UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Power-aware Error Detection Schemes for Memory Arrays

Panagiota Nikolaou

Supervising Professor
Yiannakis Sazeides

This thesis is submitted to the Graduate Faculty of University of Cyprus in partial fulfillment of the requirements for the Degree of Master of Science at Computer Science Department

December, 2011

ABSTRACT

Achieving fault tolerance is an inevitable problem in architectural arrays such as caches, which it becoming more challenging with the power constrains. This work propose to reduce energy by avoiding access to columns of on-chip SRAM arrays whose cell contents used to store the bits for error protection. We refer to this approach as Error Detection Code Sharing that relies on a lazy-based method for fault detection. We explain how the EDCS approach can be leveraged to reduce the energy needed for Error Detection Codes. Experimental analysis reveal that the proposed scheme can leverage to reduce the energy. The potential energy savings of the propose approach at 32nm often exceeds 7% for several processor arrays. The energy savings however may come at the expense of lower fault–detection coverage.

This work also proposes, EDCS-LC, an approach in lie with the EDCS that uses one vector to keep track of the codes that appear in the cache to further increase the fault detection coverage with no impact on the energy savings that EDCS technique provides.

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Power-aware Error Detection Schemes

Panagiota Nikolaou

Supervising Professor
Yiannakis Sazeides

This thesis is submitted to the Graduate Faculty of University of Cyprus in partial fulfilment of the requirements for the Degree of Master of Science at Computer Science Department

December, 2011

ACKNOWLEDGES

At this point I would like to thank my supervising professor, Mr. Yanos Sazeides that gave me the opportunity to work on this thesis. I also want to thank him for his help, guidance and advice that offered me during the preparation of this thesis, but also for the confidence that he shows in me to bring out this work. As a supervising professor gave me many comments, suggestions and advices that helps me to finish this job. Furthermore, I would like to express my thanks to the members of the Xi Computer Architecture Group for their help, feedback and comments.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Power and reliability challenges.

For the past several decades, technological developments have facilitated the continuous miniaturization of devices on silicon chips. According to Moore's Law [6], the number of transistors on a chip roughly doubles every two years in the same area.

As a result the scale of transistors gets smaller and smaller. Recently allowed the integration of large caches and many cores into the same chip. Unfortunately, the scaling of the other key design parameters has not followed suit and have elevated power and reliability into prime design constrains across all computing market segments. Specifically, power envelops are becoming so stringent that it may be impossible in the future to operate all on chip recourses. With the increasing power density of modern circuits as the number of transistors per chip scales (Moore's law), power efficiency has increased in importance [20].

The reliability trends are also ominous. With shrinking cell geometries, soft errors and hard errors have become a greater concern. **Power and reliability** are increasingly becoming intertwined.

Power dissipation has become a major concern to those designing processors for high performance desktops, servers, and battery-operated portable devices. Published reports also corroborate the fact that on–chip static RAM caches consume substantial fraction of overall chip power. For example on–chip L1, L2 caches dissipate from 25% to 50%

of the total chip power, depends from the processor [15]. Higher energy dissipation requires more expensive packaging and cooling technology, which in turn increases cost and decreases system reliability [4].

There are fundamentally two ways in which power can be dissipated: either dynamically (due to switching activity), or statically (which is mainly due to leakage in the gates). If current technology scaling trends hold [11], leakage will soon become very important source of power consumption, and as such new techniques are needed to battle this growing problem. The problem of leakage stems from the need for a tradeoff between dynamic power and performance [12].

Figure 1.2 projected leakage power consumption as a fraction of the total power consumption according to International Technology Roadmap for Semiconductor [20].

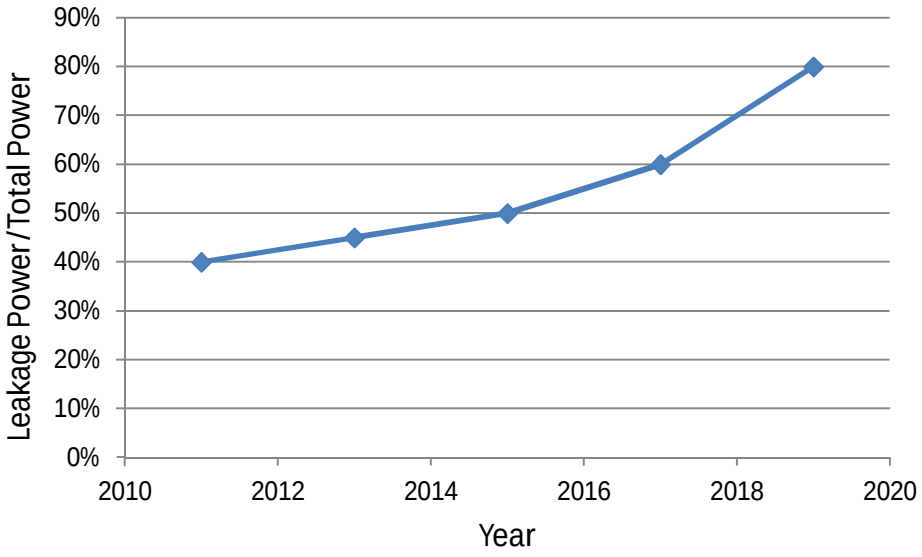


Figure 1. 1 : ITRS leakage protection.

Power has become a first-order design consideration motivating researchers to look at techniques for reducing power consumption in all components of a system design. For

memory ECC, power reduction is also an important consideration, since the ECC checker circuit is activated during each read and write access to the memory.

Existing techniques used in current processors such as ECC codes [17], sparing [7], and larger more resilient cell [8] incur considerable area and energy costs. For instance protecting an array with 72-64 SEC-DED requires 12.5% additional cells, and can increase energy consumption by 20% [9, 10]. Current power saving techniques have been mainly focused on reducing the dynamic power that needed for protecting the array.

Several techniques have been proposed to reduce the power consumption that required for fault detection in memory arrays. The importance of the power dissipation led us to address with this issue.

Clearly protecting an array entails both area and energy overheads. The overheads are due to the larger array storage needed for the error code in each entry.

Table 1.1 lists the normalized dynamic energy per read access and the static energy obtained for three different caches for their respective data and tag array. Also in Table 1.2 lists the leakage breakdown obtained for three different caches for their respective tag and data array. For each array is shown the normalized leakage for the data bits and the code bits. The data in Table 1.1 and Table 1.2 clearly indicate that error protection entails a significant energy overhead of depending on the cache array. One of the main goals of the propose techniques is to minimize the energy overhead. For the results of Table 1.1 and Table 1.2 we assume that we operate an instruction per cycle. The methodology that we used to obtain these results is given in Chapter 8.

	L1 Dcache		L1 Icache		L2 cache	
	Tag	Data	Tag	Data	Tag	Data
Dynamic Power	71	89	71	93	51	70
Leakage power	29	11	29	7	49	30

Table 1. 1: Normalized Read hit Energy Breakdown and leakage power for different types of arrays.

The data in Table 1.2 clearly indicate that error protection for leakage power entails a significant energy overhead 11-15% depending on the cache array.

One of the main goals of this thesis is to minimize this energy overhead.

	L1 Dcache		L1 Icache		L2 cache	
	Tag	Data	Tag	Data	Tag	Data
Data bits	86	89	86	89	85	89
Code bits	14	11	14	11	15	11

Table 1. 2: Normalized leakage power for different types of arrays.

1.2 Contribution.

In this thesis, we propose two approaches to reduce the leakage power that required for fault detection in memory arrays.

The first approach is the Error Detection Code Sharing (EDCS) approach. The main idea of this approach is that uses one code word to protect multiple data words for further reduction of energy and possible area used for memory array protection. In this case total power can be reduced managing the static (leakage) power consumption that has a significant portion of total power consumption. This, however, may come at the expense of lower fault-detection coverage.

So, we propose also the Error Detection Code Sharing using Legal Codes approach that increases the fault detection coverage from EDCS with small cost on energy and area savings. The key idea of this approach is that uses one vector that keeps track of the

legal *error detecting codes* that appear in the architectural arrays at any given time. The legal codes are the codes that appear at any given time in a memory array. The propose approach relies on that we don't obtain all the codes in a memory array. So using this approach we detect an error if one illegal code appears.

In general, this approach is only capable of fault detection, it cannot correct faults in arrays. Therefore it can be useful for detecting faults in architectural arrays such as caches, when it is used in combination with a backward-error-recovery scheme, e.g. checkpoint-rollback [11, 12] , that can recover the state of a memory array upon fault detection.

.

Chapter 2

Fault Tolerance

2.1 Definition of fault tolerance.

Fault tolerance has always been around, but now is becoming more important because we have more reliance on computers. So we study faults and their causes because if we don't understand them, it is much more difficult to design systems that can tolerate them!

2.1.1 Fault-Error-Failure.

Fault [22] is an incorrect step, process, or data definition in software or in hardware which causes the computer to perform in an unintended or unanticipated manner. It is an inherent weakness of the design or implementation which might result in a failure. We have fault avoidance using techniques and procedures which aim to avoid the introduction of faults during any phase of the safety lifecycle of the safety-related system. Also we can have fault tolerance that is the ability of a functional unit to continue to perform a required function in the presence of faults or errors. Fault tolerance is defined as how to provide, by redundancy, service complying with the specification in spite of faults having occurred or occurring.

Error [22] is the manifestation of a fault. A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

Failure [22] is the system level effect of an error and it is visible to the user. The inability of a system or component to perform its required functions within specified performance requirements. For example we have the production of incorrect result of computation (2+2=5).

Faults can result in errors. Errors can lead to system failures.

Errors are the effect of faults. Failures are the effect of errors.

2.1.2 Hard and Soft errors

The **Soft errors [2],** which also called **transient errors,** occur once and then disappear. The soft errors often occur from the cosmic rays, the decay of radioactive atoms which exist in trace amounts in all materials and maybe from the noise phenomenon. A soft error will not damage a system's hardware; the only damage is to the data that is being processed.

The **Hard errors [2]**, which also called **permanent errors**, occur and don't go away. They can appear like chip-level soft errors, but the difference is that the hard error is not rectified when the computer is rebooted. The solution to a hard error is to replace the memory chip or module entirely.

Finally we have the **intermittent** errors which occur occasionally.

Fault masking is a structural redundancy technique that completely masks faults within a set of redundant modules. A number of identical modules execute the same functions, and their outputs are voted to remove errors created by a faulty module.

2.2 Causes of faults.

Hardware faults are mostly *physical faults*, while software faults are *design faults*, which are harder to visualize, classify, detect, and correct. Design faults are closely related to fuzzy human factors and the design process, which we don't have a solid understanding. In hardware, design faults may also exist, but physical faults usually dominate.

Specifically the causes of faults are classified to the following categories:

1. Physical problems [2].
    a. Transient Phenomena like extraterrestrial cosmic rays which bombard the earth constantly from the far depths of the galaxy and the alpha particle radiation that comes from the decay of radioactive atoms which exist in trace amounts in all materials.
    b. Manufacturing Defects [2].

       As the technology scales with the integration of billion of transistors, transistors have smaller dimensions in order of 45nm, so it is very hard to manufacture something with these dimensions and not to cause faults. Also may have bad solder connection between chip and board.
2. Hardware design flaws like logical and timing bugs [2].
3. Operator error like the bugs in software. The operator error is the leading cause of computer system failures [2].
4. Design Flaws like the bugs in software [2].
    a. Incorrect algorithm
    b. Memory leak
    c. Reference to NULL

5. Malicious attacks [2].
6. Variations [18].
    a. Random dopant fluctuations which results from the discreetness of dopant atoms in the channel of a transistor. The transistor channels are doped with dopant atoms to control their threshold voltage. This problem decreases with the decrease in transistor size in each technology generation. So the transistor area reduces to the half and, thus the number of dopant atoms in the channel decreases exponentially over generations.
    b. The sub-wavelength lithography that we used for pattering transistors. This occurs line edge roughness and several other effects in transistors, resulting in variation.
    c. The heat flux and it depends on the functionality of the circuit block, for example the activity and compute load at any given time. Higher heat flux results in higher temperature, creating hot spots, which in turn create temperature variations across the die, affecting circuit performance. This also results in higher sub-threshold leakage, variations in the leakage across the die, and the variations in power delivery demand across the power distribution.

The first two are static and they occur during the fabrication but the third is dynamic; that is, it is time and context variant.

7. Extreme Variations [18].

    As technology continues to scale further both static and dynamic variations will continue to become worse. So we have some extreme variations that we can meet.

a. In the future, for nanometre-scale technologies, fewer transistors will actually approach the target Vt, and this distribution will flatten out. These variations would be impossible to correct for them during design.
b. Soft errors that they increase in each technology generation. These errors are more important when they occur in a logic flip-flop because it is difficult to detect and correct them.
c. Aging that has had significant impact on the transistor performance. Along the years the transistor's saturation current degrades because of oxide wear out and hot-carrier degradation effects.

2.3 Fault Tolerant Design.

There are two fundamentally different approaches to recover from errors. The first one is called Forward error recovery and the second, Backward error recovery. The objective of Forward error recovery is to mask effects of errors using redundancy and going forward in presence of errors. In the Backward error recovery approach, redundancy used to enable recovery to saved good states of system. This technique does backward to recover from errors.

The Forward error recovery may include additional hardware (hardware redundancy), additional information (information redundancy), additional design (design redundancy), more time (temporal redundancy) or a combination of these.

2.3.1 Forward error recovery

Hardware redundancy is perhaps the most used architecture in many applications. Here, fault redundancy is achieved by duplicating the critical hardware components of a system with the intention to increase the reliability of a system. The most used architecture of hardware redundancy, Triple Module Redundancy (TMR).

The basic concept of the TMR is that we have three modules that perform a process and that result is processed by a voting system to produce a single output. The voting element accepts the outputs from the three sources and delivers majority vote as its output.

If any one of the three systems fails, the other two systems can correct and mask the fault. If the voter fails then the complete system will fail.

Some ECC memory uses triple modular redundancy hardware (rather than the more common Hamming code, is explained below), because triple modular redundancy hardware is faster than Hamming error correction hardware.

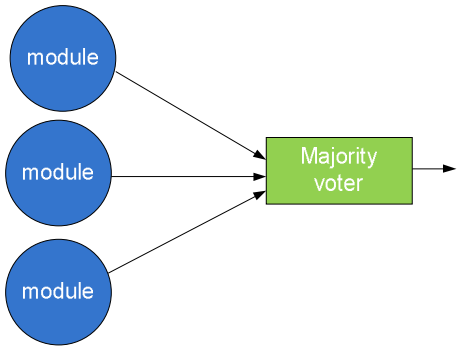


Figure 2. 1 Triple modular redundancy (TMR).

Triple modular redundancy (TMR) [2] is also commonly used form of fault masking in which the circuitry is triplicated and voted. The voting circuitry can also be triplicated so that individual voter failures can also be corrected by the voting process.

Hybrid redundancy is an extension of TMR in which the triplicated modules are backed up with additional spares, which are used to replace faulty modules -- allowing

more faults to be tolerated. Voted systems require more than three times as much hardware as non redundant systems, but they have the advantage that computations can continue without interruption when a fault occurs, allowing existing operating systems to be used.

Information redundancy involves the addition of redundant information. For example for a given k-bit piece of information, add r check bits to it that make it possible to detect/correct errors in the original k-bit information.

Design redundancy use different designs to guard against a fault in any of them.

Temporal Redundancy involves replicates of actions on a module which are using the same module, but at a different time.

2.3.2 Backward error recovery

The basic idea of Backward error recovery is to recover from a previous state of system that we know that is error-free. For doing this we use some new terminology such as checkpoints that periodically saving state of system, logging list that saving changes made to system state and finally recovery point that is the point to which we recover in case of error.

Checkpoint and Recovery [3, 13]

Checkpointing and recovery belongs to the category of error recovery for fault tolerance. Checkpointing involves occasionally saving the state of a process in stable storage during normal execution. Upon failure, the process is restarted in the saved state

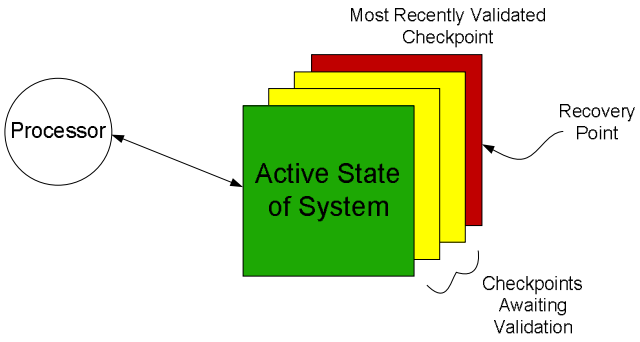


Figure 2. 2 Backward Error Recovery.

(last saved checkpoint-recovery point) that we know is error-free. The assumtion is that the error will be gone before resuming execution. This thus reduces the amount of lost work. For instance, the progressive retry technique employs checkpointing along with message logs that saves the changes made to system state. Our work can borrow many concepts from the research in the area of checkpointing. Figure 2.2 shows an abstraction of the operation of Backward error Recovery technique using checkpoint, recovery point and the active state of the System.

Chapter 3

SRAM Arrays and Energy Consumption

3.1 Organization of the cache.

This Chapter is based on the paper of Bushra Ahsan et al and on the Cacti report [1, 23].

In computer engineering, a cache is a component that transparently stores data so that future requests for that data can be served faster. The data that are stored within a cache might be values that have been computed earlier or duplicates of original values that are stored elsewhere. Cache is faster and usually static memory, SRAM that retains data bits in its memory as long as power is being supplied.

If requested data is contained in the cache (cache hit), this request can be served by simply reading the cache, which is comparatively faster. Otherwise (cache miss), the data has to be recomputed or fetched from its original storage location, which is comparatively slower.

3.2 Dynamic power

An array consists of rows and columns. In an SRAM based array, we have columns known as differential pair columns (bitlines BL0 and BL0_b) and an associated wordline. Figure 3.1 shows the dynamic power consumption during a write and read access. Dynamic power dissipation occurs during state changes (i.e., when devices are switching)The figure represents the control signals that are necessary to perform the access, the sense amplifiers and the write drivers. For simplicity we show just one cell

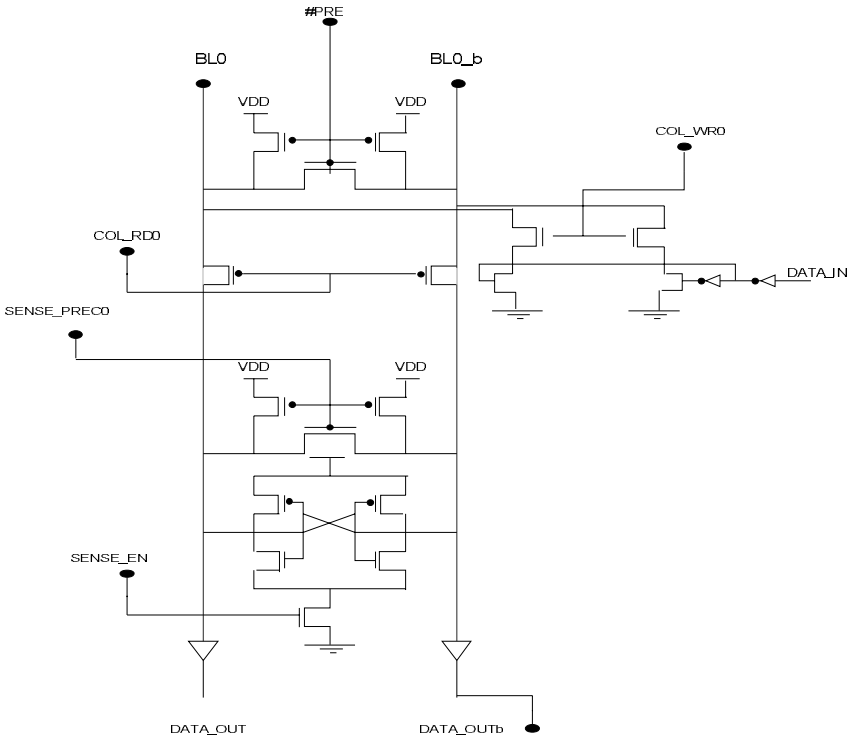


Figure 3. 1: Dynamic Power Dissipation Mechanism.

and one column pair. The address and data are transferred to the array through the buses. Before a normal access starts, the columns are presumed to be in precharged state and ready to be accessed. The decoders decode the row address from the address bits and activate the associated wordline. On a read access, the COL RD selects the column to be read (since many columns can be sharing the sense amps and the associated drivers). Depending on the value of the cell, either BL0 or BL0_b is discharged. The SENSE EN signal allows the sense amps to sense the values and output the data. Once the value has been read, the PRE and SENSE PRE0 signals precharge the column and the sense amps back to the precharge voltage.

In a write operation, the COL WR operation selects the column and data values are input through the DATA IN drivers. Unlike read, in a write access, a complete differential has to occur for the value to be written (full swing as compared to half

swing). Once the write is done, the discharged column is brought back to precharge voltage.

Several techniques have been proposed to reduce the dynamic power consumption that required for fault detection in memory arrays. We discuses one of those techniques below.

3.3 Static (leakage) power

Static power consumption has grown to a significant portion of total power consumption in recent years. Its importance, however, has grown considerable over the past five years. In part, this importance stems directly from the fact that leakage energy now represents 20-40% of the power budget of microprocessors in current and near future fabrication technologies.

Static power dissipation is a result of the various leakage modes of the MOS transistor. While there are many different leakage modes, the most important leakage mechanism in modern submicron channel length technologies is subthreshold leakage [10]. Subthreshold leakage is current that flows between the source and drain even when the transistor is off (i.e., the voltage at the gate is below the threshold voltage).

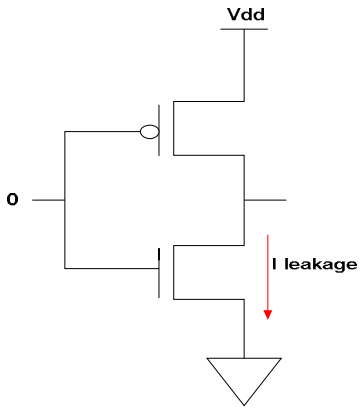


Figure 3. 2: Static Power Dissipation Mechanism [10].

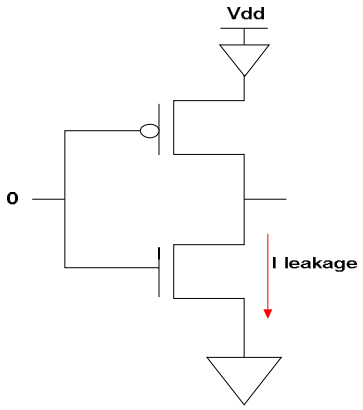


Figure 3. 3: Static Power Dissipation Mechanism with gated Vdd.

Some techniques are proposed for leakage reduction. The concept of those mechanisms is to minimize the leakage power and apparently the total energy. The two most widespread mechanisms are based on the stacking effect and the drowsy effect that manipulate the voltage across transistor terminals.

The stacking technique refers to the technique of stacking *off* transistor source to drain. Gated-Vdd is a transistor stacking technique that reduces the leakage power. This

technique allows to the transistor that it doesn't working, to *sleep* gating the ground. Turning this transistor *off* disconnects the SRAM cell from the power supply. Figure 3.3 shows the traditional power dissipation mechanism and its gated *Vdd*.

The drowsy effect refers to a low voltage mode for the memory cells. The leakage reduction of the drowsy mode is not like the gated-Vdd approach which completely cuts off the path to Vdd. It allows for some nonzero supply voltage preserves the state of the memory cell. The drowsy effect was proposed to address the disadvantage of the gated-Vdd technique that is it destroys the state.

Chapter 4

Error Checking and Correction (ECC) in Memory Systems.

There are two kinds of errors that can typically occur in a memory system. The first is called a repeatable or hard error. In this situation, a piece of hardware is broken and will consistently return incorrect results, as we explained above.

A bit may be stuck so that it always returns "0" for example, no matter what is written to it. Hard errors usually indicate loose memory modules, blown chips, motherboard defects or other physical problems.

They are relatively easy to diagnose and correct because they are consistent and repeatable.

The second kind of error is called a transient or soft error. This occurs when a bit reads back the wrong value once, but subsequently functions correctly. These problems are, understandably, much more difficult to diagnose! They are also, unfortunately, more common. Eventually, a soft error will usually repeat itself, but it can take anywhere from minutes to years for this to happen.

The only true protection from memory soft errors is to use some sort of memory detection or correction protocol. Some protocols can only detect errors in one bit of an eight-bit data byte; others can detect errors in more than one bit automatically. Others can both detect and correct memory problems, seamlessly.

These protocols called ECC "Error Correction Codes" and is a method used to detect and correct errors introduced during storage or transmission of data.

Certain kinds of RAM chips inside a computer implement this technique to correct data errors and are known as ECC Memory

Error rate in memory systems is usually decreased by applying error checking and correction (ECC) techniques which can be based on different algorithms for correcting single or multi-bit errors. Error correction reduces memory failures dramatically. **Figure 4.1** shows the block diagram of a memory system with error correction.

In this figure we assume that we have directed map cache so we don't have sets but only blocks.

When the memory system receives k information bits the encoder calculates the check bits which together with the information bits are stored in the memory.

When the memory system sends for the read access the k information bits enable the decoder which calculate again the check bits of the information bits and compare the two ECC codes. So the decoder identifies and corrects the errors which have possibly occurred in the information bits and thus provide the corrected data on the output.

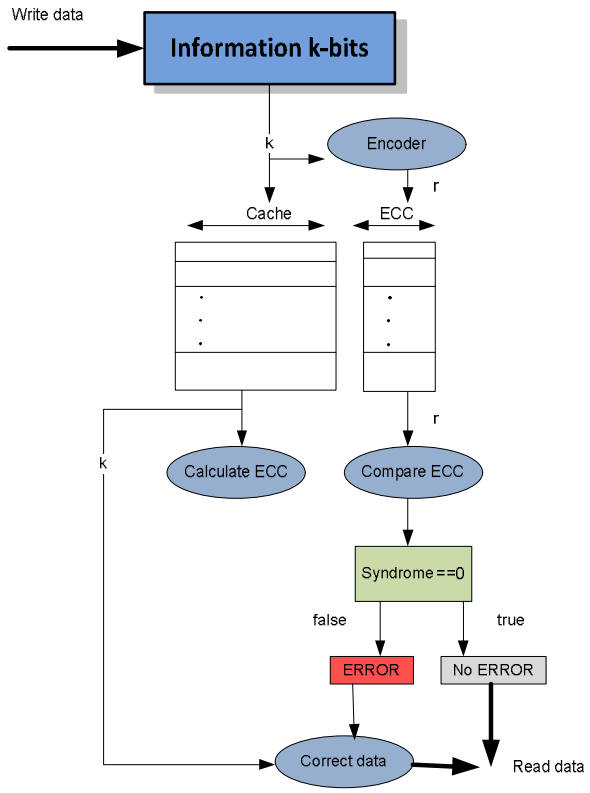


Figure 4. 1: Block diagram of a memory system with error correction.

Chapter 5

Error detecting and Correcting Codes

The problem of “doing things right” on a large scale of operations is not essentially new. Therefore we must perform a large scale of operations without a single error in the end result.

A single failure usually means the complete failure in a digital computer, in the sense that if it is detected no more computing can be done until the failure is located and corrected. But when if it escapes the detection then it invalidates all subsequent operations of the machine.

The need for error detection and correction having assume importance. It is clear that to transmit information from one place to another the machines use codes which are simply sets of symbols. So we shall assume that the transmitting equipment handles information in the binary form of a sequence of 0’s and 1’s. Thus each code symbol (information and check) will be represented by a sequence of 0’s and 1’s.

Each code symbol from these codes has exactly n binary digits, where m digits are associated with the information while the other k=n-m digits are used for error detection and correction. From these numbers we can produce the R that is the ratio of the number of binary digits used to the minimum number necessary to convey the same information.

So we have that R =n/m.

We now discuss some algorithms to construct a single error detecting, single error correcting and single error correcting plus double error detecting codes.

Firstly, if we need only to detect a single error it is easier to use *even or odd parity check algorithm*. In this algorithm we have n+1 binary digits which the n are the information bits and the 1 is the check bit. In the n-th position we place either 0 or 1, so that the entire n positions have an even or odd number of 1's. This is clearly a single error detecting code since any single error in transmission would leave an odd or even respectively number of 1's in a code symbol.

Otherwise, if we need to detect and correct a single error we use more complex algorithms such as *Hamming and Hsiao code*. So we first assign n of the n+k available position as information positions and next we assign the k remaining positions as check positions. The k is the checking number and consists of a sequence of 0's and 1's. We shall require that k will give us the position of any single error, with the zero value meaning no error in the symbol code. Thus k must describe m+k+1 different things, so that

$2^k > m + k + 1$.

We explain in more details Parity, Hamming and Hsiao codes in Section 3.2, Section 3.3 and Section 3.4 respectively.

5.1 Parity code

A parity bit is a bit, with a value of 0 or 1, that is added to a block of data for error detection purposes [9]. It gives the data either an odd or even parity, which is used to validate the integrity of the data.

Parity checking is a rudimentary method of detecting simple, single-bit error or multi bit odd errors in a memory system. This method forcing the sum of some predetermined set of bits to be an even number (called even parity) or an odd number (called odd parity).

To get the desired effect, the parity bit is set to one or zero according to the sum of the specified original bits. If the selected set of original bits sums to an even number, the extra parity bit is set to zero; if it sums to an odd number, it is set to one. Hence, in either case, the sum -- including the parity bit -- will always be an even or odd number for odd or even parity protection respectively.

The computer knows exactly which parity checking it is using. If it uses an even parity and the number of 1-bit add up to an odd number then it knows there was an error: Else if it uses an odd parity and the number of 1-bit add up to an even number then it knows there was an error. However, if an even number of 1-bit is flipped the parity will still be the same. But an error occurs. So the even/odd parity can't detect this error.

Therefore parity checking is a useful way validating data, but it is not a foolproof method. For instance, the values 1010 and 1001 have the same parity. Therefore, if we have the value 1010 and it changed to the value 1001, no error will be detected. This means parity checks are not 100% reliable when validating data. Still, it is unlikely that more than one bit will be incorrect in a small field of data.

5.1.1 Parity example.

Suppose that we have a binary bit word and we are using an odd parity:

00011

Now suppose that the encoded word is stored in a memory and on a read operation the information bit in position 3 changes from 0 to 1. **00111**

To determine if there has been a bit inversion in the data, the parity check bit is regenerated and compared with the parity bit generated when storing the data in memory. The new parity bit is 0. So the result word now is

00110

So there must have an error. There are 2 1-bit, which is an even number.

5.2 Hamming code.

A Hamming code [17] is a first linear error-correcting code named after it's inventor, Richard Hamming. Hamming codes can detect single-bit and multi-bit odd errors, and correct single-bit errors. Is possible to correct single-bit errors and detect double bit errors when the minimum Hamming distance between the transmitted and the received bit patterns is 3. The Hamming distance is the number of bits in which two words of equal length differ from each other.

Because of the simplicity of Hamming codes, they are widely used in computer memory (RAM).

So when we have an information code the first thing that we do is to calculate the number of the check bits (k) from the above equation in Section 3.1. In a 7-bit message, there are seven possible single bit errors, so three error control bits could potentially specify not only that an error occurred but also which bit caused the error.

5.2.1 General algorithm.

The following general algorithm generates a single-error correcting (SEC-SED) code for any number of bits.

The construction of the code is best described in terms of the parity- check matrix H. The selection of the columns of the H-matrix for a given number of information bits and check bits is based on the following algorithm. So the main idea is to construct an H-matrix in a way extract the check bit equation for every code.

Determine the positions which each of the various parity checks is to be applied. The checking number is obtained digit by digit, from right to left, by applying the parity checks in order and writing down the corresponding 0 or 1 as the case may be. Any position which has a 1 on the right of its binary representation must cause the first check to fail, the second parity check must use those positions which have 1's for the second digit from the right of their binary representation, and move on until we have the k-th parity check which must use those positions which have 1's for the second digit from the right of their binary representation.

1. Decide for each parity check which positions are to contain information and which the check. The choice of the positions 1, 2, 4, 8, 16,…. for check positions has the advantage of making the settings of the check positions independent of each other. All other positions are information positions.

The following general algorithm generates a single-error correcting (SEC-DED) code for any number of bits.

We can extend the previous algorithm to construct a single error correcting plus **double error detecting code**. To do this, we include an extra parity bit. So when we don't have errors all the parity checks including the last one that we add, are satisfied, also when we have single error the checking number gives the position of the error, and for two errors the last parity check is satisfied, and the checking number indicates some kind of error. However, including an extra parity bit the minimum distance of Hamming code increase to 4. This is the reason that gives to the code the ability to detect and correct a single bit error and at the same time detect (but not correct) a double bit error. It could also be used to detect up to 3 errors with a given misscorection probability, but not correct any.

The construction of a Hamming code with 8 bit information word is given below. It can be seen from the relationship that if m=8 then we need k=4 check bits appended to the information bits for a single bit correction. The bit position of the code are labelled with numbers 1 through 12.

Bit positions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Bit names	C1	C2	P1	C3	P2	P3	P4	C4	P5	P6	P7	P8

The bit positions corresponding to powers of 2 are used as check bits c1,c2 ,c3 and c4, respectively. The other bit positions correspond to the data bits P1 to P8. The parity check matrix for the Hamming single error correction-double error detection code with 16 information bits and 6 check bits that becomes from the above algorithm is:

	C1	C2	P1	C3	P2	P3	P4	C4	P5	P6	P7	P8	C5	P9	P10	P11	P12	P13	P14	P15	P16	C6
C1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
C2	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1
C3	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1
C4	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
C5	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
C6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 5. 1: Parity check matrix of the (16,6) Hamming SEC-DED code.

From the matrix the check bit equations can be extracted as follows:

C1=P1⊕P2⊕P4⊕P5⊕P7⊕P10⊕P12⊕P14⊕P16

C2=P1⊕P3⊕P4⊕P6⊕P7⊕P9⊕P10⊕P13⊕P14

C3= P2⊕P3⊕P4⊕P8⊕P9⊕P10⊕P15⊕P16

C4=P5⊕P6⊕P7⊕P8⊕P9⊕P10

C5=P9⊕P10⊕P11⊕P12⊕P13⊕P14⊕P15⊕P16

C6= P1⊕P2⊕P3⊕P4⊕P5⊕P6⊕P7⊕P8⊕P9⊕P10⊕P11⊕P12⊕P13⊕P14⊕P15⊕P16

5.2.2 Hamming example.

The construction of a Hamming code with 8 information bits is given below. It can be seen from the hamming relationship that if info word is 8 then 4 check bits appended to the information bits for a single bit correction. With the red colour we see the bit positions of the code.

Info word:1 1 0 0 0 1 0 0

P1	P2	1	P3	1	0	0	P4	0	1	0	0

The bit positions corresponding to powers of 2 are used as check bits P1, P2, P3 and P4, respectively. The other bit positions correspond to the data bits.

To calculate the check bits we use the Hamming matrix and we implement the bellow equations.

P1=XOR of (3, 5, 7, 9, 11) =**0**

P2=XOR of (3, 6, 7, 10, 11) =**0**

P3=XOR of (5, 6, 7, 12) =**1**

P4=XOR of (9, 10, 11, 12) =**1**

So the result word appears bellow.

001110010100

Now suppose that the encoded word is stored in a memory and on a read operation the information bit in position 3 changes from 0 to 1.

111100100

To determine if there has been a bit inversion in the data, the parity check bits are regenerated and compared with the check bits generated when storing the data in memory. The new check bits are P1'P2'P3'P4'=0101. So the result word now is

011011010100

To find the location of the erroneous bit a syndrome vector is calculated as given below.

S1 =XOR (P1,P1')

S2 =XOR (P2,P2')

S3 =XOR (P3,P3')

S4 =XOR (P4,P4')

Therefore the error address is 0110 (6^{th} bit) and from the word

011010010100

Information bit 3 must be inverted. In the case that, the error address is all 0s, no bit error has occurred in the stored word. This method it's only to investigate if an error occurred in the information and in the ECC bits. Hence, it can protect also if we have an erroneous bit at the ECC code.

5.3 Hsiao code.

Hsiao code [14] is another code for single-error correction and double error detection. It is equivalent to the Hamming code in the sense that both codes require the same number of check bits for a specific data word length. Hsiao code is different from Hamming code in a way that it is simpler to implement and indicates that this code is better in performance, cost and reliability than the conventional Hamming SEC-DED codes. This is also the reason that Hsiao code is suitable for applications to computer memories or parallel systems.

Hsiao code is based on Hamming code with some improvements by simplifying the hardware implementation and providing faster and better error-detection capability

The key step of generating the well-known Hsiao code is to construct a {0,1}-check-matrix. In order to have a SEC-DED code, the minimum weight requirement is 4, which implies that three or fewer columns of the H-matrix are linearly independent. The reason that the required minimum weight is 4 is related with the minimum Hamming distance that needs to correct single bit errors and detect double bit errors.

The following general algorithm generates a single-error correcting (SEC-SED) code for any number of bits.

Similar with Hamming code the selection of the columns of the H-matrix for a given number of information bits and check bits is based on the following algorithm. So the main idea is to construct an H-matrix in a way extract the check bit equation for every code. The difference from Hamming code is that the matrix resulting from the algorithm is fixed and does not change from code symbol to code symbol. Unlike the matrix for Hsiao code is not fixed but it can be constructed in a way to satisfy some constrains. Depending on the needs, the selection of H-matrix maybe changed depending on the characteristics of a data stored in memory.

5.3.1 General algorithm.

The algorithm is based on the constrains mentioned above. So the parity check matrix for the Hsiao code is constructed as follows:

1. There are no all-0 columns.
2. Every column is distinct.
3. Every column contains an odd number of 1's (hence odd weight).

The first two constrains give a Hamming distance-3 code and the third constrain guarantees the code thus generated to have Hamming distance 4.

The third constrain also is responsible for odd-weight-column codes. Therefore, choosing a minimum odd-weight for the number of 1's, the total number of 1's in every row are the minimum. So if all the rows have the minimum number of 1's and equal to each other, then we have the fastest encoding and error detection in the decoding process. Also in this situation the hardware that required for implementation is less. Less hardware also means better reliability and lower cost.

For example, if the implementation requires less hardware it has less chance to failure, since every circuit has an intrinsic failure rate.

Specifically, the above third constrain can extended to give the actual construction procedures providing a minimum odd-weight-column code.

So the parity check matrix now for the Hsiao code is constructed as follows:

1. There are no all-0 columns.
2. Every column is distinct.
3. Every column contains an odd number of 1's (hence odd weight).
    a. Every column should have an odd number of 1's; i.e., all column vectors are of odd weight.
    b. The total number of 1's in each row of the H-matrix should be a minimum.
    c. The minimum number of 1's in each row of the H-matrix should be equal, or as close as possible, to the average number, i.e., the total number of 1's in H divided by the number of rows.

The construction of a Hsiao code with 8 bit information word is given below. It can be seen from the relationship that if m=8 then we need k=4 check bits appended to the information bits for a single bit correction. The bit position of the code are labelled with numbers 1 through 12.

Bit positions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Bit names	P1	P2	P3	P4	P5	P6	P7	P8	C1	C2	C3	C4

The bit positions from 9 to 12 are used as check bits c1,c2 ,c3 and c4, respectively. The other bit positions correspond to the data bits P1 to P8.

One parity check matrix for the Hsiao single error correction double error detection code with 16 information bits and 6 check bits that becomes from the above algorithm is:

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	C1	C2	C3	C4	C5	C6
C1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	1	0	1					
C2	1	1	1	0	0	0	1	1	1	1	0	0	1	0	0	0		1				
C3	1	0	0	0	1	0	1	1	1	0	0	0	0	1	1	1			1			
C4	0	0	0	0	0	1	1	0	0	1	1	1	0	1	1	1				1		
C5	0	1	0	1	0	0	0	1	0	1	1	1	1	1	0	0					1	
C6	0	0	1	1	1	1	0	0	1	0	0	1	1	0	0	1						1

Table 5. 2: Parity check matrix of the (16,6) Hsiao SEC-DED code.

Here, it's good to mention that there is more than one construction for that parity check matrix but randomly chosen this applying the minimum odd-weight which is 3.

From the above matrix the check bit equations can be extracted as follows:

$C1=P1 \oplus P2 \oplus P3 \oplus P4 \oplus P5 \oplus P6 \oplus P11 \oplus P15$

$C2=P1 \oplus P2 \oplus P3 \oplus P7 \oplus P8 \oplus P9 \oplus P10 \oplus P13$

$C3=P1 \oplus P5 \oplus P7 \oplus P8 \oplus P9 \oplus P14 \oplus P15 \oplus P16$

$C4= P6 \oplus P7 \oplus P10 \oplus P11 \oplus P12 \oplus P14 \oplus P15 \oplus P16$

$C5= P2 \oplus P4 \oplus P8 \oplus P10 \oplus P11 \oplus P12 \oplus P13 \oplus P14$

$C6= P3 \oplus P4 \oplus P5 \oplus P6 \oplus P9 \oplus P12 \oplus P13 \oplus P16$

Some observations from the above equations are that we can observe that each data bit can affect three check bits. For example P1 affect C1, C2 and C3. With the same way P11 affect C1, C4 and C5. This observation is very important because explain the proof of the single error correction double error detection property. Another observation that

results from the equations is that the matrix is constructed in a way that taking two data bits affect 5or less check bits but no more.

To highlight the comparison between the algorithms, Hamming and Hsiao we summarize the number of XOR gates that we need for each check bit.

Check bit	Hamming	Hsiao
C1	8	7
C2	8	7
C3	7	7
C4	5	7
C5	7	7
C6	15	7

Table 5. 3: Number of XOR gates for each code.

As can been seen from the above table Hsiao algorithm use less XOR gates than Hamming. It is shown that Hsiao codes may be constructed that always have fewer 1s than Hamming codes and thus require less implementation logic. Higher processing speeds can be achieved, since logic depth is reduced by eliminating the need for a parity bit across all of the data and check bits for double error detection.

5.3.2 Hsiao example.

The construction of a Hsiao code with 16 information bits is given below. It can be seen from the relationship that if info word is 16 then 6 check bits appended to the information bits for a single bit correction.

Bit positions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Info word	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**0**

To calculate the check bits we use the H-matrix and we do the bellow equations.

C1=XOR of (1, 2, 3, 4, 5, 6, 11, 15) =**1**

C2= XOR of (1, 2, 3, 7, 8, 9, 10, 13) =**0**

C3= XOR of (1, 5, 7, 8, 9, 14, 15, 16) =**1**

C4= XOR of (6, 7, 10, 11, 12, 14, 15, 16) =**1**

C5= XOR of (2, 4, 8, 10, 11, 12, 13, 14) =**1**

C6= XOR of (3, 4, 5, 6, 9, 12, 13, 16) = **0**

So the result word appears bellow.

11000100110001001011 10

Now suppose that the encoded word is stored in a memory and on a read operation the information bit in position 3 changes from 0 to 1.

11100100110001001011 10

To determine if there has been a bit inversion in the data, the parity check bits are regenerated and compared with the check bits generated when storing the data in memory. The new check bits are C1'C2'C3'C4'C5'C6'=011111. So the result word now is

11100100110001000111 11

To find the location of the erroneous bit a syndrome vector is calculated as given below.

S1 =XOR (C1,C1')

S2 =XOR (C2,C2')

S3 =XOR (C3,C3')

S4 =XOR (C4,C4')

S5 =XOR (C5,C5')

S6 =XOR (C6,C6')

It can be seen that the syndrome bits match with the third column from the left in the parity matrix, identifying d3 as an erroneous bit.

Next, we can consider a double-bit inversion. For example, information bit 3 and 4 changed from 00 to 11. Then the recomputed check bits are:

111101001100010011111

The resulting syndrome bits are:

010010

As we can see the syndrome provides a column that does not match with any column of the parity check matrix and the vector of the modulo-2 addition that results has an even number of 1's. So we understand immediately that a double bit error is presented. Note that the syndrome vector matches with the XOR of columns d3d4 in the parity check matrix (d3d4 are the bits where bit-inversions were inserted).

5.4 Selecting Error Correction Codes to Minimize Power in Memory Checker Circuits.

The main objective of this paper that proposed from Shalini Ghosh [20] is to reduce power consumption in single error correcting, double error detecting checker circuit that perform memory ECC. Power is minimized with a little or no impact on area and delay, using the degrees of freedom in selecting the parity check matrix of the error correcting code. They proposed two approaches that are based to select the appropriate H-matrix using the degrees of freedom in selecting the H-matrix. These degrees of freedom are depending to the simple permutations to matrix columns.

Chapter 6

Lazy Based Fault Detection for Memory Arrays (LBFD).

This Chapter is based on [22].

The reduction of power consumption that required for fault detection in memory arrays has been the subject of several previous papers.

The main objectives of these proposals were to achieve the power reduction using several techniques. The most relevant paper of this work is discussed below.

Very relevant to our work is the unpublished paper that submitted at the 39th International Symposium on Computer Architecture (ISCA 2012) by Yanos Sazeides et al. [21] that introduces a lazy approach for fault detection in memory arrays enabling techniques that trade-off between a memory array's fault-detection latency-coverage and energy-area. In general LBFD approach is only capable for fault detection. Therefore it cannot correct faults in the array. This approach relies on the read-write memory array invariance. This means that every write to an array location is preceded by a read. The concept of this approach is that without checking for faults on reads but only on writes can reduce the overheads that required for fault detection in memory arrays.

On a write the error code of the value that is being overwritten is computed and compared with the exciting one. If is error free the error code of the new value is computed and stored in the array. The reason that is called lazy based approach is that with this mechanism it can't be evident the delay for two consecutive writes. To minimize this delay the authors proposed an array sweep [5] to detect faults that are not

overwritten, after a given number of instructions. Sweep operation use an algorithm to scan the content of a cache of a point set. Interval sweeping, around of 1 million cycles and more has negligible performance impact for all core arrays [19].

LBFD approach can be applied to arrays that contain architectural state only to detect faults. For error correction LBFD can naturally be combined with numerous hardware and software based variants of backward error recovery (BER) that have been proposed. The general idea of BER based schemes is to maintain a checkpoint of a validated previous state of the system which can even be the initial programme state that can revert to in the case of detecting failures.

Nonetheless, our work is distinct because it is an extension of the LBFD approach to further reduce energy.

In Figure 6.1 we show the LBFD operation during a write access, assuming a 4-way associate cache. The Figure 6.2 shows the LBFD operation during a read access.

As can be seen from Figure 6.1 a normal write has to be done. In this case no energy is saved since column is accessed in normal way. In Figure 6.2 energy is saved since there is no need to read and drive the ECC bits out for checking.

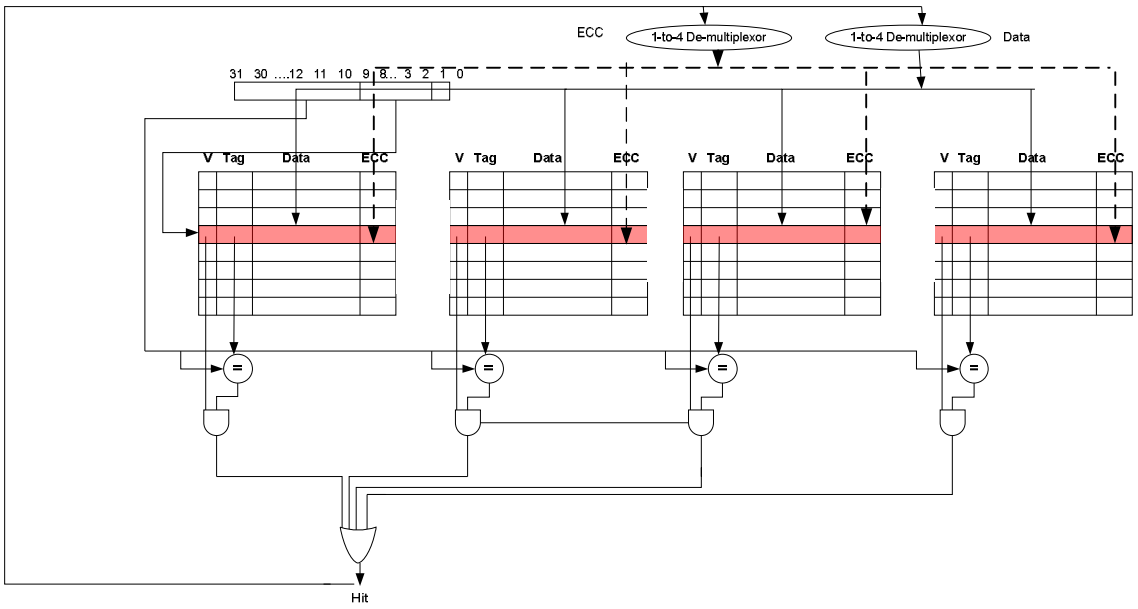


Figure 6. 1: 4-way associative cache-write access.

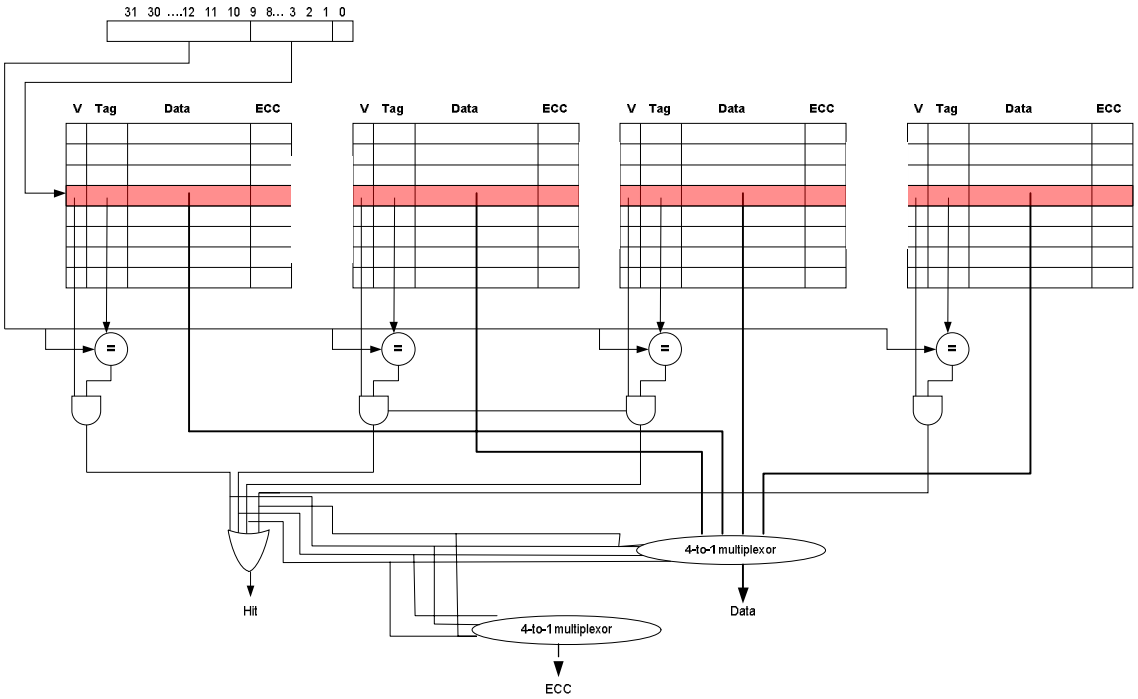


Figure 6. 2: 4-way associative cache-write access.

6.1 Adaptive scheme.

Since the benefit of LBFD depends on the dynamic program behaviour (reads vs writes) and sweep overhead, we propose to extend the dual mode protection at the granularity of intervals. An adaptive algorithm considers at the end of each interval, depending on the number of different types of accesses, if the LBFD based scheme has advantage over the baseline and decides what protection mode to use for the next interval. This algorithm shows the benefits that the LBFD scheme has over the baseline scheme.

The psefdokodik of the algorithm is:

If *energy_prev_in_lbfd > energy_now_in_baseline*

then *choose baseline;*

else *choose lbfd;*

Chapter 7

Error Detection Code Sharing (EDCS)

7.1 EDCS Approach.

As we told before, the EDCS approach is an approach that is in lieu with the LBFD approach. The main key of this approach is that EDCS protects multiple data words using the same code instead of having one code per data word. This can reduce leakage energy by power gating the EDC codes.

EDCS is also an error detection approach that introduced for arrays to protect the architectural arrays. Because is an extension of the LBFD approach inherits the basic characteristics of this approach with some exceptions.

So all writes accesses to an array become a read followed by a write. The first difference from the LBFD approach is that no error checking is performed on regular writes before it is overwritten. The only checking is enabled during an array sweep.

We refer to this read followed by a write as a read-to-remove and denote it henceforth as *R2R*. A *R2R* before a write is used to *remove* the code of the current value, about to be overwritten, from the shared code whereas a write adds the code of the new value in the shared code. To implement the additional/removal of a code from the shared code we xoring the two together. This implies that (i) if a value remains the same between a write and a R2R, its code will also be the same and, therefore, by xoring it twice its effects are removed from the shared code, and (ii) at any given time the shared code is

equal to the xor of the codes of all values sharing the code. In principle EDCS can be stretched as far as to protect a whole array with a single code. e.g. a parity bit or a SECDED code.

One drawback of EDCS over LBFD is that for typical codes, such as parity and SECDED, it can not detect errors when reading a word, all words that sharing a code need to be read to check that their overall xor matches the shared code. This requires all shared codes to have a known value at initialization. Therefore, with EDCS it is more efficient to check for errors only during sweep and not before every write. This increase in the failure detection latency may be acceptable if EDCS provides significant savings.

To highlight the difference between the two approaches it is good to present an example that is showing the basic functionality of these techniques.

We illustrate the LBFD and EDCS operation during an interval in **Figure 7.1** assuming a 4x8 array that contains initially all zeros and assuming even parity protection. At the initial state the parity for the entire array is zero. Each write access is denoted by a Wi[j] with i representing the write entry and j representing the write value. Similarly Ri[j] denotes reading from entry i the value j. In order to keep updating correctly the parity, a R2R is added before every write. The figure shows how in case of a single error in the array it is detected when the faulty entry is read by the baseline scheme, when it is overwritten in the case of LBFD, and during the sweep with EDCS.

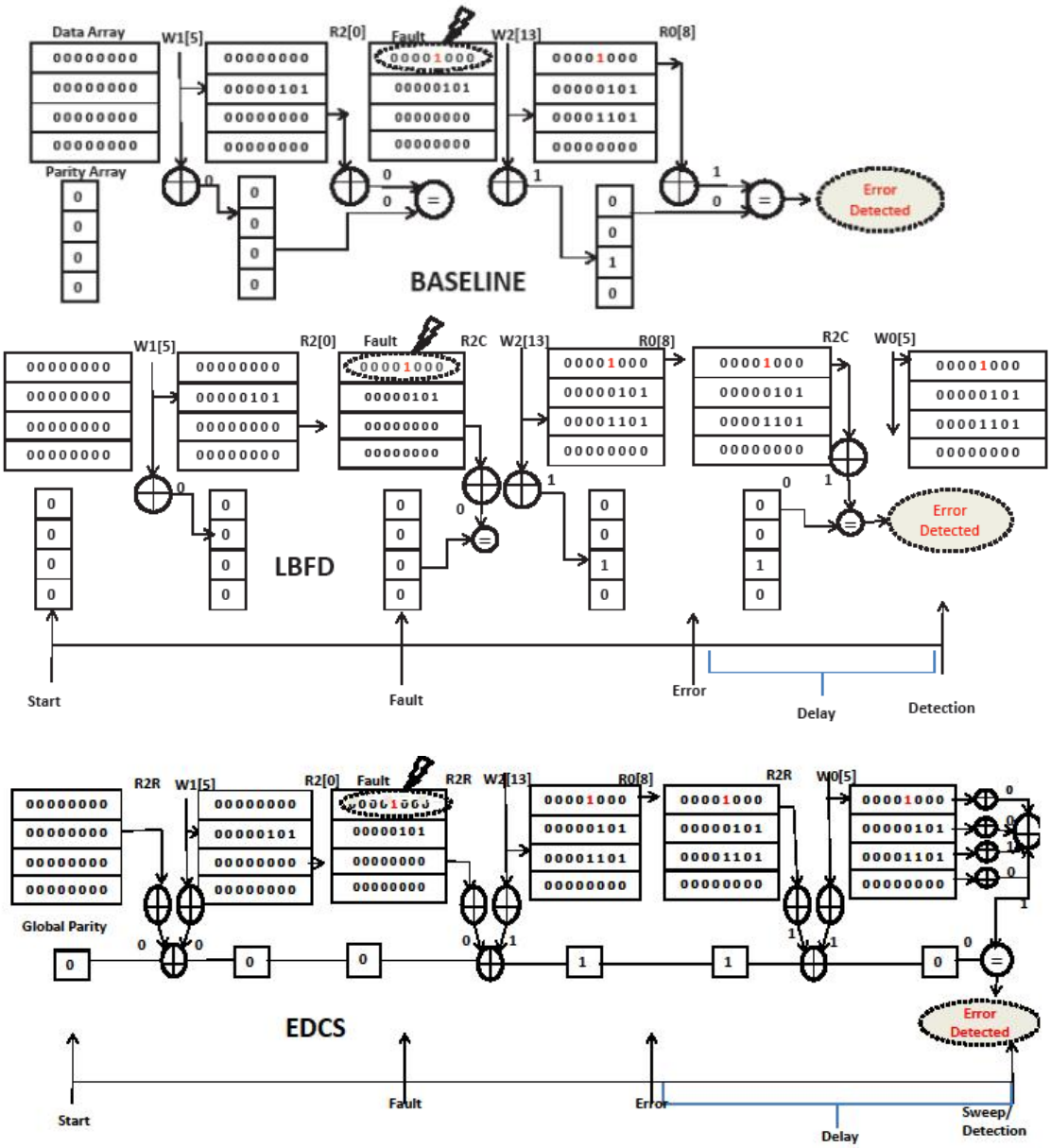


Figure 7. 1: (a) Baseline, (b) LBFD, and (c) EDCS techniques.

7.2 EDCS for Caches.

The EDCS approach can be applied to arrays that contain architectural state. To provide how EDCS can be used to detect faults in tag and data arrays for caches we made the following assumptions: a) the arrays are either protected with byte parity or a

SEC-DED code, b) the data are stored in logic blocks in the cache where each block are consisted from 64bits data, and c) the cache is n-way associated cache.

EDCS potential benefits come from that it use one code word to protect multiple data words. However, this means that the area that is used to store the EDC codes may be disabled. Figure 7.2 presents the logical organization of a cache using the EDCS technique for a way that the EDC bits are not used. The gray colour is used to represent the disabling of EDC codes as shown in the figure.

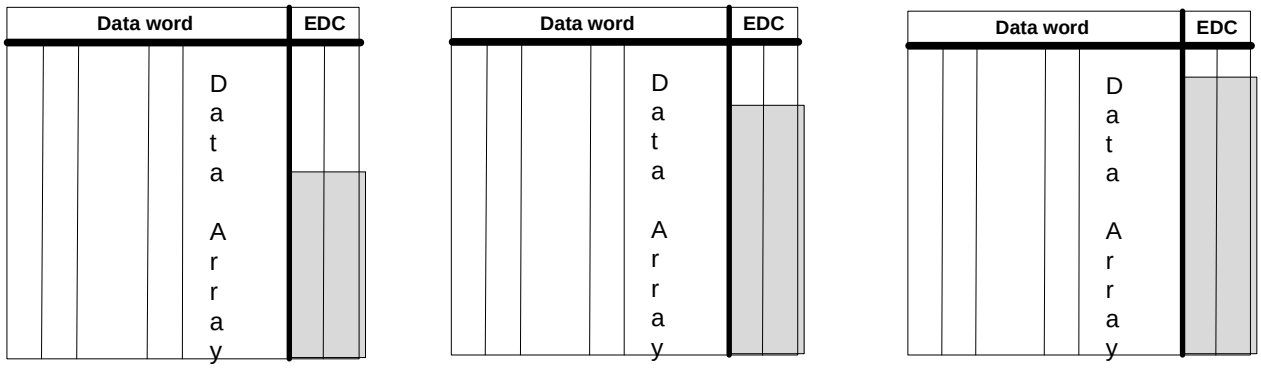


Figure 7. 2: Logical organization of a cache using the EDCS technique.

Figure 7.3 illustrates the completely logical organization of a cache using the EDCS technique during write access, respectively. We choose to not show the logical organization of a read access because is the same with the LBFD approach.

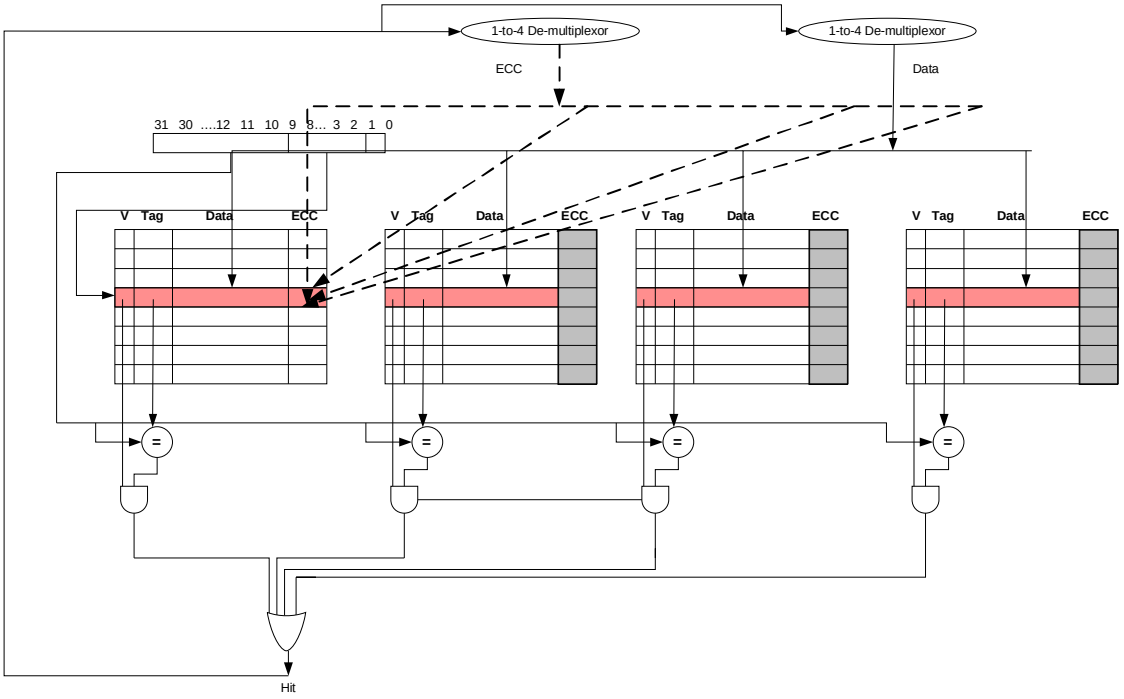


Figure 7. 3: 4-way associative cache-write access.

As we can see from the Figure 6.3 in the case of a write access, there is no need to write and drive the EDC bits in all ways. There is only a need to store the EDC bits in one way, so since the full swing is avoided to write the error detection bits, the resultant precharge is also avoided for n-1 ways. As we explain above every write to an array location is preceded by a read. Figure 6.3 shows the operation of a write access. With the grey colour are the columns that are disabled. As we can see from the figure there is only one way to store the EDC bits during the write access. Thus no energy is spent in bitlines, senseamps and the associated multiplexers. In this case energy is saved since there is no need to output the EDC bits for checking and input the EDC bits to all the ways.

The implementation of this approach refers to the technique of staking off transistor source to drain. So using the Gated-Vdd transistor stacking technique we manage to

keep the transistors that enables the EDC bits in a *sleep* mode. This transistor gates the ground. In normal operation this transistors are on. Turning the transistors off disconnects the SRAM cells from the power supply and we reduce the leakage consumption.

Like LBFD, EDCS can allow a value with a detectable error to be read and propagated until the sweep phase at the end of the interval. When this value comes from an array containing architectural state, such as a cache, this can result in state corruption, program hang, program abort, and even system crash, in the last three cases without ever getting to the sweep!

So to solve these problematic situations, EDCS approach for architectural arrays can naturally be combined with numerous hardware and software based variants of backward error recovery (BER) that have been proposed or adopted. The general idea of BER based schemes is to maintain a checkpoint of a validated previous state of the system, which can even be the initial program state, which can revert to in the case of detecting a failure. BER schemes are capable of recovering from corrupted state, hangs, program aborts etc. Such approaches are widely used and adopted for cloud based computing.

Chapter 8

Experimental Framework and Results

8.1 Experimental Framework

The experiments in this work were performed using benchmarks from the SPEC2000 and, suites with train or reference inputs. All benchmarks are compiled with gcc with -O3 optimizations. The simulator sim-alpha [22] was extended to perform measurements using a high performance out-of –order superscalar processor. For architectural arrays we evaluate the EDCS scheme to measure the times that EDCS scheme is used and how many reduction we can achieve at the leakage power by the tag and data arrays of three different caches: L2 data, L1 instruction and serially accessed L2. For these experiments we access the importance of three instruction interval lengths-100K -instructions between sweeps. The energy numbers for the architectural array analysis are obtained using CACTI [16] modelling 32nm. For all the experiments we report the results for all the bencmarks.

We analyze also the fault coverage using mathematical formulas and some form of coding for a specific array configuration: a 64KB data cache, with 512 bit data and 8 bit code for each 64 bit data. We compare the following four techniques to each other: Byte Parity, SEC-DED code, Global Parity and Global SEC-DED.

8.2 Results

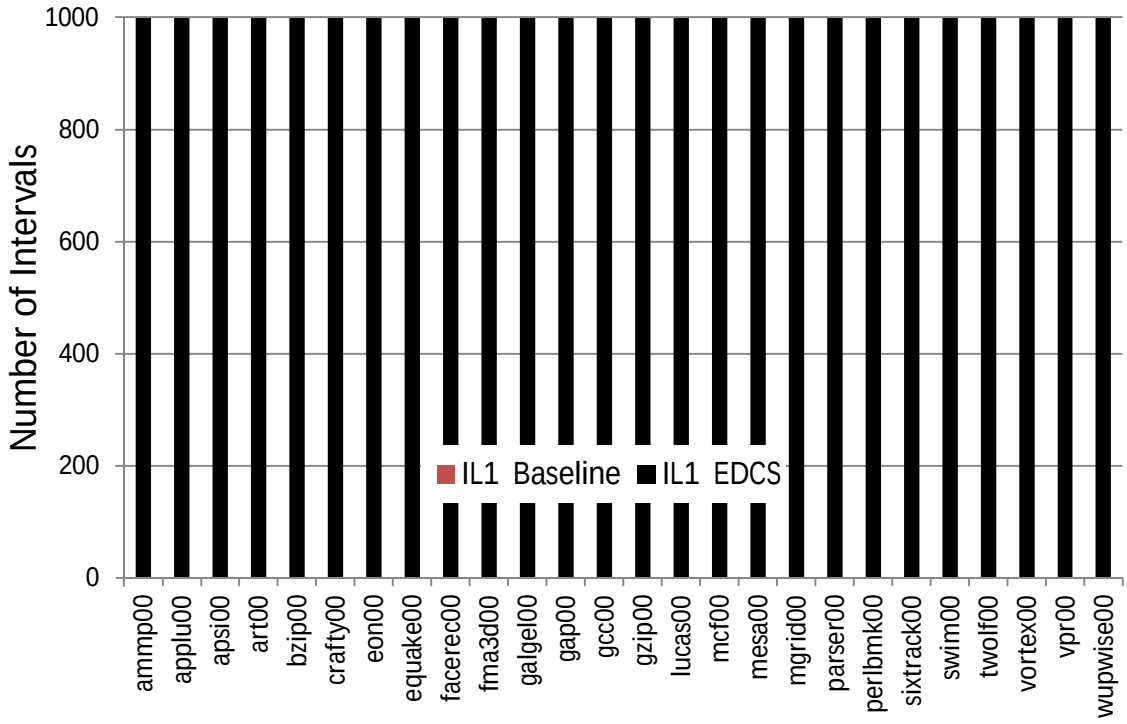
8.2.1 Energy Savings from using EDCS for different Caches

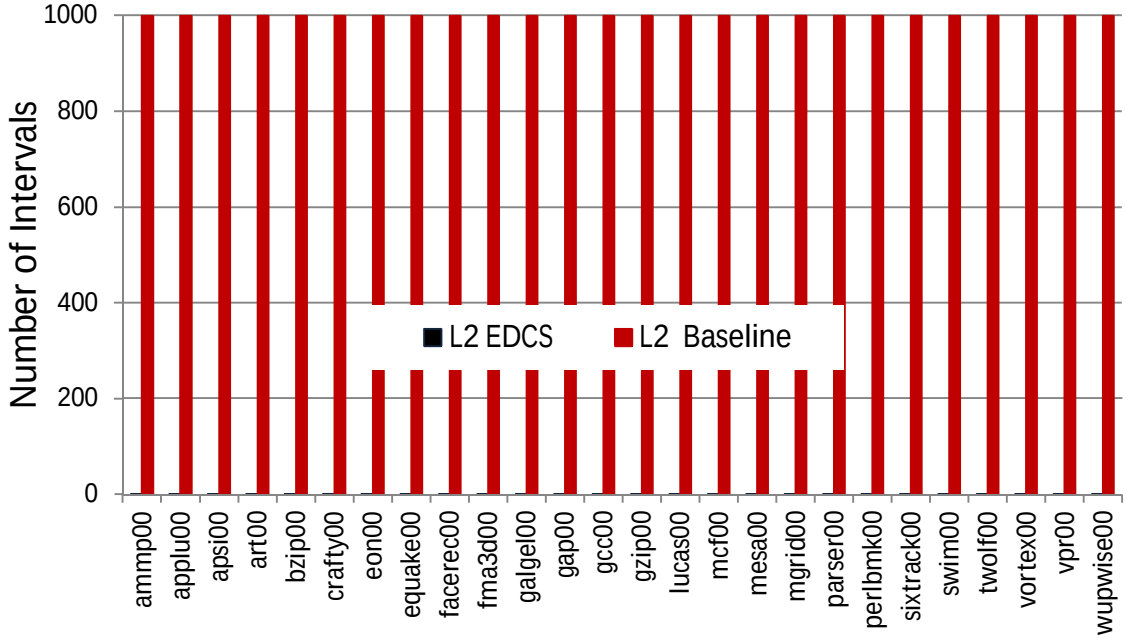
In this Chapter we present the results for each array and benchmark for three interval length.

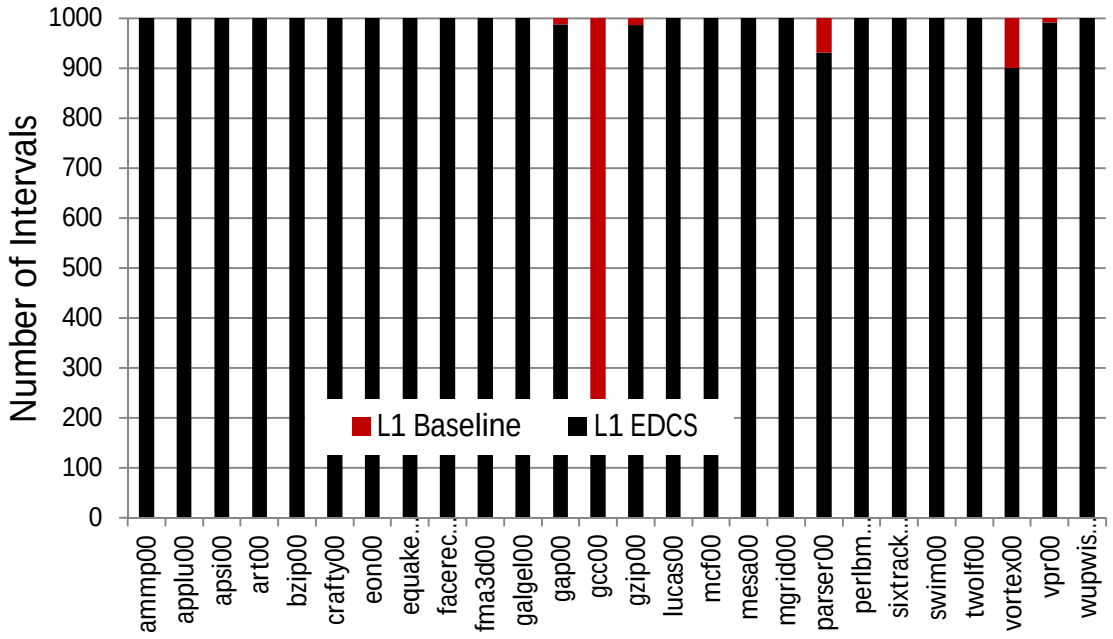
We propose like LBFD to extend the dual mode protection at the granularity of the intervals. An adaptive scheme considers at the end of each interval, depending of the number of different types of accesses, if the EDCS based scheme has advantage over the baseline and decides what protection mode to use for the next interval.

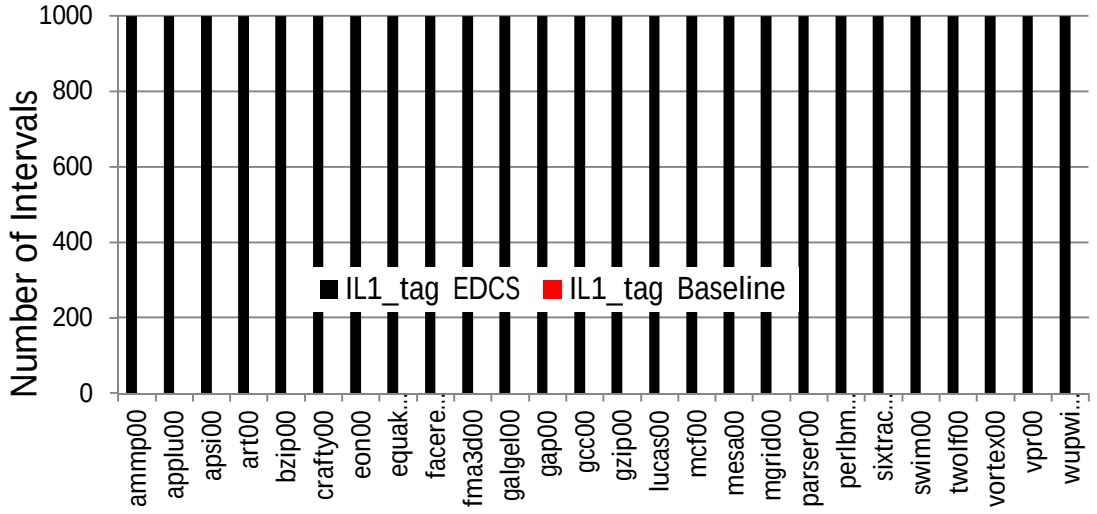
In general the benefits of the EDCS technique are become if we don’t have a lot of changes from a mode to the other.

Figure 8.1 presents how many times we change the mode for each array and benchmark for 100K interval length. This interval length was selected, because selects more times the EDCS mode than the baseline and, therefore, more likely to benefit from the techniques proposed in this report.


Number of Intervals
1000
800
600
400
200
0
IL1 Baseline
IL1 EDCS
ammp00
applu00
apsi00
art00
bzip00
crafty00
eon00
equake00
facerec00
fma3d00
galgel00
gap00
gcc00
gzip00
lucas00
mcf00
mesa00
mgrid00
parser00
perlbmk00
sixtrack00
swim00
twolf00
vortex00
vpr00
wupwise00


Number of Intervals
1000
800
600
400
200
0
L2 EDCS
L2 Baseline
ammp00
applu00
apsi00
art00
bzip00
crafty00
eon00
equake00
facerec00
fma3d00
galgel00
gap00
gcc00
gzip00
lucas00
mcf00
mesa00
mgrid00
parser00
perlbmk00
sixtrack00
swim00
twolf00
vortex00
vpr00
wupwise00


Number of Intervals
1000
900
800
700
600
500
400
300
200
100
0
L1 Baseline
L1 EDCS
ammp00
applu00
apsi00
art00
bzip00
crafty00
eon00
equake...
facerec...
fma3d00
galgel00
gap00
gcc00
gzip00
lucas00
mcf00
mesa00
mgrid00
parser00
perlbm...
sixtrack...
swim00
twolf00
vortex00
vpr00
wupwis...


Number of Intervals
1000
800
600
400
200
0
IL1_tag EDCS
IL1_tag Baseline
ammp00
applu00
apsi00
art00
bzip00
crafty00
eon00
equak...
facere...
fma3d00
galgel00
gap00
gcc00
gzip00
lucas00
mcf00
mesa00
mgrid00
parser00
perlbm...
sixtrac...
swim00
twolf00
vortex00
vpr00
wupwi...

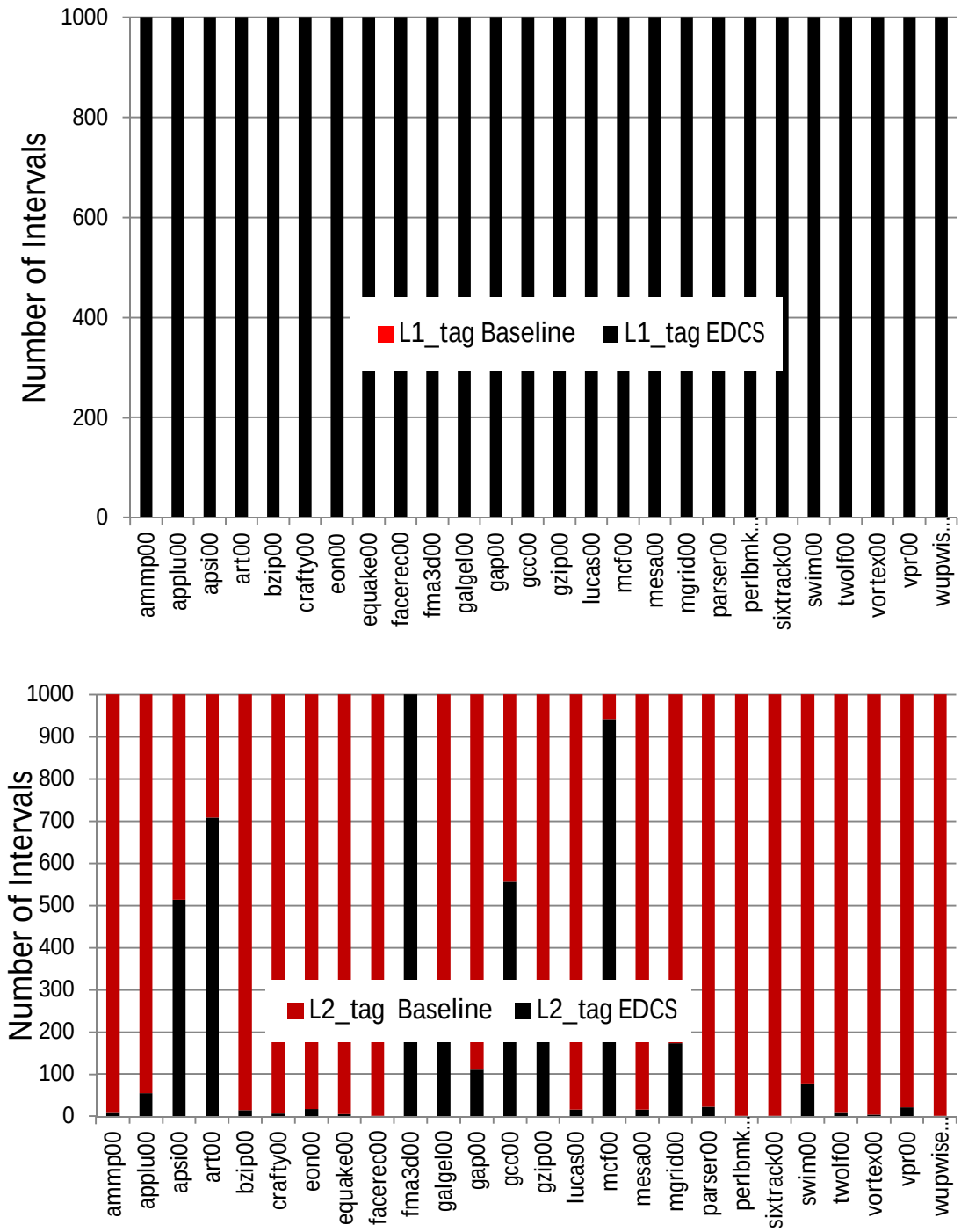


Figure 8. 1: Times that we change the mode for each array and benchmark for 100K interval length.

There results are taken using the interval algorithm that we explain in the Chapter 6.

The results shown that the EDCS can be used in all the arrays and provide leakage savings except the L2 data and tag array that uses more time the Baseline mode . L2 data works only in the baseline mode for all the benchmarks so it is not efficient to evaluate the EDCS approach for this array. For the L2 tag array we obtain that in some benchmarks use EDCS approach more than the baseline but it is also non efficient to implement the EDCS technique for this array because fort the most bencmarks it is turn only in the baseline mode. Therefore, we can use this approach for L1 DCache and L1 ICache for data and tag array.

The results in Figure 8.2 present the leakage consumption for the tag and data arrays of three different caches provided by LBFD and EDCS techniques. For simplicity we present the results for one benchmark: mesa00. The results are the same for all the benchmarks.

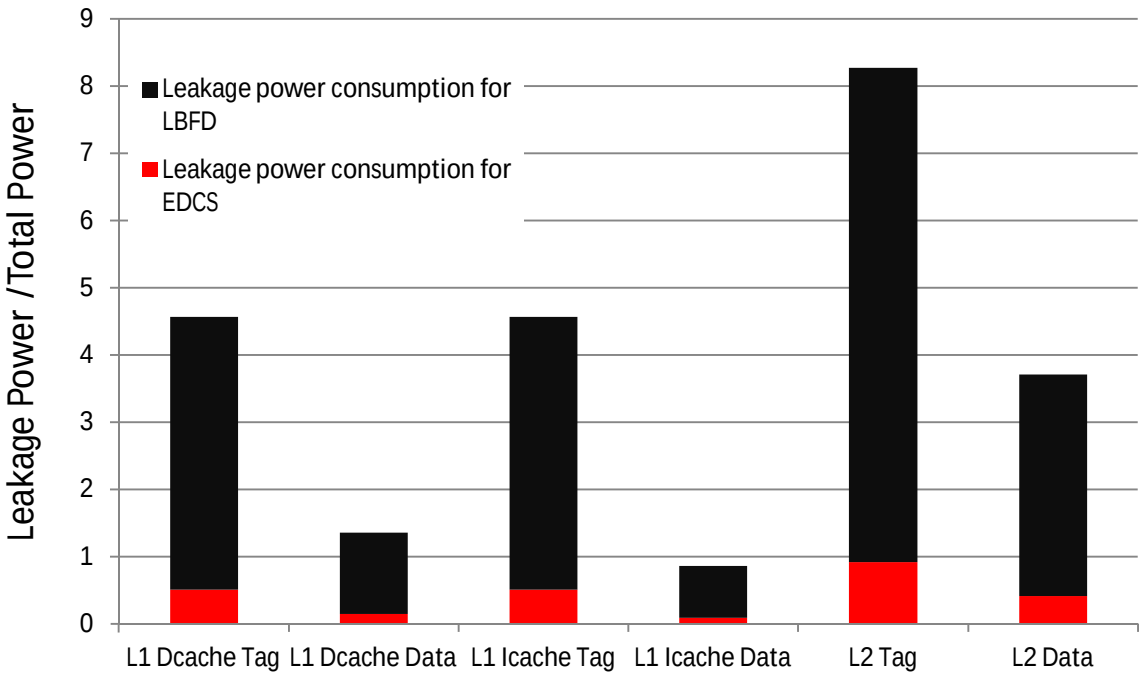


Figure 8. 2 : Relative leakage savings for EDCS.

The results show that the EDCS can provide substantial leakage savings and consequently total power savings in all arrays.

Chapter 9

Fault coverage

9.1 Fault coverage issues.

The EDCS approach can be applied to architectural arrays. The coarser granularity of EDCS can translate to more savings over LBFD but at the expense of lower fault coverage. The fault-coverage of EDCS method depends on the fault-behavior and the granularity of protection (how many words are protected by a shared code). By fault behavior we distinguish between random pfail for each cell or cluster of faults.

We analyze the fault coverage for a specific array configuration: a 64KB data cache data array, with 512 bit data a\nd 8 bit code for each 64 bit data. We consider two baseline protection schemes byte parity and a 72-64 SECDED with the check matrix in [14]. For EDCS we consider arrays where multiple bytes share a parity bit (Global-parity) and multiple words shared the same 8 bit SECDED code (Global SECDED). Recall that EDCS encodes the shared code as the xor of the code of the individual words.

The formulas that we use to obtain these data are based on binomial probability and for the SECDED code an enumeration is needed for a given number of faults how many column combinations in the check matrix provide a zero syndrome [14].

We turn now to formal mathematical settings for analyzing the fault coverage for the

EDCS approach. Any probabilistic statements must refer.

The main idea is to use the different techniques and try to figure out the probability to detect the errors. We illustrate multiple two scenarios, one for a given number of fails and the other is for a random cell pfail. We present only the scenario with a random cell pfail because is more generic.

A probability space has the following inputs.

1. *f*, that is a given number of fails.
2. *n*, that is the number of bits that correspond to a word plus the check bits that are used to protect the word.
3. *w*, which is the total number of words that are protected from a code.
4. A probability function *pwf*, that is the EDCS word failure to detect.
5. *pfail*, that is the given probability for random cell failures.

The *pfail* determines the probability to have or not a failure in a cell.

a. Byte parity

Some characteristics: The even-parity bit can detect only the odd number that occurs in a word. For example can detect 1 error, 3 errors, 5 errors,…etc.

$$pwf = \sum_{\substack{i=2 \\ + \\ i\%2=0}}^{n} \binom{n}{i} pfail^{i}(1-pfail)^{n-i}$$

b. SEC-DED (Hsiao)

Some characteristics: ECC: Detects all the odd number faults that occurs in a word and two bit errors.

$$pwf = \sum_{\substack{i=4 \\ + \\ i\%2=0}}^{n} \binom{n}{i} pfail^{i}(1-pfail)^{n-i} \frac{W(i)}{\binom{n}{i}}$$

c. Global Parity

Some characteristics: ECC: Detects all the odd number faults that occurs in a number words.

$$pwf = \sum_{f=1}^{w} \binom{w}{f} \left(\sum_{\substack{i=2 \\ + \\ i\%2=0}}^{n} \binom{n}{i} pfail^{i}(1-pfail)^{n-i} * \left(\binom{n}{0} pfail^{0} pfail^{n-0} \right)^{1-f} \right)$$

d. Global SEC-DED (Hsiao)

$$pwf = \sum_{\substack{i=2 \\ + \\ i\%2=0}}^{n*w} \binom{n}{i} pfail^{i}(1-pfail)^{n-i} \frac{W(i)}{\binom{n}{i}}$$

where the W(f) is the number of cases that will give the zero syndrome for all the words that are protected from a shared code and thus be undetected for every f.

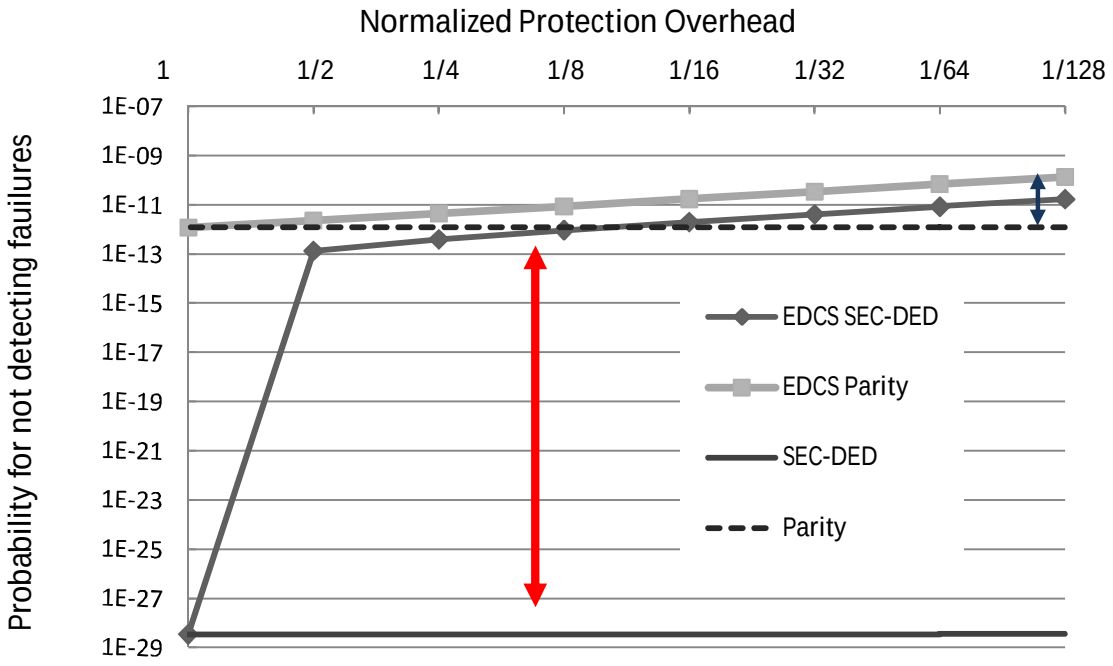


Figure 9. 1: Probability for not detecting random cell failures with pfail = 10^{-9}

Figure 6.6 shows the probability of not detecting an error for a random cell pfail=10^{-9} for the four schemes as function of the code overhead of the EDCS schemes as compared to the baseline cost. When the cost of EDCS is 1 that mean each word is protected by a code and behaves as the LBFD scheme. When it is 1/2 it means every two words are protected with 1 code etc.

The data in Figure 9.1 clearly illustrate that EDCS schemes provide lower fault coverage than their respective baseline with lowering number of codes. It is interesting also to observe that for same code size the Global-SECDED scheme is always superior than Global-parity and in some case even better than the byte parity scheme. The

Global-SECDED scheme, however, is significantly worse than the baseline SECDED due to its inability to capture in some cases 2 faults.

Chapter 10

Improving EDCS Fault Coverage (EDCS-LC)

10.1 EDCS-LC Approach.

The propose scheme opens up several directions of work including it's more detailed evaluation/optimization for different applications as well as it's combination and interaction with other error correction and detection techniques. The fault coverage that provides the EDCS scheme can be optimized. To evaluate this optimization we propose another scheme that is based on the EDCS technique with some extra details to increase the fault coverage.

As we told before, the EDCS approach is an approach that is in lieu with the LBFD approach. In particular, EDCS protects multiple data words using the same code instead of having one code per data word. This can reduce leakage energy by power gating unused codes.

The main key is to increase the reliability that EDCS scheme provides with no impact on the energy savings.

Therefore we propose the Error Detection Code Sharing using Legal Codes, (EDCS-LC) that is an error detection approach that introduced for arrays to protect the architectural arrays. Because is an extension of the EDCS approach inherits the basic characteristics of this approach with some exceptions.

The main difference in this approach is to use a *vector* that keeps the track of the legal codes that appear during an interval. Using this vector we provide further protection that we had using the EDCS approach. The state overhead of this vector is 2^k bits for a memory array, where k is the error detection bits that used to protect a word. This vector keeps track of the legal codes that appear in an architectural array at any given time.

In general writes accesses to an array become a read followed by a write. The difference from the EDCS approach is that we keep the read-to-remove and denote it henceforth as *R2R* that is used to *remove* the code of the current value, about to be overwritten, from the shared code whereas a write adds the code of the new value in the shared code but also we implement a *read-to-check* that enables the vector to keep track of the legal codes that appear in a memory array and also provides an error checking in the vector at any given time. To bound again the detection latency between of faults, an array sweep can be performed, possibly periodically and at the end of the program.

In principle EDCS-LC approach proposed to detect the errors that can not cached in the sweep phase.

To highlight the difference between EDCS and EDCC-LC it is good to present an example that is showing the basic functionality of these techniques.

We illustrate the EDCS and EDCS-LC operation during an interval in **Figure 10.1** assuming a 4x8 array that contains initially all zeros and assuming SEC-DEDcode protection. At the initial state the SEC-DED code and the vector are initialized with zero values. Each write access is denoted by a Wi[j] with i representing the write entry and j representing the write value. Similarly Ri[j] denotes reading from entry i the value j. In order to keep updating correctly the SEC-DED code, a R2R for EDCS and R2RC for

EDCS-LC is added before every write. The figure shows how in case of a single error in the array it is not detected from the EDCS approach but is detected from the EDCS-LC approach when it is overwritten.

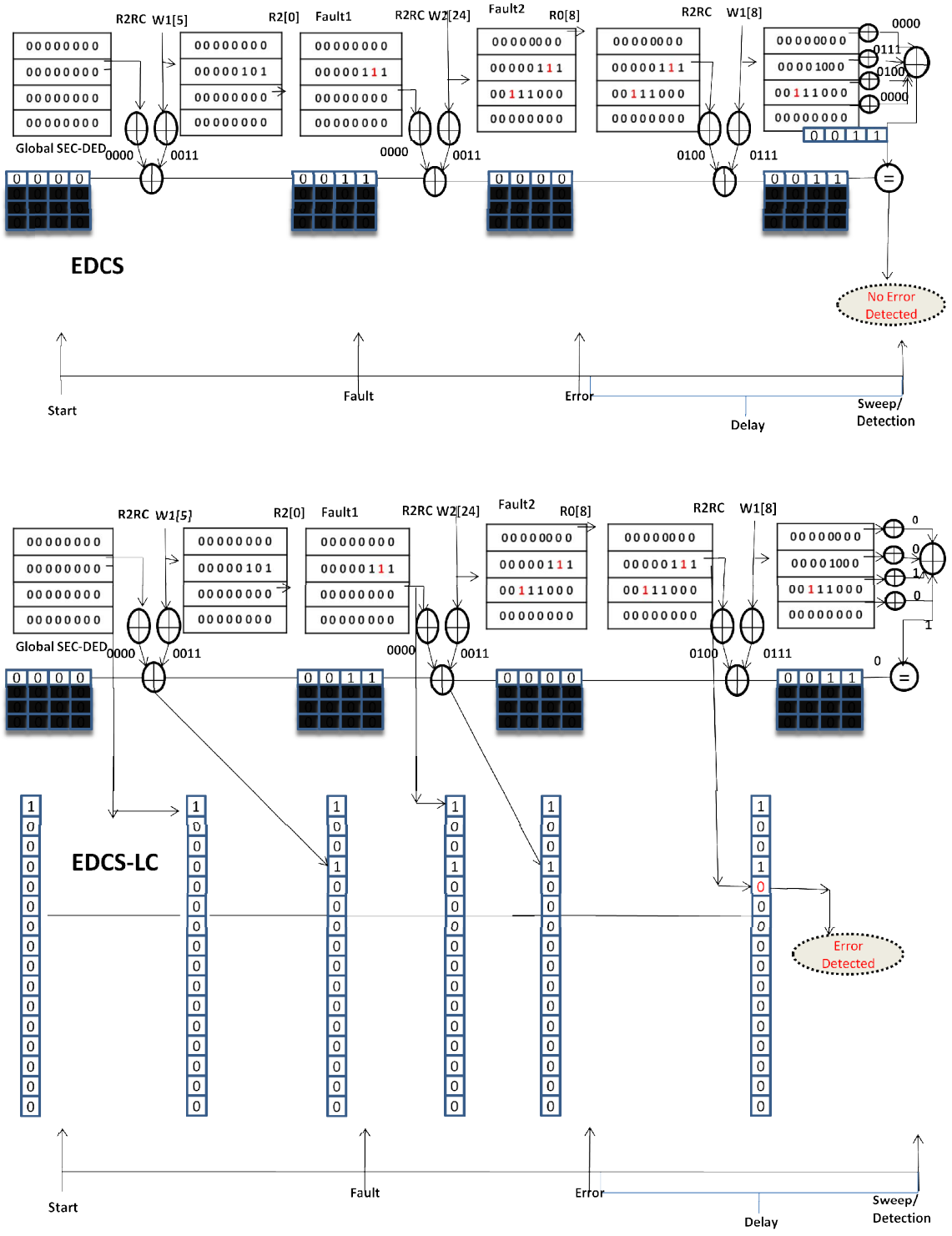


Figure 10. 1: (a) EDCS techniques (b) EDCS-LC.

The EDCS-LC like EDCS approach can be applied to arrays that contain architectural state. To provide how EDCS-LC can be used to detect faults in tag and data arrays for caches we made the following assumptions: a) the arrays protected with SEC-DED code, b) the data are stored in logic blocks in the cache where each block are consisted from 64bits data, and c) the cache is n-way associated cache.

EDCS-LC potential benefits come from that it use one code word to protect multiple data words. However, this means that the area that is used to store the EDC codes may be disabled. Figure 10.2 presents the logical organization of a cache using the EDCS-LC technique for a way that the EDC bits are not used. The gray colour is used to represent the disabling of EDC codes as shown in the figure. The organization of the vector illustrates in Figure 10.2 It has one column and one row for all the corresponding EDC code.

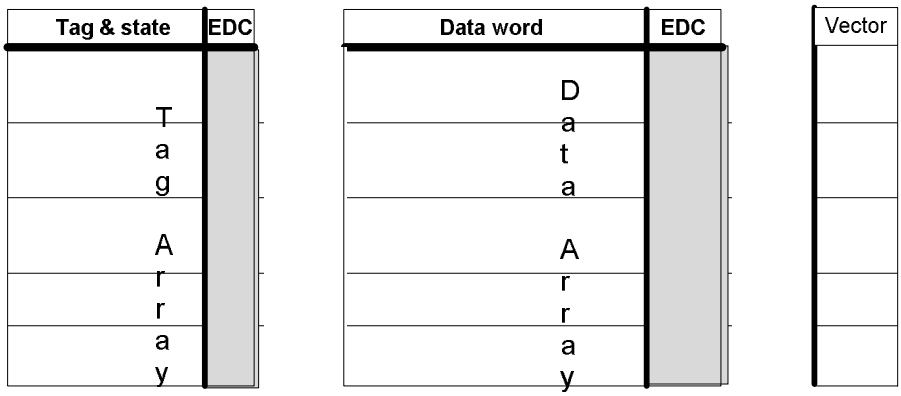


Figure 10. 2: Logical organization of a cache using the EDCS-LC technique.

Figure 10.3 illustrates the completely logical organization of a cache using the EDCS-LC technique during write access, respectively. We choose to not show the logical organization of a read access because is the same with the LBFD approach.

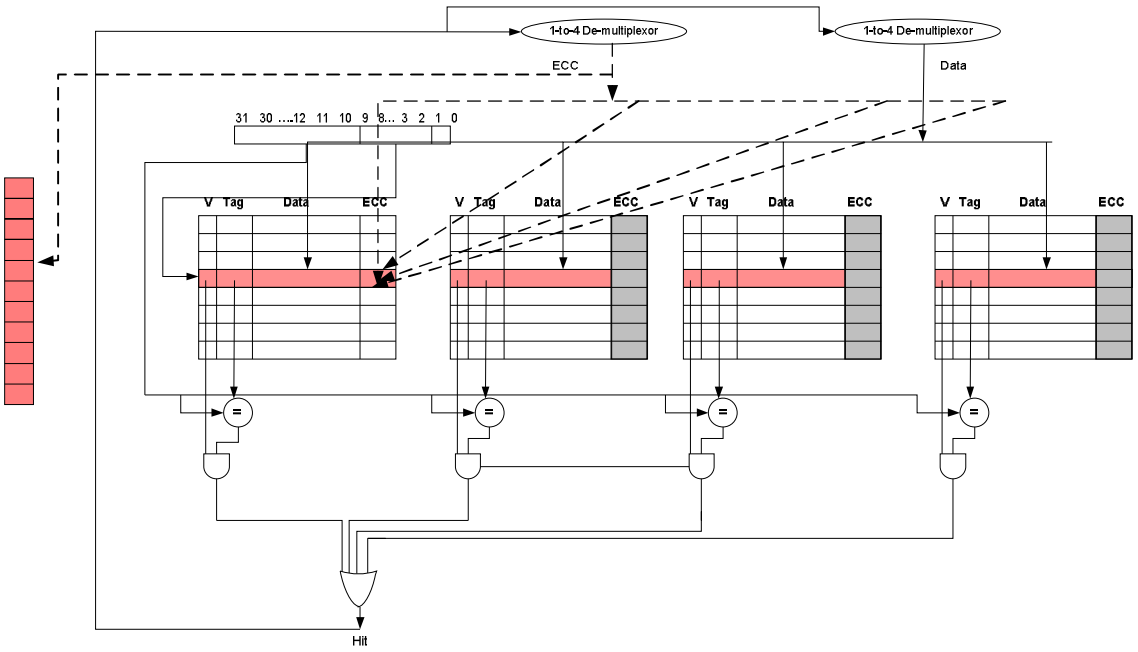


Figure 10. 3: 4-way associative cache-write access.

As we can see from the Figure 10.3 in the case of a write access, there is no need to write and drive the EDC bits in all ways like EDCS. There is only a need to store the EDC bits in one way, so since the full swing is avoided to write the error detection bits, the resultant precharge is also avoided for n-1 ways. The organization of the vector, is similar to an ECC array. For all the possible codes the vector has a corresponding row for that EDC code.

The operation of the vector is as follows:

Store hit in a memory array: Read the overwritten data and compute the EDC from the cache. Check if the EDC codes belong to the legal codes in the vector. If not we have an error. Write data in the array and update the vector with the new code.

The EDCS-LC approach can be applied to architectural arrays. The fault-coverage of EDCS-LC method depends to the EDC codes that appear during an implementation at a given time. To measure how we increase the fault coverage we have to keep track of the codes that appear in a cache at a given time, this is not in this thesis purposes but propose for future work. In the next Chapter we do some analysis for the codes to minimize the appearance of the legal codes at any given time. Assume for example that we have a space with 2^k points that represent all the EDC codes. Figure 10.4 shows these points with black colour. A sphere defines as that all the points that are on the surface are the legal codes that we can see at a given time in a cache array. The other codes are the illegal codes that at a given time it does not obtained in the cache array. In fact we are trying to minimize the sphere radius and consequently the legal codes. Figure 10.5 shows this operation.

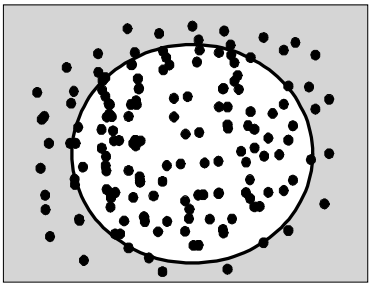


Figure 10. 4: Geometrical model that represents the EDC.

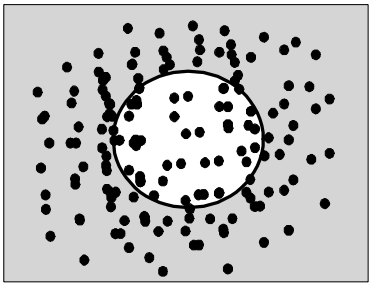


Figure 10. 5: Geometrical model that represents the EDC.

This in turn means that our technique may increase scientifically the fault coverage.

Chapter 11

Analyzing Error detecting and Correcting Codes

11.1 Evaluation of Hsiao code.

Clearly, a clever algorithm is needed, to select the best codewords, and to verify them on the fly, as they are being read. In the write access should use the algorithm to generate the codewords when they have to be store, and in the read access should use it to check them when they are load. The approach described here is due to an offline analysis of the SEC-DED codes that are described in Section 4. This offline analysis it had be done to help us to minimize the appearance of the legal codes that appear in a cache array at any given time. This is done in many ways, all of them involving some form of coding.

Hsiao is code for single-error correction and double error detection. It is equivalent to the Hamming code in the sense that for specified number k of data bits we used the same number of check bits. This code it used more in hardware because is better in performance, cost and reliability than Hamming code. Based on that we focused our analysis more on Hsiao algorithm than Hamming.

11.2 Distribution of the different ECC codes.

One important issue is to investigate the behaviour of 128 codes. Therefore we calculate and plot the ECC code for all the 2^{32} data.

Figure 11.1 shows the appearance of each ECC code for different data for Hsiao. The x-axes shows the different data and the y-axes shows the different ECC codes in integer form. With the red colour we see the appearance of each ECC code and with the white colour we see the range that respectively ECC don't appear.

The data show that the appearance of the 128 ECC codes in Hsiao algorithm is distributed.

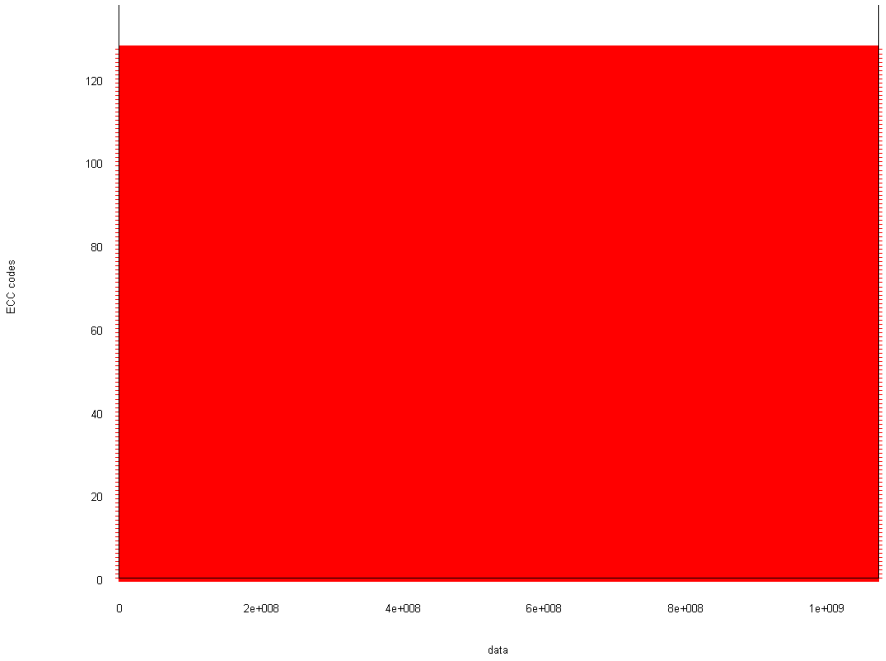


Figure 11. 1: Appearance of ECC codes using initial Hsiao

11.3 Distance of the different ECC codes.

As we can obtain from the previous graph, different data words can have the same ECC code. This is logic because the ECC bits that are used for protection are less than the corresponding data bits.

I consider the **distances** which is the number between two data words that have the same ECC code, if were listed in ascending order.

Figure 11.2 depicts the cumulate graph that shows many times each distance appears for every code in Hsiao algorithm. The x-axes shows the different distances and the y-axes shows how many times each distance appears.

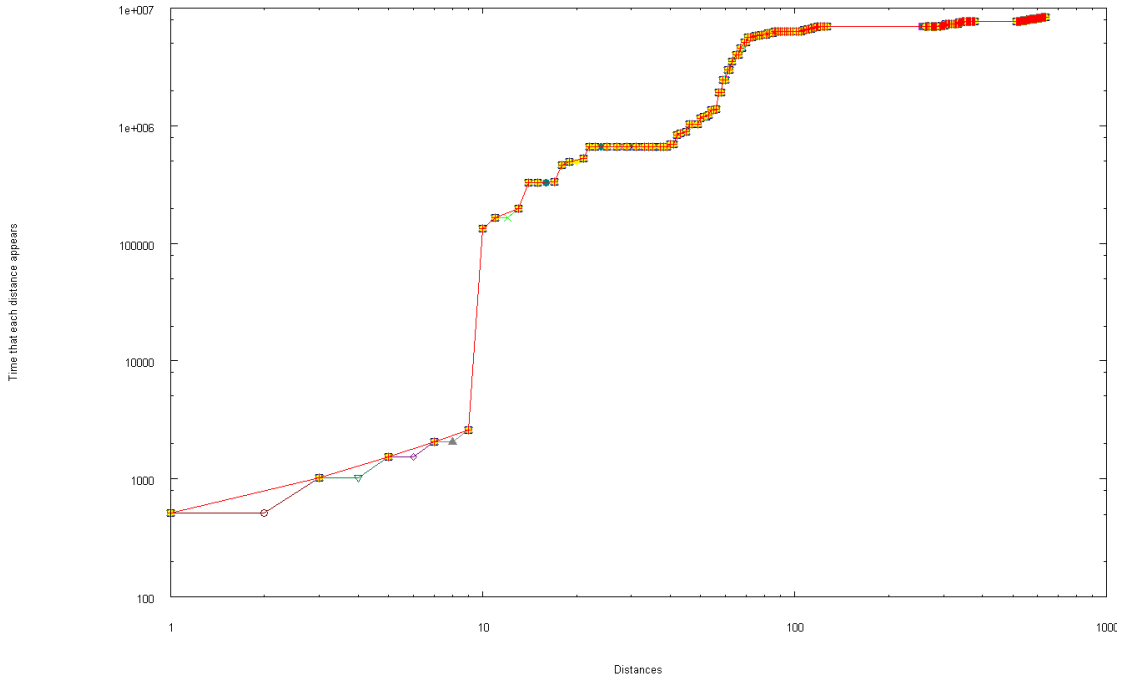


Figure 11. 2: Cumulate graph that shows many times each distance appears for every code in Hsiao algorithm

As can be extracted from Figure 11.2 the maximum distance for Hsiao is 600.

11.4 More analysis in Hsiao algorithm.

As we mention above is more efficient to investigate and processed the Hsiao code because it used more often in computer memories for it's better performance, cost and reliability.

We know that the H-matrix for Hsiao algorithm is not constant but it can be built from some requirements, unlike with Hamming that the algorithm builds only one constant matrix. So how we can know that the matrix we use is the most appropriate?

11.5 Investigating the H-Matrices for Hsiao Algorithm.

Table 11.1 shows the parity-check matrix for 32 data bits and 7 check bits. Assuming d0,d1,d2,d3,…,d31 are the information bits, and the c0,c1,c2,…,c6 are the check bits as shown below. Note that there are 32 rows corresponding to 32 of the 35 possible combinations of 3-out-of-7. The total number of 1's in the H-matrix, therefore is equal to (3*32)=96 and the average number of 1's in each column is equal to 96/7≈14.

	C0	C1	C2	C3	C4	C5	C6
d0	1	0	0	0	0	1	1
d1	1	0	0	0	1	0	1
d2	1	0	0	1	1	0	0
d3	1	0	1	0	0	0	1
d4	1	1	0	0	0	0	1
d5	1	0	0	0	1	1	0
d6	1	0	0	1	0	1	0
d7	1	1	0	0	1	0	0
d8	0	1	0	0	0	1	1
d9	0	1	0	0	1	0	1
d10	0	1	0	1	0	0	1
d11	0	1	1	0	0	0	1
d12	0	1	0	0	1	1	0
d13	0	1	0	1	0	1	0
d14	1	1	0	0	0	1	0
d15	0	1	0	1	1	0	0
d16	0	0	1	1	0	1	0
d17	0	0	1	0	0	1	1
d18	0	1	1	0	0	1	0
d19	1	0	1	0	0	1	0
d20	0	0	1	0	1	1	0
d21	0	1	1	0	1	0	0
d22	1	0	1	0	1	0	0
d23	0	0	1	0	1	0	1
d24	1	1	0	1	0	0	0
d25	0	0	0	1	1	0	1
d26	0	0	1	1	1	0	0
d27	0	0	1	1	0	0	1
d28	0	0	0	1	1	1	0
d29	0	1	1	1	0	0	0
d30	1	0	1	1	0	0	0
d31	1	0	0	1	0	0	1

Table 11. 1: Parity check matrix of the (39,32) SEC-DED [14]

11.6 Permutation Function.

The function that propose in this thesis maximizes the distances of the codes in Hsiao algorithm. Distances are maximizing with no impact on area and delay.

The key issue is to select the H-matrix in a way to maximize the distances of the appearance of the same codes. For odd-weight columns codes there is a degree of freedom in selecting the H-matrix that can be used. This degree of freedom is simply permuting the columns. This has no impact on area or delay as it does not change either the total number of 1's in the H-matrix or the balancing of 1's among the rows.

The goal is to take the initial matrix that Hsiao provide us in his paper and apply an optimization technique that it depends from some permutations.

First we start doing some trivial permutations to the matrix for example moving one bit position to another. To see all the general behaviour I made all possible combinations for this permutation. Notice that with this simply permutations we don't have huge changes to the distance but we have some changes.

Then, a matrix sorting is performed. The approach is to select a parity check matrix using some properties and implement the corresponding check matrix to maximize the distances. For this reason we propose some permutations. We can see the new matrix below.

	C0	C1	C2	C3	C4	C5	C6
d0	1	1	0	1	0	0	0
d1	0	1	1	1	0	0	0
d2	1	0	1	1	0	0	0
d3	1	0	0	1	1	0	0
d4	0	1	0	1	1	0	0
d5	0	0	1	1	1	0	0
d6	1	1	0	0	1	0	0
d7	0	1	1	0	1	0	0
d8	1	0	1	0	1	0	0
d9	1	0	0	0	1	1	0
d10	0	1	0	0	1	1	0
d11	0	0	1	0	1	1	0
d12	0	0	0	1	1	1	0
d13	1	0	0	1	0	1	0
d14	0	1	0	1	0	1	0
d15	0	0	1	1	0	1	0
d16	1	1	0	0	0	1	0
d17	0	1	1	0	0	1	0
d18	1	0	1	0	0	1	0
d19	1	0	0	0	0	1	1
d20	0	1	0	0	0	1	1
d21	0	0	1	0	0	1	1
d22	1	0	1	0	0	0	1
d23	1	1	0	0	0	0	1
d24	0	1	1	0	0	0	1
d25	0	1	0	1	0	0	1
d26	0	0	1	1	0	0	1
d27	1	0	0	1	0	0	1
d28	0	0	0	1	1	0	1
d29	1	0	0	0	1	0	1
d30	0	1	0	0	1	0	1
d31	0	0	1	0	1	0	1

Table 11. 2: Permuted parity check matrix of the (39,32) SEC-DED.

Figure 11.3 shows the data bits that each ECC code appears using the permuted matrix. We obtain that in large data set some codes don’t appear. Compared to the Hamming code the range again is not too large but we can see a big difference, so this method shows excellent results.

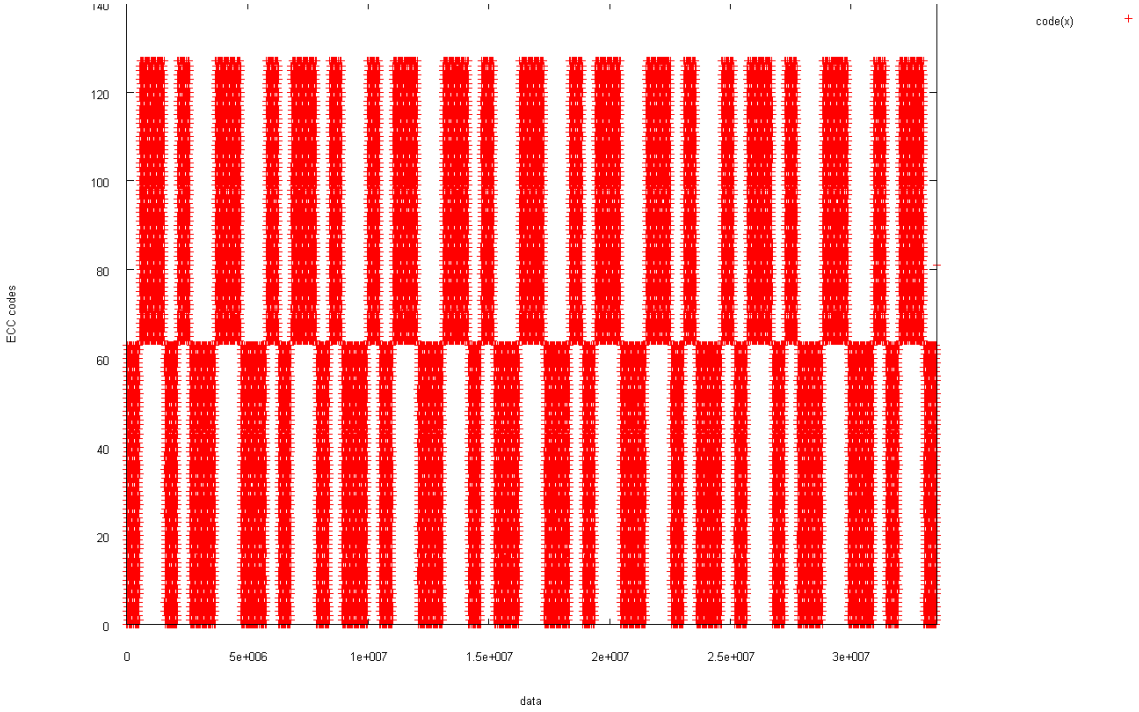


Figure 11. 3: Appearance of ECC codes using permuted Hsiao

Figure 11.4 shows the distribution of the distances. In the x axis I plot each distance and in the y-axis I plot how many times each distance appears for all the 128 ECC codes.

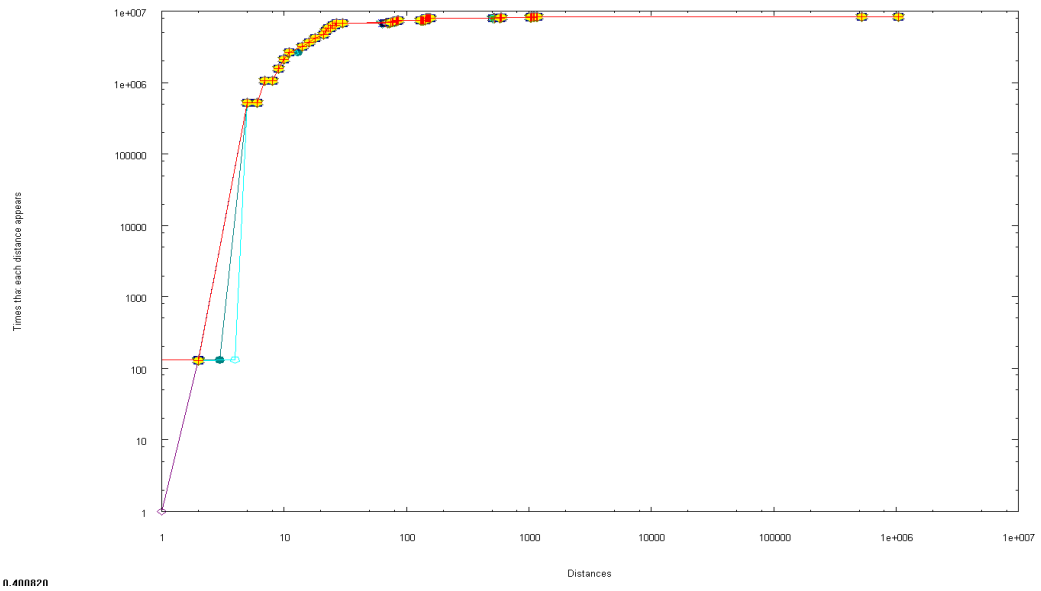


Figure 11. 4: Cumulate graph that shows many times each distance appears for every code in Hsiao algorithm

Compared the two algorithms, the Hsiao with the permuted has greater maximum distance from Hsiao with the initial matrix. The maximum distance now from 600 it began about 1,000,000.

It can be easily shown that this operation is a valid operation because we don't ignore some special properties that the codes give us, for example the ability of detection and correction. To see this it's better to make an example. So let us to develop a single error correcting code for k=8. The number of bits required to correct single bit and detect double bit inversions is r = 5. The permuted parity check matrix for the initial (16, 22) Hsiao code, it shown below.

	C0	C1	C2	C3	C4	C5	C0	C1	C2	C3	C4	C5
d0	0	0	0	1	1	1	1	1	1	0	0	0
d1	0	0	1	1	0	1	1	1	0	0	1	0
d2	0	0	1	1	1	0	1	1	0	0	0	1
d3	0	1	0	0	1	1	1	0	0	0	1	1
d4	0	1	0	1	1	0	1	0	1	0	0	1
d5	0	1	1	0	0	1	1	0	0	1	0	1
d6	0	1	1	0	1	0	0	1	1	1	0	0
d7	0	1	1	1	0	0	0	1	1	0	1	0
d8	1	0	0	0	1	1	0	1	1	0	0	1
d9	1	0	0	1	0	1	0	1	0	1	1	0
d10	1	0	0	1	1	0	1	0	0	1	1	0
d11	1	0	1	0	0	1	0	0	0	1	1	1
d12	1	0	1	1	0	0	0	1	0	0	1	1
d13	1	1	0	0	0	1	0	0	1	1	1	0
d14	1	1	0	0	1	0	1	0	1	1	0	0
d15	1	1	1	0	0	0	0	0	1	1	0	1

From the matrix the check bit equations are derived as follows:

C0= d8 ⊕d9⊕d10⊕d11⊕d12⊕d13⊕d14⊕d15

C1= d3⊕d4⊕d5⊕d6⊕d7⊕d13⊕d14⊕d15

C2= d1⊕d2⊕d5⊕d6⊕d7⊕d11⊕d12⊕d15

C3= d0⊕d1⊕d2⊕d4⊕d7⊕d9⊕d10⊕d12

C4= d0⊕d2⊕d3⊕d4⊕d6⊕d8 ⊕d10⊕d14

C5= d0⊕d1⊕d3⊕d5⊕d8 ⊕d9⊕d11⊕d13

For example if,

d0d1d2d3d4d5d6d7d8d9d10d12d13d14d15=0011001100110011

then the parity check bits are

C0C1C2C3C4C5=011010

Now let us assume bit d6 has changed from 1 to 0.

d0d1d2d3d4d5d6d7d8d9d10d12d13d14d15=0011000100110011

The check bits can be recomputed as:

C0'C1'C2'C3'C4'C5'=000000

Therefore the syndrome bits are:

e0e1e2e3e4e5=011010

It can be seen that the syndrome bits match with the seventh row in the parity matrix, identifying d6 as an erroneous bit.

It can be seen that the syndrome bits match again with the seventh row in the new parity matrix, identifying d6 as an erroneous bit.

Overall, our example demonstrate that optimizing the input permutations of the H-matrix of the memory ECC give us significant increase of distance, while simultaneously keeps the correctness of the algorithm.

Concluding it is evident that we minimize the legal codes for EDCC-LC approach.

Chapter 12

Conclusions and Future Work

This thesis introduces the notion of error detection codes and proposes EDCS and EDCS-LC mechanisms that are new approaches for memory array protection that does not check for errors on a read but lazily before a value is overwritten or during an array sweep.

EDCS, is a mechanism that uses one code word to protect multiple data words for further reduction of energy and possible area used for memory array protection. Overall our experimental results for an out-of-order processor with an n-way cache demonstrate that the reduction of leakage consumption is almost to a. This, however, may come at the expense of lower fault-detection coverage.

Therefore we propose also the EDCS-LC approach that increases the fault detection coverage. The key idea of this approach is that uses one vector that keeps track of the legal error detecting codes that appear in the architectural arrays at any given time. Analysis of error detection codes shows that we can minimize the EDC codes that appear in an architectural array and consequently the propose technique increase the fault coverage.

The thesis points to several direction of future work. One is to implement the dynamic implementation of EDCS-LC approach to measure if the fault coverage increases or not.

One other important direction of research is to consider other policy optimizations that can lead to better fault coverage and reduction of energy consumption. Finally,

dynamic power complexity issues of EDCS and EDCS-LC techniques need to be investigated.

References

[1] Bushra Ahsan, Lorena Ndreu, Isidoros Sideris, Yiannakis Sazeides, Sachin Idgunji and Emre Ozer, “Eliminating Energy of Same-Content-Cell-Columns of On-Chip SRAM Arrays”, International Symposium on Low Power Electronics and Design 2011 Fukuoka, Japan, August 1-3, 2011

[2] Daniel J. Sorin, “Fault Tolerant and Testable Computing Systems Faults and Their Causes”, Duke University, ECE 254 / CPS 225, 2008

[3] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet, “ Improving the availability of shared memory multiprocessors with global checkpoint/recovery”, In ISCA, 2002.

[4] F. Pollack, “Packaged microprocessor power density has risen exponentially for decades”, MICRO-32, Haifa, Israel, 1999.

[5] Gerth Stolting Brodal and Rolf Fagerberg, “Cache Oblivious Distribution Sweeping”, Department of Computer Science, University of Aarhus, Ny Munkegade, C, Denmark

[6] Gordon E. Moore, “Cramming More Components onto Integrated Circuits”, IEEE.

[7] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M.Schwarz, and M. T. V aden. Ibm power6 microarchitecture. IBM Journal of Research and Development, 51(6):639 –662, 2007. 2

[8] J. Adam Butts and Gurindar S. Sohi, "A Static Power Model for Architects", 33rd Annual International Symposium on Microarchitecture (MICRO-33), December 2000.

[9] Jiantao Pan, "Dependable Embedded Systems",18-849b , Spring 1999

[10] J. P. Kulkarni, K. Kim, and K. Roy, "A Fully differential, robust Schmitt trigger based sub-threshold Sram", In ISLPED '07: Proceedings of the 2007 international symposium on Low power electronics and design, pages 171–176, 2007. 2

[11] Kaxiras S., Hu Z., and Martonosi M. "Cache decay: exploiting generational behaviour to reduce cache leakage power", International Symposium on Computer Architecture (ISCA2001)., Sweden.

[12] Linwei Niu, Gang Quan, "Reducing Both Dynamic and Leakage Energy Consumption for Hard RealTime Systems", Department of Computer Science and Engineering University of South Carolina Columbia, 2004

[13]M. Prvulovic, J. Torrellas, and Z. Zhang. Revive, "Cost-effective architectural support for rollback recovery in hared-memory multiprocessors", In ISCA, pages 111 122, 2002.

[14] M. Y. Hsiao, “A Class of Optimal Minimum Odd-weight-column SEC-DED Codes,” July (1970)

[15] Milind B.Kample and Kanad Chose, “Analytical Models For Low Power Caches”, State University of New York, Bighamton, NY 13902-6000

[16] Naveen Muralimanohar, Rajeev Balasubramonian, Norman P. Jouppi, “CACTI 6.0: A Tool to Understand Large Caches”.

[17] R. W. Hamming, “Error detecting and error correcting codes”, The Bell System Technical Journal, 26(2):147–160, 1950. 2 design, pages 171–176, 2007. 2

[18] Shekhar Borkar “Designing Reliable Systems From Unreliable Components: The Challenges of Transistor Variability and Degradation”, IEEE Computer Society 2005 IEEE

[19] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec, “Performance implications of single thread migration on a chip multicore,” in SIGARCH Computer Architecture News, 2005, p. 2005.

[20] Yan Meng, Timothy Sherwood and Ryan Kastner, “On the Limits of Leakage Power Reduction in Caches”, University of California, Santa Barbara HPCA-2005

[21] Yiannakis Sazeides, Panagiota Nikolaou, Lorena Ndreu, Isidoros Sideris, Nikolas Ladas, Quentin Minster, Sachin Idgunji and Emre Ozer," Lazy Based Fault Detection for Memory Arrays", unpublished report 2011 ISCA sbmision LBFD

[22] http://vikashazrati.wordpress.com/2008/10/30/fault-failure-error/, "Byte by Byte Impressions on Technology, People and Process"