2018 IEEE International Symposium on High Performance Computer Architecture

Don't Correct the Tags in a Cache, Just Check Their Hamming Distance From the Lookup Tag

Alex Gendler, Arkady Bramnik, Ariel Szapiro
Intel Israel Design Center
Haifa, Israel

Yiannakis Sazeides
University of Cyprus
Nicosia, Cyprus

***Abstract*—This paper describes the design of an efficient technique for correcting errors in the tag array of set-associative caches. The main idea behind this scheme is that for a cache tag array protected with ECC code, the stored tags do not need to be corrected prior to the comparison against a lookup tag for cache hit/miss definition. This eliminates the need for costly hardware to correct the cache tags before checking for a hit or a miss. The paper reveals the various optimizations needed to translate this idea into a design that delivers a practical improvement in a product. An analysis of our design, as compared to state of the art methods, shows that it can provide the same correction and detection strength with less area, power and timing overheads and better performance. An Intel Core® microprocessor is implementing this technique in its second level and third level caches.**

I. INTRODUCTION

Strikes from cosmic radiation and alpha particles commonly refer to as soft-errors [17], are one of the main sources of errors in modern microprocessors [1]. Although bit-flips due to particle strikes do not damage circuits permanently, they can cause incorrect functionality because the use of corrupt values during a program execution may cause a program to crash or, even worse, produce a wrong output without any warning. Microprocessor vendors set soft error rate (SER) targets to ensure that a product can satisfy expected error rate requirements in different market segments [15].

Remaining within a SER rate budget represents an arduous design challenge that entails performing intertwined trade-offs between performance, power, area and SER for different processor units. Depending on a unit's SER contribution and cost to lower its SER, the unit may remain unprotected, be augmented with only error detection or enhanced with both error detection and correction.

Modern microprocessors employ multi-level cache hierarchy to provide high performance [4][13]. A cache is built with arrays that contain data and tag (address) information. Both data and tag correctness is necessary for correct functionality of a processor. The use of corrupt data, due to an error in a data array or tag array, can compromise correctness. Large arrays in the cache hierarchy, such as data and tag arrays in L2 and L3 caches, are typically protected with Error Correcting Codes (ECC). One of the most popular and cheapest ECC schemes that ensures acceptable product SER rate in many cases is the single-error-correction-double-error-detection (SECDED) [5].

The use of ECC techniques, however, is not free. It increases area and power and may increase cycle time or access latency and inevitably increase cost and lower performance. In the case of caches, the use of ECC can negatively impact performance as it can increase the read critical path for accessing the tag and data arrays [21][14]. Therefore, finding a cost efficient way to use ECC that avoids an increase in the cache access latency can potentially lead to a performance gain for the whole microprocessor.

Even though, both tag and data arrays are vulnerable to SER, the vast majority of publications on cache error protection focus exclusively on data array protection. This may seem plausible because data arrays are bigger and protection techniques for data are directly applicable to tag arrays. However, tags mainly serve a different purpose from data: detect matches against lookup addresses versus storing and loading data. This makes tags amenable to a different class of efficient error protection that is not applicable to data.

Specifically, our work presents how to use the fast-tag hit algorithm [20][18] to drastically reduce the overheads of tag array ECC while preserving its correction and detection strength. The main idea behind this technique is that a cache tag, protected with an error correction code, can determine if a cache access is a hit or a miss by only checking what its Hamming distance from the lookup tag is. This realization does away the expensive hardware needed for correction and, consequently, helps reduce area, power overheads, relax timing pressure and improve performance. But the gap between a concept and its implementation is wide. This paper bridges this gap while making the following contributions:

- reveal the optimizations employed in a product design to turn the fast-tag hit concept to a scheme with practical value,
- propose a novel key latency reduction optimization: use fast speculative hit/miss definition that can only be incorrect in the case of uncorrectable tag errors that are detected non-speculatively but slower,
- introduce the speculative Tag ECC approach, and
- illustrate the area, power, timing and performance benefits of fast-tag hit with product grade tools.

A clear testament of the advantages enabled by fast-tag hit ECC is its implementation — based on the design presented in this work — by Intel in the second level cache and the third level cache of a Core® microprocessor.

The rest of the paper organization is as follows. Section II discusses related work. Section III provides background information on SECDED ECC codes and cache organization. Section IV reviews the flow of a cache access with an ECC protected tag array and presents state of the art tag ECC schemes. Section V explains the fast-tag hit algorithm while Section VI introduces a specific implementation of it. The methodology used for evaluation purposes and the results of

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00055

this analysis are presented in Sections VII and VIII respectively. Other uses of the proposed scheme are discussed briefly in Section IX. The paper concludes in Section X.

II. Related Work

As far as we know, the basic concept of fast-tag hit has first appear in published literature in two papers [20][18]. The work in [20] presents the basic fast-tag hit idea and proposes to use saturating adders to implement the 1's count function needed to measure the Hamming distance between lookup and stored tags. The paper uses gate counts to compare the proposed scheme against traditional Tag ECC. The paper by [18] also introduces the idea of fast-tag hit and shows how to use it for both parity and SEC protected tag arrays. The paper performs an evaluation using 40nm commercially available libraries, but it does not specify what comparator it uses for determining the Hamming distance.

Our work goes beyond these efforts in a number of ways. It presents a detail explanation about the design and implementation of an instance of fast-tag hit in a product. This includes discussion about various optimizations employed to bridge the gap between an idea and a practical implementation. The paper elucidates the benefits from the optimizations as well as to why their use does not create correctness issues. In particular, it presents a fast and efficient design for a 1's count circuit based on carry-save-adders and explains the operation and benefits from using a fast speculative hit/miss definition in conjunction with slower double error detection that is off the critical path. It presents and compares, using product grade tools at 14nm technology, a variety of tag ECC schemes that help highlight the benefits of fast-tag hit algorithm and the significance of the efficient implementation of the 1's count circuit. The paper also discusses the implications of having up to two errors per tag as well as the proposed methods ability to mask errors that are certain misses. The paper explains how to use fast-hit to implement both non-speculative and speculative Tag ECC approaches. As far as we know this is the first work to present the speculative Tag ECC approach. This approach resembles an earlier proposal that suggests forwarding data (not tags) from caches speculatively to hide correction latency [2].

Another related work [12], proposes to measure Hamming distance in two stages using multiple butterfly formed weight accumulators based on half-adders instead of saturating adders as in [20]. The scheme in [12] measures separately the Hamming distance for tag and ECC bits, to avoid increasing the access time when the generation of the ECC bits for the lookup tag lies in the critical path. This assumption is pessimistic as it is not representative, at least for the cache designs we use, where the delay for generating the ECC for the lookup tag can overlap with the tag array access. We do not compare directly against the method proposed in [12], as it is not applicable to our cache designs, but to a variation that measures the Hamming Distance of tag and ECC bits together using butterfly weight accumulators.

One small thread of published work on tag protection, unrelated to fast-tag hit, leverages the similarity across tags to provide error protection [22][11]. Another work proposes to use a two-tier approach for when to check for tag errors that is triggered when observing a likely anomaly [8].

Admittedly, there is not much research with focus on protecting cache Tag arrays against errors. One of the paper aims is to increase awareness about the problem and its unique characteristics and stimulate further research in this area.

III. Background

This Section reviews some basic information on SECDED ECC and Cache organization.

A. SECDED ECC

The following discussion shows how to define a SECDED ECC code. Most of this discussion comes from [14] and is relevant to a Hamming based ECC code.

In general, a SEC (Single Error Correction) code length, number of information symbols (data bits) and number of parity-check symbols (ECC bits) needs to satisfy the following two equations:

$$k \leq 2^m - m - 1$$

$$n = m + k$$

where n is the code length, k is the number of information symbols and m is the number of parity check symbols. An ECC code can be uniquely defined by its check or generator matrix.

The check matrix, H, is used to produce a codeword's syndrome. A syndrome is used to check the integrity of the codeword. It is obtained by performing a product of the H matrix with the codeword. The parity-check matrix H of a code consists of non-zero unique m-tuples as its columns. In a systematic form, the columns of H are arranged in the following form:

$$H = [Im\ Q]$$

where Im is an mXm identity matrix and the sub-matrix Q consist of k columns which are m-tuples of weight 2 or more. The columns of Q may be arranged in any order without affecting the distance property and weight distribution of the code.

The generator matrix, G, is used to produce an n-bit codeword given k information bits. In a systematic form, the generator matrix of the code is

$$G = [Q^t\ Ik\ ]$$

where Q^t is the transpose of Q and Ik is the kXk identity matrix.

The check and generation operations require determining the parity for different subsets of the bits in the codeword and information symbols. The decoding procedure takes as input an m-bit syndrome and produces an n-bit vector that is set at a given position when the codeword has error at that position.

Extending a SEC code to SECDED requires an additional parity check bit as compared to SEC.

The relative overhead of ECC bits diminish with increasing number of information symbols. For a SECDED code, the overhead is 62.5%, 37.5%, 21.9% and 12.5% when the information symbols are 8, 16, 32 and 64 bits respectively.

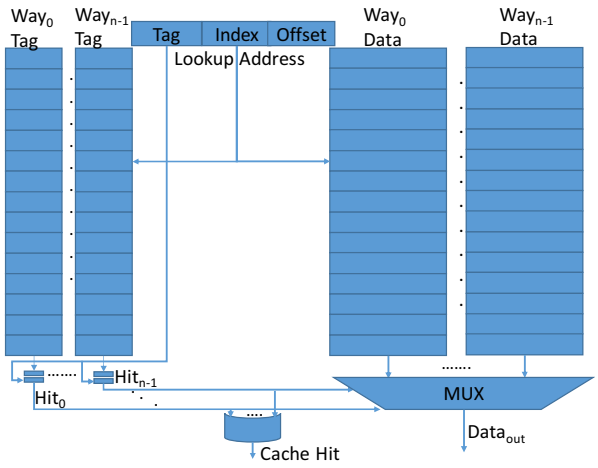

Fig. 1. A Set-Associative Cache Organization

B. *Caches*

The memory subsystem in a modern microprocessor typically consists of a multi-level cache hierarchy with the different levels usually refer to as Level 1 (L1), Level 2 (L2), Level 3 (L3), etc. Each cache is usually implemented as a multi-way set associative memory structure containing tag and data arrays. The tag arrays contain entries with address information whereas the data array entries lines of data. Both arrays have the same number of sets and ways. A tag array entry, in a given set and way, contains the address information for the corresponding line in the data array. The high-level structure of multi-way set associative cache is presented in Fig. 1.

For cache search purposes, a lookup address is divided into three fields: Tag, Set and Offset. Usually a cache line size is wider than single byte and the offset field represents the index of a single byte inside the cache line. The set field corresponds to the set index of an address in the cache. The tag field is part of the address stored in a cache way. During a cache read cycle, all the valid tags stored in the cache set, corresponding to the lookup address set, are compared with the lookup address tag field to check for a hit/miss definition: if required data exists in any cache way. Comparators generate the per way hit/miss indication. An OR of all way hit signals will generate a cache-hit signal that indicates if the required data is in the cache.

There are three basic cache flows in a set associative cache: read, write and replacement. For a cache read, the way hit signals control a way multiplexer (denoted as MUX in Fig. 1). Activating a way hit signal will transfer the read data of its corresponding way to the cache output. Modern microprocessors typically do not employ caches that read all data cache ways of a certain set in parallel due to the big power cost. A way hit signal is typically used as read enable for certain data cache way. In either case, the timing of hit/miss signal is very important for performance since it influences the cache read latency.

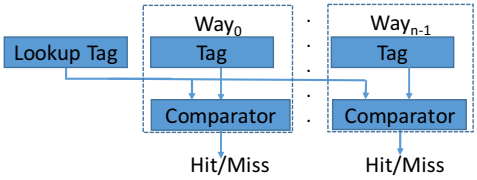

Fig. 2. Tag Array with no Error Protection

A cache write is similar to a read with the difference that data are written in the way of the cache set with the matching tag instead of reading data from it.

Replacement is performed when a new line needs to be written into a cache. This requires determining in which way to write the new data using a replacement algorithm, e.g. Least Recently Used. For the case of a writeback cache, if the selected way contains modified (dirty) data then eviction of this data is required before placement of the new.

IV. TAG ECC PROTECTED CACHES

This Section presents two state of the art approaches for protecting the tag array of a cache against errors.

We like to note that the designs discussed in this Section and the rest of the paper do not necessarily represent actual implementations — not revealed for IP protection — but are sufficiently representative to appreciate trade-offs between the different presented schemes.

Fig. 2 shows for an unprotected tag array how the tags of a selected set in the array are used to produce per way hit/miss signals by comparing the lookup tag against the stored tags in the set.

A. *Baseline Non-Speculative Tag ECC Protection*

The first baseline tag array ECC protection is shown in Fig. 3. This scheme is what is typically used for data ECC protection [14] and includes the following stages per way:

- Read tag and ECC bits from the tag array
- Use a checker to calculate syndrome corresponding to the read tag and ECC bits
- A decoder takes the syndrome and produces a repair vector that indicates which bit (if any) position needs to be repaired
- The error signal unit uses the generated syndrome to determine if there is
  - no tag error (NE)
  - a correctable tag error (CE), the tag error it meets the ECC correction strength, e.g. a single bit error is detected by SECDED ECC. A correctable error signal may be generated to the system for error logging purposes [9][10]
  - the detected error exceeds the ECC correction strength. The information about tag corruption will be sent to the system for further treatment. For

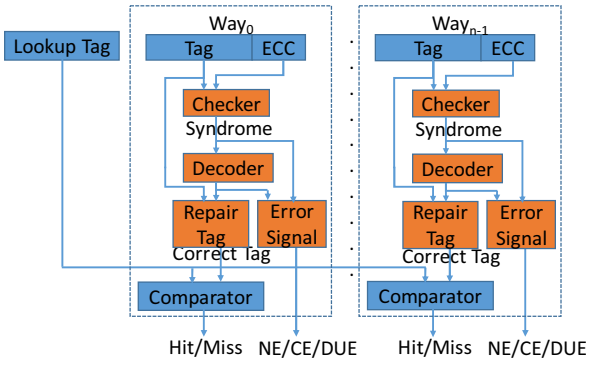

Fig. 3. Baseline Non-speculative Tag ECC

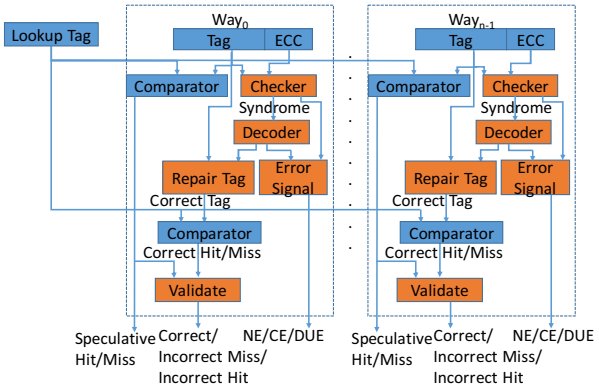

Fig. 4. Speculative Tag ECC Scheme

example, to initiate system shutdown in the case of modified data (a detectable-unrecoverable-error (DUE)), or fill in a new line in the place of corrupted line and continue normally (in the case of non-modified data).

- In the case of a correctable error, the repair unit flips the tag bit at the position indicated by the repair vector and restores the correct tag. The tag in the array is not updated with the restored tag.
- The tag read from the way (and possibly restored by ECC) and lookup tag are compared to produce the per way hit/miss signal.

Obviously, this baseline implementation of a tag array requires costly ECC check and correction hardware per way. Such hardware implements functionality similar to that presented in Section III.A. Furthermore, adding ECC increases power since each access requires error checking and decoding per way. What is more, the ECC functionality lies in the read critical path for hit/miss definition that may increase cache access latency. Next, we present an alternative approach aiming to remove the latency overhead added by ECC.

B. Baseline Speculative Tag ECC Protection

A tag ECC approach with potentially higher performance than the baseline non-speculative ECC protection scheme is shown in Fig. 4. It includes two paths for hit/miss definition: a fast but speculative, same as with the unprotected array, and a second slower one, similar to the baseline tag ECC protection. Although we treat this approach as prior art, we note that to the best of our knowledge speculative Tag ECC has not been discussed in prior literature and it represents a paper's contribution.

The fast-path provides a speculative hit/miss definition with the same latency as the unprotected array in Fig. 2, and, consequently, in the absence of errors the data can be forwarded with the same latency as with no-protection.

The slow-path validates whether the speculative hit/miss definition is correct. The slow-path includes the same stages at the baseline non-speculative protection scheme plus an additional stage for validating the speculative hit/miss definition. If no-error is detected, then the speculative hit/miss definition is correct. In the case of a correctable error, there are four validation outcomes:

- Speculative hit – actual hit: no action is needed and no performance penalty. This can happen when the error is in one of the ECC bits
- Speculative miss – actual miss: no action is needed and no performance penalty. This can happen when the lookup tag and the stored tag with the error are different both prior and after the stored tag correction. Therefore, the speculative miss definition is correct
- Speculative hit – actual miss: a recovery action is needed to avoid corruption of architectural state from using wrong data. Execution resumes normally after recovery.
- Speculative miss – actual hit: a recovery action is needed to avoid the corruption from line duplication. The missed line already exists but because of the error, a fill request is issued for it. Execution resumes normally after recovery.

In the last two cases, by the time the speculative hit/miss definition is reversed, some actions may have been performed based on the incorrect hit/miss signals. These actions may lead to architectural state corruption. Consequently, speculative tag ECC needs extra resources in the microprocessor to track the progress of actions performed speculatively in the various processor paths because of speculative hit/miss definition. In the case of mispeculation this work needs to be annulled, e.g. through pipeline and buffer flushing [2], before resuming execution.

The behavior in the case of an uncorrectable error can be identical with the baseline non-speculative ECC technique.

The speculative approach presents some interesting trade-offs. It is able to provide the same ECC strength as the non-speculative ECC scheme. In the common case, with no error, it has the same latency and performance with the unprotected case. However, as compared to the non-speculative scheme, this comes with an increase in area due to the extra resources needed for validation. The power, of this scheme is worse than the unprotected scheme but better than the non-speculative ECC. The slow path is activated only when the checker produces a syndrome that indicates an error. In the non-speculative scheme, each access goes through the checker, decoder and repair units. Unfortunately, the speculative scheme adds significant design complexity as it requires resources for tracking and annulling speculative work. Furthermore, in the case of a correctable error the

mispeculation penalty can be considerable spanning many cycles. Although, hard-errors are rare phenomena, when they occur in conjunction with the speculative ECC scheme, they may degrade performance considerably. This may happen when a frequently accessed set includes a tag with a correctable hard error. Similarly, a correctable soft-error in a frequently accessed line it will cause performance degradation until the line gets evicted or modified (updated). Note that for the non-speculative Tag ECC, the hit/miss definition path and latency without errors is the same as in the case with correctable errors.

The two state of the art ECC approaches presented in this Section have their pros and cons and a designer may choose either depending on which offers best return-on-investment for a given design. The next Section presents a technique that can be used with both non-speculative and speculative Tag ECC and enables superior in almost every respect schemes as compared to the ones in Section IV.

V. FAST-TAG HIT THEORY AND ALGORITHM

This Section reviews the fast-tag hit algorithm. First, the algorithm is derived from coding theory and then is used to construct both non-speculative and speculative tag ECC schemes. Afterwards a specific instance of fast-tag hit is presented for a non-speculative SECDED ECC. This discussion is partially based on the explanation of the fast-tag hit in [20][18]. The discussion in Section V.B about how fast-tag hit helps increase availability in the presence of uncorrectable errors is one of our work's contributions.

A. Derivation from Basic Theory

The basis of the proposed algorithm comes from coding theory and the elemental inequalities used below are known and proven in coding theory [14].

In particular, it is established from coding theory that there exists a minimal Hamming distance between two legal linear code words. The Hamming distance between two binary words is the number of positions with mismatched bits between the two words. This is equal to the sum of the result of the bitwise XOR operation between the two words. Therefore, when the Hamming distance between a legal code word and another code word satisfy inequality (1) this can be result of data corruption only in the second code word.

$$0 < dm < dmin \qquad (1)$$

where:

dm is the actual Hamming distance between the two code words;

$dmin$ is the minimum Hamming distance for a given linear code.

It is known that the minimum Hamming distance depends on the ECC code used. It is also proven, that when the actual Hamming distance is greater than 0, but less or equal to $\frac{dmin-1}{2}$ then the correct value of the corrupted code word is equal to the legal code word (inequality (2)). If the actual Hamming distance is more than $\frac{dmin-1}{2}$ but less than $dmin$ then data corruption exists but the correct value for the corrupted code word is unknown (inequality (3)).

$$0 < dm \leq \frac{dmin-1}{2} \qquad (2)$$

$$\frac{dmin-1}{2} < dm < dmin \qquad (3)$$

The fast-tag hit technique [20] [18] is based on the above ECC principles. Specifically, if we assume that the lookup tag is always correct, then for a hit/miss indication we do not need to know the corrected value of tags in the Tag array. It is sufficient to know their Hamming distance from the lookup Tag. This transformation enables many simplifications in the design of a tag ECC.

The proposed fast-tag hit flow includes the following steps:

- Read the tag and ECC bits from the tag array
- Compare with the lookup tag and lookup ECC bits. The lookup ECC bits are generated in parallel with the tag array access (not in the critical path). The following are the possible outcomes of the comparison:
  - The lookup and read tag-ECC are matched (there is a hit) if their Hamming distance is equal to 0 (full match – no error)
  - It satisfies the criteria of inequality (2). This says that if there is an error that is correctable then there is a hit: the real value of the stored tag is equal to the lookup tag
  - If the Hamming distance meets the criteria of inequality (3), then the correction strength is exceeded and information about tag corruption is sent to the system for further treatment according to system policy
  - If the Hamming distance is greater than or equal to the minimal Hamming distance, the lookup and read tag are mismatched (there is miss).

Fig. 5 shows a generic description for non-speculative Tag ECC based on the fast-tag hit algorithm. When there is no error or the error is correctable there is a hit, otherwise, there is a miss. In the case of an error that exceeds the ECC strength a signal is send to the system for further actions like in the case of uncorrectable error for the baseline tag ECC (Section IV.A).

The overview of a speculative Tag ECC scheme that employs fast-tag hit is shown in Fig. 6. The reduction in the complexity of the slow validation path is evident (compare to Fig. 4).

Clearly, the proposed fast-tag hit algorithm eliminates the need for (i) calculation of the syndrome, (ii) decoding the syndrome, and (iii) restoring the correct tag value, all of which are required by the baseline non-speculative and speculative tag ECC methods in Section IV.

The fast-tag hit logic, both for the non-speculative and speculative ECC (Figs. 5 and 6), is similar to the tag hit for unprotected tag arrays (Fig. 2) but the tag comparison includes ECC bits in addition to the tag bits. Thus, fast-tag hit introduces some overhead for generating and comparing the ECC bits of the lookup tag. Nevertheless, this penalty is significantly smaller than the overheads, of the baseline ECC

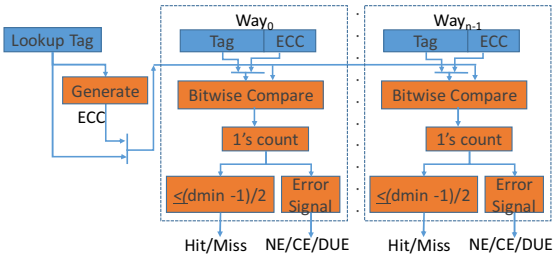

Fig. 5. Fast-Tag Hit Non-Speculative Tag ECC Scheme

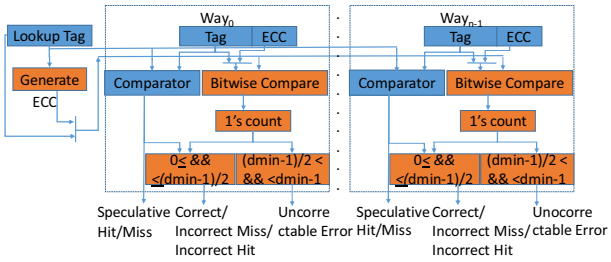

Fig. 6. Fast-Tag Hit Speculative Tag ECC Scheme

techniques, that are eliminated by the fast-tag hit scheme. More specifically, fast-tag hit needs ECC generation only for the lookup tag, whereas in the state of the art Tag ECC schemes the checker, syndrome and repair modules are needed per way. Furthermore, the lookup tag ECC generation does not hurt timing; it is not in the read critical path because it is performed in parallel with the tag array access.

Another difference of the fast-tag hit is that it uses a 1's count function to summarize the results of the bitwise comparator instead of an AND operation. This is a more complex function with likely longer delay than an AND. Nevertheless, as shown in the next two Sections (V.B and VI), the function can be simplified depending on the ECC strength required.

For a writeback cache, modified line eviction requires sending both the correct tag and data of the evicted line to a higher-level cache or to external memory. Therefore, for fast-tag hit, but also for all other tag ECC schemes, the tag replacement flow needs correcting any detected correctable tag error. This can be implemented using a traditional tag ECC algorithm, with similar stages as in Fig. 3, but without the comparator stage.

Overall, fast-tag hit algorithm seems capable to reduce both area and power penalty as well as relax timing pressure of the baseline ECC schemes. The various overheads of the different schemes as well as their performance are evaluated and compared in Section VIII.

B. Fast-Tag Hit Non-Speculative SECDED Tag ECC

This Section describes the workings of the fast-tag hit algorithm when used to design a SECDED Tag ECC. Note that the minimum Hamming distance for SECDED ECC is dmin=4. The remaining discussion, in Sections V and VI, focuses on the non-speculative SECDED Tag ECC due to space limitations and the similarity between the speculative and the non-speculative schemes. Nonetheless, both designs are analyzed in the experimental results in Section VIII.

Table I explains how the proposed scheme reacts to different cases of Hamming distance between the lookup and stored tag/ECC codewords for a SECDED ECC algorithm assuming that the SER can cause at most 2-bit corruption in a tag. This is a reasonable assumption given that large arrays often employ both or either logical and physical interleaving that render unlikely more than two upsets in the same tag [19].

Distance equal to 1 occurs when the lookup tag/ECC word and tag/ECC word read from the tag array are matched in all but one-bit position. This may occur when 1-bit in either the tag or ECC bits read from the cache is corrupted. The fast-tag hit flow will identify such scenario as hit. Information about a correctable error can be sent to the system for error logging purposes.

The distance equal to 2 can be the result of a 2-bit corruption in the stored data/ECC. In this case a hit/miss outcome cannot be resolved because the corrupted tag can have this distance from many legal code words, one with distance two and several with distance four [14]. In this case, the fast-tag hit mechanism will send to the system the information that 2-bit corruption exists. The system will then check the modified status of the cache line with the 2-bit corrupted tag and react as the traditional ECC scheme for the case that the error exceeds its correction strength (Section IV.A).

The distance equal to 3 (3-bit mismatch). This can be the result of a 1-bit corruption in the stored tag that reduces its real Hamming distance from the lookup tag/ECC word from 4 to 3. A 2-bit corruption can also cause a 3-bit mismatch if the original Hamming distance is 5. In either case, the lookup and stored tag are different and this scenario is defined as a miss. A subtle implication of this reaction is that double errors in the case of a certain miss are not reported thus increasing the availability of the system. More specifically, when the tag of an unmodified line suffers a double error, if in all subsequent

Table I. Interpretation of Hamming-distance between lookup Tag-ECC and Stored Tag-ECC for SECDED fast-tag hit algorithm

Bitwise compare (Hamming distance)	Hit/miss (match/mismatch) definition	Reasoning for hit/miss definition
0	Hit	Fully match, no corrupted bit is found
1	Hit	1 corrupted bit is found, lookup and cache tags defined as matched
2	No definition/ Don't care	2 bits corruption, no hit/miss definition
3	Miss	lookup and cache tags defined as mismatched
4 or more	Miss	lookup and cache tags defined as mismatched

tag comparisons it is involved with the Hamming distance is not equal to 2, then the double error gets masked.

The distance is greater or equal to 4 (4-bit mismatch). This scenario occurs when either (i) there are no corrupted bits in the read tag/ECC word and its real Hamming distance from the lookup tag/ECC bits is greater or equal to 4, or (ii) there are 1 or 2 corrupted bits but the real Hamming distance is greater than or equal to 5 or 6 respectively. This scenario is also treated as a miss since in all cases the real stored tag and lookup tag are different. This reaction, similar to the case with Hamming distance 3, can help mask double errors.

The ECC strength of the non-speculative SECDED fast-tag hit mechanism is similar with a baseline non-speculative: single error correction, dual error detection. SECDED fast-tag hit guarantees correct hit/miss indication in the case of 1-bit corruption and correct detection of 2-bit corruption.

Protection against corruption of 3 or more bits is not supported by SECDED fast-tag hit. Such corruption, for the traditional, speculative and proposed SECDED fast-tag hit, can lead to a false hit/miss indication [7] and to a Silent Data Corruption (SDC).

The next Section presents design details about a specific implementation of the fast-tag hit algorithm with SECDED ECC.

VI. Fast-Tag Hit Implementation

This Section presents details for a possible fast-tag hit implementation when using a SECDED code and the tag/ECC codeword is 38-bits (31 bits for tag, 7 bits for ECC). The implementation we propose it is based on carry-save-adders (CSA) [3]. A number of optimization are presented that illuminate how the concept of fast-tag hit is transformed to a practical ECC scheme. The proposed design and associated optimizations are some of the main contributions of this paper.

A. Implementation Optimizations

At a high level, what is needed for fast-tag hit it is to perform a bitwise XOR of the lookup and stored tag/ECC bits and, using a 1's count summary function, determine the Hamming distance between the two codewords. A generic 1's count function with 38-bit input and 6-bit output may appear sufficient but is complex and unnecessary. Several optimization opportunities exist and needed to simplify and speed-up the 1's count function used in fast-tag hit:

Optimization 1: An inspection of Table I reveals that what is needed is something simpler than a full 1's count: determine if 1's count is 0, 1, 2 or greater than 2. This enables simplification of the 1's count function implementation.

Optimization 2: A further simplification is possible by realizing that for hit/miss definition what is performance critical is to detect whether the 1's count is less or equal to 1 (way hit) or greater than 1 (way miss). This removes from the critical path the double error signal definition. Double errors in a case of a miss can be detected with a slight delay, e.g. a cycle later, after the original miss definition. Such a delay does not compromise correctness because in the case the line with the double tag error: (i) is modified, the system will shut down which renders inconsequential the hit/miss definition in a previous cycle, (ii) is unmodified, there are two scenarios to consider: (1) no other tag matched the lookup tag (cache miss), the new incoming line will be filled in the place of the line with the tag error, and (2) another tag matched (cache hit), which implies the Hamming distance of the lookup tag with the tag with error is four (Section V.A) and, therefore, a true miss. In either case, for unmodified lines with double tag error the delay in error detection does not compromise correctness. This optimization may be useful for any tag ECC scheme; use a separate circuit for double error detection to help hit/miss definition get faster.

Similarly, single error definition can benefit from this optimization because it is only used for error logging purposes [9][10] and, therefore, generating and reporting the error with a small delay does not affect correct functionality.

Optimization 3: Another simplification becomes feasible by allowing speculative hit definitions that can be incorrect only when there is a double error. This can further simplify and speed-up the circuit used for hit/miss definition. For example, dividing the 38-bit 1's count function into two 19-bit functions that their results are ANDed together. More details about the functionality of this scheme are given subsequently. This optimization, similar to optimization (2), relies on having separate circuits for hit/miss definition and double error detection and with the double error detection circuit been off the critical path. A speculative hit definition that turns out to be incorrect, due to an uncorrectable double error, does not compromise functionality as long as the double error definition is resolved before forwarding a wrong but valid line. In general, such time windows exist in caches because additional time after hit definition is needed either to multiplex the data of a selected way or to access the selected way (Section III.B).

Fig. 7 shows a high-level description of the circuit for hit/miss definition leveraging the optimizations (1) and (3). It contains two parallel 19-bit functions each simply indicating whether the Hamming distance, in its corresponding half of the tag comparison, is $\leq$1 (0 or 1) (optimizations (1) and (2)). Such circuit requires considerably less area, power and timing than a full 1's count function of 38 bits. The circuit in Fig. 7 assumes, without loss of generality, that the one function checks the Hamming distance for bits in even bit positions and the other in the odd positions. Table II describes the circuit's behavior.

It is clear from the table that the functionality of the circuit is correct in all cases except when both the even and odd functions indicate a Hamming distance $\leq$1 and the circuit indicates at its output a hit. In such case, this can be an actual hit or a double error. As discussed earlier (optimization 3), the incorrect definition in this case will always lead to a double error detection by a separate circuit that can handle this, and all other cases with double errors, correctly (shown in Fig. 8 and discussed later).

A carry-save-adder (CSA) [3] is a type of digital adder that is suited for computing the sum of three or more n-bit binary numbers. It is distinct from other adders in that it outputs numbers of same dimensions as the inputs. This makes it an effective candidate for Hamming distance calculation, i.e. bitwise sum, as well as for the simplified form, determining if

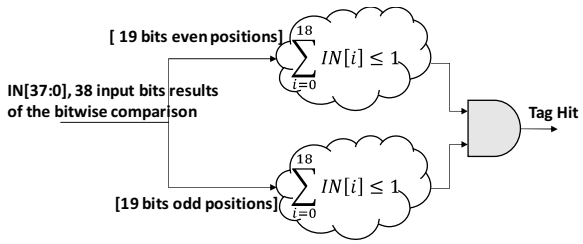

Fig. 7. Hit/Miss Definition with Even/Odd Functions

Table II. Behavior of Split Even and Odd 1's Count Function

Even≤1/Odd≤1	TRUE	FALSE
TRUE	Hit (Hit or double error, each half has distance ≤1)	Miss (Odd half has distance ≥2)
FALSE	Miss (Even half has distance ≥2)	Miss (Both halves have distance ≥2)

the bitwise sum is less or equal to 1, which is needed by the two functions in Fig. 7.

B. Carry-Saved-Adder Based 1's Count Function

A CSA based function that determines if the sum of 19 1-bit inputs is less or equal to 1 is shown in Fig. 9. The circuit uses four 4to2 CSAs (a 4to2 CSA is actually a 5 input and 3 output circuit) for the first level and a 3to2 CSA for the second level. Three of the first level CSAs takes as input an exclusive 5-bit subset of the 19-bit input. The fourth CSA takes as inputs 4 bits from the 19-bit input and the SUM output of one of the other first level CSAs. The second level CSA takes as input the unprocessed SUM bits of the first level CSAs. There are only three such SUMs since the fourth SUM is input to a first level CSA.

The number of 1's at the inputs of a single CSA is given by the expression $\sum_{i=0}^{c-1} 2\,x\,Carry_i + SUM$, where c is the number of carry output signals. The SUM bit of a CSA indicates whether the number of 1s in the CSA's input are odd or even and the number of set carry signals the number of non-overlapping pairs of 1's in the CSA inputs. Consequently, the carry outputs of the first level CSAs detect the number of non-overlapping pairs of 1s in their own inputs, whereas the second level CSA detects such pairs across the SUM bits of three different first level CSAs.

The actual overall Hamming distance can be obtained using the carry signals from all CSAs and the SUM bit of the second level CSA as follows:

$$2\,x\,number\ of\ set\ carry\ signals + second\ level\ SUM.$$

It is obvious, that the circuit in Fig. 9 does not include, there is no need for it, hardware to do the actually computation of the Hamming Distance (optimization 1). Rather, the circuit simply determines if the Hamming distance of its inputs is ≤1 by taking the NOR of all carry signals. If all such signals are zero it means the number of 1's in the input, and hence the Hamming Distance, is ≤1. When at least one carry output is set the output of the circuit becomes zero, Hamming Distance is ≥2. The SUM output, of the second level CSA, indicates whether the 1's count is odd or even, and when the NOR output is a 1 this indicates precisely whether the Hamming distance is 1 or 0.

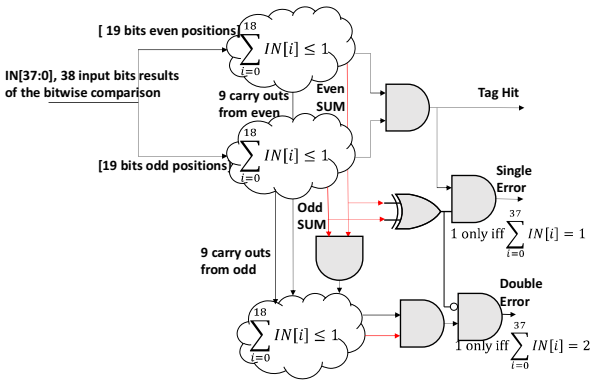

Fig. 8. Fast-Tag Hit Single and Double Error Detection

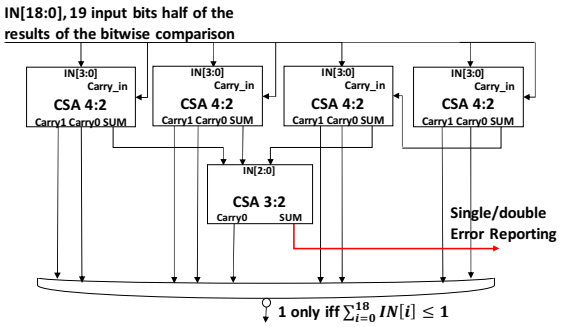

Fig. 9. Details of 19-bit ≤1 1's count function

The path that needs to be fastest in the logic circuit in Fig. 9 is the one leading to the NOR output since that is used for the performance critical hit/miss definition. This the reason for choosing the specific CSA topology in Fig. 9 since it is the one that reduces the delay of that path. To help illustrate this, consider an alternative CSA based design with identical functionality and cost shown in Fig. 10. The only difference with Fig. 9 is in the topology; a 3to2 CSA is used in the first level and a 4to2 CSA is used in the second level to add the SUMs from the first level CSA and one of the 19-bit inputs. Unlike the design in Fig. 9, in Fig. 10 there is no dependence between any first level CSAs. Fig. 11 shows one of the critical paths for each design to the carry output (C0) of the second level CSA in terms of a 2-input XOR gate equivalent delays (assuming a specific CSA implementation). The careful tuning of the topology of the first design (in Fig. 9) allows removing from the critical path the top XOR gate of the 3to2 CSA. This results in a 5 gate delay for the design in Fig. 9 whereas the second design (in Fig. 10) has a 6 gate delay. This example helps highlight the two sides of what design entails: diverse opportunities for optimization in an implementation but at the same time complex and large design space that needs to be navigated.

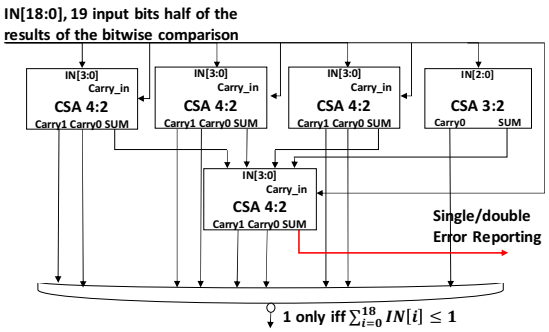

Fig. 10. Alternative 19-bit ≤1 1's count function

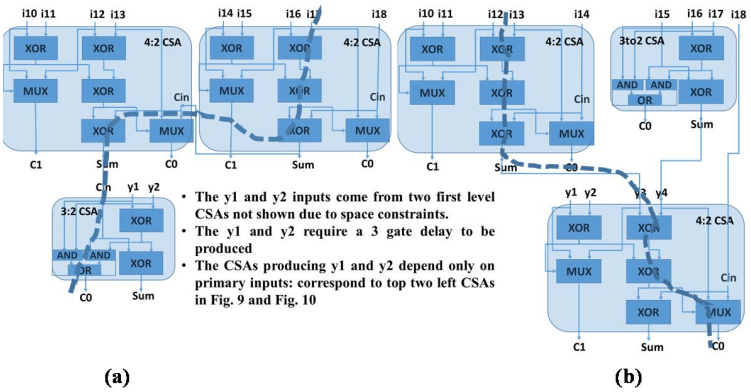

Fig. 11. Critical Path for (a) Circuit in Fig. 9 (b) Circuit in Fig. 10

C. *Single and Double Error Reporting*

The SUM output and the carry signals from both even and odd functions are used for detecting and reporting single and double errors.

Fig. 8 shows the circuit that detects double errors and reports single errors for the fast-tag hit algorithm. These signals are produced with some delay after the original hit/miss definition is complete. These signals, though, do not lie in the critical path and having separate circuit for them simplifies and makes faster the circuit used for tag hit/miss definition (optimization 2).

The circuit generates a single error signal when both the even and odd functions indicate Hamming distance ≤1 (i.e. both functions have a 1 at their NOR output) and one of the functions has a zero hamming-distance, indicated by a 0 SUM bit.

The presence of a double error is detected only when one carry signal is set and the two SUM bits have the same value. The first condition is evident why it is necessary but the second may be not so obvious. The absence of the second condition allows a Hamming distance of 3 to be reported as double error. According to Table II this should be reported as a miss.

To determine that only one carry signal is set, a 19 1-bit 1's count function is used (same as the one used for Tag Hit definition in Fig. 7). This function takes as inputs all the carry outputs from the even and odd functions (9 from each) and for 19th input the logical AND of each functions SUM bit. The latter input is essential to detect double error when there is a single error in both the even and odd functions. This ensures detecting the case of incorrect hit definition (optimization 3). A single set carry is detected when the output of the function is 1 (i.e. the number of set carry outputs is ≤1) and the SUM bit is also 1.

The next Section presents the evaluation methodology.

VII. METHODOLOGY

The various schemes have been evaluated with in-house area, power and timing tools as well as performance and power simulators used for product development. The technology parameters used for the analysis are for a 14nm technology node.

For area, power and timing analysis, the results are presented only for the logic used to implement the tag hit/miss definition. The results are normalized relative to the values of these parameters for the comparator used for the unprotected scheme. The analysis also considers qualitatively the design complexity and effort required by the various schemes. This is loosely defined to reflect the number and type of resources the design of a scheme requires as well as the number of units they require design changes.

For performance and power simulation, a state of the art microprocessor is evaluated that includes a modern out-of-order core microarchitecture and the cache hierarchy configuration shown in Table III. Representative regions of SPEC Integer and Floating Point benchmarks [6] are simulated. The simulations are used to compare the performance and power of a baseline ECC and fast-tag hit ECC scheme with CSAs (Section VI.A). The simulations assume, supported by the findings of this work (Section VIII.B), that the baseline non-speculative ECC scheme requires an extra cycle latency for accessing the L2 and L3 as compared to the latency of non-speculative fast-tag hit. Aggregated simulations results are presented using the geometric mean.

The fast-tag hit based on CSAs has been implemented by an Intel Core® microprocessor. The technique has been applied to the L2 and L3 caches. This is a clear proof of the benefits of fast-tag hit over other state of the art approaches. Data from the implementation are used to extract information about design tradeoffs and estimating the overall area savings of fast-tag hit, as compared to baseline ECC.

Table III. Processor Cache Hierarchy used in Simulations

Level	Capacity	Associativity (ways)
L1 Data	32 KB	8
L1 Instruction	32 KB	8
L2 (unified)	256 KB	8
L3 (shared)	8 MB	16

VIII. RESULTS

A. Overall Performance, Power and Area Improvements

Table IV shows the performance gain of non-speculative fast-tag hit ECC as compared to non-speculative baseline ECC obtained using performance simulation. The results show a small but still positive improvement for both the integer and floating point benchmarks from fast-tag hit. The one cycle reduction in L2 and L3 access latency, due to removing syndrome, decoding and repair from the critical path, leads to this improvement. Benefits for floating point programs is on average higher because they are more memory intensive.

Table IV. Performance of Fast-Tag Hit vs Traditional ECC

Benchmark	SPECINT	SPECFP	Overall
SpeedUp	0.25%	1.0%	0.45%

We like to note that the performance simulator used in this study is highly accurate, i.e. the above performance improvement is not in the range of statistical noise. Feature design decisions for products are based on the performance simulator used in this study that has been validated against actual silicon performance measurements.

Besides performance, fast-tag hit offers power and area reductions. The power simulation analysis and area estimates from the actual implementation of CSA based fast-tag hit reveals overall power and area reduction in the order of 1%. The power benefits over baseline ECC are netted on every access, either to L2 or L3, with energy savings coming from each way in the tag array (8 in L2 and 16 in L3). In the same vein, eliminating most of the ECC logic from each way in the L2 and L3 tag arrays leads to a non-negligible area decrease.

The fast-tag hit cannot improve performance over the speculative baseline ECC scheme since the fast-tag hit improves the latency of the slow path that is critical in the rare case with errors. The fast-tag hit offers, nonetheless, area and power savings comparable to the improvements offered by the non-speculative fast-tag hit over the baseline non-speculative ECC.

B. Area, Power, Timing and Complexity Analysis

This Section compares the area, power, timing overheads and complexity of SECDED ECC for non-speculative and speculative Tag ECC without and with fast-tag hit. The fast-tag hit designs evaluated are based on CSA. The comparison with different fast-tag hit 1's count circuits is presented in Section VIII.C.

Recall, that this analysis only considers the implications on the logic used to build the ECC in each scheme. It is noted that the analysis for the speculative schemes does not account for the extra logic required to track the progress of speculative work and to cancel it in case of mispeculation (but the comparison attempts to account for this in terms of higher complexity). Table V summarizes the various findings for non-speculative and speculative Tag ECC schemes.

The data show that the baseline schemes require a considerable area increase (8.8x and 10.1x respectively) whereas the increase of fast-tag hit is much more modest (3.1x and 4.4x). Most of the area in the baseline schemes (Fig. 3 and Fig. 4) are due to the checker, decoder and repair modules. The speculative scheme's additional area is mainly due to the extra comparator needed in the slow path to determine the correct comparison outcome. The data clearly show that the 1's count functions, that dominate the area of the fast-tag hit scheme, are much more area friendly than the checker, decoder and repair units required by the previous state of the art proposals.

Table V. Area, Power, Timing Overheads of various ECC Schemes

	Non-Speculative		Speculative	
	Baseline	**Fast-Tag Hit CSA**	**Baseline**	**Fast-Tag Hit CSA**
Area	8.8x	3.1x	10.1x	4.4x
Power	8.6x	2.9x	5.3x	3.9x
Delay	2.7x	1.6x	1x	1x
Complexity	Medium	Low	High	High

The results about power show a substantial increase by the baseline ECC schemes and a smaller increase by the fast-tag hit. The power trends between schemes are similar to the area trends except for the baseline speculative scheme. The entire path of the other designs is activated on every access. For the baseline speculative scheme, the slow path after the checker is activated only when there is an error detected. As a result, the power of the baseline speculative scheme in the absence of errors, the typical use case, is considerably lower as compared to the baseline non-speculative scheme (5.3x vs 8.6x). Nonetheless, the power increase of the fast-tag hit schemes is still smaller (only 2.9x and 3.9x). This underlines the superior power efficiency of the CSA based 1's count function used by fast-tag hit circuit as compared to traditional checker, decoder and error repair circuit.

Regarding end-to-end delay, Table V shows that for the non-speculative ECC schemes the baseline it increases it by 2.7x, much more than the fast-tag hit using CSA. The fast-tag hit causes a small increase, 1.6x, as compared to an unprotected tag array. In terms of cycle count, for a specific product design considered in our study, the fast-tag hit delay reduction translates to one cycle saving per cache access.

The speculative schemes do not affect the end-to-end delay in the case without error, the typical case. Recall that a speculative scheme has fast and slow paths (Section IV.B).

An analysis of the delay benefits of optimization 3 (Section VI.A), that uses two smaller 1's count functions instead of one larger monolithic, shows that it can help reduce the critical path of fast-tag hit with CSAs by around 6%.

Consequently, the non-speculative fast-tag hit can provide better performance than the baseline non-speculative tag ECC. Additionally, fast-tag hit provides the same ECC strength for both non-speculative and speculative tag ECC with significantly less area and power overheads as compared to state of the art ECC approaches. The fast-tag-hit does not require adding ECC correction logic as the baseline schemes.

The non-speculative option has lower design complexity than the speculative one, because it does not require adding extra logic for tracking and canceling work but it has longer delay. The choice between non-speculative and speculative

tag ECC it depends on design parameters and constraints beyond the scope of this study. What clearly this work shows, however, is that both can benefit from the use of fast-tag hit.

C. Comparison of 1's count circuits for Fast-Tag Hit

Table VI compares various implementations of the non-speculative fast-tag hit that use different 1's count functions. The data show that the proposed CSA based design is more area and power efficient as compared to SA [20] and BWA [12] designs. The use of many full adders and half adders tax the power and area of the two alternative designs. Another main difference is in the end-to-end delay. The CSA based scheme is about 44% and 13% faster as compared to SA and BWA respectively. This is mainly due to the optimizations used to enable faster hit/miss definition and at the same time remove the double error detection from the critical path (Section VI.A). In contrast, the SA and BWA designs use the same circuit and path to determine tag hit/miss as well as single and double error, which elongates its critical path. This underlines the significance of efficient implementation of a promising concept to turn into a practical component in a product.

Table VI. Area, Power, Timing Overheads of Fast-Tag Hit with various 1's count circuits

	Baseline	Fast-Tag Hit CSA	Fast-Tag Hit SA	Fast-Tag Hit BWA
Area	8.8x	3.1x	6.4x	7x
Power	8.6x	2.9x	6.3x	7x
Delay	2.7x	1.6x	2.3x	1.8x
Complexity	Medium	Low	Low	Low

D. Trends with changing Tag/ECC Size

This Section analyzes the sensitivity to the tag/ECC size of the non-speculative fast-tag hit with CSAs relative to an unprotected array. The results, shown in Table VII, clearly show that the area and power overheads are larger for smaller tag sizes but decrease at diminishing rate as tag size grows. The trends with timing are opposite, the delay is shorter for smaller tag size but it grows at a diminishing rate with increasing tag size.

The main reason for the area and power trends is that for small tag size the relative overhead of ECC is higher. For example, the 16-bit data SECDED protection requires 6 check bits, 27% overhead, whereas 32-bit data require 7 check bits, 18% overhead (Section III.A). The timing overhead of fast-tag hit grows at a diminishing rate with increasing tag size because the number of CSAs and levels needed grow at discrete steps (Section VIII). This also explains the seemingly anomaly from 8 to 16 bits. In particular, when growing the tag from 8 to 16 bits the critical path for fast-tag hit does not increase since the number of CSA levels remains the same. However, for the unprotected case the more the tag bits the more time is needed to summarize their bitwise XOR.

Table VII. Area, Power, Timing as a function of Tag/ECC size

	4	8	16	32	38	48
Area	4.5x	3.75x	3.23x	3.15x	3.1x	3.04x
Power	4.3x	3.6x	3.02x	3.0x	2.9x	2.85x
End2End Delay	1.25x	1.4x	1.33x	1.5x	1.6x	1.6x

IX. Other Uses

The fast-tag hit algorithm is not limited to SECDED ECC. It can work also with DECTED algorithm (dual error correction, triple error detection) [16]. DECTED fast-tag hit algorithm will guarantee correct hit/miss indication in the case of 1 or 2-bits corruption and correct indication of 3-bit corruption with no hit/miss definition. In general, the fast-tag hit algorithm is applicable to any linear block and cyclic codes type of codes based on a correction matrix [14]. The technique can be used for caches, as done in this work, as well as in other structures that employ tag matching, such as TLBs.

Another use of fast-tag hit is for parity protected tag arrays [18]. Specifically, the comparison for tag match it can be extended to include the parity of the stored tag and the lookup tag, generated during the tag array access. This helps save area and power used for parity generation per way and also relax the critical path for reads.

X. Conclusions

This work presents the design of an efficient error correction technique for tag arrays. The main idea behind fast-tag hit is that a cache tag protected with ECC does not need to be corrected to be used for a cache access hit/miss definition. It is sufficient to know what its Hamming distance from the lookup tag is. Consequently, the fast-tag hit scheme eliminates the need for costly hardware, used in previous state of the art approaches, for error correction, such as syndrome generators and decoders and error repair units. The work presents design details for a specific implementation instance of the proposed algorithm for SECDED ECC and discusses various ways to optimize it and the significance and implications of these optimizations. An experimental evaluation shows that fast-tag hit is superior in almost every respect as compared to previous state of the art tag ECC approaches. A very strong indication of the benefits offered by fast-tag hit, over alternative tag ECC schemes, is its implementation by an Intel Core® microprocessor.

XI. Acknowledgements

Part of this work has been performed during a sabbatical of the last author at Intel Israel Design Center in Haifa, Israel. The last author is partially supported by the European Union Horizon 2020 project Uniserver grant no. 688540 and the University of Cyprus.

References

[1] Robert C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies." IEEE Transactions on Device and materials reliability 5.3 (2005): 305-316.

[2] Henry Duwe, Xun Jian, Rakesh Kumar. "Correction prediction: Reducing error correction latency for on-chip memories." IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015

[3] John G Earle. "Latched carry save adder". IBM Technical Disclosure Bulletin, 7 (10): 909–910, March 1965.

[4] Eyal Fayneh, Marcelo Yuffe, Ernest Knoll, Michael Zelikson, Muhammad Abozaed, Yair Talker, Ziv Shmuely and Saher Abu Rahme. "14nm 6th-generation Core processor SoC with low power consumption and improved performance." IEEE International Solid-State Circuits Conference (ISSCC), 2016

[5] R. W. Hamming, "Error detecting and error correcting codes," in The Bell System Technical Journal, vol. 29, no. 2, pp. 147-160, April 1950.
[6] John L. Henning. "SPEC CPU2006 benchmark descriptions." SIGARCH Comput. Archit. News 34, 4, 1-17, September 2006.
[7] Mu-Yue Hsiao. "A class of optimal minimum odd-weight-column SEC-DED codes." IBM Journal of Research and Development 14.4 (1970): 395-401.
[8] Luong Dinh Hung, Masahiro Goshima, and Shuichi Sakai. "Mitigating soft errors in highly associative cache with CAM-based tag." IEEE International Conference on Computer Design, 2005.
[9] Intel, MCA Enhancements in Future Intel Xeon Processors, white paper, June 2013
[10] Intel, 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C & 3D): System Programming Guide, September 2016.
[11] Jesung Kim, Soontae Kim, and Yebin Lee. "SimTag: exploiting tag bits similarity to improve the reliability of the data caches." Proceedings of the Conference on Design, Automation and Test in Europe. European Design and Automation Association, 2010.
[12] Byeong Yong Kong, Jihyuck Jo, Hyewon Jeong, Mina Hwang, Soyoung Cha, Bongjin Kim, and In-Cheol Park. "Low-Complexity Low-Latency Architecture for Matching of Data Encoded With Hard Systematic Error-Correcting Codes." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 22, no. 7 (2014): 1648-1652.
[13] G.K. Konstadinidis, H.P. Li, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. P. Masleid, C. Zheng, Y. D. Lin, P. Loewenstein and H. Park. SPARC M7: A 20 nm 32-Core 64 MB L3 Cache Processor. IEEE Journal of Solid-State Circuits, 51(1), pp.79-91., 2016
[14] Shu Lin, Daniel J. Costello , "Error Control Coding: Fundamentals and applications," Prentice-Hall Series in Computer Applications in Electrical Engineering, 1983, pp.51-123
[15] Shubu Mukherjee. Architecture design for soft errors. Morgan Kaufmann, 2011.
[16] Riaz Naseer, and Jeff Draper. "Parallel double error correcting code design to mitigate multi-bit upsets in SRAMs." Solid-State Circuits Conference, 2008. ESSCIRC 2008. 34th European. IEEE, 2008.
[17] Timothy J. O'Gorman, John M. Ross, Allen H. Taber, James F. Ziegler, Hans P. Muhlfeld, Charles J. Montrose, Huntington W. Curtis, and James L. Walsh. "Field testing for cosmic ray soft errors in semiconductor memories." IBM Journal of Research and Development 40, no. 1 (1996): 41-50.
[18] Pedro Reviriego, Salvatore Pontarelli, Marco Ottavi, and Juan Antonio Maestro. "FastTag: A Technique to Protect Cache Tags Against Soft Errors." IEEE Transactions on Device and Materials Reliability 14, no. 3 (2014): 935-937.
[19] N. Seifert, B. Gill, K. Foley and P. Relangi. "Multi-cell upset probabilities of 45nm high-k + metal gate SRAM devices in terrestrial and space environments." IEEE International Reliability Physics Symposium, 2008
[20] Wu Wei, Dinesh Somasekhar, Dinesh and Shih-Lien L. Lu. "Direct compare of information coded with error-correcting codes." IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2012.
[21] Hongbin Sun, Nanning Zheng, and Tong Zhang. "Realization of L2 cache defect tolerance using multi-bit ECC." 2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems. IEEE, 2008.
[22] Shuai Wang, Jie Hu, and Sotirios G. Ziavras. "Replicating tag entries for reliability enhancement in cache tag arrays." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 20.4 (2012): 643-654.