replacement can affect the coverage. We tested a pseudo-random policy. The LRU strategy performs slightly better on the D-cache for SPECint95. It is, however, a little less efficient than the pseudo-random policy in the other cases. From the results, we conclude that the replacement policy does not significantly affect the performance. In section 3.4, we discussed the entry selection issue for error scrubbing. As shown in the graph, the performance gain from LRU entry selection for scrubbing is insignificant. The results without scrubbing are also shown. Scrubbing mostly improves the coverage at the cost of hardware complexity.

Thus far we have discussed the effect of other parameters on the parity cache performance only in the case of the direct error model. However, similar effects of parameters were noted from the results of simulations under the other three models. We omit them here due to space limitation.

5.2. The performance of shadow checking

We have also conducted a set of simulations for shadow checking to investigate how replication architecture with unequal sized modules performs under the presence of soft errors following different error models. Errors were injected in the shadows as well as the main cache. Data items that are supposed to be identical under the normal condition were exposed to independent error injections and are compared for error checking.



(a) 4KB shadow (b) 8KB shadow

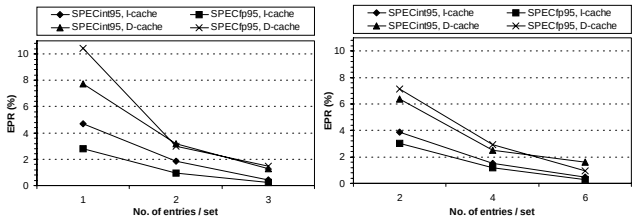
Figure 11. EPR under shadow checking.

Figure 11 shows the average EPRs under shadow checking with two shadows. The results for two shadow sizes are compared. Clearly, larger shadow misses fewer errors. Note that the performance variation among different error models are similar to the case of parity caching (Figures 8c and 8d). The RARs of the shadow cache with 4KB shadows are 0.3 and 0.15 for the I-cache and D-cache, respectively. In the case of 8KB shadows, the RARs are 0.55 and 0.38, respectively. Here, we also observe the EPRs in the direct model are very low, but the same is not the case for the other models. However, we still confirm that shadow checking is very area efficient in all cases. If the designer needs to enhance cache reliability against the types of errors that require replication, but only a small area can be budgeted, then the shadow cache is worth considering.

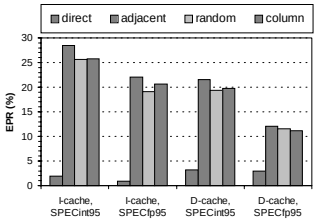
5.3. The performance of selective checking

Figures 12a and 12b show the relationship between average EPRs and the number of entries per set in the direct error model. With only half of the check code required for the uniform structure, EPRs of less than 4% are obtained. However, this is lower coverage than a parity cache of the same area can provide. The reason for this is that in selective checking, a line is selected to assign check codes within the scope of a set rather than the entire cache. Recall that an advantage of selective checking is that it only needs a simple modification to the conventional architecture.

From Figure 12c, unlike the first two checking schemes, we note that the protection coverage of two check codes per 4-way set varies very little under the three error models. This is also due to the fact that the check codes are managed independently for each set. Although EPRs are relatively high in the three models, they are still lower than 29%, which is much higher coverage than an intuitive expectation with about a half size check code array. This in-



(a) 4-way set (b) 8-way set



(c) Effect of four error models

Figure 12. EPR under selective checking.

dicates that our locality-based checking scheme efficiently uses the given check code area.

6. Conclusions

In conventional architectures, as the size of the primary cache grows, the redundant code for data integrity checking also needs to be increased proportionally. We have proposed new architectural solutions for the situations where enough area cannot be budgeted to support this uniform organization and further expansion of the protection code. In our schemes, check code can be designed for a given area in such a way that the most frequently accessed cache lines, which are likely to be the most error-prone and are likely to have the lowest error propagation latencies, take precedence in integrity checking over less frequently used lines. Prediction for line selection is performed by taking advantage of locality in cache accesses.

We have considered four possible error models and applied them to our simulated systems. From the simulation, we have found that with α% check codes of the uniform error checking architecture, the proposed schemes achieve far more than α% in error protection coverage. In particular, significantly low EPRs are obtained under the direct error model with a small area. We have also shown that adding the error scrubbing technique is more beneficial for a system with a small number of protection code entries. Parity caching and shadow checking schemes are more effective in the adjacent error model than selective checking. However, selective checking requires the simplest organization and management, and is thus easy to implement for multi-way set associative caches. Despite the unbiased error injection in time and location in the random and column models, our schemes that are tuned for the protection of the MFU lines are still area effective for such error models.

We have shown that our locality-based configuration schemes for the check codes can be adapted to current systems with a small overhead. An important advantage of the proposed architectures over the conventional uniform structure is the flexibility given to the system designer in planning cache systems of the desired capacity in terms of size and reliability. We have given an area estimate for the schemes based on an area model. In order to fully validate the benefit of the schemes obtained from controllability of area occupancy, the geometry of physical chip area in the VLSI design needs to be investigated further.

Acknowledgments: The authors would like to thank Johan Karlsson for his comments on our error injection model, Matt Virgo for verifying our trace analyzers, and the anonymous reviewers for providing useful comments. This work was funded in part by NSF Grants MIP-9630058 and MIP-9896025.

References

[1] P. Chow. *The MIPS-X RISC Microprocessor*. Kluwer, Boston, 1989.

[2] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. *Performance Evaluation Reviw*, 22:128–137, May 1994.

[3] M. Hamada and E. Fujiwara. A class of error control codes for byte organized memory systems-SbEC-(Sb+S)ED codes. *IEEE Trans. on Computers*, 46(1):105–110, January 1997.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 1996.

[5] H. Imai. *Essentials of Error-Control Coding Techniques*. Academic Press, San Diego, CA, 1990.

[6] J. Karlsson, P. Ledan, P. Dahlgren, and R. Johansson. Using heavy-ion radiation to validate fault handling mechanisms. *IEEE Micro*, 14(1):8–23, February 1994.

[7] J. M. Mulder, N. T. Quach, and M. J. Flynn. An area model for on-chip memories and its application. *IEEE J. Solid-State Circuits*, 26:98–106, February 1991.

[8] S. Park and B. Bose. Burst asymmetric/unidirectional error correcting/detecting codes. *Proc. Int'l Symp. Fault-Tolerant Computing*, pages 273–280, June 1990.

[9] D. Patterson, T. Anderson, N. Cardwell, R. Formm, K. Keeton, K. Kozyrakis, R. Thomas, and K. Yelick. Intelligent RAM (IRAM): Chips that remember and compute. *Proc. Int'l Symp. Solid-State Circuits*, pages 224–225, February 1997.

[10] J. C. Pickel and J. T. B. Jr. Cosmic ray induced error in MOS memory cells. *IEEE Trans. Nuclear Science*, NS-25:1166–1171, December 1978.

[11] A. M. Saleh. Reliability of scrubbing recovery-techniques for memory systems. *IEEE Trans. Reliability*, 30(1):114–122, April 1990.

[12] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation*. Digital Press, Beford, MA, 1992.

[13] A. K. Somani and K. Trivedi. A cache error propagation model. *Proc. Int'l Symp. Pacific Rim Fault Tolerant Computing*, pages 15–21, December 1997.

[14] J. Sosnowski. Transient fault tolerance in digital systems. *IEEE Micro*, 14(1):24–35, February 1994.

[15] P. Sweazey. SRAM organization, control, and speed, and their effect on cache memory design. *Midcon/87*, pages 434–437, September 1987.

[16] URL:. `http://www.specbench.org`.