SELF-ADAPTIVE CACHE MEMORIES

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea "Politehnica" din Timişoara
în domeniul CALCULATOARE și TEHNOLOGIA INFORMAȚIEI
de către

ing. Liviu AGNOLA

Conducător ştiinţific:	prof.univ.dr.ing Mircea Vlăduțiu
Referenţi ştiinţifici:	prof.univ.dr. Mircea Petrescu
	prof.univ.dr.ing. Liviu Miclea
	prof.univ.dr.ing. Horia Ciocârlie

Ziua susţinerii tezei: 15 Decembrie 2012

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică şi Telecomunicaţii
8. Inginerie Industrială
9. Inginerie Mecanică
10. Ştiinţa Calculatoarelor
11. Ştiinţa şi Ingineria Materialelor

Universitatea „Politehnica” din Timişoara a iniţiat seriile de mai sus în scopul diseminării expertizei, cunoştinţelor şi rezultatelor cercetărilor întreprinse în cadrul şcolii doctorale a universităţii. Seriile conţin, potrivit H.B.Ex.S Nr. 14 / 14.07.2006, tezele de doctorat susţinute în universitate începând cu 1 octombrie 2006.

România, 300159 Timişoara, Bd. Republicii 9,
tel. 0256 403823, fax. 0256 403221
e-mail: editura@edipol.upt.ro

Foreword

This PhD thesis was elaborated during my activity in the Faculty of Automation and Computer Science at "Politehnica" University of Timișoara.

This thesis was elaborated during my activity carried on as part of the doctoral program entitled "Self-Adaptive" Cache Memories. It presents the starting point for the research activity, its directions and obtained results. Reliability is the most important part in computer systems these days. A branch of reliability is fault tolerance. Fault tolerance techniques deal with errors and ensure the correct functioning of any computer systems.

This thesis presents the basic characteristics of cache memories that will be used. Also it introduces the notions and definitions that are used in reliability. The introductory notions end with the methods and techniques used for memory testing. A review of previous research that has been done in the field of memory and cache memory fault tolerance techniques in the last few years is also presented before introducing the original concepts and techniques that together form this thesis. The original contributions of this thesis consist of: a new method for increasing the fault tolerance and reliability of cache memories, a new mathematical model that can predict the position of faults in any memory system, and an improved method of a variation of the triple modular redundancy technique is also presented. All of these results are accompanied with a series of experiments and results.

Timişoara, December 2012 Liviu Agnola

Acknowledgements

The completion of this thesis would not have been possible without the continuous help, support and encouragement of my advisor, prof. dr. ing. Mircea Vladutiu. Without his advices and his immense knowledge of computer hardware design and test engineering, this thesis would not have been accomplished.

I also owe thanks to my colleagues at the ACSA laboratory: Mihai Udrescu and Lucian Prodan. Their expertise in the field of computer hardware design, as well as their important assistance in preparing conference papers and presentations was essential to my work.

Last but not least I would like to thank God, and my family for their moral support and understanding, throughout this difficult and important period of my life.

This work was partially supported by the strategic grant POSDRU/88/1.5/S/50783, Project ID50783 (2009), co-financed by the European Social Fund – Investing in People, within the Sectoral Operational Programme Human Resources Development 2007-2013.

Agnola, Liviu

Self-Adaptive Cache Memories

Teze de doctorat ale UPT, Seria 14, Nr. 12, Editura Politehnica, 2012, 117 pagini, 82 figuri, 21 tabele.

ISSN: 2069-8216
ISSN-L: 2069-8216
ISBN: 978-606-554-593-9

Cuvinte cheie: cache memories, reliability, fault tolerance, built-in self-test, graceful degradation, triple modular redundancy, testing

Rezumat,
Throughout the PhD thesis are addressed topics of very high relevance and actuality regarding the fault tolerance of cache memories. The state of the art in cache memory testing is presented and comparisons between the state of the art and the developed methods are also presented. The first version of the Self Adaptive cache Memories technique is improved twice, obtaining even better results. Another problem that is treated in thesis is the distribution of faults inside a memory, and an original probabilistic method for error location prediction is also presented in this thesis.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

1 Introduction

Ever since digital systems were created there were problems in making sure that the systems are working correctly, i.e. the results offered by the machines are accurate and correct.

With ever growing computing power and memory size this issue has become of great importance. Given the fact that in the last years memory size and speed were increased considerably and also the memory in a computing system accounts for somewhere about 50% of the power that the system uses, and taking into account Moore's law (the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years) it is imperative that the memory works correctly and without faults.

The doctoral program addresses the domain of Computer Science, with emphasis on Computer Hardware Design and Built-In Self-Test/Repair. In the last few decades the main focus in computer systems has shifted from performance towards reliability, yield and robustness. As memory systems continue to decrease in size and increase in capacity, the probability of hard, permanent faults increases, especially in SRAM cells [1]. Due to this fact the usual method of using spare rows or columns, for preventing hard faults can become obsolete [1] [2]. The hard or permanent errors can appear due to process variation [1] [3] and aging [4].

Since in the last few years the memory sizes have increased considerably, also the possibility of errors both during fabrication and normal functioning time has increased. This has directed the main manufacturers of memory chips to use fault tolerant techniques in order to counteract the effects or errors. The most common technique is the use of spare rows and columns that replace the faulty memory locations [5].

The techniques for error detection and correction, such as parity bits or error correction codes only support a limited number of errors that can be detected. For example the use of a parity bit can detect an odd number of errors. The ratio of parity bits to the number of bits is usually 1:8 [5].

Because in very large systems the probability of complete failures as well as the probability of multiple failures can become significant, the main companies that manage these large systems have introduced new solutions. For example IBM has introduced *Chipkill* to treat this issue, this technology is also used by SUN and Google Clusters, Intel uses a similar technology called *SDDC* [5]. These technologies even though are different in name they are very similar in process: they distribute the information stored in the Error Correcting Codes throughout the memory chip such that if the chip fails, the information in the chip can be recreated using these Error Correcting Codes in a different memory chip in the large system [5].

The research in the field of memory reliability and fault tolerance has become very important and has even found a place in the 2012 edition of one of the

most important books in the field of Computer Architecture [5]. This provides more than enough motivation to justify new research direction in the field of memory testing and fault tolerance for memory systems.

My work focuses on improving the reliability and yield of set associative cache memories. In order to address this issue first we will need to present the basics of cache memories, memory testing, built-in self-test solutions and graceful degradation solutions.

The first original contribution of this thesis is to propose a new method that can be implemented on any set associative cache memory and that provides an increase in reliability, yield and functioning time of the memory chip. All of these benefits will be at only a small cost in performance, due to the fact that it is a case of graceful degradation [6] [7] [8]. The increase in reliability, yield and functioning time is achieved by removing from use any faulty cell that has been diagnosed as an incurring hard error [9]. The small cost in performance is achieved from the reorganization of the memory cell array. This is done both for maintaining a high reliability, yield and functioning time of the memory chip. Also it is done for maintaining a relatively high performance of the memory, by reducing the number of misses and increasing the number of cache hits. To this end, we will assume that the cache memory is equipped with a concurrent built in self-test mechanism capable of detecting the hard error that may appear during the use of the chip and also during the production stage [9].

The second original contribution of this thesis is a technique called Simplified Selective Fault Tolerance, which addresses and improves a state of the art technique called Selective Fault Tolerance [10] [11]. This technique relies on the triple modular redundancy technique that has been used for reliability improvement [12]. We will also show a case study of how to use this original technique in order to improve the reliability and yield, while on the other hand reducing the area and energy overhead.

The third original contribution of this thesis is the use of probability theory in determining the most probable distribution of errors in any type of memory systems. This method of determining the most probable distribution of faults in any memory type is applied on the cache memories in conjunction with the first two original contribution of this thesis.

The thesis is structured as follows: chapter 0 will provide the basic notions that will be used throughout this thesis. It will provide a description of the basic notions and concepts on faults and dependability, as presented in [13]; the basic notions and ideas behind cache memories, and provide the description of how they work, as presented in [14] and [15]; also this chapter will provide our readers with an introduction to memory testing, as described in one of the most important books in the field [16]. Chapter 3.2 will provide an introduction in the fields of built-in self-testing for memories; this chapter will also provide an overview of the graceful degradation technique. Chapter 0 will provide an overview of the state of the art in graceful degradation techniques for cache memories. Chapter 5 will provide the basic description of our original method called Self Adaptive cache Memories (SAM)

which greatly improves the state of the art described in chapter 0 in terms of reliability. Chapter 6 describes an original method that applies probability theory in order to determine the most probable distribution of errors in a memory system. Chapter 7 will provide two original methods for improvement of the SAM method described in chapter 5, these improvements reduce the area and energy overhead, while increasing the performance of the original SAM method. Chapter 8 provides an original technique called Simplified Selective Fault Tolerance, which addresses and improves a state of the art technique called Selective Fault Tolerance. Chapter 9 will conclude this thesis and also will provide future research directions.

2 Memory Faults and Testing

2.1 Basic notions and concepts on faults and dependability

In this section we will start by giving some general definitions on faults, errors, failures; also the basic means for fault detection, correction and fault tolerance.

2.1.1 Failures, Errors and Faults

A system is an entity that is interacting with other entities, the other entities may be: humans, other entities, software, hardware, and the external world or physical world [13]. The function of a system is described by the functional specification, and it is what the system is intended to do in terms of functionality and performance [13]. The service that the system is delivering is its behavior as it is perceived by the user, where a user is another system, which receives the service provided by the first system.

In order to be able to define faults, errors and failures we must first state what a correct service of a system is. A system is said to deliver a correct service when the service implements the system function. A failure or a system failure is an event that happens when the delivered service deviates from correct service [13]. A system fails in one of two cases: either the specification did not adequately describe the system function; or because it doesn't comply with the functional specification [13]. A service failure is a transition from correct service to incorrect service [13]. A service outage is the period of delivering an incorrect service, a service restoration is the transition from incorrect service to a correct service [13].

When a system deviates from the correct service state the deviation is called an error. The hypothesized or adjudged cause of an error is called a fault [13]. A fault can be either external or internal of the system. An error is the part of the total state of the system that can lead to its subsequent service failure [13]. A fault is active when it causes an error; otherwise it is called dormant. Many errors don't reach the system's external state and cause a failure [13].

A degraded mode that still offers a subset of needed services to the user is when the functional specification of a system includes a set of several functions and the failure of one or more of the services implementing the functions may leave the system degraded [13]. The specification may identify several such modes, for example: limited service, slow service, emergency service, and others [13].

The manifestation and creation mechanism of faults, errors, and failures are depicted in Figure 2.1, these mechanism presented in Figure 2.1 enable the "chains of threads" to be completed, as illustrated in Figure 2.2.

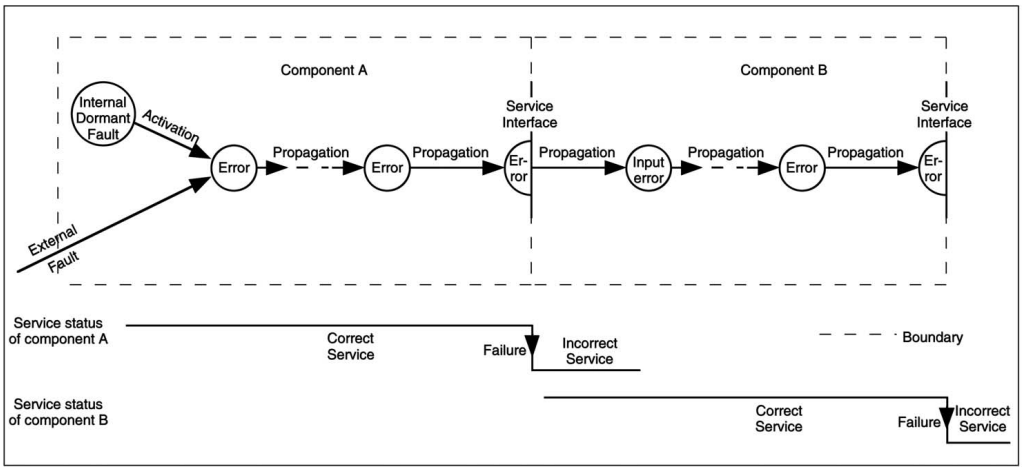

Figure 2.1: Error propagation, from [13].

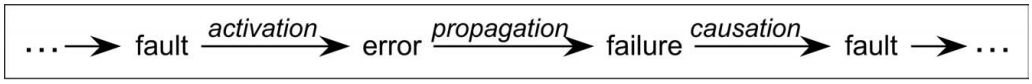

Figure 2.2: The fundamental chain of dependability and security threads, from [13].

Eight basic viewpoints classify all faults that may affect a system during its life, leading to elementary fault classes, as depicted in Figure 2.3.

For a simpler representation we can group the combined fault classes, presented in Figure 2.4, into three groups [13]:

- Interaction faults, that include all external faults
- Physical faults that include all fault classes that affect hardware
- Development faults that include all fault classes occurring during development

2.1.2 Dependability and Security

As presented in [13] there are two valid definitions of dependability, the first, and original definition of dependability is the ability of a system to deliver service that can justifiably be trusted. The other definition for dependability is the ability to avoid service failures that are more frequent and severe than is accepted. The latter definition is providing a criterion for making a decision if a system is dependable or not, while the first definition is stressing the importance of justification.

According to [13] the dependability of a system is an integrating concept that includes the following attributes:

- Availability
- Reliability
- Safety
- Integrity
- Maintainability

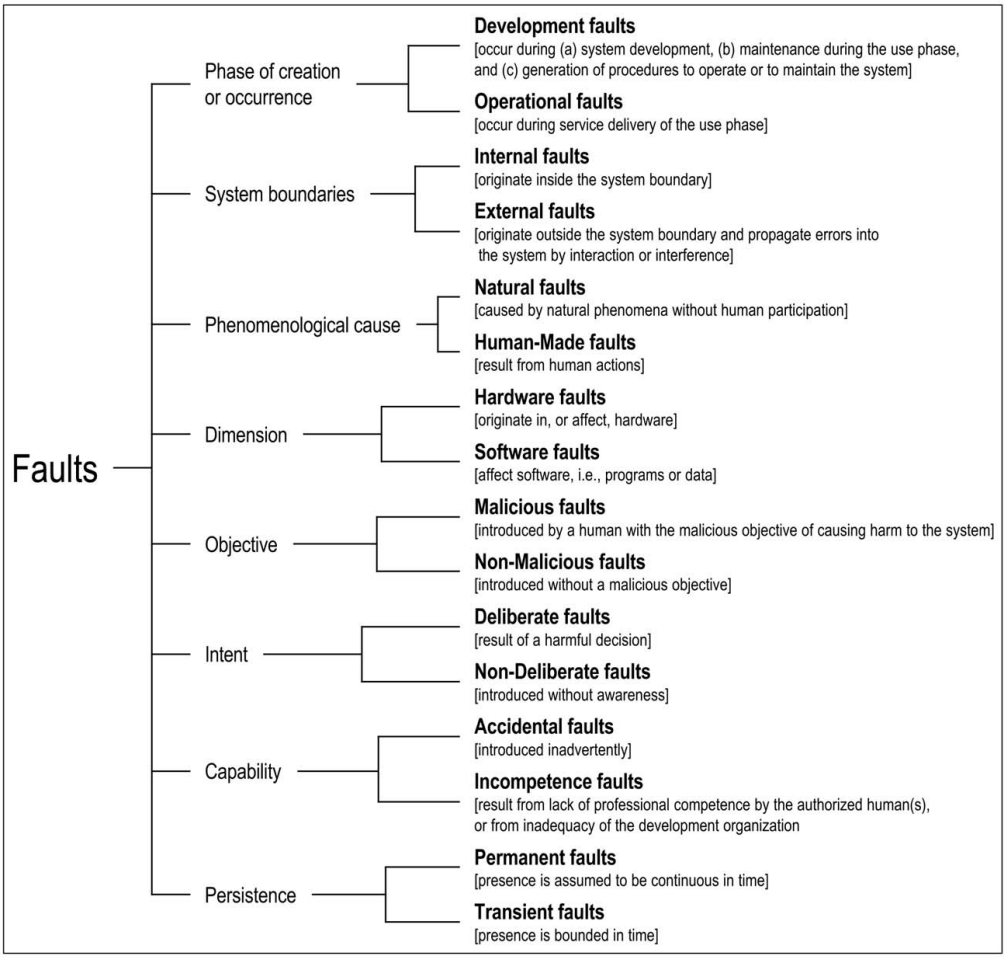

Figure 2.3: The elementary fault classes, from [13].

In the followings we will present the definition of security as illustrated in [13]. Security is a composite of the attributes of confidentiality, integrity, and availability, requiring the concurrent existence of: availability for authorized actions only; confidentiality; and integrity. In Figure 2.5 is summarized the relationship between security and dependability.

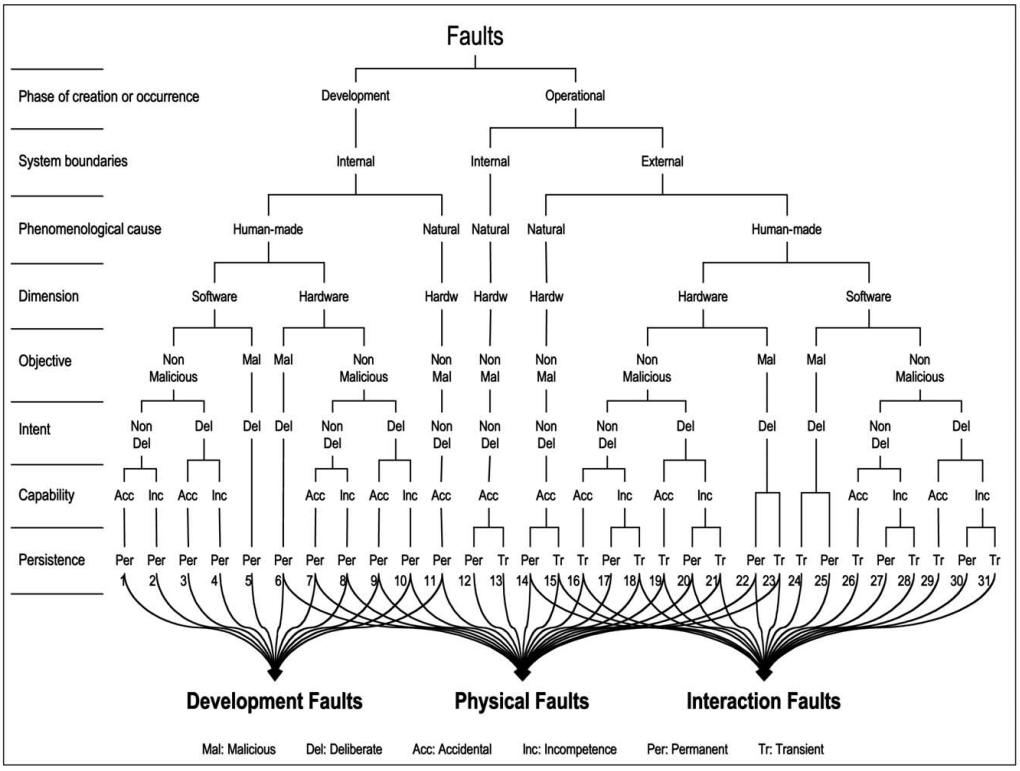

Figure 2.4: The classes of combined faults, from [13].

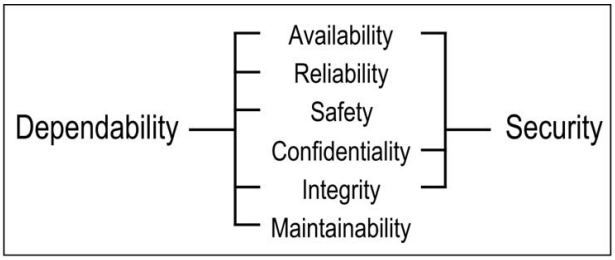

Figure 2.5: Dependability and security attributes, from [13].

The means to attain dependability and security are: fault prevention, i.e. a way to avoid the beginning or happening of faults; fault tolerance, i.e. a way to avoid, in presence of faults, the service's failures; fault removal, i.e. a way to reduce the severity and number of faults; and fault forecasting, i.e. a way to approximate the current number, the future occurrence, and the likely consequences of faults.

Before passing on to the next subsection we will present two more definitions of dependability as they appear in the ISO standards. The first one appears in [17]: the collective term used to describe the availability performance

and its influence factors: reliability performance, maintainability performance and maintenance support performance. The second definition is from [18]: the extent to which the system can be relied upon to perform exclusively and correctly the system task or tasks under defined operational and environmental conditions over a defined period of time, or at a given instance of time. The ISO definition, i.e. the first one, is focused mainly on availability [13]. Due to the unavoidable presence of faults, no system is totally available, safe, secure, or reliable [13].

2.1.3 Means to Achieve Dependability and Security

From the means to achieve dependability and security listed in the previous subsection, in this section we will focus mainly only on fault tolerance and fault removal, the other two methods will be given only a short description.

Fault prevention, as a way to avoid the beginning or happening of faults, is a part of general engineering [13], so it is mainly utilized by the manufactures in order to increase yield and causes of faults. The faults occurring in a system can be recorded by that system and used by the producer to eliminate the fault causes via process modification [19] [20].

Fault tolerance, which purpose is to avoid failures of the system, is implemented via error detection or correction and through system recovery [13] [21]. The techniques involved in fault tolerance are presented in Figure 2.6.

The focus of this thesis will be on isolation of the faults and reconfiguration of the system afterward. Also for this we will need an error detection mechanism and to be more specific, a mechanism for concurrent fault detection, capable of detecting errors and even correcting some of them as they appear. We will also provide an option for diagnosis to be sent back to the manufacturer for future improvements to their products.

Many approaches and schemes have been proposed over the decades for fault tolerance and for the many parts of fault tolerance. There exist a large number of synonymous for fault tolerance: self-repairing and self-healing are just two of them. Also in [22] the term recovery-orienting computing has been presented, this term defines a fault tolerant method for the goal of overall system dependability.

The fault removal technique aims at reducing the number of faults and their severity. Hardware testing is mainly aimed at removing production faults [13]. An important part of fault removal is the fault removal during use. The fault removal during use aims at removing the faults without stopping the system for maintenance. Also this technique increases a system dependability and functioning time. This technique, along with fault tolerance is very useful when a proper maintenance of a system cannot be done, for example a deep space probe cannot be returned back to earth each time an error occurs, and so that system needs to have very efficient fault removal and fault prevention techniques in order to be able to function in an inaccessible, for maintenance, environment.

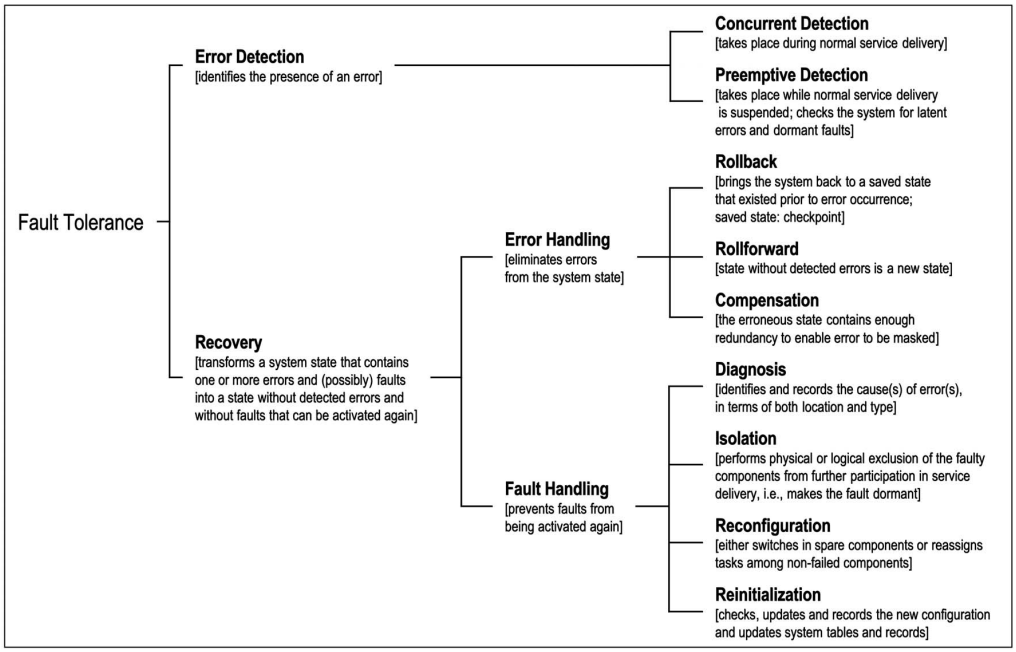

Figure 2.6: Fault tolerance techniques, from [13].

As a conclusion to this section Figure 2.7 shows a refined dependability and security tree, from the definitions and techniques presented in this section [13].

2.2 Cache memories

Since our thesis describes a self-repair method for set associative cache memories, in this section we will provide a brief introduction that will contain the basics on cache memories.

First of all we will start by presenting the memory hierarchy that is used in modern computers, Figure 2.8. In this hierarchy from top to bottom the storage devices get slower in speed, larger in capacity and cheaper in cost per byte. When computer system first started to develop only three levels of memory existed: CPU registers, DRAM or main memory, and the local hard disk [15]. Since the 1980's when the speed of the CPU registers and the speed of the main memory were almost equal, the gap between these two elements of a computer system has increased constantly, see Figure 2.9.

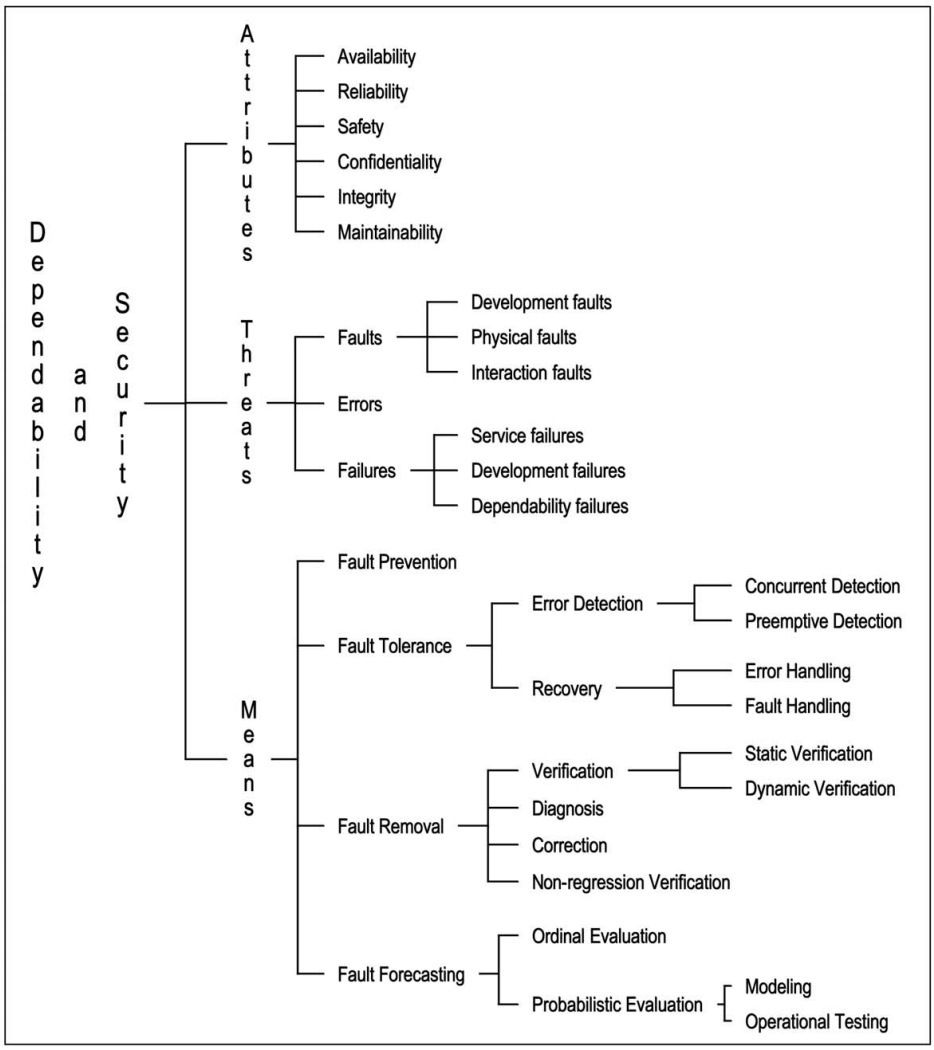

Figure 2.7: A refined dependability and security tree, from [13].

Because of this gap in performance and speed between the main memory and CPU registers, in order to increase the performance of the whole computer system, producers had to introduce a new level in the memory hierarchy, an SRAM memory type, called cache level 1. This level 1 cache was able to increase performance but not for too long, because the gap, in speed, between this level and the main memory also started to increase. A new cache level was needed, the level 2 cache. In the last few years producers needed again to introduce the so called level 3 cache memory, and probably in another three or four years we will see the level 4 and so on.

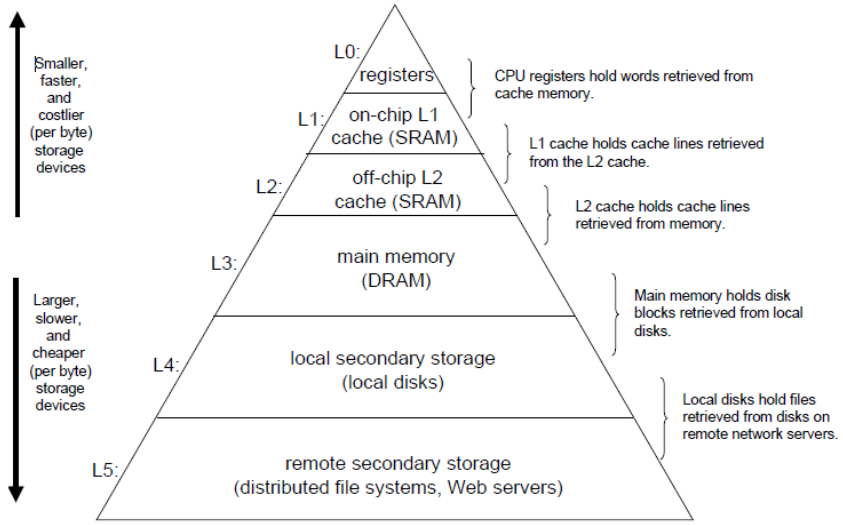

Figure 2.8: The memory hierarchy, from [15]

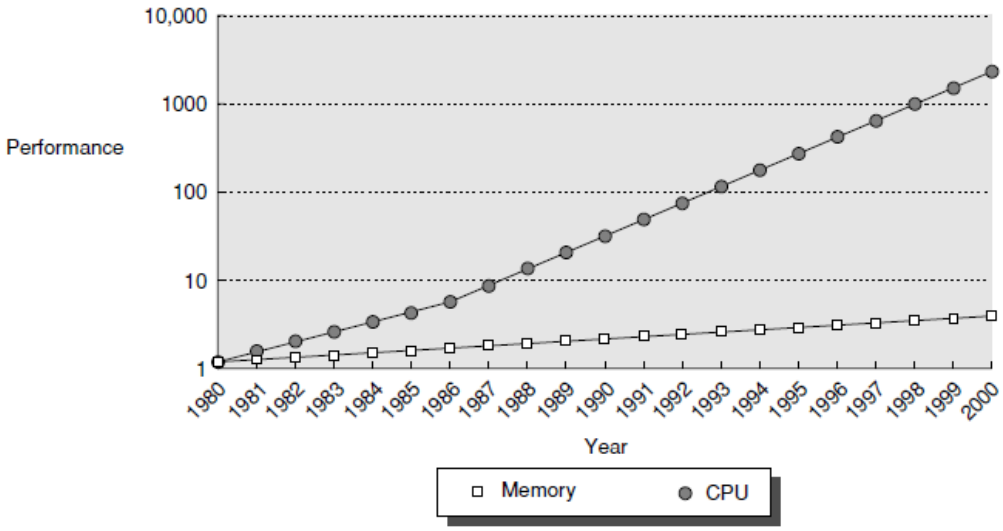

Figure 2.9: The gap in performance between memory and CPU, from [14]

So in order to conclude, a definition for cache memory: is a SRAM type memory placed between CPU registers and main memory (DRAM), it is superior in speed, compared to the main memory, but has a lower capacity. The cache memory contains copies of the locations in the main memory in order for the system to gain in speed and performance. So every byte that is processed by the CPU is passed

through the cache system, for this reason the dependability of the cache system becomes crucial.

2.2.1 Cache memory organization

Usually the level 1 cache is located on the same chip as the CPU, and can be accessed in one or two clock cycles. The cache level 2 is usually placed outside the CPU chip, and so it has greater access times, to the order of 10 clock cycles [15]. Figure 2.10 shows a typical structure for a computing system with a two level cache system.

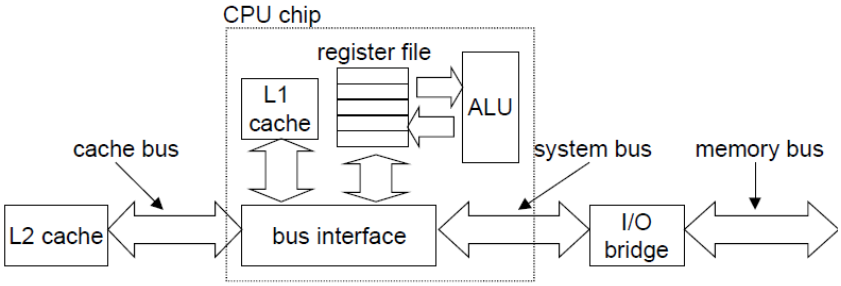

Figure 2.10: Typical structure for two level cache, from [15].

Now we will take a closer look at what is inside a cache memory. Before we start we must state the number of bits *m* that uniquely identifies every line of memory in that computer system. This *m* bits permit access to $M=2^m$ address lines or memory locations in the system. A cache memory for this system will have $S=2^s$ cache sets, within each of these sets there will be a number of *E* cache lines, each line will have a data block of $B=2^b$ bytes, $t=m-(b+s)$ tag bits, that are used to uniquely identify the block stored in the cache line, and one valid bit that is used to indicate if the cache line either has or hasn't significant information [15]. An example of such a cache memory is illustrated in Figure 2.11.

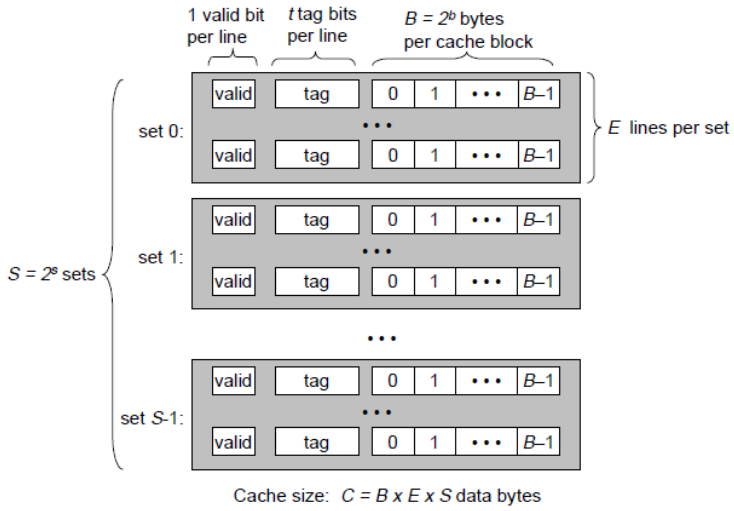

Figure 2.11: General organization of a cache memory, from [15].

Usually a cache memory's organization and size can be characterized by these four parameters: *S, E, B,* and *M*. Figure 2.12 illustrates the organization of the address of such a cache memory with the parameters discussed above.

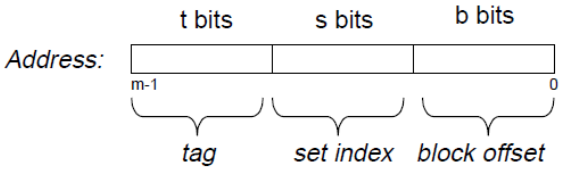

Figure 2.12: Address organization of a cache memory, from [15].

A summary of the most usual cache memory parameters is presented in Figure 2.13.

Fundamental parameters	
Parameter	Description
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits

Derived quantities	
Parameter	Description
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes) not including overhead such as the valid and tag bits

Figure 2.13: Cache parameters, from [15].

This concludes the present subsection of our thesis; we will not go any further in detail, in presenting the organization of cache memories, for this we will refer the reader to [15] [14].

2.2.2 Set associative caches

The most usual method to group cache memories is after *E*, the number of lines in each set of the cache memory. After this classification the cache memories are split into three major groups: direct mapped cache memories, where *E*=1; set associative cache memories, where *E*>1, and also *S*>1; and in the last group are fully associative cache memories where *S*=1, i.e. there is only one set and a location from the main memory can be mapped in any line without restriction. An example of the differences in mapping between the three groups of cache memories is depicted in Figure 2.14.

We will start by providing the reader with a short description of direct mapped cache memories; our focus will mainly be on set associative cache memories, them being the object of this thesis. For a more detailed approach to direct mapped and fully associative cache memories the reader is referred to [15] [14].

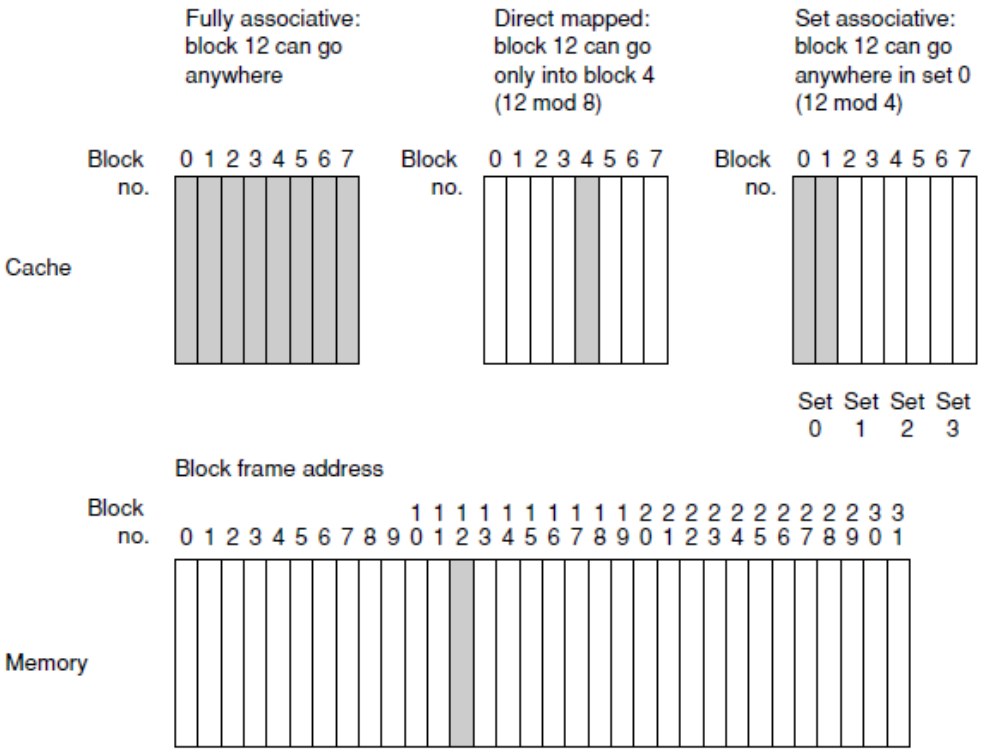

Figure 2.14: Mapping differences between groups of caches, from [14].

As stated before a direct mapped cache memory is a cache memory that only has one line per set, i.e. E=1. Such a memory is depicted in Figure 2.15. This type of cache memory is the simplest and easiest to understand [15].

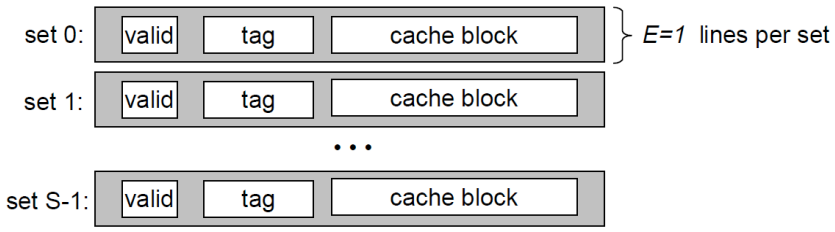

Figure 2.15: Direct mapped cache, from [15].

Set associative cache memories are those caches for which E>1, and also S>1, i.e. there is more than one line in each set of the memory. This provides an advantage from the direct mapped caches because a location from the main memory can be mapped in more than one place in the cache. This is being

particularly useful when working with array that have two or more dimensions. In Figure 2.16 is presented a 2-way set associative cache memory.

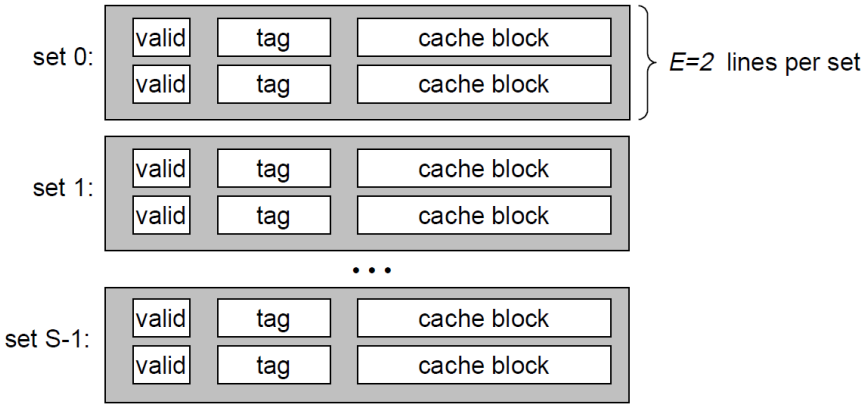

Figure 2.16: Organization of a 2-way set associative cache memory, from [15].

The access in a set associative cache memory is similar as in any other type of memory. First the set is selected as shown in Figure 2.17. After the set is selected the second task is to see if any line in that set matches the tag of the address requested by the CPU. If we have a line matching, which is also known as a cache hit, we proceed to the extraction of the word from the cache block. This is shown as an example in Figure 2.18.

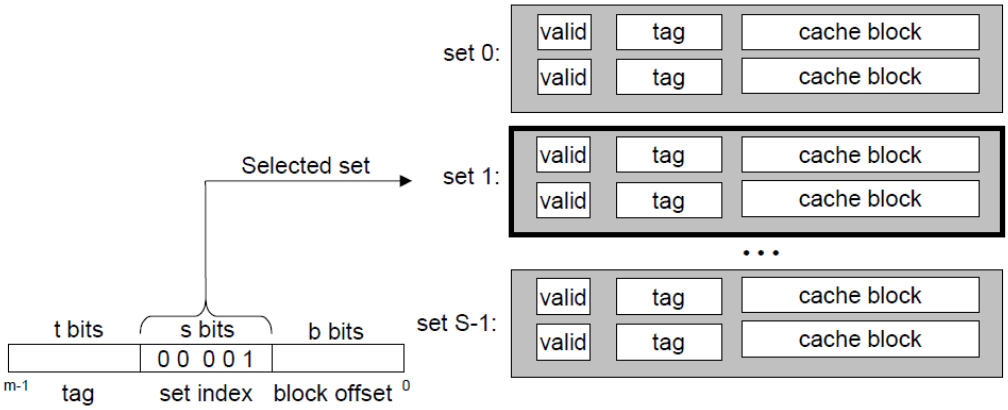

Figure 2.17: Set selection in a set associative cache memory, from [15].

We will conclude this subsection with an example of a set associative cache memory from the microprocessor Alpha 21264. This is a 2-way set associative cache that contain 64KB of data, with the block size of 64 bytes. The organization of this memory is presented in Figure 2.19.

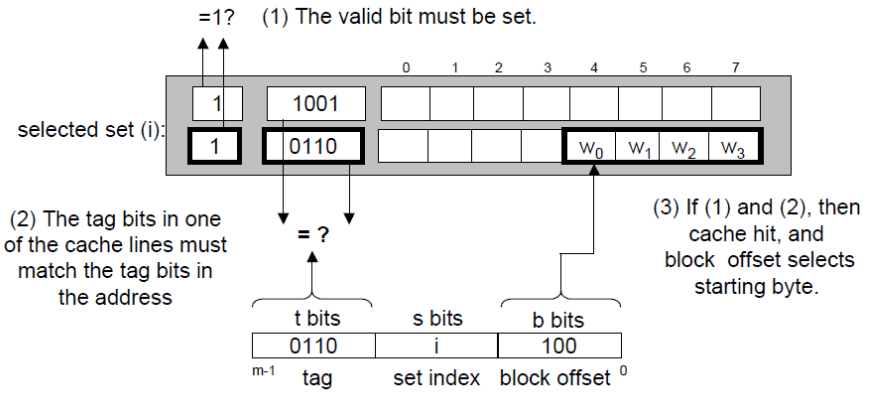

Figure 2.18: Line matching and set selection in a set associative cache memory, from [15].

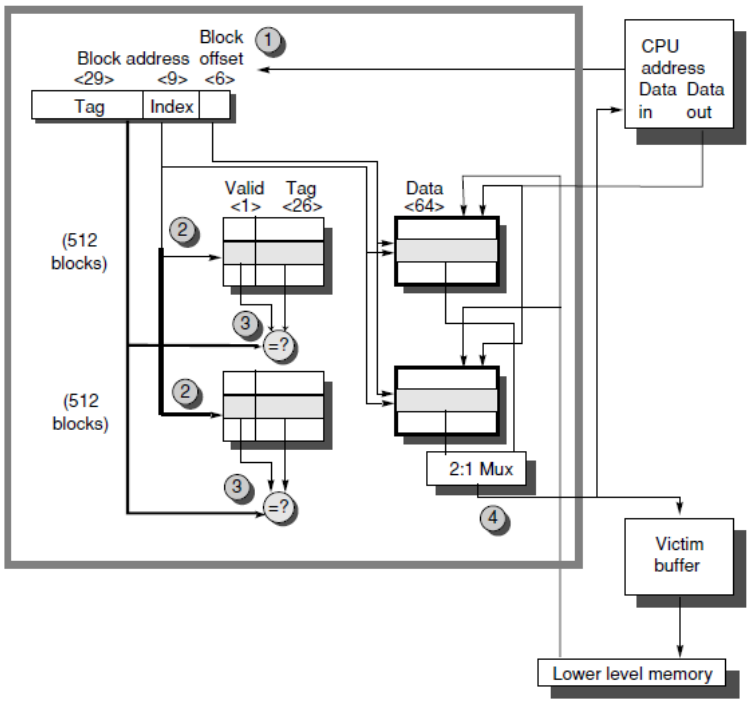

Figure 2.19: The organization of the cache in Alpha 21264 microprocessor, from [14].

3 Memory testing, Built-In Self-Tests, and Graceful Degradation

3.1 Memory testing

In this section we will provide our reader with the basics on functional models of memory chips, the errors that can appear in accordance with these models, and also some test methods that are used for memory testing.

3.1.1 Functional RAM chip models and faults

We will first present the functional model for a RAM memory with all of the main components, Figure 3.1 illustrates this.

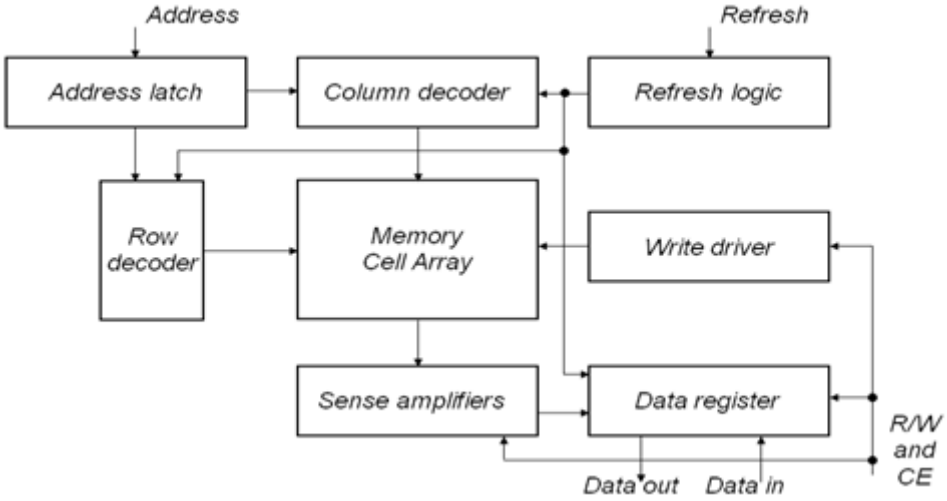

Figure 3.1: DRAM memory model, from [16].

Since we will be working with cache memories that are SRAM memory types, from the DRAM memory model we will exclude the refresh logic, since the SRAM is non-volatile. Figure 3.2 shows a memory model for a SRAM type of memory.

Some of the functional faults that can appear in a RAM memory are illustrated in Table 3.1, the list is not complete. Note that we refer to a cell as an entity that stores data, and to a line as an entity that is used to transmit data from one entity to another.

As can be seen from Table 3.1, the list not being complete, the number of functional faults is very large. Given the large number of functional faults and the fact in order to test for each individual group of faults can be very expensive and very time consuming we can start grouping some of the elements of the memory as shown in Figure 3.3. As can be seen in Figure 3.3 the address latch, column decoder, row decoder and the connections between them are grouped in the address decoder, the memory cell array remains unchanged and the read/write logic has the following elements: write driver, sense amplifiers, data register and the connections between them.

The reduced functional model from Figure 3.3 generated the following types of errors: stuck-at faults, transition faults, coupling faults and neighborhood pattern sensitive faults. Table 3.2 presents the reduced functional faults. As can be seen in this table the number of potential types of faults is reduced considerably, leaving only four categories of faults, that include all the other types of faults. This is a clear advantage, because with a smaller number of functional faults it is easier, cheaper and faster to test the memory chips.

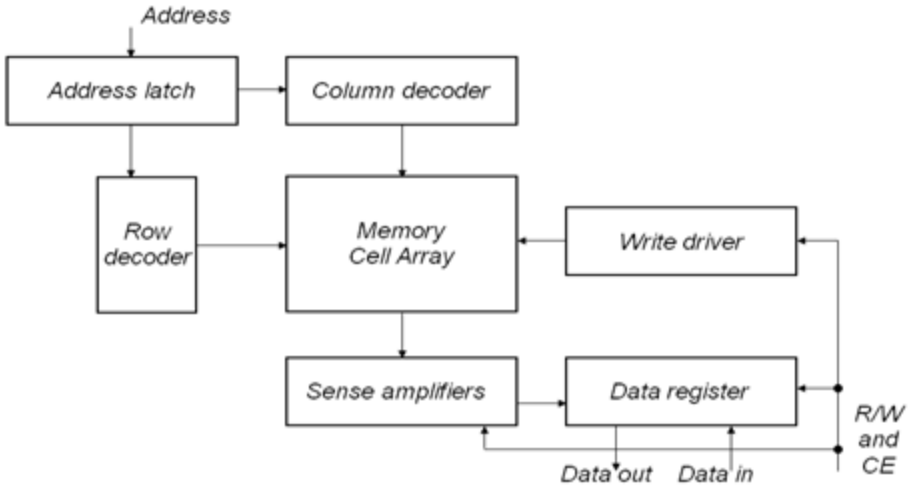

Figure 3.2: SRAM memory model, from [16].

Table 3.1: RAM functional faults, from [16].

	Functional Fault
a	Cell stuck
b	Driver stuck
c	Read/write line stuck
d	Chip-select line stuck
e	Data line stuck
f	Open circuit in data line
g	Short circuit between data lines
h	Crosstalk between data lines
i	Address line stuck

j	Open in address line
k	Shorts between address lines
l	Open decoder
m	Wrong address access
n	Multiple simultaneous address access
o	Cell can be set to 0 but not to 1 (or vice versa)
p	Pattern sensitive cell interaction

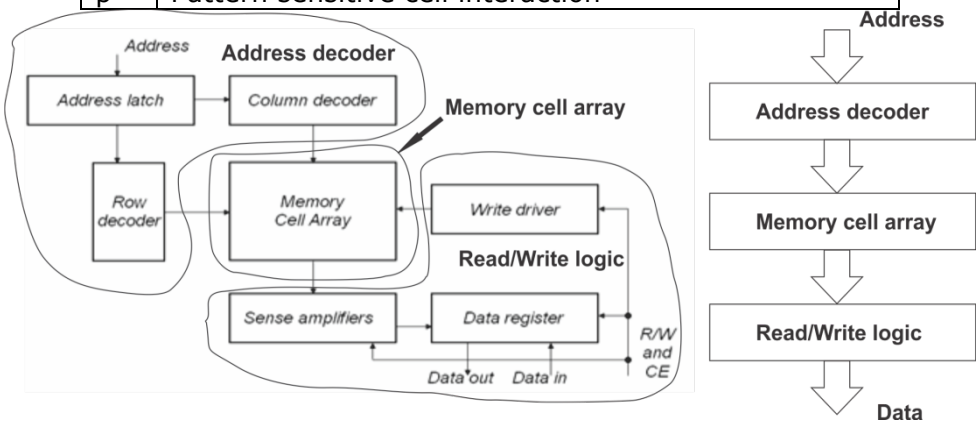

Figure 3.3: Reduced functional model, from [16].

Table 3.2: Reduced functional faults, from [16].

1. SAF	Stuck-At Fault
2. TF	Transition Fault
3a. CF	Coupling Fault
3b. NPSF	Neighborhood Pattern Sensitive Faults

We can furthermore group the type of faults from Table 3.2 into three categories: faults involving one cell, faults involving two cells, and faults involving *n* cells. The classification is as follows [16]:

- Faults involving one cell:
  - Stuck-At Faults (SAF)
  - Transition Faults (TF)
- Faults involving two cells:
  - Coupling Faults (CF)
- Faults involving *n* cells:
  - The *n* cells are allowed to be located anywhere in the memory. These are the *n*-coupling, bridging and the state coupling faults
  - The *n* cells are clustered together in a physical neighborhood. These are the Neighborhood Pattern Sensitive Faults (NPSF)

Table 3.3 describes the standard notations used when describing faults and types of faults as presented in [16].

This concludes this subsection of our thesis. In the following subsection we will provide the reader with a short description of each category of the reduced functional faults.

Table 3.3: Standard fault notations, from [16].

0	denotes that the cell is in a logical state 0
1	denotes that the cell is in a logical state 1
x	denotes that the cell is in a logical state x, where $x \in \{0,1\}$
↑	denotes a write 0 operation to a cell containing 1
↓	denotes a write 0 operation to a cell containing 1
↕	denotes a write $\bar{x}$ operation to a cell containing an *x*
→	denotes a write 0 operation to a cell containing an 0
→	denotes a write 1 operation to a cell containing an 1
⇒	denotes a write *x* operation to a cell containing an *x*
∀	denotes any operation; $\forall \in \{\uparrow, \downarrow, \updownarrow, \rightarrow, \Rightarrow\}$
<...>	denotes a particular fault; "..." describes the fault
<I/F>	denotes a fault in a single cell I describes the condition for sensitizing the fault: $I \in \{\uparrow, \downarrow, \updownarrow, \rightarrow, \Rightarrow\}$ F describes the value of the faulty cell: $F \in \{0,1,\uparrow,\downarrow,\updownarrow\}$
<I1, I2, …, In-1; In/F>	denotes a fault involving *n* cells I1,…, In-1 describes condition on the *n*-1 cells to sensitize the fault in cell *n* In describes the condition for the fault to be sensitized in cell *n*. It may be empty (In=[]) in which case In/F=[]/F can be written as F

3.1.2 Reduced functional faults

Stuck-At Faults

The most common definition of a stuck-at fault is: the logic value of a stuck-at line or cell has always the same logic value, either 0 (SA0 faults) or 1 (SA1 faults) [16]. The notation for a SA0 fault is $< \forall/0 >$; and for a SA1 fault $< \forall/1 >$. A test that can detect and locate all stuck-at faults in a memory chip has to read a 0 and a 1 from each memory cell [16].

Figure 3.4a shows a state diagram for a healthy memory cell. In Figure 3.4b and Figure 3.4c are shown the state diagram for SA0 and SA1, respectively. A cell has the logic value 0 in state 0 (S_0), and the value 1 in the state S_1.

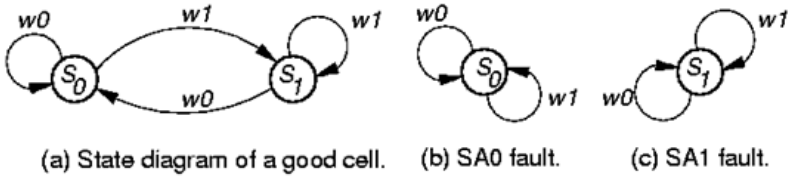

Figure 3.4: State diagram for SAF, from [16].

Transition Faults

The definition of transition faults is: A cell or line which fails to undergo a $0 \rightarrow 1$ transition when it is written is said to contain an up transition fault; similarly, a down transition fault is the impossibility of making a $1 \rightarrow 0$ transition [16]. The notation for the up TF, as shown in [16] is $<\uparrow/0>$, and for the down TF $<\downarrow/1>$.

The transition faults are a special case of stuck-at faults, in order for a better understanding of this we will provide the reader with a short example [16].

Example

Figure 3.5 shows a Set/Reset (S/R) flip-flop with the Reset stuck-at 0. In this situation the fault may be classified as a $<\uparrow/1>$ fault because the S/R flip-flop will fail to make a $1\rightarrow 0$ transition.

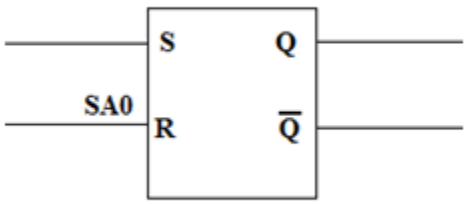

Figure 3.5: A flip-flop as a model for a transition fault, adapted from [16].

Transition faults cannot be treated as SA*x* faults because other faults, such as coupling faults, may bring the cell back into state $\overline{x}$. So in order to test for transition faults we have to use a slightly more complex algorithm. A test that has to detect and locate al TFs, should satisfy the following requirements, according with [16]: Each cell must undergo a $\uparrow$ transition (state of the cell goes form 0 to 1), and a $\downarrow$ transition (state of the cell goes from 1 to 0), and be read after each transition before undergoing any further transitions.

The state diagram of a memory with a $<\uparrow/0>$ transition fault is illustrated in Figure 3.6. The notations are the same as for the stuck-at faults.

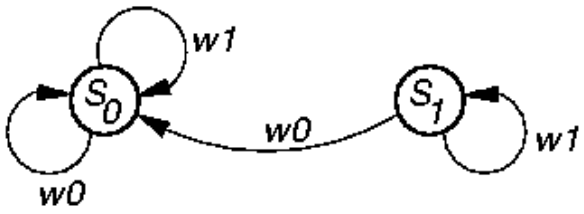

Figure 3.6: State diagram for TF, from [16].

Coupling Faults

Coupling faults are grouped according to these assumptions:

1. A read operation will not cause an error.
2. A non-transition write operation will not cause a fault.
3. A transition write operation may cause a fault.

The coupling faults that involve two cells, and that is used in [16] [23] [24] [25], has a definition as follows: a write operation which generates a ↑ or a ↓ transition in one cell changes the contents of a second cell.

The coupling fault that involves two cells is a special case of the more general case *k*-coupling fault that involves *k* cells and is defined as follows: is the same as the two coupling fault, but in addition the transition is only performed when the other *k*-2 cells are in a certain state [24]. If there is no restriction on the placement of the *k* cells the *k*-coupling fault is very complicated to test for [26].

The two coupling faults can be grouped in two types: inversion coupling faults and idempotent coupling faults, which will be briefly discussed. Special cases of coupling faults are state coupling faults and bridging faults, for detailed perspective these types of coupling faults we refer our reader to [16].

The inversion coupling faults (CFin) has the following definition: a ↓ (or ↑) transition in one cell inverts the contents of a second cell [16].

A test that detects all CFins must satisfy the following: "for all cells which are coupled cells, each cell should be read after a series of possible CFins may have occurred (by writing into the coupling cells), with the condition that the number of transitions in the coupled cell is odd (i.e. the CFins do not mask each other)" [16].

The idempotent coupling faults (CFid) has the following definition: A ↓ (or ↑) transition in one cell forces the contents of a second cell to a certain value, 0 or 1 [16].

A test that detects all CFids must satisfy the following: "for all cells which are coupled cells, each cell should be read after a series of possible CFids may have occurred (by writing into the coupling cells), in such, a way that the sensitized CFids do not mask each other" [16].

As a conclusion to the state coupling faults Figure 3.7 illustrates the state diagram of two good cells (a), the state diagram of a <↑;↕> CFin (b); and the state diagram of a <↑;1 > CFid (c).

Neighborhood Pattern Sensitive Faults

The neighborhood pattern sensitive fault is a special case of the *k*-coupling fault, in the sense that the *k*-1 cells, beside the base cell are in the immediate vicinity of the base cell. In Figure 3.8 the NPSF terminology, as presented and used in [16], is depicted. There are three cases of NPSF: ANPSF (Active Neighborhood Pattern Sensitive Faults), PNPSF (Passive Neighborhood Pattern Sensitive Faults), and SNPSF (Static Neighborhood Pattern Sensitive Faults). In the following we will present a short description of each of these types of NPSF along with a testing requirement for each one, again for a more ample description we refer our readers to [16].

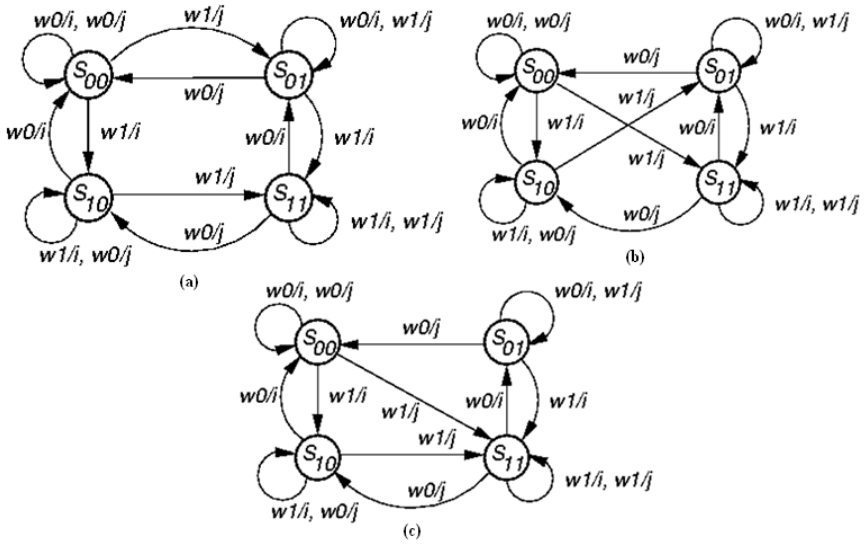

Figure 3.7: State diagrams involving two cells, from [16].

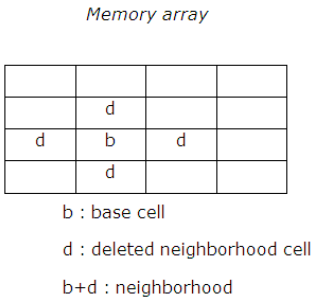

Figure 3.8: NPSF terminology, from [16].

In ANPSF due to a change in the deleted neighborhood pattern the base cell changes its contents. The change in the deleted neighborhood is a transition while

the rest of the deleted neighborhood cells and the base cells have a certain pattern. In order to detect and locate ANPSFs a test must satisfy the following requirement: "each base cell must be read in state 0 and in state 1, for all possible changes in the deleted neighborhood pattern" [16].

In PNPSF due to a certain neighborhood pattern the content of the base cell cannot be changed. In order to detect and locate PNPSFs a test must satisfy the following requirement: "each base cell must be written and read in state 0 and in state 1, for all permutations of the deleted neighborhood pattern" [16].

In SNPSF a state of the deleted neighborhood pattern forces the content of the base cell to a certain value. In order to detect and locate SNPSFs a test must satisfy the following requirement: "each base cell must be read in state 0 and in state 1, for all permutations of the deleted neighborhood pattern" [16].

With this we conclude the present subsection dedicated to describing the most important possible types of faults. The next and last subsection of this chapter is dedicated to describe some traditional tests and some march tests along with their test times.

3.1.3 Traditional and March Tests

In this subsection of our thesis we provide our reader with a brief description of the traditional test: zero-one, checkerboard, GALPAT and Walking 1/0, sliding diagonal, and butterfly. Also we will provide a short description of the march test MATS and MATS+, concluding this subsection with a comparison between the traditional tests and a couple of march tests. Table 3.4 summarizes the notation used throughout this subsection.

Table 3.4: Notation and abbreviations used in memory testing

B	The number of bits (cells) in a memory word, thus the width of the memory
N	The number of address bits; the number of addresses will thus be 2^N
n	The total number of bits (cells) in the memory, which equals $B \cdot 2^N$
k	The size of the neighborhood
A	An address
C	A cell
M	A set of cells, words or addresses
r	A read (operation)
w	A write (operation)

Zero-One

This is the simplest test for a memory chip. It consists of writing 1s and 0s in the memory cell array. The algorithm consists of four steps, see Figure 3.9. This algorithm is also known as MSCAN (Memory Scan) [16] [27].

Step 1: **write** 0 in all cells

Step 2: **read** all cells

Step 3: **write** 1 in all cells

Step 4: **read** all cells

Figure 3.9: Zero-One test algorithm, from [16].

This test detects all SAF, and also it detects some TF, and some CF. The test has a length of $4 \cdot 2^N$, and it is of order *O(n)* [16].

Checkerboard

For this test we first need to split the memory in two groups: group 1 and group 2, in a checkerboard pattern, as shown in Figure 3.10. Figure 3.11 presents the algorithm of the checkerboard test. The fault coverage is similar with the zero-one test, and also the number of operations is the same as the zero-one test, giving the checkerboard test an order of *O(n)* [16].

1	2	1	2
2	1	2	1
1	2	1	2
2	1	2	1

Figure 3.10: Cell numbering for checkerboard algorithm, from [16].

Step 1: **write** 1 in all cells-1 and 0 in all cells-2

Step 2: **read** all cells (words)

Step 3: **write** 0 in all cells-1 and 1 in all cells-2

Step 4: **read** all cells (words)

Figure 3.11: Checkerboard algorithm, from [16].

GALPAT and Walking 1/0

These two tests are similar, that is why we present them together. First the memory is filled with 1s (or 0s), except for one cell, called the base cell that has the opposite value. For both these tests the base cell covers the whole memory. The difference between these two tests appears when the base cell is read: in GALPAT the base cell is read after each cell is read, while in Walking 1/0 the base cell is read only once after all the other cells have been read. This is depicted in Figure 3.12. The fault coverage for both these test, according with [16] is: all SAF, TF, CF are detected and located. Note that the tests are performed twice once with a 0

background and the second time with a 1 background. The order of both these test is $O(n^2)$ [16] [28].

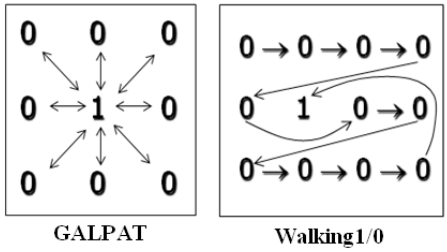

Figure 3.12: Read actions for GALPAT and Walking 1/0, from [16].

Sliding Diagonal

The sliding diagonal has been developed as a shorter alternative to GALPAT, so instead of a single base cell as in GALPAT the sliding diagonal test uses an entire diagonal of base cells, making it faster but less efficient. Figure 3.13 shows the read actions for the sliding diagonal test. As stated before the fault coverage is smaller than the GALPAT: some CF are detected and located, but not all of them; also this test detects and locates all SAF and TF. Due to the fact that sliding diagonal uses an entire diagonal instead of a single base cell the time order of this test is reduced to $O(n^{3/2})$ [16].

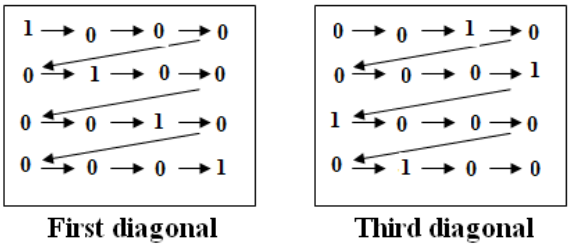

Figure 3.13: Read actions for sliding diagonal, from [16].

Butterfly

The butterfly test has been designed in order to reduce even more the test time of the GALPAT test, but with the purpose to only find SAF [16]. We will not go in detail with this algorithm, providing only a very short description of the reading of the cells. From GALPAT, only the reading of the cells differs, in that only the neighboring cells with the base cell are read. So the algorithm can detect and locate all SAF. The test order of the butterfly is *O(nlogn)* [16].

Before moving on to MATS and MATS+ test we will make a short observation regarding all of the march type tests. These tests are called march test because they "march" throughout the memory. A march element as described in

[16] is "a finite sequence of the operations applied to every cell in the memory before proceeding to the next cell". The order of the addresses can either be increasing (⇑), decreasing (⇓), or unimportant (⇕). An example of a march element can be ⇓(w1,r1), that means that in every cell of the memory starting with the highest address and deceasing is first written a 1 and immediately is read a 1.

MATS

The MATS test or Modified Algorithm Test Sequence is the shortest march test [16], it detects all SAF. This test requires a number of 4*n* operations, having the test time order *O(n)*. The basic scheme of the MATS test is illustrated in Figure 3.14.

$$\{\Updownarrow (w0);\ \Updownarrow (r0,w1);\Updownarrow (r1)\}$$

Figure 3.14: MATS test scheme, from [16].

Looking at the MATS in comparison with Zero-One or Checkerboard, which have the exact same number of operations performed on the memory cell array we can see a net superiority of the MATS test in the fault coverage [16].

MATS+

MATS+ is a special version of the MATS test, used when the technology of the memory chip is unknown [16] [29]. This test uses 5*n* operations, so has an order of *O(n)*. The fault coverage is the same as the MATS test. The scheme of the algorithm is depicted in Figure 3.15.

$$\{\Updownarrow (w0);\ \Uparrow (r0,w1);\ \Downarrow (r1,w0)\}$$

Figure 3.15: MATS+ test scheme, from [16].

We will conclude this section with a summary of the tests described in this section alongside with some other march tests described in [16]. This summary is presented in Table 3.5. As can be observed from Table 3.5 the test times for even a small memory chip can be very high. Also in order to be able to apply these tests there are necessary special equipment outside the memory chip, these test equipment are very expensive because they are usually used only in one generation of chips, needing change after each technological improvement. Also in the last years the size of the memory has increased considerably without a corresponding increase in speed, this making the tests lengthy and sometimes even obsolete. Due to these facts and many other disadvantages the producers have started exploring alternatives to the old testing methods, these alternatives have developed in a general method called Built-In Self-Test that is integrated on the memory chip and permits the test of the chip, only by adding some extra pins, without the special equipment, or with some equipment that permit the production cost to be reduced. The Built-In Self-Test methods along with others of similar type will be presented in the next chapter.

Table 3.5: Comparison of memory test algorithms, from [16].

Algorithm	Fault Coverage					Test Time	
	AF	SAF	TF	CF	Others	Order	1Mb
Zero–One	–	L	–	–		n	0.42s
Checkerboard	–	L	–	–	Refresh	n	0.52s
Walking 1/0	L	L	L	L	Sense amplif. rec.	n^2	2.5day
GALPAT	L	L	L	L	Write recovery	n^3	5.1day
GLAROW	LS	L	L	L	Write recovery	$n\sqrt{n}$	7.2day
GLACOL	LS	L	L	L	Write recovery	$n\sqrt{n}$	7.2day
Sliding Diag.	LS	L	L	–		$n\sqrt{n}$	10s
Butterfly	–	L	–	–		2n	3.6m
MATS	DS	D				n	0.42s
MATS+	D	D	–	–		n	0.52s
Marching1/0	D	D	D	–		n	1.5s
MATS++	D	D	D	–		n	0.63s
March X	D	D	D	D	Unlinked CFins	n	0.63s
March C-	D	D	D	D	Unlinked CFins	n	1.0s
March A	D	D	D	D	Unlinked CFs	n	1.6s
March Y	D	D	D	D	Linked TFs	n	0.85s
March B	D	D	D	D	Linked CFs	n	1.8s

L=Locate LS=Locate Some D=Detect DS=Detect Some

3.2 Built-In Self-Testing and Graceful Degradation

Throughout the remainder of this chapter we will discuss the various methods used for Built-In Self-Test (BIST) for memory testing. Also we will provide a description of a method called graceful degradation, which, as its name suggests, allows the memory to continue functioning even after faults appear.

3.2.1 Memory Built-In Self-Test

We will start this section with a basic description of what BIST means and what it implies, and we will continue with a more detailed presentation of BIST methods used for memory testing.

Introduction to BIST

In the digital world everything eventually breaks down and stops functioning correctly. The most important thing to know is when to trust the result that a digital device provides to be correct and when not. The methods described in the previous

section, though useful, are not practical because they need special equipment in order to be able to test a device. In order to eliminate this inconvenient the industry has provided a solution called Built-In Self-Test, which adds extra logic needed for the test sequence on the chip of the circuit under test (CUT). The first digital systems to have a BIST were two Hewlett-Packard digital voltmeters, as described in [30]. The development cost and time increased by 1%, also there was a 1% increase in part costs, but the total costs dropped by 5% because the modularity of the system was no longer needed. Frohwerk describes in [31] a method for determining the correctness of a circuit by analyzing a *signature*. A *signature* is a statistical property of a circuit. In order to build BISTs for integrated circuits he applied the work of Peterson and Weldon [32] and Golomb [33] on error correcting codes and shift registers [34].

A digital system is diagnosed and tested during its lifetime on countless occasions. The tests and diagnosis must be quick and they need to have a very high fault coverage [34]. A way to ensure these restrictions is to specify a test as one of the system functions, so it becomes a self-test [34]. Many of the digital systems designed at AT&T around 1987 had self-tests, usually implemented in the software [34] [35]. Although this approach provided flexibility and its fault coverage and diagnosis weren't as high as expected [34]. This led to the building of the self-test function into the hardware [36] [37]. The earlier in the design stage the testing is considered the more efficient it is and the more the cost is reduced, this is because of the reduced number of prototypes and re-fabrications that are needed.

In the last few years due to the large integration the need for testing is greater than ever, that is why the great majority of the manufacturers, if not all of them, use BIST methods on a very large scale. The BIST solutions for testing can be applied to any digital system, but due to the fact that in our thesis we only discuss memory testing we will stop with this general introduction of BIST here, refereeing the reader for a more detailed description to [16] [34] [38].

Memory Built-In Self-Test

Random Access Memories (RAM) memories are perhaps the hardest elements in digital systems to test; this is because memory testing requires delivery of a huge amount of pattern stimuli to the memory. Also it requires the readout of an enormous amount of information [34].With the memory Design for Testability (DFT) the most time consuming part is implemented on-chip, and it reduces the order of test time by a magnitude order [16]. The area overhead for memory DFT for a 4Mb DRAM is 1% [39]. The area overhead for memory BIST can be expected at around 2% [34].

The most important difference between memory BIST and memory DFT is that the memory BIST is completely self-contained, which means that all the functions required for the BIST are contained in the chip such that the test can be performed autonomously [16]. For DFT parts of a test are implemented on chip, these are the ones that provide the largest reduction in test time. So this way the

inner loops of a test algorithm can be executed by the DFT on the chip, while the other parts of the algorithm are executed, by externally providing certain control and test data and/or observing certain response data [16]. This is why the test times are in favor of the BIST when compared to DFT [39].

The most important advantages of BIST are: the test time, which is minimized (i.e. it is from 2 to 3 orders of magnitude faster than the conventional tests [16]); and the test is completely contained on the memory chip. The disadvantages of the BIST are: the area overhead is larger than DFT, usually with a factor of 2 [16]; it is only capable of implementing the tests for which it was designed.

The types of memory BIST are:

- Concurrent BIST
- Non-Concurrent BIST
- Transparent Testing

The concurrent BIST is a memory test mechanism where the memory can be tested concurrently with the normal system operation. This means that faults occurring during normal use of the memory can be detected, and depending on the complexity of the test even be corrected. For this type of BIST usually a form of information redundancy is used in the form of a parity bit or an error correcting code (ECC), which also increase the area overhead due to the extra information that has to be stored. The advantages for the concurrent BIST are that all faults, within the restrictions of the method used, are detected and/or corrected. This means that all permanent and transient faults are detected and/or corrected when they appear.

The disadvantages for the concurrent BIST are: the large area overhead needed, the performance penalty because of the constant need of checking the ECC, also the number and type of faults that can be corrected is limited, and so even if we have a complex concurrent BIST we cannot guarantee that the memory will be completely fault free. Note that the 100% certainty that the memory is fault free cannot be achieved by any kind of test.

The non-concurrent BIST is a memory test mechanism that requires interruption of the normal system function in order to perform the testing. The original memory contents are lost. The advantages of this kind of BIST are: maximum freedom in the data pattern used, more complex fault models can be detected. The disadvantages of the non-concurrent BIST are: the faults not covered by the BIST algorithm are not detected; the transient faults that occur between BIST periods are not detected, so only the permanent faults can be detected by this kind of BIST.

Transparent testing is a memory test mechanism that requires interruption of the normal system function for testing. The original memory contents are preserved in the memory after the testing is finished. Due to the fact that this is a particular method of non-concurrent BIST the advantages and disadvantages of the non-concurrent BIST also apply.

3.2.2 Graceful Degradation

The graceful degradation technique deals with hard/permanent faults, and it is an alternative to reconfiguration. In reconfiguration the defective component is replaced by a spare one, if this one is available. In graceful degradation the defective component is just switched off and never used again, thus maintaining the reliability at the cost of performance [38]. The provision of spare units that remain unused in the reconfiguration technique is called backup sparing. For graceful degradation the backup units either do not exist or they are used to provide an increase in performance [38].

The graceful degradation is a special part of fault tolerance techniques, because all other techniques have one thing in common: they have redundant units that are used for replacement, error detection or error correction. The graceful degradation technique does not make use of redundant units, in the sense that either these units are used to increase performance, or they do not exist at all [38].

There are two comparable but different aspects to graceful degradation. For the first one the design of the system is such that it will provide the possibility of continued operation at the cost of a degraded performance. This will result in a slower working system, but even a slower system is preferred to a non-working system. The second case of graceful degradation is when extra resources are added in the design of the system, such that even in the circumstance of errors appearing the system will be able to provide a minimum performance level with a high probability. The system will start by providing a higher performance if no errors are detected, and this performance will start to degrade, but still be above a certain minimum threshold.

The choice between the two aspects of graceful degradation depends on the application that needs to run on a certain system. For example the first aspect of graceful degradation may be used in a non-performance critical system; an example of such a system is a personal computer. The second aspect of graceful degradation is used for performance critical systems; an example of such a system is a space satellite, or an aircraft controller [38].

The way the graceful degradation technique is used in industry is through software. This is because of the reduced cost of this solution and the fact that it is easy to modify. As a shortcoming for this approach is the extra performance cost that is introduced in this manner. For example in the VAX-11/780, which has a 2-way set associative memory, if errors are detected in one way then that way is disabled leaving the cache memory as a direct map cache. This is not very efficient, since in the eliminated way there are still locations that function correctly that could be used. Also the Univac 1100/60 has the ability to not use portions of the cache [38]. These are examples of machines that make use of the first aspect of graceful degradation.

For the second aspect of graceful degradation an example can be found in the multiprocessor system Pluribus [40], that was designed for the ARPANET [38]. This multiprocessor system has an extra processor that provide an increase in throughput and also allows the system to maintain a minimum performance

requirement if one of the processors fails. Some other examples can be found in the SIFT, Stratus and Tandem computers [38].

There are cases where a system cannot be classified in one or the other instances of graceful degradation. Examples of such are the multiprocessor systems: Cm* and C.mmp [41] [42]. This is because both of these systems make use of both aspects of the graceful degradation technique [38].

The first thing that designers have to keep in mind when designing systems that use the graceful degradation technique is that there is not a standard way of evaluating the system. This is because the evaluation will depend on the application that is served by that system. The metric that evaluates the system will consist of both performance and reliability aspects and the weight of each of these aspects will vary from application to application [38]. Also one has to keep in mind that the performance will vary in time, depending on the number of errors encountered and the location of those errors.

3.2.3 Conclusions

Throughout this thesis we will make use of the first aspect of the graceful degradation technique. We will use this in providing an efficient mechanism for set associative cache memories that will allow taking out of use of each location at a time, instead of a whole set. By doing this we will ensure that the performance of the system is maintained as much as possible, while providing an increase in reliability and speed of the whole system.

This technique that we have developed is called SAM (Self Adaptive cache Memories), and will also make use of a BIST that resides inside the cache memory capable of detecting errors as they appear. By doing this we will provide a reliability increase in the cache memory, while maintaining a stable performance. The method that we have developed can also be changed and applied to any type of memory, such that the reliability gain and also the performance degradation rate are maintained.

4 State of the art in reliability techniques for cache memories

In this chapter we will present an overview of the state of the art work on cache memory reliability and its importance and relevance to the computing world.

4.1 Adaptive Cache Design to Enable Reliable Low-Voltage Operation

In this paper the authors propose an adaptive cache design, which addresses the operating system. The operating system is thus allowed to optimize for energy efficiency or performance while maintaining the reliability of the cache memory [43].

The basic goal is to tolerate both persistent (hard) and non-persistent (soft) failures at low voltages, by enabling a special cache design. This design exploits both power-reliability tradeoff and power-performance tradeoff. The solution proposed by the authors makes use of error correcting codes (ECC) in order to tolerate both hard and soft faults at low voltages; this solution is called multi-bit segmented ECC (MS-ECC). The basic idea for doing this is to use some ways in each cache set in order to store error-correcting codes for the other ways in that set, during the low-voltage operation time. In this way the remaining ways have an increase in reliability against high failure rates. Depending on the Vccmin and the desired reliability the number of ways used for storing ECC can be adaptively selected by the operating system, thus making this method very flexible. This number has to be carefully selected because the cache will decrease its capacity with every way that is taken out of normal use, and chosen for storing ECC [43].

In order to reduce the complexity of error correcting the authors make use of the Orthogonal Latin Square Codes (OLSC) that were proposed in [44]. By using more check bits the OLSC can encode and decode faster than traditional ECC. Also OLSC can be used to adaptively correct a varying number of errors, because it uses modular error correction hardware. To further simplify the ECC implementation, the authors use the MS-ECC in order to divide a cache line into segments and use these segments for finer granularity. By making use of this finer granularity the errors that appear can be corrected with lower latency and complexity [43].

The main contributions that were brought by the authors are: the yield loss and lifetime reliability are both quantified from the perspective of persistent and non-persistent faults. The new fault tolerant mechanism that was discussed in the previous paragraph is another contribution; this mechanism allows for multi-bit

error correction for cache tags and small cache line segments. They also analyze the trade-off between maintaining the cache capacity high or using it for fault tolerance techniques, this is done via the Operating System, by allowing this Operating System to actively adapt the cache for fault tolerance, low voltage and high performance. The most important results in terms of number obtained by the authors is the operation of a cache memory at 520 mV, with minimal latency and high reliability, by using half of the cache's capacity for fault tolerance [43].

The main downfall of the method presented in this paper is that it takes from use cache lines, in order to maintain the reliability of the chip, while decreasing the supply voltage. By doing this, even though, the reliability is maintained the performance will be decreased, because the number of misses in the cache will increase. This number increases because the capacity of the cache is reduced; in some cases it can be even reduced to half of its original size.

4.2 Reliability-driven ECC Allocation for Multiple Bit Error Resilience in Processor Cache

In this paper the authors propose a reliability-driven ECC allocation scheme that matches the relative vulnerability of a memory block with ECC protection. The memory block is determined using post-fabrication characterization. In order to reduce the number of check bits, the authors make use of shortened Bose-Chaudhuri-Hocquenghem (BCH) cyclic code with zero padding. Also in order to reduce the impact on energy, performance and are, the authors propose an efficient circuit/architecture-level optimizations of the ECC encoding and decoding logic [45].

The authors address the problem of runtime errors. So far this problem has been addressed by using either parity bits or Hamming codes which can detect double errors and correct single errors (SECDED). But these techniques are becoming less effective due to technology scaling and the increase of MBUs (multiple-bit upsets). Also the overhead added by these techniques increases when all cells are treated equally, for the most pessimistic case. Because of the process variations of inter and intra-die, different sections of a memory cell array can move to various process corners [46]. This in turn will have an impact on the distribution of the vulnerability of cells. This means that different cells will have different vulnerabilities because of their spatial position. An example of this is depicted in Figure 4.1 where the typical block size varies from 1 to 8 KB. These are the reasons why the authors try to account for this distribution of vulnerability and try to allocate the ECC according with this distribution [45].

The effectiveness of a multi-bit error tolerance scheme depends on the ECC choice. To address this issue the authors make use of shortened BCH cyclic codes with zero padding for variable ECC protection [47]. In order to increase the performance and reduce energy consumption the correction logic is invoked only when the decoding logic detects an error. Another contribution is the effective sharing of hardware resources used for encoding and decoding.

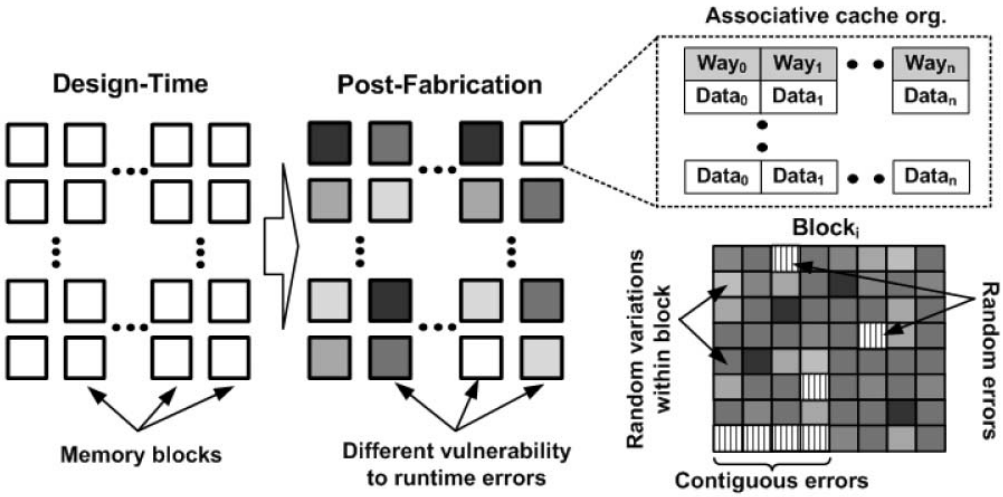

Figure 4.1: Due to intra and inter-die variations blocks of a cache can move to different reliability corners. The cache is *k*-way set associative. The errors can appear from either contiguous or random failures, from [45].

In order to summarize the contributions of this paper we can say that: the authors presented an efficient variable ECC scheme that uses the distribution of memory block reliability. Also they have chosen appropriate ECC techniques in order to minimize area and energy consumption, while at the same time increasing performance. This technique can tolerate both random and contiguous errors [45].

This paper provides a technique for adaptively tolerating errors in the processor cache, by analyzing the distribution of vulnerability and allocating ECC accordingly. What it doesn't account for is the fact that the vulnerability also has a ware and tare component, which in time may change the distribution of the vulnerabilities, and thus making the cache memory susceptible to incurring errors.

4.3 Process Variation-Aware Adaptive Cache Architecture and Management

The authors of this paper are trying to solve the problem of the current design methodologies, which are tuned for the worst-case scenario, and which are becoming extremely pessimistic from a performance standpoint. The use of the worst-case scenario might not be a viable solution in the future, because of technology scaling, which in terms implies an increase in overhead, both area and energy. The contributions of this paper aim at solving this very problem. First, the authors propose an adaptive cache management policy, based on non-uniform cache access. And second, it proposes a latency compensation approach in order to change the access latency of some cache lines. The results presented show that by applying these methods the authors can recover a significant amount of the performance loss [48].

Even though the read, write and hold failures are very important, the most frequent type of failure in a cache is the access timing failure [49] [50]. So in order to avoid these types of failure the worst-case scenario technique has been in use. This technique prolongs the access time in order to avoid these failures. But beside this technique there are at least two others: adapting the execution to variable access latencies; and modifying the latency of selected cache locations, in order to increase performance, but this is done at the expense of aging and/or increase energy consumption. The authors approach these last two methods in order to increase performance, and maintain the reliability of the cache memory [46].

This paper makes the following contributions: exploiting variable access latency; selective latency compensation; and experimental evaluation. We will discuss briefly each of these contributions and present our reader with some results. The exploitation of variable access latency makes use of both address prediction and the March tests. It tries to access a low latency cache location as early as possible, by issuing instructions that depend on load. The selective latency compensation is a mechanism that trades power consumption and reliability for reduced latency, but this can lead to increase aging and important power consumption. The experimental evaluations in this paper are made using the SPEC 2000 benchmarks. These techniques can be implemented for both manufacturing defects and variations that degrade over time, making this a case of graceful degradation. It can also be implemented for L2 caches and TLB (Translation Lookaside Buffer). The performance recovery obtained from the experiments is shown to be over 60% [48].

One aspect that is not treated in this paper is the time degradation. This time degradation can lead to a decrease of the reliability of the cache memory. Also by applying the method described in this paper could lead to a rapid aging of the cache memory.

4.4 Replication Cache: A Small Fully Associative Cache to Improve Data Cache Reliability

In order to avoid the costly solution of N modular redundancy, the authors of this paper propose the use of a small fully associative cache memory, that replicates every write in the L1 cache. This solution is used for protection against soft errors. The impact of the newly introduced cache memory is quite small on performance, and because of data locality it can be proven to be a very efficient solution for soft errors [51].

The newly introduced cache memory is called replication cache; Figure 4.2 depicts the place of the newly introduced replication cache (R-cache) [51].

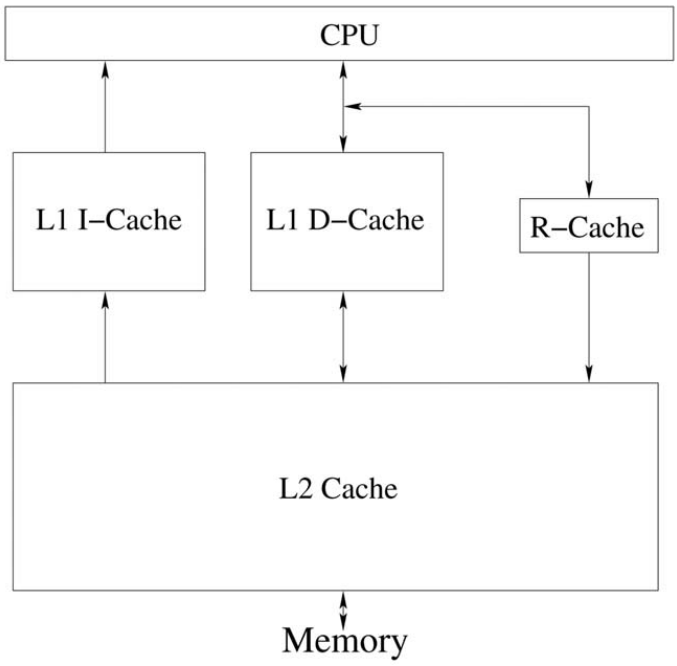

Figure 4.2: Replication cache, that protects the L1 data cache, from [51].

The experiments conducted throughout this paper show that the area overhead introduced by the replication cache is smaller than an N modular redundancy solution. Also it is proven in this paper that this solution can protect up to 97% of the entries. This is done by the data replication that this replication cache provides [51].

Even though the solution provided in this paper might protect a significant number of entries, it introduces a notable, though smaller than N modular redundancy, area overhead. Beside this shortcoming if an error is detected then the mechanism would not know which data is corrupted, which will introduce a performance penalty. Also, by not having a third module, like the triple modular redundancy technique, there might be cases when a faulty entry can pass through this mechanism.

4.5 On the Characterization and Optimization of On-Chip Cache Reliability

In this paper a new framework is proposed for characterization and for conducting comprehensive studies on the reliability behavior of cache memories. This is useful for providing insight into cache vulnerability to soft errors, and also producers and architects can use it in order to improve the reliability of the cache memory. The authors aim at developing a new lifetime model, for both data and tag

arrays, which are located not only in data caches, but also in instruction caches [52].

Throughout this paper a comprehensive number of scenarios are analyzed and based on the results obtained from simulations the new schemes, for tracking and avoiding the soft errors, are developed. The authors define a number of metrics, like the temporal vulnerability factor, and with the help of these metrics they define different vulnerabilities of the cache memory. Beside this, the authors also propose a number of schemes that aim at reducing the vulnerabilities discovered. For example the clean cache line invalidation scheme that was developed aims at reducing the time when clean cache lines stay in the vulnerable read-read phase [52].

Even though this paper provides a very comprehensive study of the errors in the cache memory, and proposes a number of schemes that can address these errors; the hard errors that can appear throughout the lifetime of a cache memory are not discussed in any way. These hard errors can appear due to aging, process variation, or a number of other causes.

In the field of graceful degradation techniques a few ideas made themselves notable. Among these, the most important regarding SRAM memories and cache memories was presented in 2005 in an article called: "A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies" [1]. In this article the authors describe a new graceful degradation method that is applied to cache memories, in order to improve their yield. In the following we will provide an ample description of the method presented in their article alongside with our comments, observations and remarks, providing our reader with a full overview of both the advantages and downfalls of the discussed method.

4.6 Description of a Process-Tolerant Cache Architecture Method

The method described in "A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies", will be referred from hereon as PTCA (Process-Tolerant Cache Architecture). The PTCA method is used to improve the cache memories yield; the results are quite remarkable, from a basic 33% yield, by applying the PTCA the yield will increase up to 94% [1]. But only having a direct intervention in the cache's memory architecture structure, in order to provide replacements for faulty cache cells, can do this.

The fault distribution for a cache memory is depicted in Figure 4.3. This is used by the authors in order to design their method and also for showing their results after the implementation of their methods.

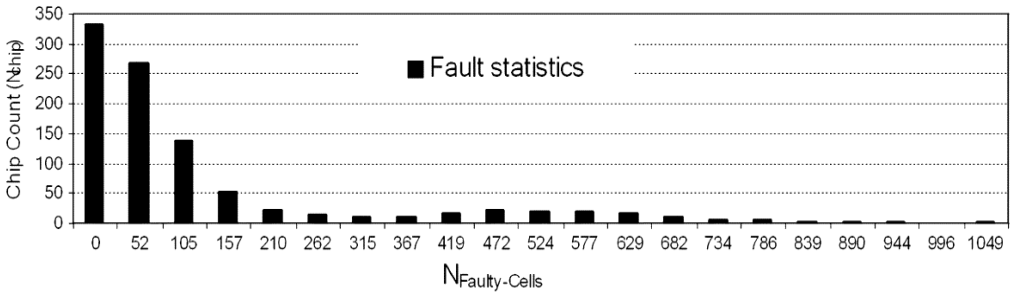

Figure 4.3: Fault statistics of a 64-K cache, from [1].

The basic idea behind PTCA is to replace a faulty cache cell with a healthy neighbor cell, e.g. if there are 8 cells in a column and one of them becomes faulty it will be replaced by one of the remaining seven cells in that column. When all cells in a row become faulty the entire memory becomes faulty. The results and architecture presented for PTCA relate to a 64 kB direct mapped cache memory, as in Figure 4.4:

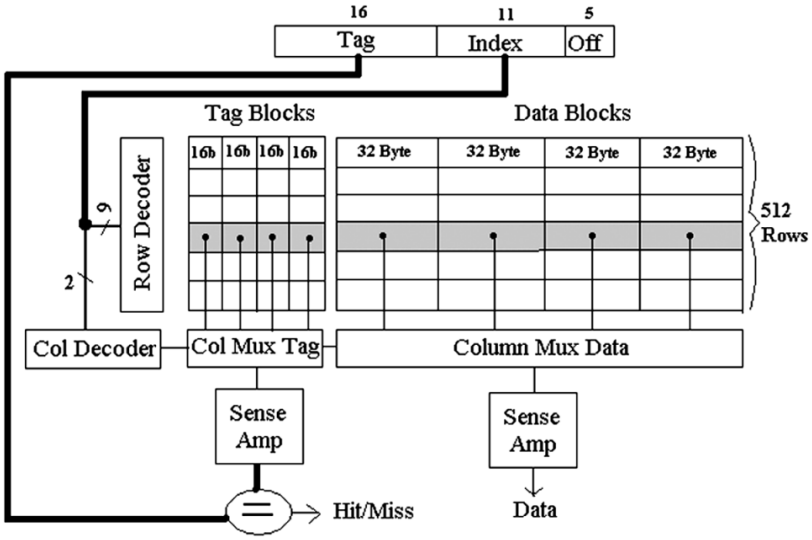

Figure 4.4: Block diagram of a 64-K cache macro, from [1].

The proposed architecture is depicted in Figure 4.5. In Figure 4.5 the BIST used can be any BIST that can recognize a given number of faulty bits in a cell [53] [54]. The Config Storage is used in order to maintain the configuration of the faulty cache cells on the hard drive, even when the processor is turned off, and then reload that configuration on the next booting of the system. The Configurator configures the way that the faulty cells will be mapped in their row and gives information to the tester if the chip is faulty or not.

The convention is to store multiple cache blocks in the same row and access them simultaneously by the use of a word line [55]. So when a cache cell that is

healthy is accessed nothing happens besides the normal cache access, but if a faulty cell is accessed then the Column Multiplexer forces the cell that will be accessed to be the healthy cell that replaces the faulty one. An example of this is presented in Figure 4.6.

With this type of remapping there can appear a few problems: if for example two cache cells in the same row have the same tag, one of them is faulty (cell "one"), and is replaced by the other one (cell "two"); then a problem appears if we are looking at this set of instructions:

STORE D "one"
LOAD "two" Register

The problem is that since both cells have the same tag, then cell "one" will be stored at the location of cell "two", then the processor will deal with a cache hit instead of a cache miss, due to the fact that they have the same tag, see Figure 4.7 (a). In order to deal with this problem the authors have proposed to include the column address bits into the tag bits, which will increase the size of the tag, see Figure 4.7 (b). After applying these modifications to the mapping, then the problem discussed, will be resolved as depicted in Figure 4.7 (c).

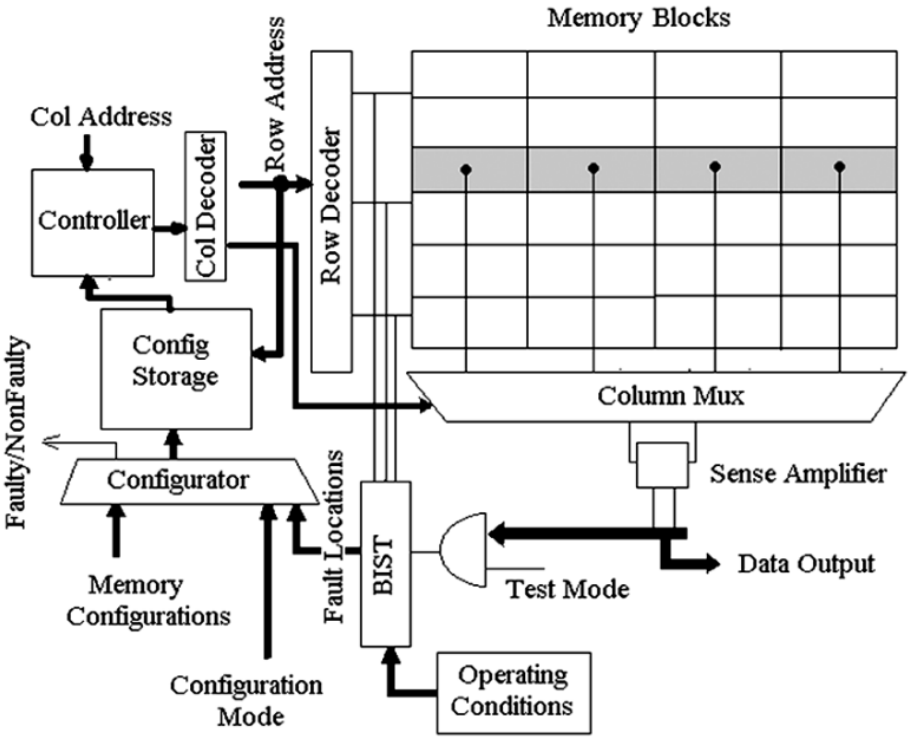

Figure 4.5: Architecture of a 64-K process-tolerant cache, from [1].

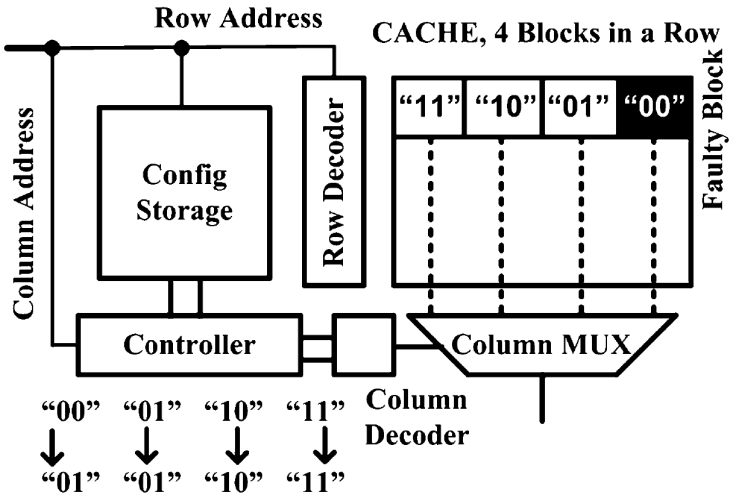

Figure 4.6: Resizing the cache, from [1].

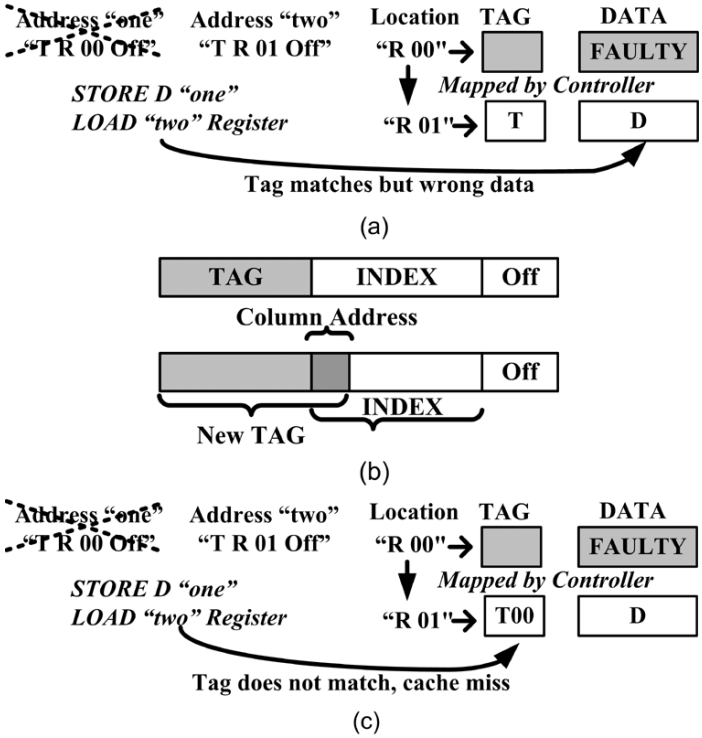

Figure 4.7: Resizing of cache based on the fault location. (a) Mapping problem. (b) Extending tag bits. (c) Resolving the mapping problem, from [1].

The fault tolerance of the PTCA method is proportional with the number of columns in a row, so the more columns the higher the fault tolerance will become.

In order to implement the Config Storage the authors have proposed two methods:

1) content addressable memory (CAM) implementation
2) one-bit implementation (OBI)

These two implementations are presented in Figure 4.8.

For the CAM implementation the fault locations (index bits) will be stored into a CAM [56]. The size of the CAM will depend on the total number of faults that need to be tolerated. A 100-entry CAM is depicted in Figure 4.8 (a).

The OBI adds one bit per cache block, which tells the Controller if that block is faulty or not. An example of the OBI is depicted in Figure 4.8 (b).

Table 4.1 shows a comparison in terms of energy and performance between the CAM implementation and the OBI. While in Table 4.2 is presented how a four block per row cache is evolving when encountering four errors in the same row.

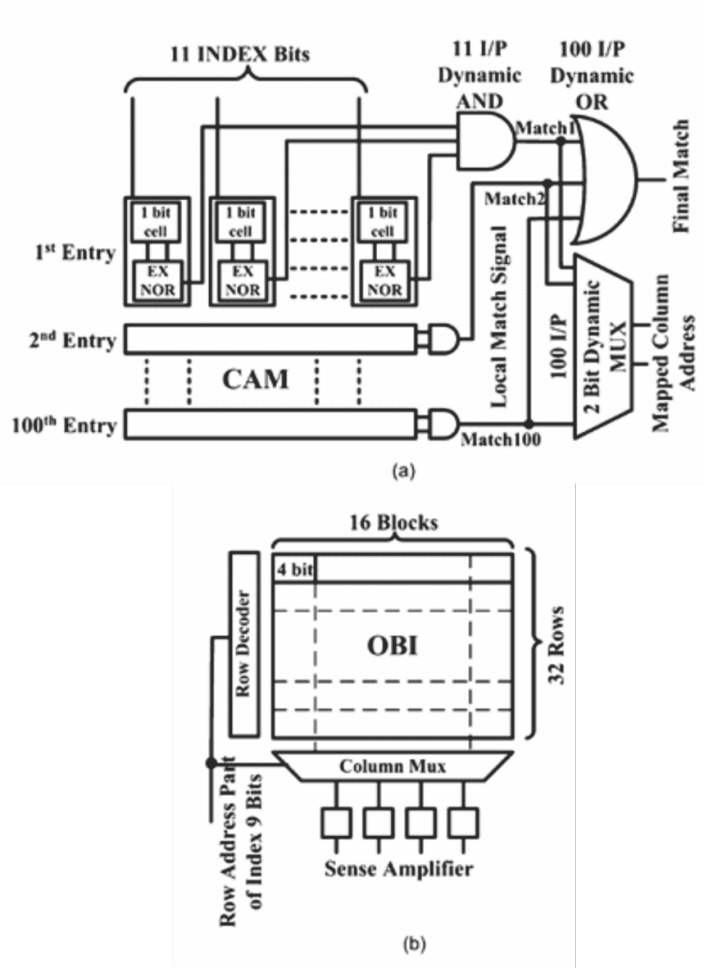

Figure 4.8: Config Storage. (a) CAM, an example to store 100 faults, (b) OBI, from [1].

In Figure 4.9 there are presented some probabilistic results that show the improvements resulted from the use of the PTCA method. Figure 4.9 (a) compares

the PTCA with ECC (error correcting codes) and redundancy, Figure 4.9 (b) depicts the behavior when the OBI is used, Figure 4.9 (c) presents the PTCA with a redundancy used in the cache memory, while Figure 4.9 (d) depicts the ECC method with a redundancy option.

Table 4.1: Comparison of Energy and Performance between different Config Storage, from [1].

Energy and Performance		64KB Cache	CAM 100 entry	CAM 200 entry	OBI 2k bit
Delay (ns)		3.86	0.88	1.11	1.81
Energy (nJ)	Match	1.89	0.036	0.074	0.034
	No-Match		0.031	0.063	
Energy overhead	Match	NA	1.9%	3.9%	1.8%
	No-Match		1.6%	3.3%	

Table 4.2: Column address selection based on fault location, from [1].

Faulty Blocks in Accessed Row	Fault Info by Config Storage	Accessed Column Address			
		00	01	10	11
		Forced Column Address			
		↓	↓	↓	↓
None	0000	00	01	10	11
3rd Block	0100	00	01	00	11
2nd &3rd Block	0110	00	00	11	11
1st, 2nd & 3rd Block	0111	11	11	11	11
All four Blocks	1111	NA	NA	NA	NA

4.7 Results of the Process-Tolerant Cache Architecture Method

In Figure 4.9 there are presented some probabilistic results that show the improvements resulted from the use of the PTCA method. Figure 4.9 (a) compares the PTCA with ECC (error correcting codes) and redundancy, Figure 4.9 (b) depicts the behavior when the OBI is used, Figure 4.9 (c) presents the PTCA with a redundancy used in the cache memory, while Figure 4.9 (d) depicts the ECC method with a redundancy option.

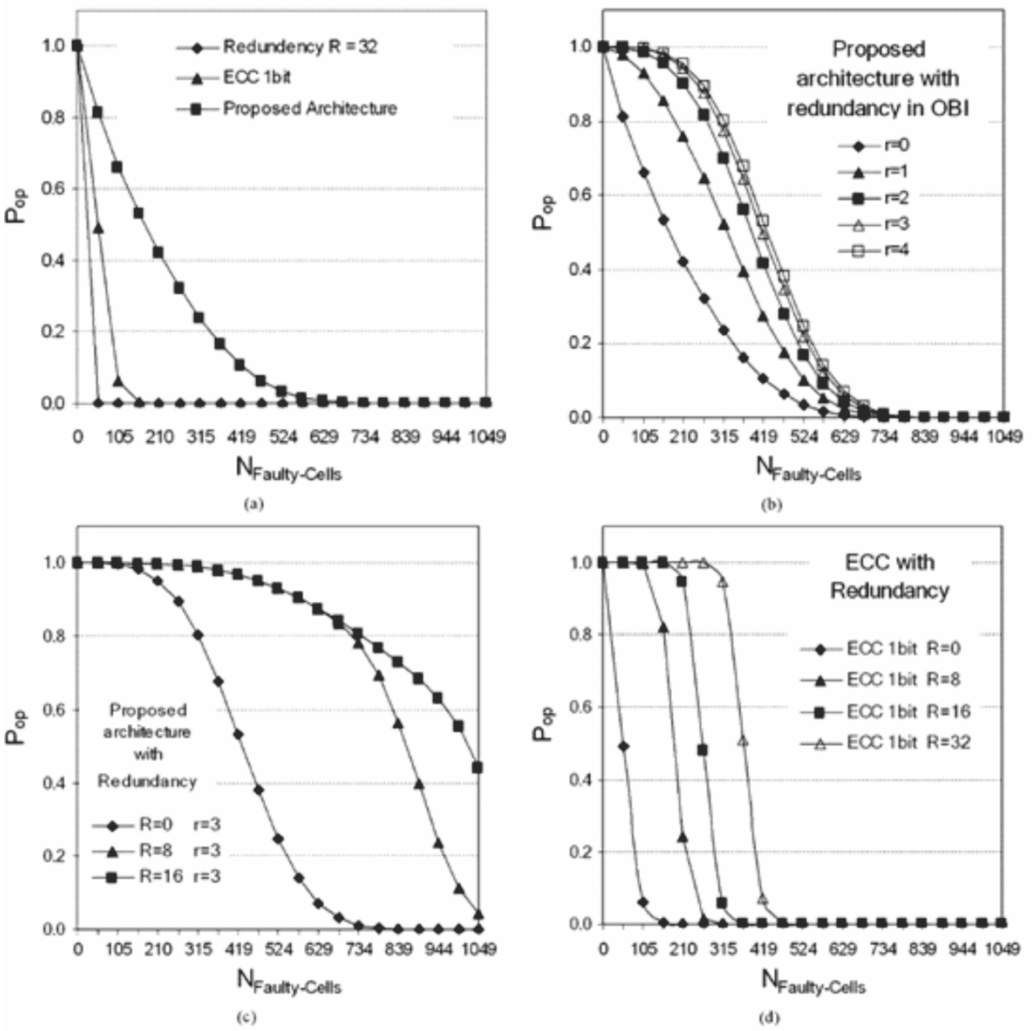

Figure 4.9: Probability of salvaging a chip versus fault probability for a 64-K cache. (a) Proposed architecture, ECC, and redundancy. (b) Proposed architecture with redundancy in OBI. (c) Proposed architecture with redundancy in cache. (d) ECC with redundancy, from [1].

In Figure 4.10 (a) the results in terms of yield are presented when OBI is used, while Figure 4.10 (b) depicts a comparison between the PTCA architecture implemented with OBI and redundancy, compared to the ECC method and the simple redundancy method in terms of yield.

Figure 4.11 shows the number of chips that can be saved after implementing the PTCA method, compared to the initial statistics of the faults of a cache memory.

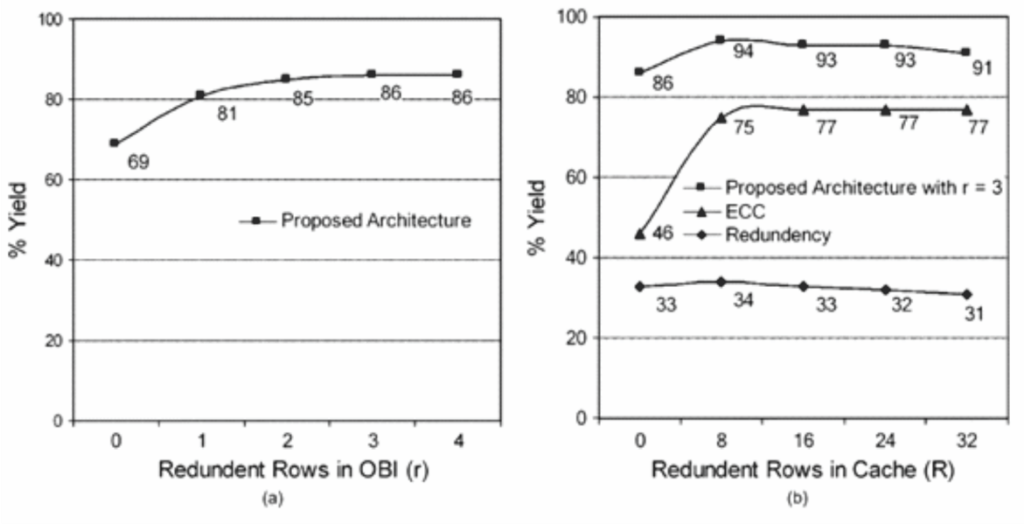

Figure 4.10: Effective yield improvement. (a) Using proposed architecture along with redundancy to OBI. (b) Using different schemes with redundancy to cache. Plot 1: Redundancy to cache. Plot 2: ECC along with redundancy to cache. Plot 3: Proposed architecture along with redundancy to cache and OBI (r = 3), from [1].

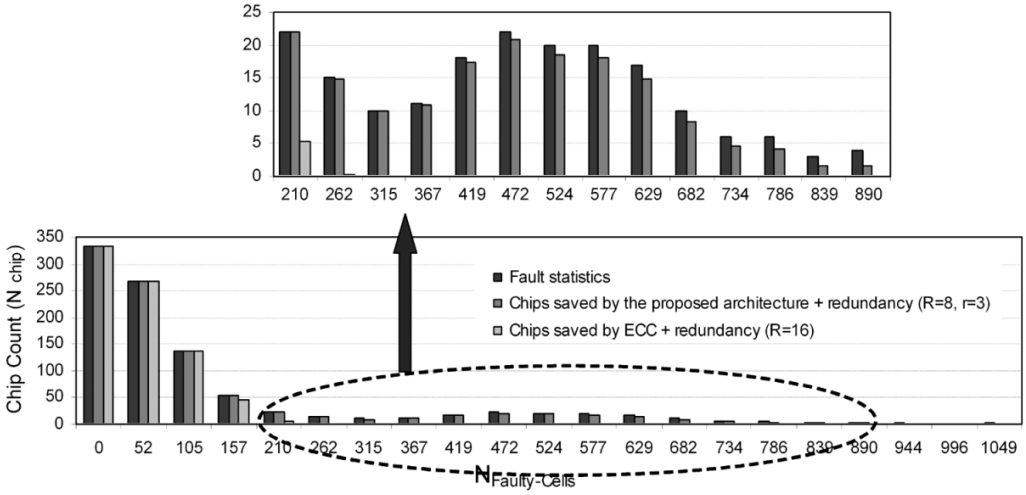

Figure 4.11: Number of chips saved by proposed architecture and ECC with optimum redundancy versus number of faulty cells for a 64-K cache, from [1].

4.8 Conclusions and discussion

As a conclusion to the article discussed we can say that even though it presents a number of improvements to the yield of cache memory chips, it also presents a set of disadvantages. The first and most important one is that in order to

implement the PTCA method the manufacturer has to modify the reading and writing process of the cache memory in order to accommodate the changes that come with PTCA. A second disadvantage is that the yield improvement is dependent on the number of cache blocks that are accommodated within a row. But the most important downfall, in our opinion, is that the PTCA method does not address a very important issue, that is, usually errors appear in patterns (e.g. Neighborhood Pattern Sensitive Faults), and so it is more probable that a neighboring cell of a faulty cell to become faulty than any other. So the method by which a faulty cell is replaced with a neighboring healthy cell might, in time, not to be very efficient.

In order to address these downfalls of PTCA, and not only, we will propose a method called Self Adaptive cache Memories, or SAM, this will be presented in the following chapter.

5 Self-Adaptive cache Memories

In this chapter we discuss an original graceful degradation method applied to k-way set associative cache memories. The method is called "Self Adaptive cache Memories" (SAM); it works by removing the faulty locations from use, while reorganizing the memory to maintain a high performance. For the proposed contribution, the analysis provided herein reveals a significant reliability increase for the cache memory, while the entailed overhead remains small in comparison with the attained goals.

5.1 Introduction

As memory systems continue to decrease in size, the probability of hard, permanent faults increases especially in SRAM cells [1]. Due to this fact the usual method, using spare rows/columns, for preventing hard faults can become obsolete [1] [6]. The hard errors can appear due to process variation [1] [4], aging [2], or other.

We propose a new method called SAM (Self Adaptive cache Memories), which is used to disable from use the faulty cells that have been diagnosed as incurring hard errors. To this end, we will assume that the cache memory has a concurrent built in self-test (BIST) capable of detecting the errors that may occur. Being a case of graceful degradation, this method will have a loss in performance because the size of the cache memory is decreasing [7] [8] [3]. The research presented herein aims at reducing that loss to a minimum by remapping some memory locations, and by the fact that the memory will be continuously adapting to new faulty locations.

5.1.1 L-Zone

First we need an extra bit for each memory cell; we will call this bit an 'L' bit. This bit allows us to separate the faulty cells from the non-faulty cells: if the L-bit of a cell is '1' it means that the cell is faulty and if the L-bit is '0' it means that the cell is working correctly. All of the L-bits form the so-called L-Zone.

For a simpler representation of the memory, we will separately present the L-Zone from the memory cell array. Taking a *k*-way set associative cache memory with *n* locations in each set; we consider having 5 faulty cells – represented by shaded cells in Figure 5.1 (a). Figure 5.1 (b) represents the corresponding L-Zone of the memory cell array.

The L-Zone is filled with zeros when the entire memory works correctly. When a hard error that cannot be corrected appears in a memory cell, the cell's corresponding L-bit becomes 1. An error is dealt with in the following way: if the

concurrent BIST detects an error which it cannot correct, then the error type (hard or soft) will subsequently be determined; this can be done as simple as another read/write from/to the same cell. If it was a soft error, then at the next access at the cell address, the error has a very high probability of disappearing. If it disappears, it means that we don't have a hard error, so the memory can resume its normal functions. If at the next access the error persists, this means that we have to deal with a hard error and that particular memory cell can't be used any longer without possible data corruption. This is the point where the actual SAM method is taking over. If the BIST logic of the memory chip has a non-concurrent testing option and if the system in which the cache memory is working in allows it to be shut down for a period of time, than the non-concurrent BIST can be used as the next two (optional) steps of the algorithm: they consist of shutting down the memory for some time, so a non-concurrent test can determine the type of error that has been found and can generate a report to be processed by the CPU; this feature can be used by the manufacturer to make future improvements to the product. The algorithm described in this paragraph can be seen in Figure 5.2. The final step is to make the cell's corresponding L-bit '1'. This step triggers the following operations:

- Checking if more than one location in a cache set is faulty. A cache set represents all of the cache locations in which a main memory location can be mapped (see section 5.1.2).
- Taking the preemptive measures in order to assure that no more than one location per set will be faulty (section 5.2.1).
- If there is no way that we can avoid more than one faulty location per set, then we have to decrease the set associativity of the cache. See section 5.2.2 for details.
- In 5.2.3 we'll propose a method of reorganizing the memory in order to eliminate from use the faulty locations

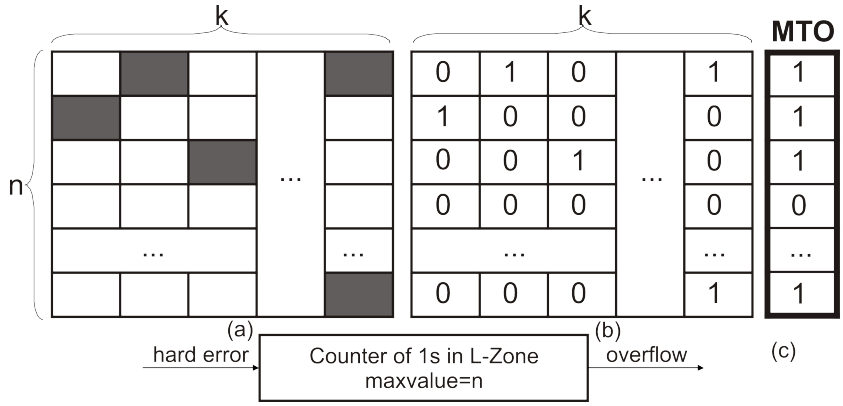

Figure 5.1: SAM description. (a) Memory cell array; (b) L-Zone; (c) MTO column and counter, from [9].

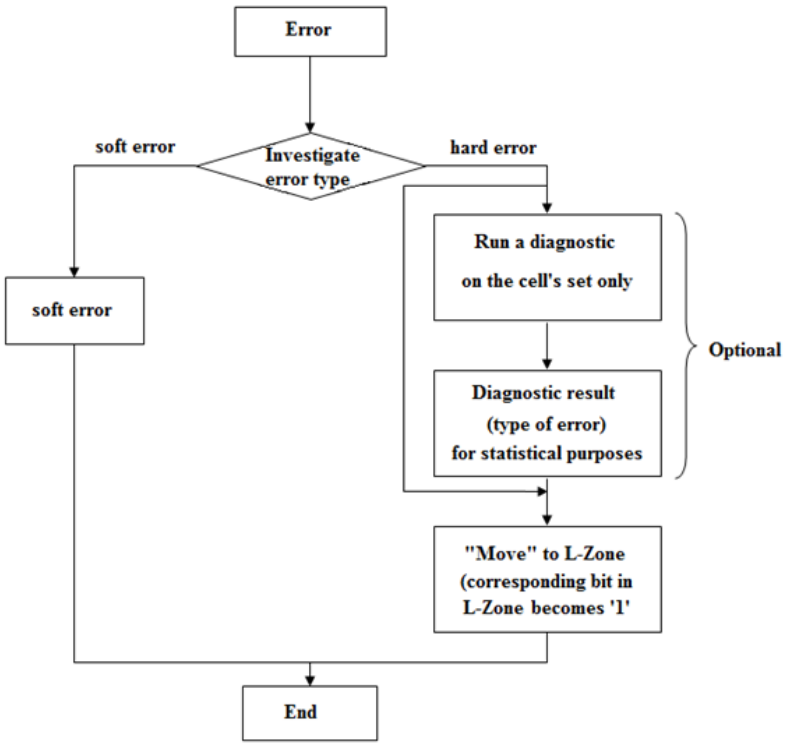

Figure 5.2: Algorithm for handling errors.

5.1.2 "More Than One" column

If we have only a faulty cell in a set, which means that the set associativity of that set is reduced by one, is a situation that needs no further operations. But we will encounter a problem if in some sets all the cache lines work correctly while in other sets we face two or more faulty cache lines. A method for avoiding this situation will be discussed in section 5.2. In this section we will provide MTO (More Than One) column, as an instrument for preventing the problems listed above, which consists of one extra bit for each set of the cache memory, the so called MTO-bit.

Besides this column we need a counter to keep track of the numbers of 1s in the MTO column with the *maxvalue=n*, where *n* is the number of sets, *n=(number of locations in cache)/k* with the cache being *k*-way set associative. This counter will hold the number of encountered faults. We could use another method: a cascade of AND gates from the MTO column which will indicate if all MTO-bits are '1'; this can reduce the logic of the circuit, but it has a downfall: the exact number of faults that had occurred will be unknown, see Figure 5.1 (c).

The MTO-bit of a set becomes '1' when an error is found on that set, and it stays '1' until all of the MTO-bits are '1' and an error is discovered, then the whole MTO column will be reset to '0'. The MTO column along with the hard error signal

will generate the following behavior: if the MTO-bit is '0' then it becomes '1'; else if the MTO-bit is '1' and we don't have an overflow from the counter, the MTO will generate a signal called L-Zone_output which will indicate that we have more than one error in a line, and we need to perform a remapping of the newly discovered hard error, see Figure 5.3. The last case is when the MTO-bit is '1' and we have an overflow from the counter, this means that we need to reduce the set associativity of the cache memory. The algorithm is depicted in Figure 5.3.

5.2 Modifications of the Set Associativity

5.2.1 Maintaining the set associativity

Maintaining the set associativity in a continuously degrading memory is a difficult task even if we can eliminate the faulty cells from use, because if – for instance – an entire line is eliminated the memory, it will work slowly or it won't work at all.

If we take the scenario described above, depending on the write policies we can have a very slow working system in case of a look-aside policy, and a faulty system in case of a look-through policy. In order to avoid such a case we implemented a replacement policy; see the algorithm in Figure 5.3.

```
if (hard_error)
    if (MTO[line]==0)
    {
        MTO[line]=1;
        L-bit[address]=1;
        counter=counter+1;
    }
    else if (overflow==0)
    {
        modify_address_to_first_not_0_in_MTO_column;
        counter=counter+1;
    }
    else
    {
        counter=counter+1;   //reset counter
        for (i=0; i<n; I++)
        MTO[i]=0;
    }
```

Figure 5.3: SAM algorithm, from [9].

We will focus in this section on the "modify_address_to_first_not_0_in MTO_column" instruction for this we will use an example. Considering the situation from Figure 5.4 (a) and we have a new uncorrectable error in line two set two. The memory contents will look like in Figure 5.4 (b), which will decrease the set associativity of line two with two while we still have lines with an intact set associativity; this is unacceptable. Therefore we search for the first line with the

MTO-bit '0', in this case this is line one, and we'll need to "switch" the faulty cell with a healthy cell from the same line, in which case the transformation of the memory will look like Figure 5.4 (c).

The "modify_address_to_first_not_0_in_MTO_column" instruction does the followings: it searches for the first '0' in the MTO column (it is found because the counter hasn't reached an overflow), and it makes a "switch" between the last available memory location in that line with the faulty location. Note: the actual memory doesn't switch the locations physically, so the memory still looks like Figure 5.4 (b) for the considered example. It is a virtual switch because the faulty location cannot actually be replaced with the healthy location, but instead all of the operations on the faulty cell will be performed on the healthy cell. Section 5.2.3 explains the way to implement the switch.

5.2.2 Reducing the set associativity

If we encounter a number of *m* faulty locations, where *m* is a multiple of the number of sets, n (i.e. $m=n \cdot l$, $l \in \{1, \ldots, k\}$, where *k* is the number of cache lines per set), in order to maintain a stable performance we are obliged to reduce the set associativity of the cache memory. This varies from cache memory to cache memory, mainly depending on the replacement policy that is being used. In this chapter we will only discuss the reducing of set associativity for cache memories that use LRU (Last Recently Used) as replacement policy. A similar method can be used for FIFO (First In First Out) replacement policy, due to their similar implementation. Note that LRU is the preferred mechanism for the replacement policy.

One of the implementations of the LRU algorithm is depicted in Figure 5.5 (a). The main idea is to maintain a list of cache set indices sorted from LRU to MRU (Most Recently Used) [57]. When a cache set is accessed its set index *s* is presented to the list, and that index is rotated to the MRU position at the end.

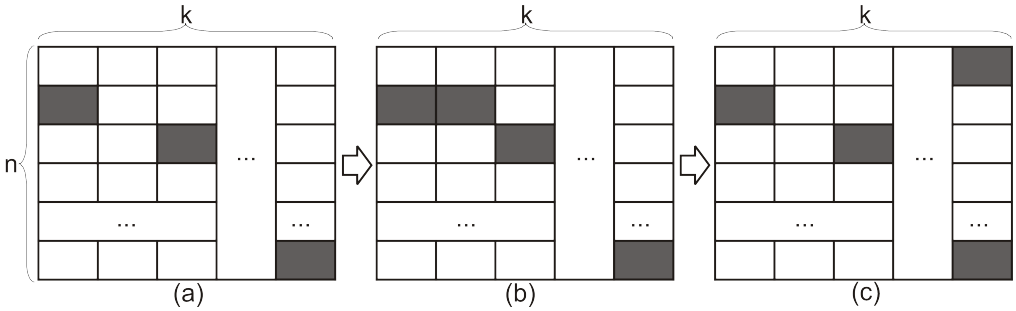

Figure 5.4: SAM remapping. (a) initial memory; (b) unacceptable error distribution;

(c) acceptable error distribution, from [9].

For using the SAM method it is more convenient to reduce the set associativity of each line, instead of just waiting until we encounter *n* errors. The

reduction will be performed by moving the faulty cell address in the LRU index, and after that, the LRU-1 will become the LRU column, as in Figure 5.5 (b). This means that we will eliminate from use one line in that set. By performing this operation we will ensure that the faulty location will be placed in the LRU column, which will never be accessed again, ensuring this way that the faulty cache line will never be accessed.

5.2.3 Reorganizing the memory

One final step that we have to discuss is the "switching" of the locations. The proposed method is somehow similar to the TLB (Translation Lookaside Buffer), meaning that we have a table with two columns: within the first we have the address of the faulty location, whereas within the second one we have an address of a healthy location. This location is taken from the first set in the memory cell array with the MTO-bit equal to '0'.

See Figure 5.6, which is a simple example of cache memory with faulty locations. In this example the actual switching doesn't occur until the memory location (2,4) is accessed; then its L-bit being '1' and the address being found in the table, the location (4,4) is used instead.

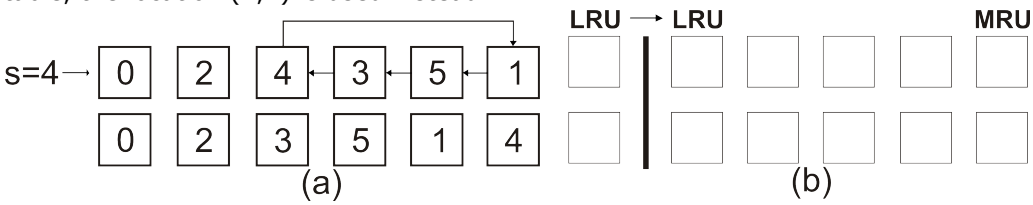

Figure 5.5: (a) LRU algorithm (b) reducing the set associativity, from [9].

5.3 Overhead

Giving the fact that the SAM method eliminates faulty memory cells from use, it is no reason to worry about encountering any other faulty locations besides the ones already eliminated. Our main concern is to find the most efficient size for the switching table. To this end, we need to take into consideration the overhead that the table is generating and the number of faults that need to be tolerated.

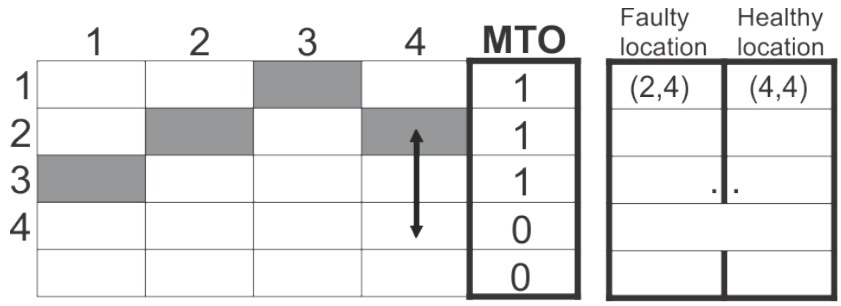

Figure 5.6: Switching table, from [9].

In order to find the most efficient size of the switching table we resort to some probabilistic calculations. It is necessary to find the most probable distribution of the errors in the memory, after a number of *l* errors already occurred. We will consider that a new faulty location can appear anywhere in the memory with the same probability. In order to do that we will "split" the memory in two parts: one that contains faulty sets, and the other part that contains only healthy cells. An example of this is given in Figure 5.7.

If we have a memory like the one in Figure 5.7, after *l* errors the possible locations in the faulty lines becomes: $possibleF=x \cdot k-l$ while the one in the healthy locations: $possibleH=(n-x) \cdot k$, in order to be in the most probable case scenario, after *n* errors, the two have to be equal: $possibleH=possibleF$ which implies that:

$$x = \frac{n(k+1)}{2 \cdot k} \qquad (5.1)$$

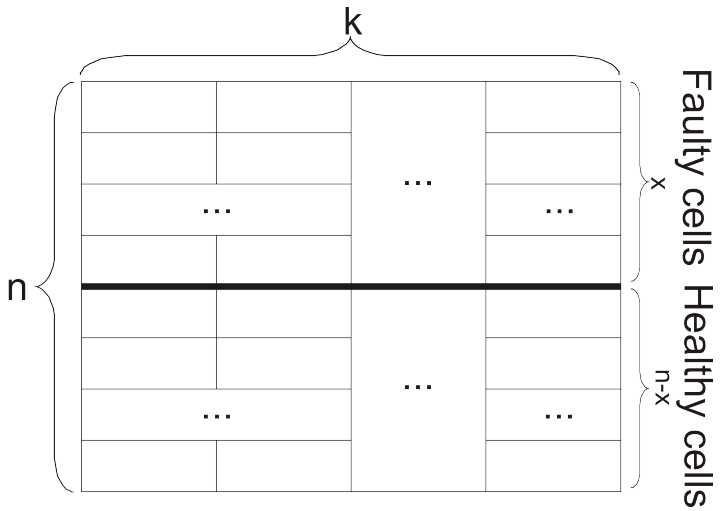

Figure 5.7: Faulty/healthy cells memory organization, from [9].

Example. Consider a L2 cache, 2MB 8-way associative, with 256B block size, as described in [6]. We will calculate the overhead for this memory, for the most probable case scenario, as discussed above. The number of bits in the

switching table will be $\log_2(1024 \cdot 8)=13$, thus making the size of a row in the switching table equal to $2 \cdot 13=26$ bits. This number will be multiplied by the number of locations necessary in the switching table. We will calculate the overhead necessary in order to reduce the cache from an 8-way set associativity to a direct mapping. There are 448+439+427+410+384+342+256=2706 locations necessary in the switching table, thus making its size $2706 \cdot 26=70356$ bits, see Table 5.1. These bits are added to the ones from the MTO and L-Zone: $n(k+1)=9216$ bits, resulting in 79572 overhead bits. This will result in an overhead of 0.474% without taking into consideration the valid bit and the tag bits used by any standard cache memory, see Figure 5.8.

Table 5.1: Numbers of locations required in the switching table, from [9].

k	8	7	6	5	4	3	2
x	576	585	597	614	640	682	768
n-x	448	439	427	410	384	342	256

Compared to the method described in [1], where if a whole row becomes faulty, the yield will be decreased, SAM can maintain a cache memory working even if a whole set becomes faulty; this is done by the use of the switching table.

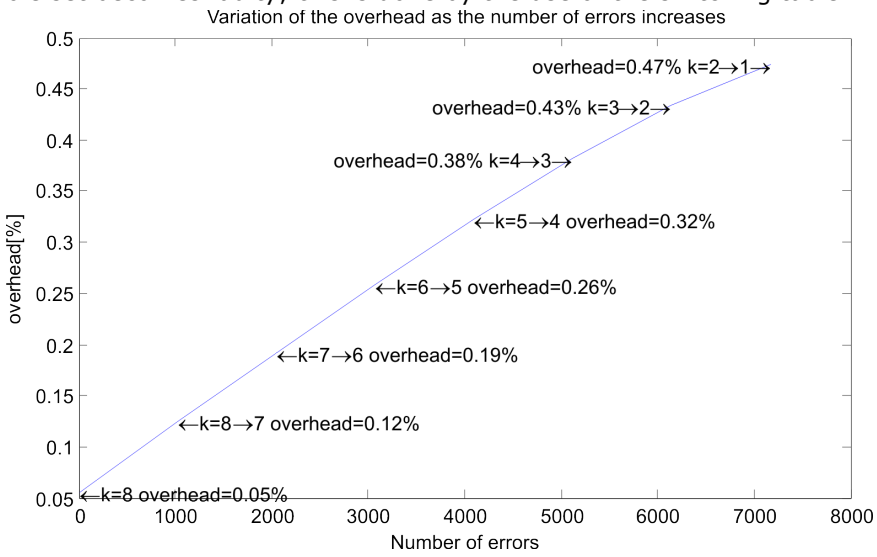

Figure 5.8: Overhead for each reduction of the set associativity, from [9].

5.4 Conclusions

The main goal of this chapter was to establish the theoretical foundations of the SAM method. By applying the SAM BIST method to a cache memory we increase its reliability by eliminating the faulty cells from the memory.

In order to give a rough estimate of the reliability of the memory we make a set of assumptions: the faults appear independent within the memory, without correlation, we take into consideration only the memory cell array. The SAM method is applied in order to sustain the reduction of the set associativity until direct mapping. We can say that the memory will stop working correctly after a number of $(k-1)\cdot n+1$ faults. A fault in the memory appears with a p probability, so instead of a reliability $R=1-p$ [58], we obtain a reliability $R=1-p^{(k-1)\cdot n+1}$, which means that we obtain a much more reliable memory system. Considering that a fault appears every 10 hours of continuous memory functioning, after introducing a concurrent BIST to the memory we increase that period to 100 hours. This can suffice to an application in which the reliability isn't as important as the performance but, for an application where the importance of reliability is paramount, this doesn't suffice. After introducing the SAM method to that memory system we can keep the memory functional not for 100 hours but for $100\cdot((k-1)\cdot n+1)$ hours (e.g. $k=8$, $n=1024$ $\Rightarrow 100\cdot((k-1)\cdot n+1)=716900$ hours, which means an improvement of 7169 times. This improvement is created at the cost of reducing the capacity of the memory. It is necessary to find a critical point at which the performance will decrease too much and the memory chip will need to be replaced. This critical point will differ from application to application.

By introducing a BIST, which detects and corrects more errors, we can avoid wrongly eliminating some of the healthy cells in the memory; this can happen if another soft fault appears in the re-reading of the memory cell. Another way we can reintroduce some cells in the normal use is by the use of a non-concurrent BIST test, which determines if the cells in the L-Zone are truly faulty or have been eliminated by mistake. If any cells like this exist they can be taken out of the L-Zone and re-possess their place in the memory, hence increasing the reliability and the performance of the system.

The overhead introduced by the SAM method can be considered as small, given the reliability which it provides, as it is presented in section 5.3. Because we seldom need to reduce the performance of the cache memory to a direct mapping, the overhead can be approximated by the one obtained at $k/2$ set associativity for which the overhead in the example proposed is 0.32%.

In short, the advantages brought by the SAM method greatly exceed the disadvantages and the shortcomings that were also identified in this chapter. Another perspective on the contribution is that by creating a few extra misses in the memory cache we obtain a huge increase of the reliability of the memory.

6 Applied Probability Theory for Fault Tolerant Memory Systems

This chapter proposes to use probability theory, combined with physical designs and methods, with the purpose of accounting failures in the memory component of computer systems.

We will discuss how and where the errors in the memory system appear. We will focus on applying probability theory to determine how a memory system is more probable to look after a number of *l* errors, and how do they distribute within the memory. We will conclude this chapter by showing some simulation results, that we have obtained using the equations and relations that will be provided.

6.1 Method description

An error in the storage component of the memory system can appear anywhere with almost the same probability. We will consider that all the errors in a memory can appear with the same probability anywhere, i.e. the errors are uniformly distributed.

The most important part of the error distribution is to know whether we have faulty rows or not, and if we have faulty rows how many columns in those rows are actually faulty. This is very useful when manufacturers are designing fault-tolerant techniques. With this aim we will "split" the memory into two parts: a part that contains faulty rows, and a part that only contains healthy rows. This is depicted in Figure 6.1.

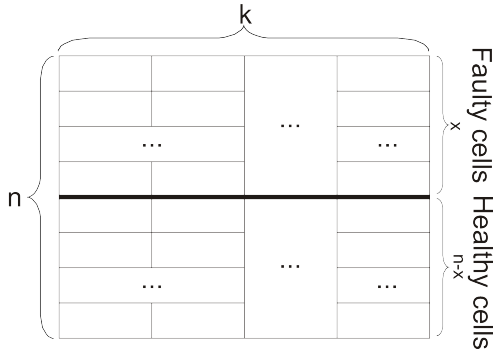

Figure 6.1: Partitioning of the memory into two parts: one that contains only faulty cells or locations, and the other where only healthy locations can be found.

In Figure 6.1 the following notations have been used: n is the number of rows; k is the number of columns; and x is the number of rows that contain faulty cells. If we take into account that l is the number of faulty cells in the memory we will obtain (6.1) , by equalizing the number of healthy cells in the two partitions of the memory. Note that, we changed x into x_1 in order to simplify future notations. So x_1 represents rows that have at least one faulty cell or column in them.

$$x_1 \cdot k - l = (n - x_1) \cdot k$$
$$x_1 = \frac{n \cdot k + l}{2 \cdot k} \tag{6.1}$$

From (6.1) we can find out how many rows have faulty cells, but what if this isn't enough? What if we need to find how many rows have two, or three faulty cells? In order to do that, we will rely on a similar equation and algorithm. First we determine x_1 from Equation 1, this x_1 will tell us how many rows have faulty cells. If l (the total number of errors) is greater than x_1 than, we will continue using Equation 1 with a new setup. That is k will become k-1, l will become l-x_1, and n will be x_1. Equation (6.2) summarizes these changes.

$$x_2 \cdot (k-1) - (l - x_1) = (x_1 - x_2) \cdot (k-1)$$
$$x_2 = \frac{x_1 \cdot (k-1) + l - x_1}{2 \cdot (k-1)} \tag{6.2}$$

In (6.2), x_2 represents the number of rows with at least two faulty cells, x_1 represents the number of faulty cells with at least one faulty cell, the rest of the notations are the same with the ones in (6.1). We can extend (6.2) to a general case, in which we will determine the number of rows with at least r + 1 faulty cells, this is represented in (6.3). For a better understanding we also have illustrated the recursive problem used in Figure 6.2.

$$x_{r+1} \cdot (k-r) - (l - \sum_{i=1}^{r} x_i) = (x_r - x_{r+1}) \cdot (k-r)$$
$$x_{r+1} = \frac{x_r \cdot (k-r) + l - \sum_{i=1}^{r} x_i}{2 \cdot (k-r)} \tag{6.3}$$

In (6.3) x_r represents the number of rows with at least r faulty cells, while x_{r+1} represents the number of rows with at least r + 1 errors. In Figure 6.2 the bolded lines separate the problems. The first problem is when we need to determine x_1, the number of rows with at least one error. After solving this problem we move on to the next problem in which we need to find the number of rows that have two or more errors, this number is represented by x_2. In order to do that, we have to take into account that the only rows that might contain errors are the x_1 rows that we have determined so far, the rest of the rows in the memory being free of errors. In order to determine x_2 we change the parameters in (6.1), to account for the changes in the new problem. Basically we now have a memory with x_1 rows and k-1

columns, and the number of errors in this memory is l-x_1, thus obtaining (6.2). For the general case, i.e. to determine the number of rows with at least r + 1 errors we have the following problem. We have a memory with x_r rows, k-r columns, and the total number of errors left in this memory is l-x_1-x_2-…-x_r, having these parameters and inserting them into (6.1) we obtain (6.3), which represents the general case.

In the following we will provide two lemmas that will help identify the number of rows with an exact number of errors. This means that at any point in time and after an arbitrary number of errors we will know for example how many rows have exactly two faulty cells.

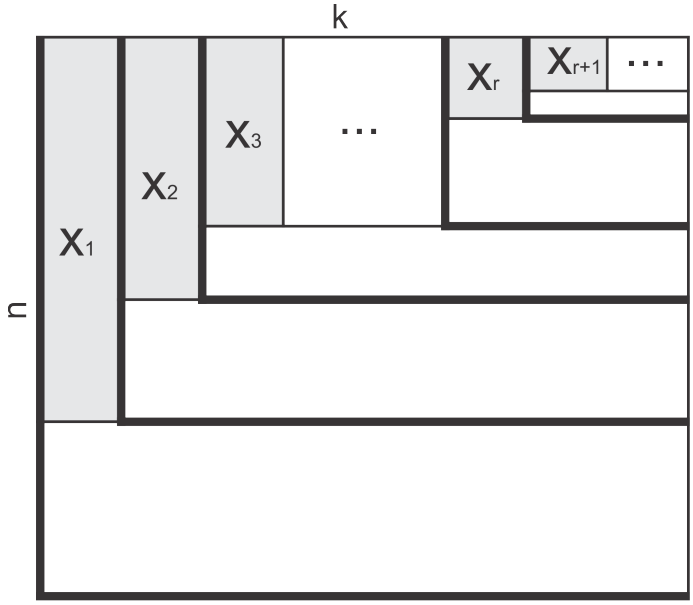

Figure 6.2: Recursive partitioning of the problem, into smaller problems; in order to be able to determine the number of rows with at least r errors.

Lemma 1

In a memory with a number of n rows and a number of k columns, which has a number of l errors that are uniformly distributed throughout the memory, the number of rows, t_1, with exactly 1 error is $t_1 = \frac{1}{4} \cdot \frac{n \cdot k - l}{k - 1}$.

Proof

The number of rows with exactly one error is the difference between the number of rows with at least one error x_1 and the number of rows with at least two errors, x_2.

$$t_1 = x_1 - x_2$$

$$x_1 = \frac{nk + l}{2k}$$

$$x_2 = \frac{x_1(k-1)+(l-x_1)}{2(k-1)}$$

$$x_1 - x_2 = \frac{nk+l}{2k} - \frac{x_1(k-1)+(l-x_1)}{2(k-1)}$$

$$x_1 - x_2 = \frac{nk+l}{2k} - \frac{\frac{nk+l}{2k}(k-1)+(l-\frac{nk+l}{2k})}{2(k-1)}$$

$$x_1 - x_2 = \frac{nk+l}{2k} - \frac{\frac{nk+l}{2k}(k-2)+l}{2(k-1)}$$

$$x_1 - x_2 = \frac{(nk+l)(k-1)-\left[\frac{nk+l}{2}(k-2)+lk\right]}{2k(k-1)}$$

$$x_1 - x_2 = \frac{nk^2-nk+lk-l-\frac{nk^2}{2}+nk-\frac{kl}{2}+l-lk}{2k(k-1)}$$

$$x_1 - x_2 = \frac{\frac{nk^2}{2}-\frac{kl}{2}}{2k(k-1)}$$

$$x_1 - x_2 = \frac{\frac{1}{2}k(nk-l)}{2k(k-1)}$$

$$x_1 - x_2 = \frac{\frac{1}{2}(nk-l)}{2(k-1)}$$

$$x_1 - x_2 = \frac{1}{4}\cdot\frac{nk-l}{k-1}$$

$$\Rightarrow t_1 = \frac{1}{4}\cdot\frac{nk-l}{k-1}$$

Lemma 2

In a memory with a number of *n* rows and a number of *k* columns, which has a number of *l* errors that are uniformly distributed throughout the memory, the number of rows t_r with exactly *r* errors is $t_r = \left(\frac{1}{2}\right)^{r+1} \cdot \frac{n \cdot k - l}{k - r}$.

Proof

As in the previous Lemma the number t_r *with* exactly *r* errors is the difference between the number of rows with at least *r* errors and the number of rows with at least *r* + 1 errors.

$$t_r = x_r - x_{r+1}$$

$$\Rightarrow x_{r+1} = x_r - t_r$$

Substituting x_{r+1} into (6.3) we obtain:

$$x_{r+1} \cdot (k-r) - (l - \sum_{i=1}^{r} x_i) = (x_r - x_{r+1}) \cdot (k-r)$$

$$(x_r - t_r) \cdot (k-r) - \left(l - \sum_{i=1}^{r-1} x_i - x_r\right) = t_r \cdot (k-r)$$

$$x_r \cdot (k-r) + x_r - x_r - t_k(k-r) - \left(l - \sum_{i=1}^{r-1} x_i\right) + x_r = t_r \cdot (k-r)$$

$$\underline{\underline{x_r \cdot (k-(r-1))}} - x_r - t_k(k-r) - \underline{\underline{\left(l - \sum_{i=1}^{r-1} x_i\right)}} + x_r = t_r \cdot (k-r)$$

$$\underline{\underline{(x_{r-1} - x_r) \cdot (k-(r-1))}} = 2 \cdot t_r \cdot (k-r)$$

$$t_{r-1} \cdot (k-(r-1)) = 2 \cdot t_r \cdot (k-r)$$

$$\frac{t_{r-1}}{t_r} = 2 \cdot \frac{k-r}{k-(r-1)}$$

$$\Rightarrow \frac{t_{r-1}}{t_r} \cong 2 \Rightarrow t_r \cong \frac{1}{2} \cdot t_{r-1}$$

Ignoring the last approximation we continue the proof.

$$\frac{t_{r-1}}{t_r} = 2 \cdot \frac{k-r}{k-r+1}$$

$$t_r = \frac{1}{2} \cdot \frac{k-r+1}{k-r} \cdot t_{r-1}$$

$$t_{r-1} = \frac{1}{2} \cdot \frac{k-r+2}{k-r+1} \cdot t_{r-2}$$

$$t_{r-2} = \frac{1}{2} \cdot \frac{k-r+3}{k-r+2} \cdot t_{r-3}$$

.....

$$t_3 = \frac{1}{2} \cdot \frac{k-2}{k-3} \cdot t_2$$

$$t_2 = \frac{1}{2} \cdot \frac{k-1}{k-2} \cdot t_1$$

By multiplying these equations we obtain:

$$t_r = \left(\frac{1}{2}\right)^{r-1} \cdot \frac{k-1}{k-r} \cdot t_1$$

Substituting the result from the previous lemma into the last expression we obtain:

$$t_r = \left(\frac{1}{2}\right)^{r+1} \cdot \frac{k-1}{k-r} \cdot \frac{nk-l}{k-1}$$

$$t_r = \left(\frac{1}{2}\right)^{r+1} \cdot \frac{n \cdot k-l}{k-r}$$

Observation 1

Since the number of errors and locations in a memory is limited, we will reach a point where the number of faulty locations, or errors, will be exceeded by the sum of faulty rows, i.e. $l < \sum_{i=1}^{r} x_i$. This will mean that we already covered all the errors in the memory, and the maximum number of faulty columns in a row is *r*. Also when we encounter this case the last x_r will be computed as follows:

$$x_r = l - \sum_{i=1}^{r-1} x_i \tag{6.4}$$

Observation 2

As we are dealing with a physical system (memory) we cannot have a fractional number of rows, but in some cases x_i and t_i can be fractional numbers. In order for this not to happen we will define x_i^{row} and t_i^{row} which will be defined as: $x_i^{row} = [x_i + 0.5]$ and $t_i^{row} = [t_i + 0.5]$, where $[x]$ represents the greatest integer that is smaller or equal to *x*.

6.2 Simulation results

We have simulated the results presented in section 6.1, using as simulation environment Matlab. The results for a 2MB cache memory with the block size of 256B, which has 1024 rows and 8 [9] columns are depicted in Table 6.1 and Table

6.2, also the memory after 2048 errors is depicted in Figure 6.3. Moreover the results for a memory with 1024 rows and 16 columns are depicted in Table 6.3 and Table 6.4, while Figure 6.4 illustrates how the memory will look after 5120 errors.

Table 6.1: Simulation results for a memory with 1024 rows and 8 columns.

	l=1024	l=2048	l=3072	l=4096	l=5120	l=6144	l=7168
x_1	576	640	704	768	832	896	960
x_2	320	421	521	622	722	823	923
x_3	128	293	414	537	658	780	902
x_4	0	216	350	485	620	755	889
x_5	0	168	310	453	596	739	881
x_6	0	136	284	432	580	728	876
x_7	0	112	264	416	568	720	872
x_8	0	62	225	383	544	703	865

Table 6.2: Simulation results for a memory with 1024 rows and 8 columns.

	l=1024	l=2048	l=3072	l=4096	l=5120	l=6144	l=7168
t_1	256	219	183	146	110	73	37
t_2	192	128	107	85	64	43	21
t_3	0	77	64	52	38	25	13
t_4	0	48	40	32	24	16	8
t_5	0	32	26	21	16	11	5
t_6	0	24	20	16	12	8	4
t_7	0	50	39	33	24	17	7

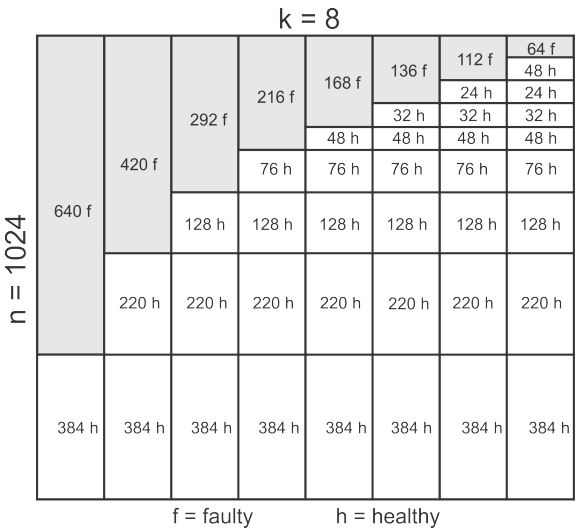

Figure 6.3: The most probable distribution of faults, in a 1024 rows by 8 columns, memory; after a number of 2048 errors.

Table 6.3: Simulation results for a memory with 1024 rows and 16 columns.

	l=1024	l=3072	l=5120	l=7168	l=9216	l=11264	l=13312	l=15360
x_1	544	608	672	736	800	864	928	992
x_2	288	386	484	582	681	779	877	975
x_3	151	267	384	500	617	733	849	966
x_4	41	203	330	456	582	708	834	961
x_5	0	169	300	432	563	695	826	958
x_6	0	150	284	419	553	688	822	957
x_7	0	139	275	412	548	684	820	956
x_8	0	133	270	408	545	682	819	956
x_9	0	130	268	405	543	680	818	955
x_{10}	0	128	266	404	542	679	818	955
x_{11}	0	127	265	403	541	679	817	955
x_{12}	0	127	265	403	541	679	817	955
x_{13}	0	127	265	403	541	679	817	955
x_{14}	0	127	265	402	540	679	817	955
x_{15}	0	126	264	402	540	679	817	955
x_{16}	0	125	263	401	539	677	816	954

Table 6.4: Simulation results for a memory with 1024 rows and 16 columns.

	l=1024	l=3072	l=5120	l=7168	l=9216	l=11264	l=13312	l=15360
t_1	256	222	188	154	119	85	51	17
t_2	137	119	100	82	64	46	28	9

t_3	110	64	54	44	35	25	15	5
t_4	0	34	30	24	19	13	8	3
t_5	0	19	16	13	10	7	4	1
t_6	0	11	9	7	5	4	2	1
t_7	0	6	5	4	3	2	1	0
t_8	0	3	2	3	2	2	1	1
t_9	0	2	2	1	1	1	0	0
t_{10}	0	1	1	1	1	0	1	0
t_{11}	0	0	0	0	0	0	0	0
t_{12}	0	0	0	0	0	0	0	0
t_{13}	0	0	0	1	1	0	0	0
t_{14}	0	1	1	0	0	0	0	0
t_{15}	0	1	1	1	1	2	1	1

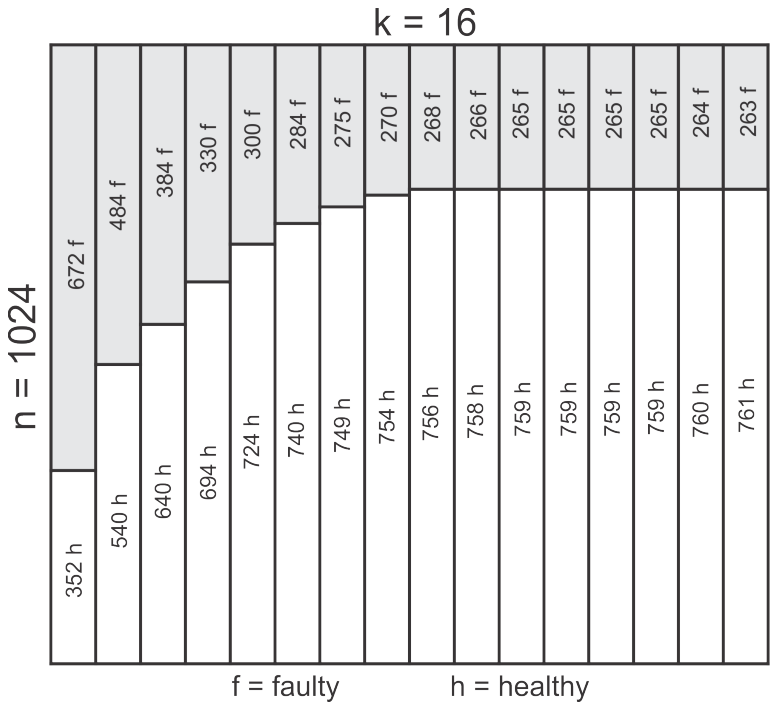

Figure 6.4: The most probable distribution of faults, in a 1024 rows by 16 columns, memory; after a number of 5120 errors.

6.3 Conclusions

Throughout this chapter we have developed a new method for predicting the most probable case scenario for the distribution of faulty cells in a memory system. We have done this by relying on probability theory and by computing our results for

the most probable case scenario. By applying our method, producers and designers of memory chips can find the most probable distribution of faults and plan accordingly in order to counteract their effects of the correct functioning of the system.

To the best of our knowledge a model for accurate prediction of the fault distribution in a memory cell array has not been developed, our original model being the first.

The main advantage of this method is that at any point in time, and for a given number of faulty cells in a memory system, our algorithm can tell its user how many rows have an exact given number of errors in it. Also our model can be easily implemented in simulators and fault injection techniques in order to provide scalability and better accuracy of the results.

The results presented in this chapter have been used throughout chapters 5, 7, and 8, which demonstrated the applicability of the developed method.

7 Improving the Self Adaptive cache Memories Mechanism

In this chapter we provide our readers with two ways of improving the performance and reducing the overhead of the self-adaptive cache memory mechanism. The first method introduces an extra bit for both the L-Zone and the MTO column, and even though it might seem counterintuitive by adding these extra bits we will both reduce the overhead and increase the performance. The second method that we will describe is one that reorganizes the switching table and keeps the records within the switching table to a minimum. The last section of this chapter will provide a merge between these two methods in order to achieve an even greater increase in performance and smaller overhead.

7.1 Algorithm Description for Switching Bits

A major disadvantage of using the switching table in SAM is that every time a faulty cell is accessed, a search in the switching table is performed, and that means a process that induces a significant time penalty. In this section we describe a method of further reducing the number of accesses in the switching table. We also present a snapshot before and after the switching bits are introduced in the cache memory [59].

7.1.1 Switching Bits

In order to be able to reduce the number of accesses in the switching table, more information is required for the case when a faulty cell is accessed. For each line, besides the L-bit, we will add an extra bit called Switching Bit (SB); for each set, besides the MTO bit, we will add an extra bit called Set Switching Bit (SSB), as presented in Figure 7.1. These added bits encode, for each line and set, four functioning states, instead of the two that were acknowledged within SAM (faulty and healthy). Table 7.1 summarizes the four functioning states along with a short description.

7.1.2 Before Introducing the Switching Bits

In this subsection we present the algorithms used by SAM for accessing the memory. First, we will present the algorithm that is used in the case of no error being detected by the concurrent BIST (see Figure 7.2).

If the L-bit of the cache line is 0 – meaning that the line is healthy – then the access is normal, i.e. SAM doesn't insert any changes to the memory access.

But if the L-bit is 1 – meaning the cell is faulty – then a search is performed in the switching table. If the location is found in the switching table's faulty part (that is the only place we are looking for it) then the location from the healthy part of the switching table will be used instead. If the location is not found in the switching table, this means that either the location is faulty or it is substituting a faulty location; either way we will have a miss in the cache memory and we will have to choose some other location from the same set instead.

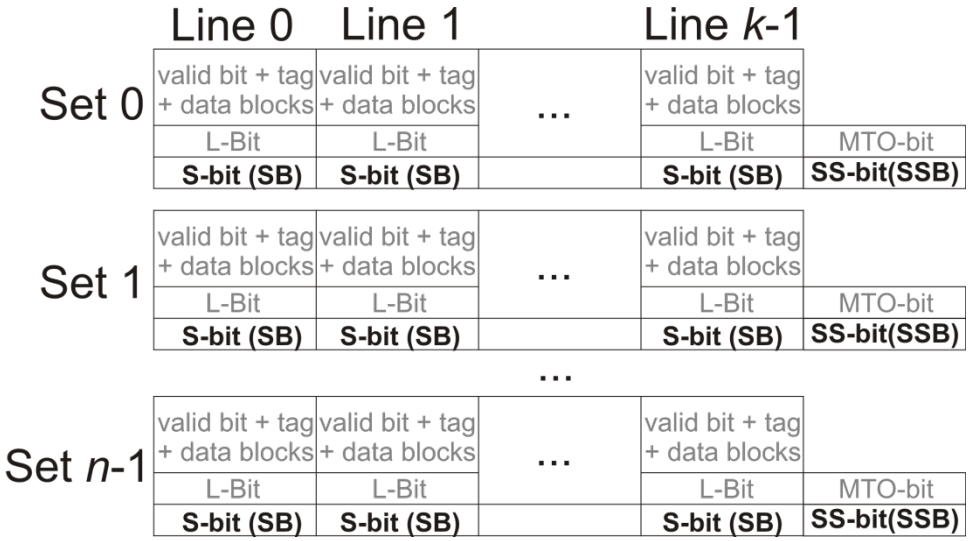

Figure 7.1: Cache memory, after introducing the switching bits

Table 7.1: Description of Switching Bits

	Bits value	Description
Set States	MTO=0 SSB=0	Healthy set
	MTO=0 SSB=1	Switched set (healthy set that has to be used to maintain performance in a faulty set)
	MTO=1 SSB=0	Faulty set (at least one cell in that set is set is faulty)
	MTO=1 SSB=1	Set with a double switched cell
Line States	LB=0 SB=0	Healthy cell
	LB=0 SB=1	Switched cell (faulty cell that has to be maintained functional)
	LB=1 SB=0	Faulty cell (it is usually the first faulty cell encountered when the MTO of the set is 0)
	LB=1 SB=1	Switched cell (healthy cell that has to replace a faulty cell from another set)

Now we will present the algorithm that is used if an error is encountered by the original SAM algorithm, this situation that can be seen in Figure 7.3. First we look at the L-bit: if it is 0, meaning that we are dealing with a healthy cell, then we take into consideration the values of the MTO-bit and the overhead used for the reduction of the set associativity, as described in [9]. If the value of the pair (MTO-bit, overhead) is (0, 0) then the L-bit of the cache line becomes 1 and the MTO-bit of the set also becomes 1. If the pair (MTO-bit, overhead) is (0, 1) then the L-bit of the line becomes 1 and a reduction of the set associativity is performed. If the value of the pair (MTO-bit, overhead) is (1, 0) then the L-bit of the line becomes 1 and a new entry is added to the switching table, thus making this line available for future accesses. The last case is when the pair (MTO-bit, overhead) has the value (1, 1), which means that the L-bit becomes 1 and a reduction of the set associativity is performed.

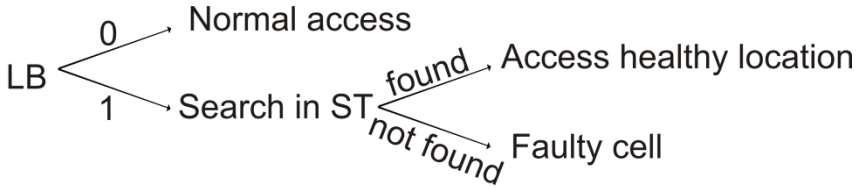

Figure 7.2: Original SAM algorithm for an access of the cache memory, without any errors, from [59].

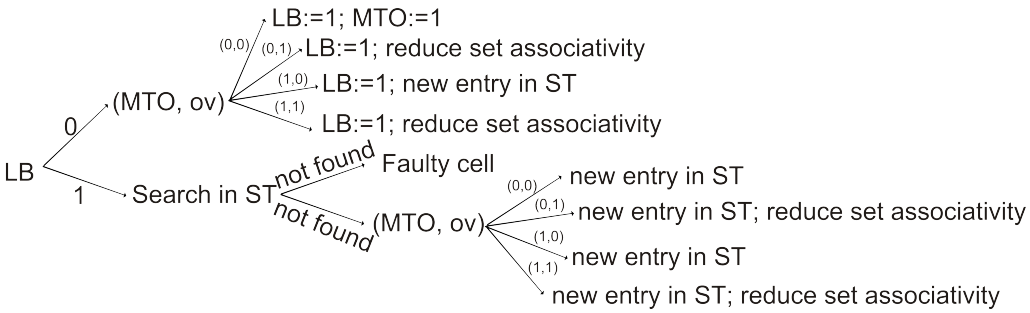

Figure 7.3: Original SAM algorithm for an access of the cache memory, when a hard error is encountered, from [59].

The second case is when the L-bit is already 1, which means that the cache line is already substituting a faulty cache line or it is a faulty cell. For this case we have to perform a search in the switching table within the faulty part. If the cache line is found then this means it is substituting a faulty line. In this case, depending on the value of the pair (MTO-bit, overhead), we have the following two cases. If the pair is (0, 0) or (1, 0) then a new entry is created in the switching table. If the pair (MTO-bit, overhead) is (0, 1) or (1, 1) then a new entry is created in the switching table, followed by a reduction of the set associativity, as described in [9]. The second case only appears if the cache line wasn't found in the switching table's faulty part, which means that the cell was already found as faulty (it was not a new

error) and, in order to access it, we have to use some other location from the same set instead.

As a conclusion to this subsection we provide some notes on the disadvantages of this algorithm used by SAM. First, if an error isn't found and a location is faulty then searches that aren't necessary will be performed in the switching table, thus introducing time penalties. Second, there are a couple of new entries in the switching table that can be avoided, e.g. a new entry isn't required always when (LB, MTO-bit, overhead) is (1, 1, 0); a more detailed explanation of this case will be presented in Section 7.1.5. Third, the searches in the switching table aren't always required if an error is found and the LB is 1; in order to fix this drawback, by adding the so-called switching bits, we can know if that particular location is in the switching table or not.

7.1.3 Algorithms after the Switching Bits

In this subsection we present the algorithms described in Section 7.1.2 which were modified to accommodate the newly added switching bits. The first one is the algorithm used when the concurrent BIST does not detect any error, as depicted in Figure 7.4.

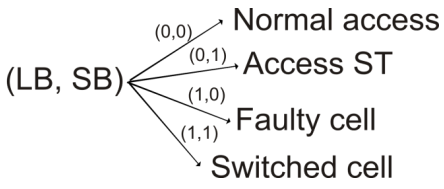

Figure 7.4: Modified SAM algorithm for an access of the cache memory, without any errors, from [59].

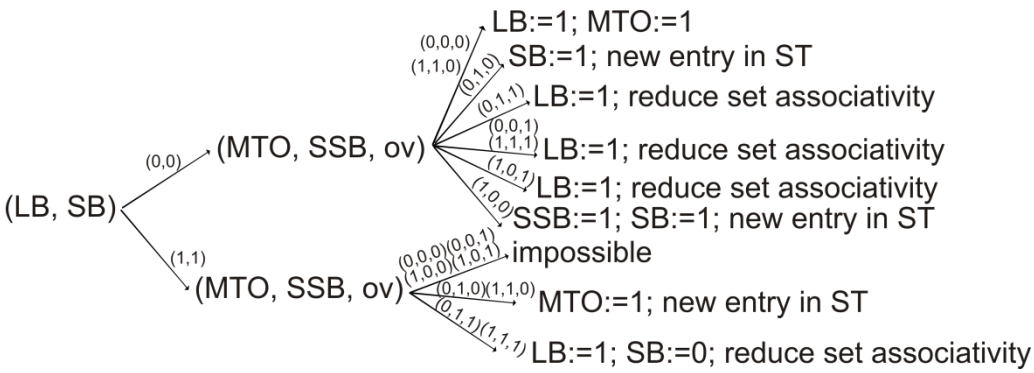

Figure 7.5: Modified SAM algorithm for an access of the cache memory, when a hard error is encountered, from [59].

In order for the algorithm to make a decision, it has to check the value of the (LB, SB) pair. If the value of this pair is (0, 0) the algorithm proceeds to the normal access of the memory location. If the (LB, SB) pair has the value of (0, 1), it

means that we are dealing with a switched cell, as it can be seen from Table 7.1, and therefore we have to perform a search in the switching table in order to find a healthy location to use instead. If the (LB, SB) pair has a value of (1, 0), this indicates a faulty cell that cannot be accessed (a situation that is also present within Table 7.1), and we have to use a healthy cell from the same set instead. If the (LB, SB) pair has the value of (1, 1), this also points to a switched cell (see Table 7.1), but in this case we also have to check the MTO-bit and the SSB. If at least one of these bits is 0 then we have to look for a new cell to be used in the same set; this means that the current cache line cannot be accessed. If both the MTO-bit and the SSB are 1 then we have to access the switching table for a new location to use instead of this one. This last case is quite rare and improbable because this means that the location used to replace a faulty cell becomes faulty itself, for example if the probability for a hard error is *p* then for this case the probability becomes $p^2 \cdot \frac{k-r}{n \cdot k-l}$ $p^2 \cdot \frac{k-r}{n \cdot k-l}$, where *k* is the set associativity, *r* is the number of reductions of the set associativity, *n* number of sets, and *l* total number of hard errors.

The second proposed algorithm is depicted in Figure 7.5 and presents the case when a hard/permanent error is encountered at an access. First we must state that regarding the (LB, SB) pair, there can be only two cases: (0, 0) and (1, 1). The other cases will not be treated the same, because in the case of (0, 1) it means that the cell is already faulty and it becomes redundant to find it as faulty for a second time; and the case of (1, 0) means that the cell is taken out of use, and therefore a new cell in the same set must be accessed instead.

If both LB and SB are 0 we will have the possibilities created according to the (MTO-bit, SSB, overhead) triplet, as presented in Figure 7.5. If this triplet is (0, 0, 0) or (1, 1, 0) then the L-bit of the cache line detected as faulty becomes 1. In the first case the MTO-bit becomes also 1, in the second case the MTO-bit is already 1. If the triplet (MTO-bit, SSB, overhead) is (0, 0, 1) or (1, 1, 1) then the L-bit of the cell becomes 1 and a reduction of the set associativity is performed, as described in [9]. If the (MTO-bit, SSB, overhead) triplet is (0, 1, 0) then the SB of the cache line becomes 1 and a new entry to the switching table is added. If the value of the (MTO-bit, SSB, overhead) triplet is (0, 1, 1) then the L-bit becomes 1 and a reduction of the set associativity is performed, as described in [9]. If the triplet (MTO-bit, SSB, overhead) is (1, 0, 0) then both SB and SSB become 1 and a new entry in the switching table is made. The last case is when the triplet (MTO-bit, SSB, overhead) is (1, 0, 1), therefore the L-bit becomes 1 and a reduction of the set associativity is performed [9].

The other case is when both SB and LB are 1. As in the previous cases, we have to act according to the value of the (MTO-bit, SSB, overhead) triplet. The cases of (MTO-bit, SSB, overhead) being (0, 0, 0), (0, 0, 1), (1, 0, 0) and (1, 0, 1) are not possible. This means that we are left with only four cases, which can be grouped in two parts. If the (MTO-bit, SSB, overhead) triplet is (0, 1, 0) or (1, 1, 0) then the MTO-bit becomes 1 and a new entry in the switching table is added. The last two cases are defined by the values of (1, 1, 1) or (0, 1, 1) for the (MTO-bit,

SSB, overhead) triplet; in this situation SB becomes 0, and the set associativity is reduced [9].

A description of each state that can be encountered in the SAM algorithm is provided in Table 7.2. While Table 7.3 depicts each state that can be encountered in the modified version of SAM alongside with a short description of each of this state.

Table 7.2: Description of each state that can be encountered in the original SAM algorithm

LB	MTO	overhead	Description
0	0	0	Healthy line in a healthy set, no reduction of the set associativity is required
0	0	1	Impossible
0	1	0	Healthy line in a set that contains faulty or switched lines, no reduction of the set associativity is required
0	1	1	Healthy line in a set that contains faulty or switched lines, the reduction of the set associativity is required
1	0	0	Impossible
1	0	1	Impossible
1	1	0	Faulty or switched line, no reduction of the set associativity is required
1	1	1	Faulty or switched line, the reduction of the set associativity is required

Table 7.3: Description of each state that can be encountered in the modified SAM algorithm

LB	SB	MTO	SSB	overhead	Description
0	0	0	0	0	Healthy line in a healthy set, no reduction of the set associativity is required
0	0	0	0	1	Impossible
0	0	0	1	0	Healthy line in a switched set, i.e. there is at least one switched in that set
0	0	0	1	1	Healthy line in a switched set, the reduction of the set associativity is required
0	0	1	0	0	Healthy line in a faulty set, i.e. there is at least one faulty line in that set, no reduction of the set associativity is required
0	0	1	0	1	Healthy line in a faulty set, the reduction of the set associativity is required
0	0	1	1	0	Healthy line in a set with a double switched cell, no reduction of the set associativity is required
0	0	1	1	1	Healthy line in a set with a double switched cell, the reduction of the set associativity is required
0	1	0	0	0	Impossible
0	1	0	0	1	Impossible
0	1	0	1	0	Impossible
0	1	0	1	1	Impossible

0	1	1	0	0	Switched line in a faulty set, no reduction of the set associativity is required, this line is faulty and replaced by a healthy one
0	1	1	0	1	Switched line in a faulty set, the reduction of the set associativity is required, this line is faulty and replaced by a healthy one
0	1	1	1	0	Switched line in a double switched set, no reduction of the set associativity is requires
0	1	1	1	1	Switched line in a double switched set, the reduction of the set associativity is requires
1	0	0	0	0	Impossible
1	0	0	0	1	Impossible
1	0	0	1	0	Impossible
1	0	0	1	1	Impossible
1	0	1	0	0	Faulty line in a faulty set, no reduction of the set associativity is requires
1	0	1	0	1	Faulty line in a faulty set, the reduction of the set associativity is requires
1	0	1	1	0	Faulty line in a double switched set, no reduction of the set associativity is requires
1	0	1	1	1	Faulty line in a double switched set, the reduction of the set associativity is requires
1	1	0	0	0	Impossible
1	1	0	0	1	Impossible
1	1	0	1	0	Switched line in a switched set, no reduction of the set associativity is required
1	1	0	1	1	Switched line in a switched set, the reduction of the set associativity is required
1	1	1	0	0	Impossible
1	1	1	0	1	Impossible
1	1	1	1	0	Switched line in a double switched set, no reduction of the set associativity is required
1	1	1	1	1	Switched line in a double switched set, no reduction of the set associativity is required

7.1.4 Advantages of Using Switching Bits

This subsection presents the basic theoretical advantages and gains from the use of the switching bits. First of all, even though it seems a paradox, the introduction of the switching bits decrease the area overhead with over 35%, this is achieved by the modification of the algorithm presented throughout section 7.1. Another advantage is the huge increase in performance; this is also due to the modification of the algorithm by adding the switching bits. The increase in performance can be of over 75%. Also for the first $n/2$ hard error the probability of adding a new entry in the switching table decreases significantly from the previous version of SAM.

Each of these improvements is endorsed by theoretical and simulation results that are presented in Sections 7.1.5 and 7.1.6.

7.1.5 Theoretical results

We start with some important remarks. First, we note that the value of the L-bit before adding the switching bits is given by the logical OR between the L-bit after adding the switching bits and the switching bit, as in (7.1). Second, the value of the MTO-bit before the switching bits were added is also a logical OR between the MTO-bit after adding the switching bits and the set switching bit, see (7.2).

$$LB_{after}\ OR\ SB\ =\ LB_{before} \tag{7.1}$$

$$MTO-bit_{after}\ OR\ SSB\ =\ MTO-bit_{before} \tag{7.2}$$

Taking all these aspects into consideration, we will now analyze the number of entries added in the switching table, before and after the adding of the switching bits. Before adding the switching bits there were two cases to deal with when a new location was added in the switching table. As Table 7.4 summarizes, after the switching bits are added, the number of cases to deal with becomes four.

Even though the number of cases where a new location is added to the switching table becomes bigger after adding the switching bits, according to the observations made at the start of this subsection, the number of entries in the switching table actually decreases. This is because case 1B, from Table 7.4, includes cases 1A and 2A, and case 2B, from Table 7.4, includes cases 3A and 4A. Table 7.5 presents the cases for which, after adding the switching bits, a new entry in the switching table is not required.

Because of the random distribution of errors in the memory cell array, one cannot determine the exact amount gained in terms of locations in the switching table. In order to provide an estimate we will resort to some probabilistic computations. Table 7.6 presents the probabilities of the SAM method situations without the switching bits, while Table 7.7 will present the probabilities of the SAM method after adding the switching bits. Note that, in order to save space, we have separated the probabilities in Table 7.6 and Table 7.7 into three parts, and – in order to get the overall probability – we just need to multiply the probabilities from the three parts. Throughout Table 7.6 and Table 7.7 the following notations have been used: n for the number of sets, k for the number of lines per set, l as the total number of errors, r as the number of reductions of the set associativity, x as the number of faulty lines since the last reduction of the set associativity, ST_{total} as the total number of entries in the switching table, and ST_i as the number of entries in the switching table since the last reduction of the set associativity.

Table 7.4: Cases for new entries in the switching table, from [59].

	(LB, MTO-bit, overhead)		(LB, SB, MTO, SSB, ov.)
1B	(0, 1, 0)	1A	(0, 0, 0, 1, 0)
2B	(1, x, x)	2A	(0, 0, 1, 0, 0)
		3A	(1, 1, 0, 1, 0)
		4A	(1, 1, 1, 1, 0)

Table 7.5: Cases for new entries in switching table, from [59].

LB	SB	MTO-bit	SSB	ov.
0	0	1	1	0
1	0	0	0	x
0	1	0	0	x
1	1	0	0	x
0	1	(0, 1, 1)	(1, 0, 1)	1
1	0	(0, 1, 1)	(1, 0, 1)	1
1	1	(0, 1, 1)	(1, 0, 1)	1
0	1	(0, 1, 1)	(1, 0, 1)	0
1	0	(0, 1, 1)	(1, 0, 1)	0
1	1	1	0	0

The ratio between the probabilities of having an entry in the switching table before and after introducing the switching bits is given in (7.3. These equations are deducted from the changes that were made in the algorithm presented throughout the sections from 7.1.1 to 7.1.4.

$$\frac{after}{before} = \frac{ST_i \cdot ST_{total} + (nk - l - ST_{total})\left(l - nr - \frac{ST_i^2}{n \cdot k}\right)}{nk(l - rn)} \tag{7.3}$$

The ratio between the accesses in the switching table before and after adding the switching bits is presented in (7.4.

$$\frac{after}{before} = \frac{ST_{total}}{l + ST_{total}} \cdot \left(1 + \frac{ST_i^2}{n^2 \cdot k}\right) \tag{7.4}$$

7.1.6 Simulation results

The simulations results are based on the probabilistic computations presented in section IV.A and are applied to the same cache memory as in [9] and [6]. The cache memory is a 2MB, 8-way set associative, with the block size of 256B.

The overhead added by the introduction of the switching bits is small for the above-described cache memory; the area overhead is of less than 1.2 KB, which means 0.05% of the total memory size. But even if we add these bits, the overall overhead decreases because of the reduction of the size required by the switching table. This happens because, due to this extra logic, some of the locations will not require a new entry in the switching table. The corresponding ratio is presented in (7.3. On the other hand, Figure 7.6 presents the comparative analysis of the probability of adding a new location in the Switching Table before and after the switching bits were added, which translates in overhead improvement.

Even though the overhead gains are modest, the performance gains become quite notable, a reduction of the switching table accesses of over 75%. This happens because the access in the switching table will not be made every time an L-bit has the value 1. The switching table will be accessed in two instances. The first

one, which is the most probable, is when the pair (LB, SB) has the value of (0, 1). The second case, and this has a very low weight with regards to the first case, if the quadruple (LB, SB, MTO-bit, SSB) has the value (1, 1, 1, 1), which means that we have to deal with a location that is switched more than once.

Table 7.6: Probabilities for regular SAM

LB	probability	MTO	probability	ov.	prob
0	$\frac{n \cdot k - l - ST_{total}}{n \cdot k}$	0	$\frac{n(r+1)-l}{n}$	0	$\frac{n-1}{n}$
1	$\frac{l + ST_{total}}{n \cdot k}$	1	$\frac{l - n \cdot r}{n}$	1	$\frac{1}{n}$

Table 7.7: Probabilities for SAM with switching bits

LB	SB	probability	MTO	SSB	probability	ov	prob
0	0	$\frac{n \cdot k - l - ST_{total}}{n \cdot k}$	0	0	$\frac{n(r+1)-l}{n}$	0	$\frac{n-1}{n}$
0	1	$\frac{ST_{total}}{n \cdot k}$	0	1	$\frac{ST_i}{n} \cdot \frac{n \cdot k - ST_i}{n \cdot k}$		
1	0	$\frac{l - ST_{total}}{n \cdot k}$	1	0	$\frac{x}{n}$	1	$\frac{1}{n}$
1	1	$\frac{ST_{total}}{n \cdot k}$	1	1	$\frac{ST_i}{n} \cdot \frac{ST_i}{n \cdot k}$		

The gain in terms of performance is presented in Figure 7.7. In Figure 7.7 the performance is plotted in terms of accesses in the switching table for the SAM method, with and without the switching bits. Figure 7.8 presents the gains both in terms of overhead and performance percentagewise.

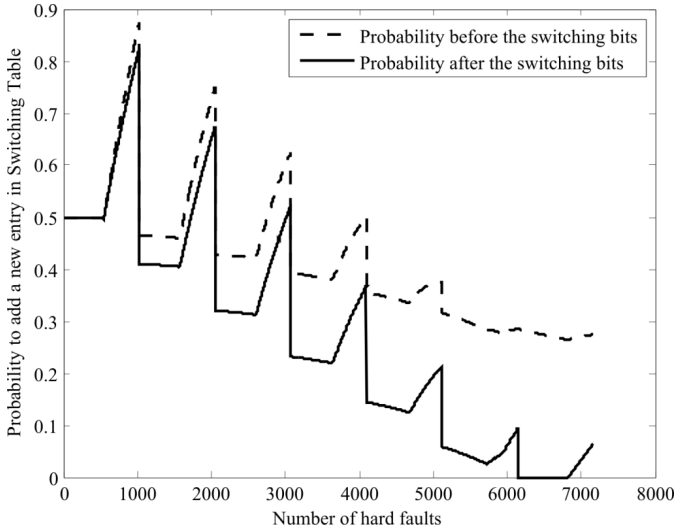

Figure 7.6: Probability for new entry in switching table, from [59].

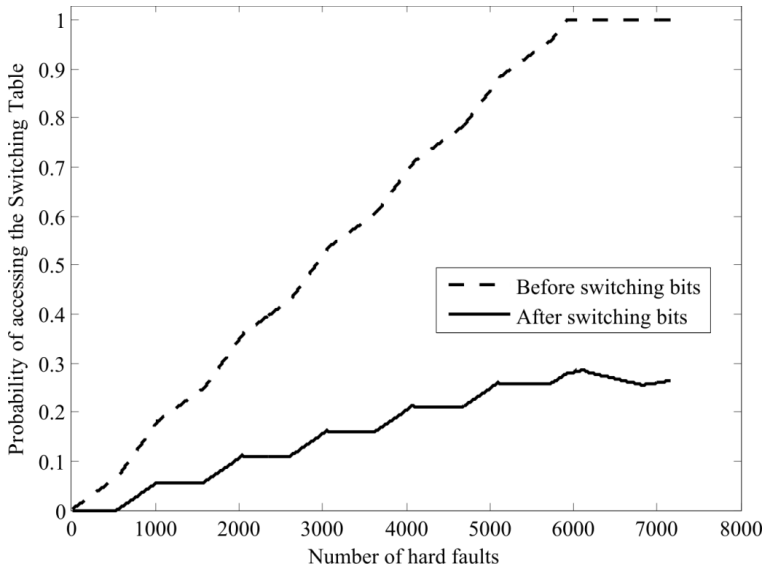

Figure 7.7: Probability for accessing the switching table, from [59].

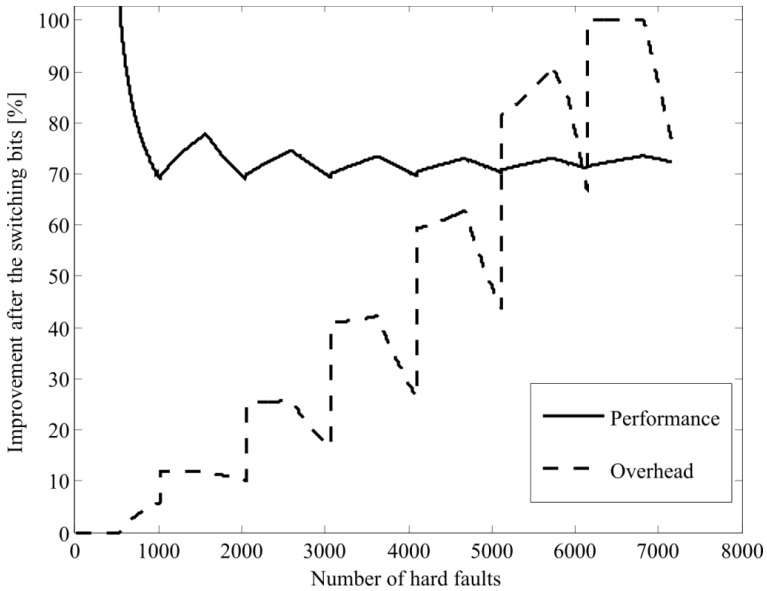

Figure 7.8: Improvement from original SAM, from [59].

7.1.7 Conclusions

The first goal of this first section of this chapter was to perform an analysis of the SAM method [9], in terms of overhead and performance. This analysis was performed throughout sections 7.1.1, 7.1.2, and 7.1.3. As a conclusion to our analysis, we have observed that the previous version of SAM has some

shortcomings, of which the main one was the large loss in performance (every time the L-bit is 1 an access in the switching table is required).

The second goal of this first section of this chapter was to introduce the switching bits to the SAM mechanism in order to improve the performance cost of this method. We have succeeded to also improve the overhead added by the switching table, the overall area overhead actually decreasing with over 35%; this figure is supported by theoretical calculations and simulations, for more details see Sections 7.1.5 and 7.1.6 and Figure 7.6. The performance gains on the other hand are really significant. We have a mean of over 75% reduction in the number of accesses in the switching table, as it can be rendered by examining Figure 7.8 and Figure 7.7, and for some cases they can be completely eliminated, as seen from section 7.1.4. The cases for which the accesses are most probable to disappear are the first $n/2$ hard errors that appear in the cache memory, where n represents the number of sets in the cache memory.

7.2 Methods for Reducing the Switching Table

This section proposes an analysis for the Self Adaptive cache Memory (SAM) mechanism, in the context of employing a set of improvements aimed at decreasing the size of SAM's switching table. This objective is achieved by eliminating some of the switching table redundant/idle entries, which generate unnecessary performance degradation and unnecessary increase of the area overhead. We also present a comparative analysis for the SAM method with and without these improvements, in terms of overhead reduction and performance increase. The simulation results have shown that the number of entries in the switching table can be reduced with up to 68%. Simulation also reveals that the time penalty can be reduced by over 80%. At the same time, we describe how SAM can also be used for yield improvement [60].

New entries in the switching table are created every time a remapping is necessary. In this context, the switching table is never searched for idle or redundant entries. This complicates the switching table and reduces its performance by increasing its access time.

From here on, for a more suitable description we will refer to a cell's address not by its physical location, but as a pair made of its set and line numbers. For example, the address of line 2 in set 0 will be given as (0, 2). We call an *idle entry*, a switching table entry that is never used and which only occupies space. We call a *redundant entry*, an entry that is not compulsory, and therefore consumes both time and space. For example, if location (1,0) is remapped to (3,1) and then location (3,1) is remapped to location (4,3), then (3,1) becomes redundant. If we have location (2,3) remapped to location (4,7), and afterwards location (2,3) becomes faulty, then location (2,3) is no longer necessary; this is the case of an idle location in the switching table.

In this section, we will discuss three cases where the contents of the switching table are changed. There are other cases for which the reduction of the

switching table is possible, as the SAM method is developed right now, the ones presented here will produce the best results in terms of interventions over gains.

Each of these three cases of switching table modifications will be accompanied by a description and an example of its use.

a. First case: the healthy cell in the second column becomes faulty. This case produces redundant entries.
b. The second case: reduction of the set associativity. This case produces idle entries.
c. The third case: another location in the same set with the healthy cell becomes faulty. This produces redundant entries.

Note that in all the examples provided throughout this section the cache lines depicted in dark grey are the ones that have been faulty before the modification of the switching table was necessary. On the other hand, those depicted in light grey are the last ones to become faulty and require the modifications in the switching table accordingly.

7.2.1 First case

For case *a.* from section 7.2 we have to search the switching table every time a new faulty cell appears in the memory, in order to see if it is located in the healthy part of the table. If this is not the case, applying the standard algorithm suffices. If it is found, instead of adding a new entry in the switching table, we can modify the existing entry and simply mark this cell as being faulty afterwards. This will lead to discarding a redundant entry from the switching table. An example of this case is illustrated in Figure 7.9, while the algorithm used in case *a* is depicted in Figure 7.10.

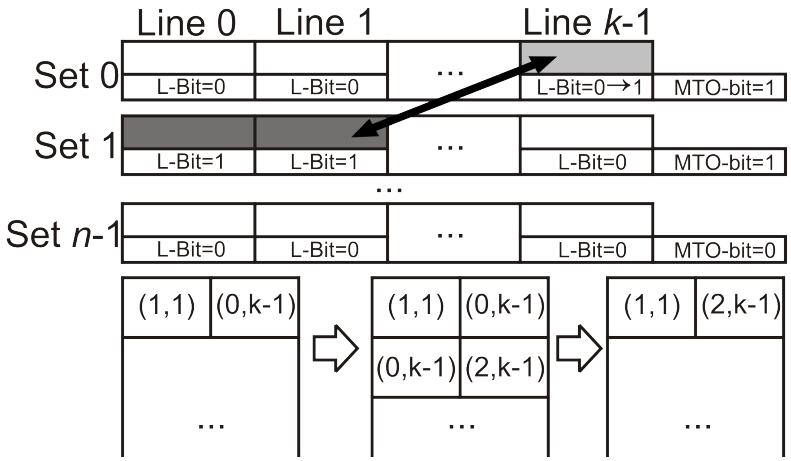

Figure 7.9: First case example, from [60].

```
if (new fault at line j in set i) {
        t= – 1;
        for (l=0; l<Switching Table size; l++) {
                if ((i,j) in Switching Table Healthy part) {
                        t=l;
                        exit loop;
                }
        if (t != – 1)
                modify entry t's healthy part from Switching Table to
                a location in first set with MTO =0;
}
```

Figure 7.10: First case algorithm, from [60].

7.2.2 Second case

For case *b.,* referring to the reduction of the set associativity of the cache memory, which is quite rare (once every *n* hard faults), we can look in the switching table and reorganize it by removing the newly formed idle locations, that will never be accessed again. In the following, we will present an example of the advantages brought by this reorganization of the switching table. To this end, we will need an extra counter to inform on how many reductions of the set associativity have been performed so far, including the current one. The algorithm required for reorganizing the switching table is depicted in Figure 7.12, together with a corresponding example, Figure 7.11.

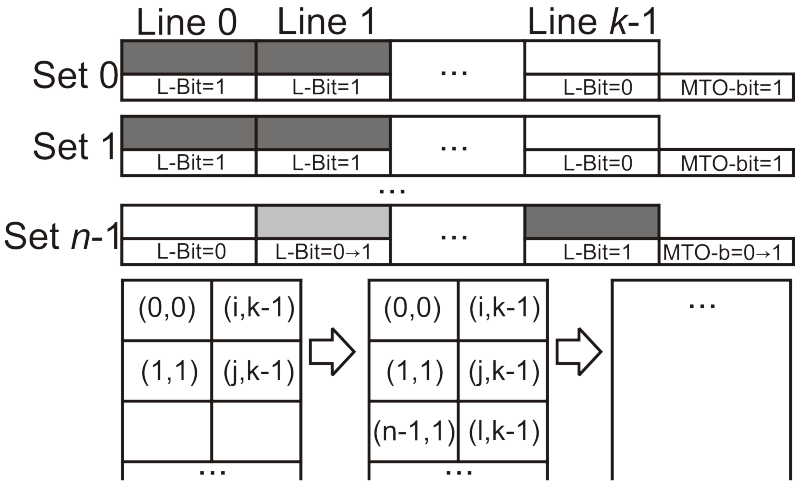

Figure 7.11: Second case example, from [60].

```
if (overflow_counter_n) {
      for (i=0; i<n; i++) {
      counter = 0;
            for (j=0; j<k; j++)
                  if ((i,j) in Switching Table) counter ++;
            if (counter <= counter_k)
                  remove all entries in ST with i in the faulty column;
            else
                  remove k entries from the ST that have i in
                  the faulty column
      }
}
```

Figure 7.12: Second case algorithm, from [60].

7.2.3 Third case

Case *c.,* when a different location from the same set as the healthy cell becomes faulty is the simplest of the three. This is because when we encounter a faulty cell that needs a new entry in the switching table, we only have to look in the table to see if there is an entry in the healthy column part of the switching table that belongs to the same set with the currently detected faulty cell. If such an entry exists, we simply modify that entry with another one from a healthy set. The healthy set is chosen according to the principles stated in [9] as the first set that has the MTO-bit 0, where the line is the last one that is available in that set. An example of this case is depicted in Figure 7.13, along with the employed algorithm in Figure 7.14.

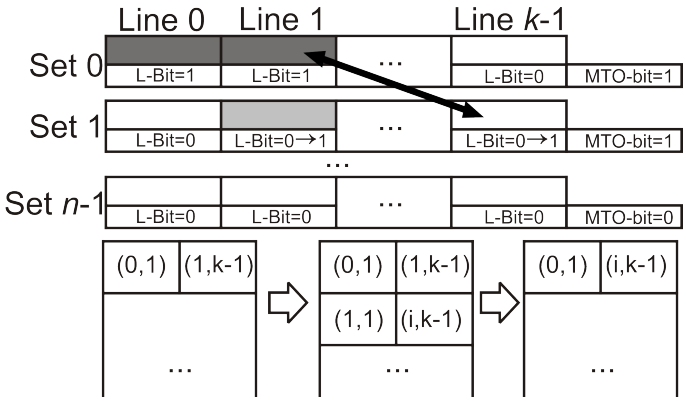

Figure 7.13: Third case example, from [60].

```
if (entry in Switching Table of line j in set i) {
        counter=0;
        ok=0;
        t= – 1;
        for (l=0; j<Switching Table size; l++) {
                if (t == – 1)
                        t=l;
                counter++;
        }
        if ((counter – counter_k)>0)
                modify entry t's healthy part from Switching Table to
                a location in first set with MTO =0;
}
```

Figure 7.14: Third case algorithm, from [60].

7.2.4 Improvements

Table 7.8 provides the number of locations in the switching table, determined probabilistically, as it was performed in [9]. In order to present the analysis results, we consider a L2-cache memory of 2MB capacity, 8-way set associative with the block size of 256B, same as the ones described in [9] and [6].

Note that case *a.* from section 7.2 refers only to a cell in a set, and case *c.* from section 7.2 refers to a set without a cell. From this, it becomes obvious that the proportion between their corresponding gains will be of degree *k*, where *k* is the set associativity left of the cache memory (i.e. the gain for case *c.* is *k* time more probable then the gain from the case *a.*). This discrepancy in gain holds true both in the number of switching table entries and in speed.

Table 7.8: Results obtained using the described improvements, from [60].

k	8	7	6	5	4	3	2
Number of faulty locations	1024	2048	3072	4096	5120	6144	7168
Locations needed in Switching Table before improvements	448	997	1500	1941	2325	2667	2923
Locations needed in Switching Table after improvements	444	981	1423	1668	1667	1399	835
Difference in number of locations	4	16	77	273	658	1268	2088
Reduction of time penalty [%]	38.5	39	41.2	46.7	55.5	67.5	82.3

Mean time penalty per access before improvements	0.06τ	0.16τ	0.29τ	0.47τ	0.75τ	1.30τ	2.85τ
Mean time penalty per access after improvements	0.06τ	0.15τ	0.27τ	0.40τ	0.54τ	0.68τ	0.85τ

7.2.5 Overhead Gains

As presented in Section 7.2, in each of the three cases we can reduce the number of entries in the switching table. The first and third cases have the potential of reducing the switching table by one entry at a time. The second case has a superior potential of reducing the number of entries, as presented in this subsection.

In order to be able to determine the overhead improvement, we will resort to probabilistic computations, using a method that is similar to that from [9]. To this end, we will look at the memory as being split in two parts: a part that contains sets with faulty cache lines and another part that contains only healthy sets. In order to simplify the computations, we consider that an error can occur anywhere in the memory with the same probability. We need this partitioning of the cache memory in order to determine the most probable distribution of the faults in the cache lines and sets. Therefore, in order to have the most probable distribution of faults, we need to have an equal or almost equal probability of fault occurrence within one of the two partitions. In that respect, the two partitions of the cache memory need to have the same number of healthy cells (or, at least, to differ with no more than one).

After writing the equations, we obtain that after *l* errors in the cache memory we have *x* sets with faulty lines, as in (7.5.

$$x = \frac{n\,k + l}{2\,k} \tag{7.5}$$

In (7.5, *n* is the number of sets in the cache memory, *k* is the number of lines per set, or the set associativity and *l* is the number of errors encountered so far. An example of how the L2-cache memory, which was described above, is most probable to look after 2048=2*n* errors is illustrated in Figure 7.15. After applying (7.5 to this cache memory, we obtain the results depicted in Table 7.8. Figure 7.16 illustrates a comparison in terms of overhead between the original SAM [9] and its improved version, which is described in this subsection.

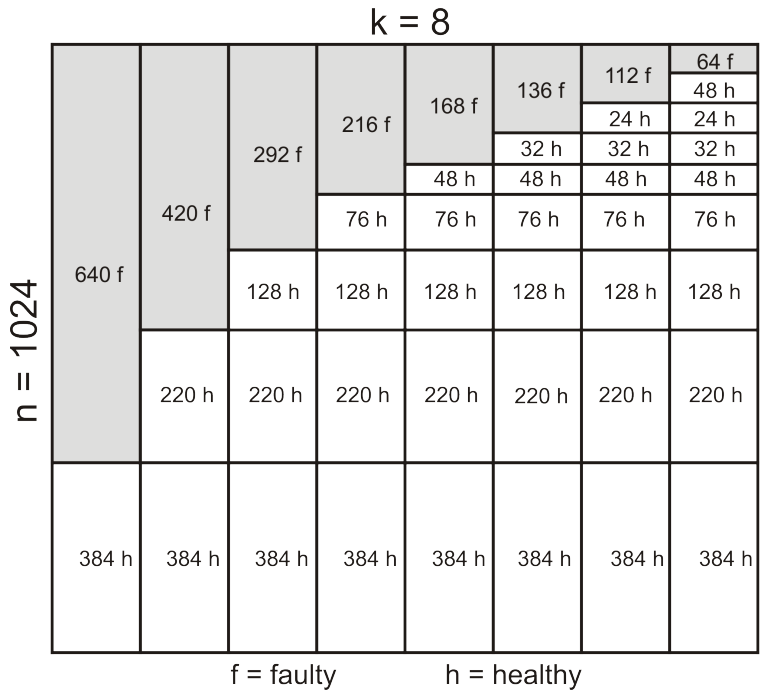

Figure 7.15: Example of fault distribution, from [60].

7.2.6 Performance Gains

The gain in performance will be obtained from the redundant cases (first and third), because the number of accesses in the switching table will be reduced, thus speeding-up the memory access. The gain from the second case can be translated into performance gain by achieving overall reduction of switching table size. This reduction of table size will translate into faster table access, thus reducing the time penalty of every switching table access, as presented in Table 7.8. Table 7.8 contains the results obtained by simulations for the same fault distribution as in the previous subsection. In these computations, we take into consideration both the lost time due to the increased algorithm complexity, and the speed gain generated by the size reduction of the switching table. Without losing generality, for our evaluation purposes we consider that the reduction in access time is proportional with the size of switching table reduction.

$$\text{Time\_penalty}_{\text{before}} = ank\tau\xi \tag{7.6}$$

$$\text{Time\_penalty}_{\text{after}} = a'nk\tau'\xi \tag{7.7}$$

(7.6 shows the time penalty of the switching table before improving SAM, while (7.7 illustrates it afterwards. In these expressions a and a' are the number of

entries in the switching table before and after respectively, n the number of sets, k the number of lines per set, τ and τ' are the access times of the switching table before and after the improvements respectively, and ξ is the mean number of accesses in the cache memory between finding two consecutive faults.

The difference in performance can be proven as being even bigger, because for simpler simulations we ignored some gains obtained from our improvements, like the size of the switching table at all times (i.e. we have only taken into consideration its final size).

Table 7.8 presents the reduction of time penalties as percentages. It also shows that the time penalty reduction obtained by the modifications of the switching table is up to over 80%. Figure 7.17 illustrates the two time penalties; before and after our improvements; due to the reduction of the switching table size, the improvements can be observed even from the first errors.

In Figure 7.18 we have summarized the improvements brought by our method percentage-wise, both in terms of overhead and performance. As it can be seen in Figure 7.18, the performance improvement varies from 37% to over 80%, while the improvement in the number of switching table required locations can reach a maximum of 68%.

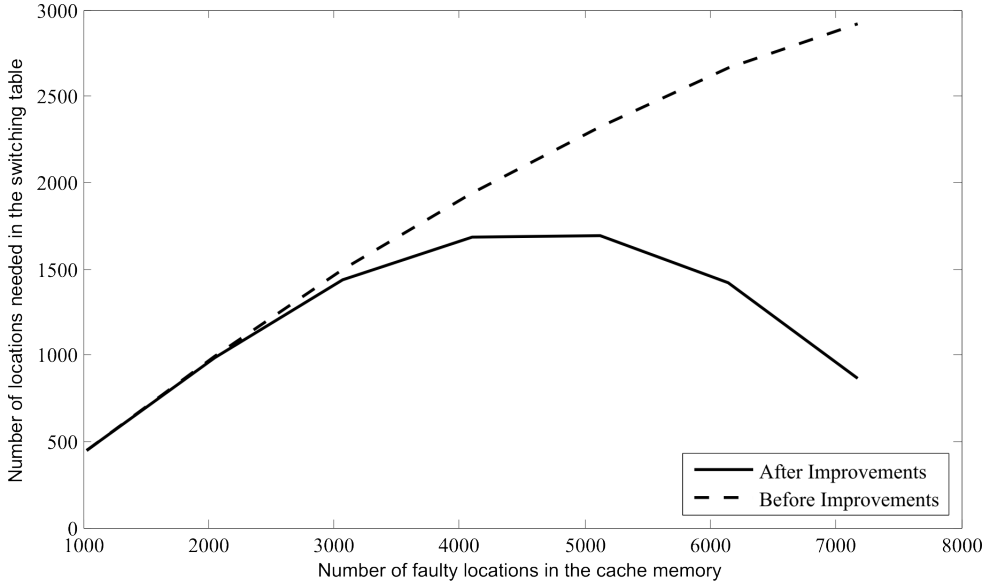

Figure 7.16: Overhead improvement, from [60].

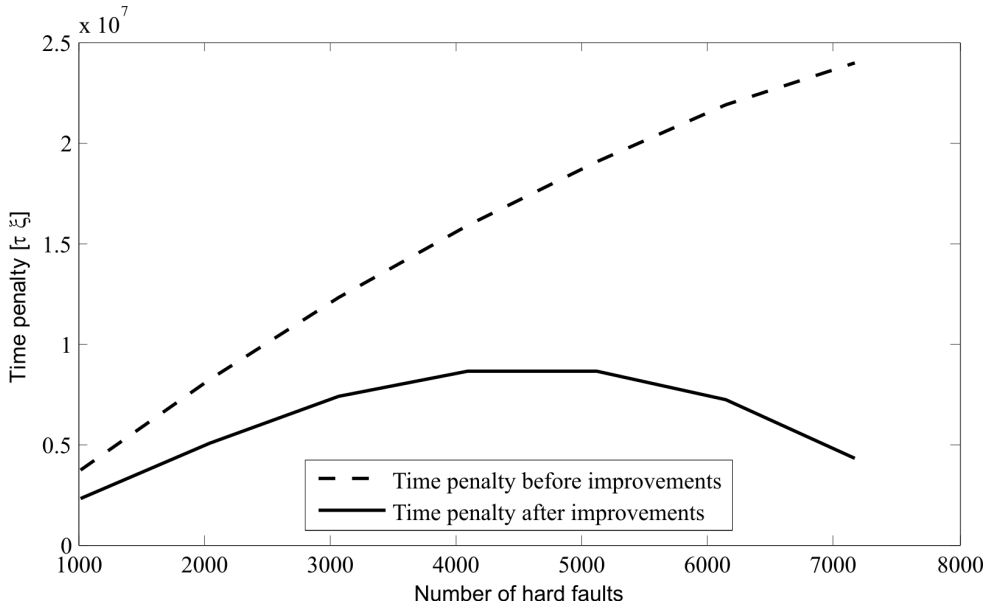

Figure 7.17: Performance improvement, from [60].

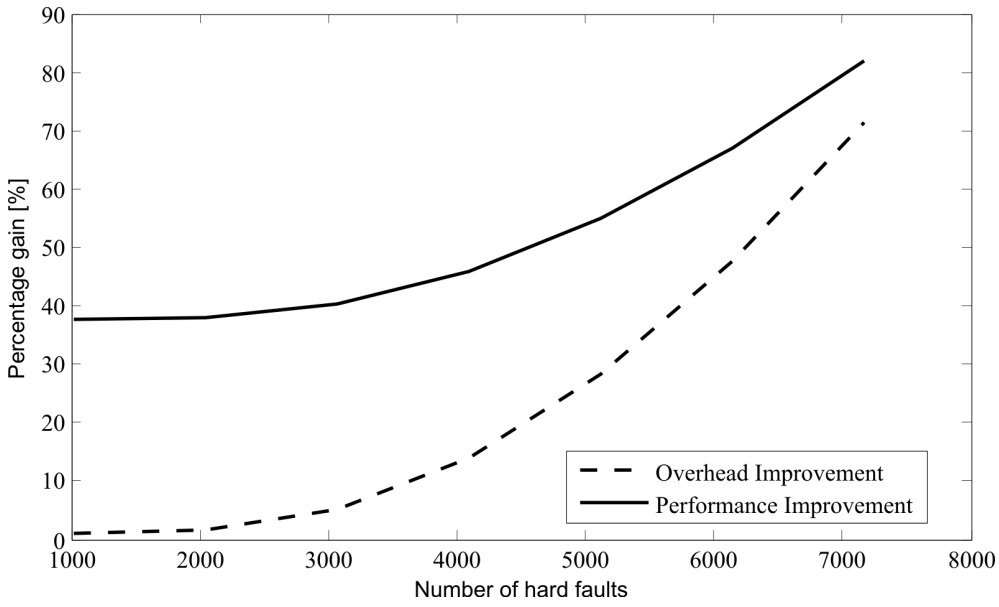

Figure 7.18: Improvements obtained, from [60].

7.2.7 Using SAM for Yield Improvement

Another useful feature of the SAM method consists of improving the chip yield. In order to be able to use SAM for this purpose, the method can be

maintained as it is and can be run before the manufacturer delivers the chip, or it can be simplified by reducing the size of the switching table. We present an analysis for using a reduced version of SAM in order to deliver a better chip yield.

Due to the fact that errors in a chip tend to cluster [61] a method for yield improvement like the one proposed in [1] for direct mapped caches is not very efficient. The method from [1] is also based on the principles of graceful degradation, and relies on the cache memory architecture in order to replace faulty cache blocks with their neighbors. The neighbors are selected as blocks that are physically located on the same row as the faulty block. Because of the fault clustering, there is a higher probability for the neighbors of a healthy block to become faulty themselves. The method proposed in [1] resembles SAM in that it also uses an extra bit (like SAM's L-bit) for the identification of the faulty block.

By using the switching table, SAM avoids relying on the neighbors of a faulty cache block, as it can be seen in chapter 5's presentation. The only problem of the SAM method with respect to reliability is the use of the MTO-bit and the switching table, which are unprotected and thus susceptible to errors. A 4-way set associative, 64kB cache memory, with a block size of 32 Bytes would be similar to the one described in [1], with the exception that it is a set associative cache memory. Moreover, although the memory from [1] uses direct mapping, it provides four blocks in a row in order to be used for remapping, thus resembling a set-associative organization. The SAM algorithm with a switching table capable of sustaining 1536 faulty blocks (75% of the whole memory) introduces an overhead of 10062 bits, which represent less than 1.92% of the size of the cache memory.

One potential disadvantage of using the SAM method is the vulnerability to errors of the L-bits, MTO-bits and Switching Table, because they have no redundancy support. This can be corrected by the use of even a triple modular redundancy for the vulnerable elements, with a total area overhead of under 6%, which will further improve their reliability. We will discuss this further in the next chapter. This disadvantage is not particular to SAM: every chip that has an integrated BIST with no self-testing capabilities has the same vulnerability.

The advantages of using SAM for Yield improvement instead of that presented in [1] are:

- Knowledge regarding the physical architecture is not required in order to implement the method
- SAM can be applied to any physical implementation of a memory chip
- SAM is able to deal with clustering faults
- The worst-case scenario for SAM depends on the switching table size and can be avoided by increasing the size of the switching table. In a worst-case scenario, the method described in [1] can fail after just 4 errors for the above-described memory

As a comparative analysis between these two methods (SAM and [1]), in terms of yield improvement without taking into consideration the logic overhead, we can say, based on [9] and [1], that for a cache memory as the one described in section 7.2.4, the method described in [1] can sustain a maximum of about 800

faulty cache blocks with no added redundancy, while SAM can sustain a number of 1536 faulty blocks (an almost double amount). The only potential drawback consists of introducing the switching table that is susceptible to errors.

By reducing the number of switching locations, we can still maintain a high yield and decrease the area that is vulnerable to errors. For the memory described in section 7.2.5, as can be seen by inspecting Figure 7.16, if we limit the number of locations in the switching table to 1000 we can assure that even in the presence of 2000 faulty blocks the cache is still functional. This is, of course, not the worst-case scenario, because we still have a probability that the faults that will appear afterwards can still be mapped; this way, the number of supported faulty blocks can be further increased. The number of faults is taken according to a uniform fault distribution within the memory, which usually provides a lower yield for a mathematical analysis [61].

7.2.8 Conclusions

The main contribution of this section consists of introducing three methods for reducing the negative impact of the switching table for the Self Adaptive Cache Memory (SAM) method, in terms of overhead and performance. We change the switching table when encountering one of the following three cases: if the healthy cell in the second column becomes faulty, if there is a reduction of the set associativity, and if a faulty cell appears in the same set as a location from the second column of the switching table.

The simulation results have shown that the number of entries can be reduced up to 68%, with an actual reduction of the switching table size of over 37%. Accordingly, we have achieved an improvement in both the size and speed of the switching table. With regard to performance gains and reduction of time penalty introduced by the switching table, we have shown that we can reduce the time penalty with up to over 80% in comparison with the SAM version presented in [9].

The reliability improvement of SAM with these modifications of the switching table remains the same as in [9]. As described in [58] for a memory without SAM the cache system reliability is $R=1-p$, where p is the probability of a faulty block. Whereas if the SAM mechanism is added, the reliability becomes $R=1-p^{(k-1)\cdot n+1}$ [9], where n is the number of sets in the cache memory, and k is the numbers of lines per set.

Also we have shown that the application of SAM is not limited to the increase of reliability of a cache memory, it can also be used to increase the yield of the memory chips.

8 Simplified Selective Fault Tolerance Technique for Protection of Selected Inputs via Triple Modular Redundancy Systems

In this chapter we will present a novel method for reducing the overhead of Triple Modular Redundancy technique by protecting selected inputs, also we will show how to apply this new method to the SAM technique described throughout chapters 5 and 7.

We will start with a presentation of our own method, which is a modified version of the Selective Fault Tolerance method and achieves substantial area reduction over the state of the art. The simulation results show that we achieved an improvement of up to over 20% in terms of area and energy overhead, compared with the state of the art. Also compared to a classic TMR we obtain improvements of up to 65%, with a mean improvement of 25% in terms of area and energy reduction.

8.1 Introduction

A solution for the increase in hardware faults is the use of triple modular redundancy (TMR) techniques [62]. As the name suggests this technique for fault tolerance relies on three identical modules that are used at the same time and the result of the three modules is passed to a voter, which decides the correct result that will be passed on. Due to the fact that at each moment instead of one system we have three modules running we have an area and energy overhead that is more than 300%.

As a consequence there has been a lot of research in order to reduce the area overhead and power consumption [63] [64] [65]. As a result of this research there emerged a technique that uses TMR in order to protect a reduced set of inputs, called Selective Fault Tolerance [10] [11]. Throughout this chapter we will provide the achieved improvements in terms of area and energy overhead compared to the state of the art.

In this chapter we will also provide a practical use of our simplified selective fault tolerance method applied to set associative cache memories, along with the results in terms of reduction of the area and energy overhead.

8.2 Triple Modular Redundancy

The basic idea for Triple Modular Redundancy is illustrated in Figure 8.1. The three modules from Figure 8.1 (module 1, module 2, module 3) are identical, while

the role of the voter is to select the correct output of the circuit [12]. So if all three modules have the same output then the output of the circuit would be that output, and if two of then have the same output then the output of the circuit will be that output. We can conclude that the triple modular redundancy would fail only when two of the three modules will fail for the same inputs.

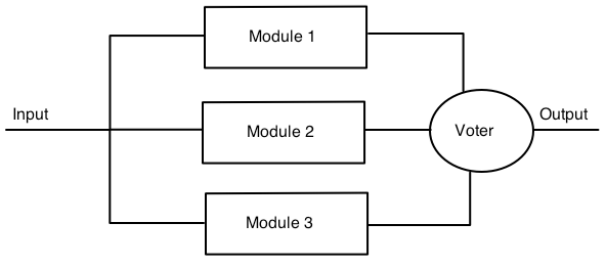

Figure 8.1: Triple Modular Redundancy elements, from [66].

The reliability of a TMR system is expressed in Equation (8.1), where RM is the reliability of a module [12].

$$\mathrm{R} = \mathrm{R}_{\mathrm{M}}^{3} + 3\mathrm{R}_{\mathrm{M}}^{2}\left(1 - \mathrm{R}_{\mathrm{M}}\right) = 3\mathrm{R}_{\mathrm{M}}^{2} - 2\mathrm{R}_{\mathrm{M}}^{3} \tag{8.1}$$

The overhead of the TMR system can be computed as in Equation (8.2) for area overhead and Equation (8.3) for energy overhead.

$$AreaOverhead\ [\%] = \frac{TMR_{area}}{Module_{area}} \cdot 100 \tag{8.2}$$

$$EnergyOverhead\ [\%] = \frac{TMR_{energy}}{Module_{energy}} \cdot 100 \tag{8.3}$$

It can be inferred from Equation (8.2) and Equation (8.3) that both area and energy overhead are 300%, without taking into consideration the voter and the links between the modules. These overheads are justified only if the reliability of that system is paramount. For everyday systems other techniques are used. In the next section we present such a technique that has as starting point TMR.

8.3 Selective Fault Tolerance

In this section we will provide a short description of the technique presented in [10] and [11], called Selective Fault Tolerance (SFT).

This method is a variation of the classic TMR, in which instead of protection all the inputs of a system it protects only a restricted set, provided by the designer of the system. This restricted set is called a critical set [10].

All the inputs in the critical set are protected by the TMR technique and have the same reliability as the TMR while the other inputs may or may not be protected by other techniques, we will consider throughout this chapter that the rest of the inputs will not be protected.

We will use the following notations: all the possible inputs are denoted by the set X, while the critical inputs are denoted by the set X_c. The relation between X and X_c is denoted in Equation (8.4).

$$X_c \subseteq X \tag{8.4}$$

If we have a combinational circuit S and we want to protect this circuit with selective fault tolerance, as described in [10], we will need the followings. First we will need a combinational circuit identical to S, we will refer to this circuit as S_1, and after this we will need two smaller circuits s_2 and s_3. The s_2 and s_3 circuits are designed with these restrictions: for a critical input these circuits generate the same output as the S_1 circuit, while for any other input at least one of the s_2 and s_3 has the same output as the S_1 circuit [10]. The concept for this can be summarized in Equation (8.5) and Equation (8.6), from [10].

$$S(x) = S_1(x) = s_2(x) = s_3(x), \qquad if\ x \in X_c \tag{8.5}$$

$$(S_1(x) = s_2(x)) \vee (S_1(x) = s_3(x)), if\ x \notin Xc \tag{8.6}$$

The inclusion of an input x in the critical set X_c can be determined by the use of a characteristic function χ [10], with this function being described in Equation (8.7).

$$\chi(x) = 1, if\ x \in X_c \quad else\ \chi(x) = 0 \tag{8.7}$$

Figure 8.2 shows the basic concept of selective fault tolerance as presented in [10], while Figure 8.3 shows the modified concept from [11], which takes into account the characteristic function in order to select the right output from the multiplexor.

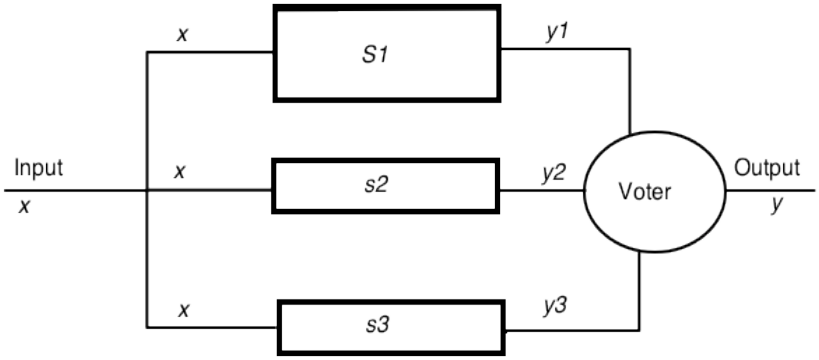

Figure 8.2: Selective Fault Tolerance with Input Detection, based on the Triple Modular Redundancy mechanism, from [66].

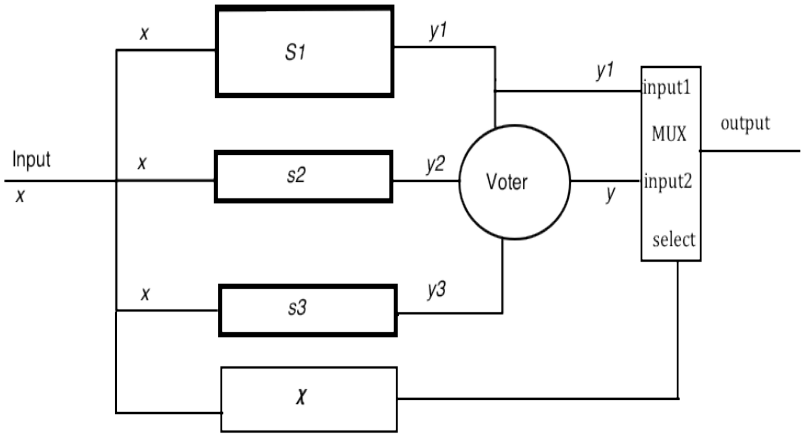

Figure 8.3: Selective Fault Tolerance with Input Detection, based on the Triple Modular Redundancy mechanism, from [66].

8.4 Simplified Selective Fault Tolerance

The method proposed in this chapter uses the Selective Fault Tolerance as a starting point. For a combinational circuit S with a set of inputs X we will protect a subset X_c of the input set X. In order to do that we will use three combinational circuits: the first one S_1 that is identical with the combinational circuit that we want to protect, and the other two circuits will be c_2 and c_3. The c_2 and c_3 circuits are identical, and are the minimal combinational circuits that, for an input from X_c, have the same output as S_1. Equations (8.8) and (8.9) describe these three circuits.

$$S(x) = S_1(x) = c_2(x) = c_3(x), if\ x \in X_c \tag{8.8}$$

$$(c_2(x) = not\ care) \wedge (c_3(x) = not\ care), if\ x \notin Xc \tag{8.9}$$

Compared with state of the art method described in section 8.3, in order to save area and energy consumption, we use the minimal circuit from s_2, and s_3, and we multiplex its output with the output from S_1. So we will use the minimal circuit between s_2 and s_3, and that circuit will be used as both c_2 and c_3. This allows us not to care what the output for the c_2 and c_3 circuits will be in the case of any other value of x which is not in X_c. Figure 8.4 illustrates our method. The module χ has the same functionality as the one described in section 8.3.

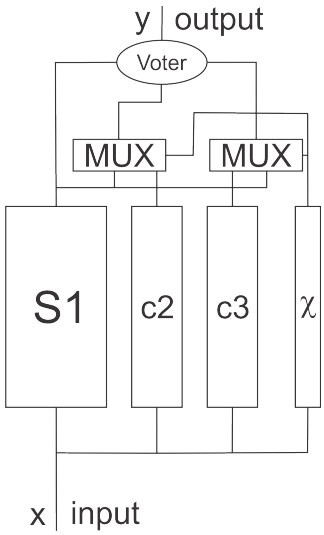

Figure 8.4: Simplified Selective Fault Tolerance, based on the Triple Modular Redundancy mechanism, from [66].

For the Simplified Selective Fault Tolerance technique we have two possible combinations:

- when $x \in X_c$, it is the case when we have inputs protected by triple modular redundancy
- when $x \notin X_c$, it is the case when the output will not be protected

For the first case, when the inputs are protected by TMR, the reliability of the system is the same as the reliability is for a system with TMR, see Equation (8.1). When the input arrives it is passed to the four modules: S_1, c_2, c_3 and χ. The outputs are computed for each of these modules as: for S_1, c_2, c_3 the output is the output of the circuit (faulty or correct), while for the fourth module, χ, the output is '1'. The output of χ will trigger the two multiplexors to select as their output the result from the c_2 and c_3 modules. The output will be compared in the voter, the same way as for the TMR technique.

For the second case, when the inputs are not protected by TMR, the reliability will be the same as for the state of the art method presented in section 8.3, as $R=R_M$. When the input arrives at the four modules the outputs are computed, but in this case the output for the χ module will be '0'. This will cause the multiplexors to select as their exit the output from circuit S_1, while the voter will have three identical inputs, as the output from S_1.

8.5 Simulation Results and SAM Application

In order for our results to be relevant, compared to the state of the art technique, we will use the same simulation environments as in [10] [11]. So we used the benchmark LGSynth91; also in order for implementation of our circuits we

used Verilog for description and Synopsis for synthesis. The results of the simulations are depicted in Table 8.1.

Table 8.1: Benchmarks results (LGSynth91)

Circuit		xor5	TMR	SFT	Our Method	Improvement from TMR [%]	Improvement from SFT [%]
Pins	in	5					
	out	1					
Number of protected inputs [%]	0	area [μm]	387	129	129	66.66	0
	10		387	273	268	30.75	1.83
	20		387	315	304	21.45	3.5
	30		387	387	387	0	0
	40		387	385	374	3.36	2.85
	50		387	364	352	9.04	3.3
	60		387	387	387	0	0
	70		387	387	387	0	0
	80		387	387	387	0	0
	90		387	387	387	0	0
	100		387	387	387	0	0
Circuit		Z9sym	TMR	SFT	Our Method	Improvement from TMR [%]	Improvement from SFT [%]
Pins	in	9					
	out	1					
Number of protected inputs [%]	0	area [μm]	1701	567	567	66.66	0
	10		1701	1034	1018	40.15	1.55
	20		1701	1176	1168	31.33	0.68
	30		1701	1291	1102	35.21	14.64

	40		1701	1383	1286	24.4	7.01
	50		1701	1351	1346	20.9	0.37
	60		1701	1450	1432	15.81	1.24
	70		1701	1515	1378	18.99	9.04
	80		1701	1491	1338	21.34	10.26
	90		1701	1509	1366	19.69	9.48
	100		1701	1701	1701	0	0
Circuit		max46	TMR	SFT	Our Method	Improvement from TMR [%]	Improvement from SFT [%]
Pins	in	9					
	out	1					
Number of protected inputs [%]	0	area $[\mu m]$	3984	1328	1328	66.66	0
	10		3984	1695	1687	57.66	0.47
	20		3984	1980	1961	50.78	0.96
	30		3984	2311	2281	42.75	1.3
	40		3984	2644	2639	33.76	0.19
	50		3984	2725	2637	33.81	3.23
	60		3984	3193	3193	19.85	0
	70		3984	3545	3445	13.53	2.82
	80		3984	3570	3535	11.27	0.98
	90		3984	3835	3795	4.74	1.04
	100		3984	3984	3984	0	0
Circuit		t481	TMR	SFT	Our Method	Improvement from TMR [%]	Improvement from SFT [%]
Pins	in	16					
	out	1					

Number of protected inputs [%]	0	area $[\mu m]$	3171	1057	1057	66.66	0
	10		3171	1647	1625	48.74	1.34
	20		3171	2065	1941	38.79	6
	30		3171	2508	2193	30.84	12.56
	40		3171	2389	2311	27.12	3.27
	50		3171	2463	2218	30.05	9.94
	60		3171	2481	1953	38.41	21.28
	70		3171	2467	1929	39.16	21.81
	80		3171	2846	2795	11.86	1.79
	90		3171	3041	3039	4.16	0.07
	100		3171	3171	3171	0	0
Circuit		parity	TMR	SFT	Our Method	Improvement from TMR [%]	Improvement from SFT [%]
Pins	in	16					
	out	1					
Number of protected inputs [%]	0	area $[\mu m]$	35082	11694	11694	66.66	0
	10		35082	16932	16910	51.8	0.13
	20		35082	17768	17564	50	1.15
	30		35082	21209	21136	39.75	0.34
	40		35082	24030	23934	31.78	0.4
	50		35082	25041	24075	31.38	3.86
	60		35082	27782	27664	21.14	0.42
	70		35082	29608	29608	15.6	0
	80		35082	31639	31620	9.87	0.06
	90		35082	33602	33308	5.06	0.88
	100		35082	35082	35082	0	0

Figure 8.5 and Figure 8.6 illustrate the reduction in area overhead, as follows: Figure 8.5 represents the comparison between our method and the TMR technique in terms of area overhead reduction, while Figure 8.6 represents the

comparison between our method and the state of the art method described in [10] [11], in terms of area overhead.

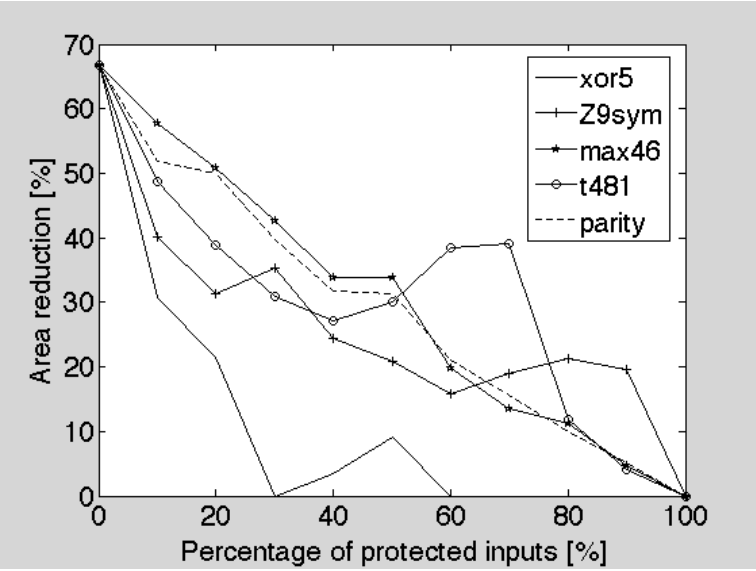

Figure 8.5: Area overhead reduction of Simplified Selective Fault Tolerance, compared to TMR, from [66].

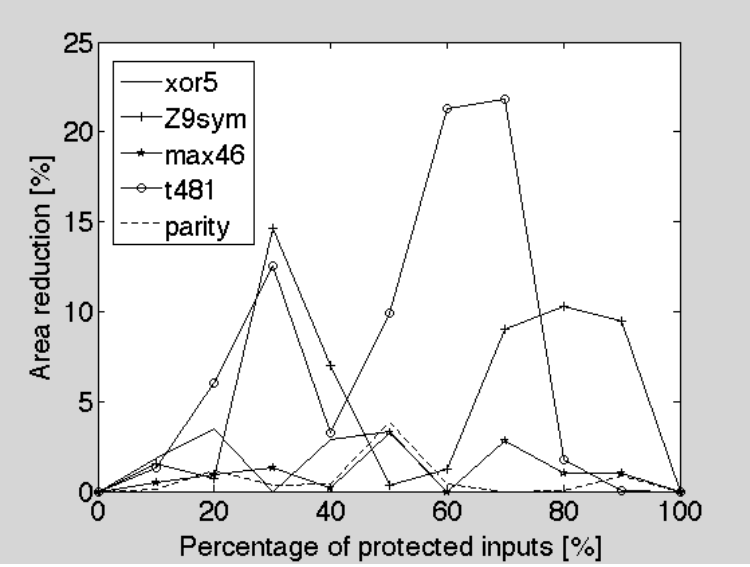

Figure 8.6: Area overhead reduction of Simplified Selective Fault Tolerance, compared to Selective Fault Tolerance, from [66].

The simulations have shown that we can achieve an improvement up to over 20% compared with the state of the art SFT technique. These improvements are both in terms of area overhead and energy consumption. The energy consumption

can be furthermore improved by powering down the c_2 and c_3 modules while the inputs are not from the protected set, but this will lead to some small performance degradation, and it will not be discussed further.

In the reminder of this chapter we will discuss how to apply this method in order to increase the reliability of the SAM method presented throughout chapters 5 and 7.

We remind our readers that the reliability of the SAM method as described in chapters 5 and 7 is very high compared to a standard memory. The reliability of SAM is shown in Equation (8.10), while the reliability of a standard memory is depicted in Equation (8.11).

$$R_{SAM} = 1 - p^{(k-1)\cdot n+1} \tag{8.10}$$

$$R_{Mem} = 1 - p \tag{8.11}$$

In Equation (8.10) and Equation (8.11), p represents the probability of having a faulty cell; k represents the set associativity of the cache memory, i.e. the number of lines in a set, while n represents the number of sets in the cache memory.

Equation (8.10), which refers to SAM reliability is actually only taking into account the memory cell array, without considering the Switching Table, which will lead to a decrease in reliability because is totally unprotected. The next section shows how to apply simplified selective fault tolerance to a memory that has a SAM mechanism, compared with a memory that has implemented a TMR mechanism for the whole memory.

8.6 Appling Simplified Selective Fault Tolerance to SAM

We will apply our simplified selective fault tolerance to a k-way set associative cache memory, with n sets that has a SAM mechanism. This simplified selective fault tolerance will be applied only for the Switching Table, and not for the whole memory as in classic TMR. In the following we will compute the new reliability of the SAM memory in Equation (8.12), and the reliability for the same cache memory with a TMR technique in Equation (8.13), while Equation (8.14) represents the comparison of the two reliabilities.

$$R_{SAM} = \frac{\left(1 - p^{(k-1)\cdot n+1}\right)\cdot nk + R_{ST}\cdot f\cdot nk}{nk(1+f)} \tag{8.12}$$

$$R_{TMR} = 3R_{Mem}^2 - 2R_{Mem}^3 \tag{8.13}$$

$$\frac{R_{SAM}}{R_{TMR}} = \frac{1}{1+f}\left(\underbrace{\frac{\left(1 - p^{(k-1)n+1}\right)}{1 - 3p^2 + 2p^3}}_{>1} + f\right) > 1 \tag{8.14}$$

In Equations (8.12)-(8.14) p represents the probability of having a faulty cell; k represents the set associativity of the cache memory, while n represents the number of sets in the cache memory. In Equation (8.12) R_{ST} is the reliability for the

switching table, and it is equal to R_{TMR} because is protected by the simplified selective fault tolerance technique, and *f* represents the overhead of the Switching Table. In our case we will consider f as the upper boundary of the overhead, which is 1%, so *f*=0.01.

As it can be observed from Equations (8.12)-(8.14) the reliability for SAM with simplified selective fault tolerance is greater than the reliability of the TMR alone. We have performed a simulation varying *p* from 0.01 to 0.015 and compared the results for a 2MB 8-way set associative cache memory with a block size of 256B, as the one described in [9]. The results for reliability are simulated for both TMR and SAM with implemented simplified selective fault tolerance and is plotted in Figure 8.7 on a logarithmic scale for the *x*-axis; this is done in order to better observe the behavior of the two reliabilities. This simulation confirms the theoretical results presented from Equations (8.12) (8.13) and (8.14).

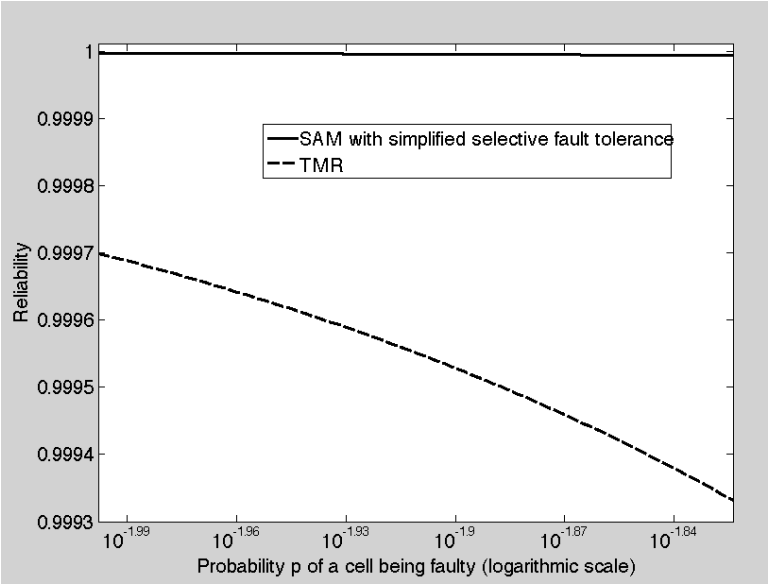

Figure 8.7: Reliability comparison between TMR for a cache memory and SAM with simplified selective fault tolerance, from [66].

Equation (8.15) describes the computation of the total area of the SAM method and (8.16) describes computations needed in order to find the total area of the memory protected via TMR.

$$A_{SAM} = A_{Mem} + 3 \cdot A_{ST} = (1 + 3 \cdot f) \cdot A_{Mem} \tag{8.15}$$

$$A_{TMR} = 3A_{Memory} \tag{8.16}$$

As can be seen from Equations (8.15) and (8.16) the reduction in terms of area overhead depends on *f*. For SAM *f* is under 1%. Therefore, when using the simplified selective fault tolerance, this leads to an improvement in area overhead of 65.66% compared to a conventional TMR. All these while increasing the overall

reliability of the cache memory in comparison with TMR. At the same time, we note that, in digital design, the improvement in area overhead translates to a similar decrease in energy consumption.

8.7 Conclusions

Throughout this chapter we have presented an improvement for the state of the art method called Selective Fault Tolerance [10]. This method proposes the use of Triple Modular Redundancy technique only for a restricted set of inputs of a combinational circuit. Our method called Simplified Selective Fault Tolerance reduces the area overhead and energy consumption of the state of the art by up to over 20%. Also we have compared our method to Triple Modular Redundancy and have obtained an improvement in area and energy overhead of up to 65% in the best-case scenario, and have obtained a mean improvement in both area and energy compared to TMR of over 25%.

In order to obtain these results we have used LGSynth91 benchmark, Verilog for description, and Synopsis for synthesis. At the same time we have shown that while reducing the area overhead we maintain the reliability the same as the state of the art method presented in section 8.3.

In this chapter we have also shown how to use our method in conjuncture with the graceful degradation technique for cache memories, described in chapters 5 and 7 in order to improve both reliability and decrease area overhead for set associative cache memories. These results have been proven both by theoretical approaches and by simulations.

For our future work we intend to implement an efficient methodology for reducing the energy consumption of the simplified selective fault tolerance method by powering down the extra modules, while they are not in use. We want to perform simulation and compute the time penalty associated with this powering down of the module and analyze if this tradeoff between performance and energy consumption is feasible.

9 Conclusions and Future Work

Throughout this thesis we have provided our readers with the basic notions of cache memories, faults, errors and failures, fault tolerance techniques, and memory specific tests and fault tolerant processes. We have also made a widespread analysis of the state of the art regarding fault tolerance techniques for cache memories, which was the field in which we brought our contributions.

The original contribution of this thesis was presented throughout chapters 5, 6, 7, and 8. We will outline here the major results and contributions of this thesis. First of all we have developed a new mechanism in order to improve the reliability of set associative cache memories. This new mechanism is called Self Adaptive cache Memories (SAM) and has as a starting point the graceful degradation technique. By adding SAM to a set associative cache memory, alongside with a build-in self-test (BIST), the cache memory can tolerate a large number of faults (both hard and soft errors). For example an 8 way set associative cache memory with a total number of 8196 cache lines, can tolerate, by implementing SAM, up to 7168 faulty lines; these lines having been diagnosed as hard errors. In order to obtain these results the SAM mechanism makes use of a Switching Table, which is used for remapping the faulty locations in order to maintain performance and correctness of the cache memory.

The second original contribution to the field of cache reliability of this thesis was creating a mechanism that can improve the performance of the original SAM method. This improvement is trifold: reduction of the area overhead, reduction of energy consumption, and increase in performance. We have proposed two methods that can be used one at a time, or combined, in order to obtain these improvements. The improvements are obtained, mainly through a more efficient management of the Switching Table.

Since the only part that was left vulnerable to errors was the Switching Table, we have adapted and improved a technique that uses Triple Modular Redundancy to protect a critical set of inputs. So the third original contribution was to improve the state of the art for selective fault tolerance techniques, which use triple modular redundancy. Also we have applied the newly developed simplified selective fault tolerance technique to a cache memory that has the SAM mechanism. So instead of protecting the whole cache memory with a triple modular redundancy technique, which implied an overhead both in area and energy of over 300%, we have applied the simplified selective fault tolerance technique only to the Switching Table. The overhead thus obtained is around 1% instead of 300%; also because we used SAM the reliability of the cache memory was improved, compared to a cache memory that uses only triple modular redundancy.

The third original contribution of this thesis was to adapt notions from probability theory and apply them in the field of memory testing. We have developed an original algorithm that can be applied to any type of memory (not only

cache memories). This algorithm provides the most probable distribution of errors inside a memory system after a number of errors. Of note: the number of errors can be chosen from zero to the total number of locations in the memory.

All of these results have been published throughout the PhD program at various conferences [9] [59] [60] [66] [67].

For future work we plan to furthermore study the field of cache memory reliability. We will look at new ways to improve the already developed mechanisms and technique, and also we plan to develop some new methods and schemes that can improve even more the reliability of the cache memory.

The reliability of the cache memory is of paramount importance. This is because in a computer system the memory hierarchy is responsible for more than 50% of faults. Also every bit that is processed has to pass through the cache memory, and this is why the reliability of this cache memory is so important.

10 Bibliography

[1] A. Agarwal, B.C. Paul, H. Mahmoodi, A. Datta, and K. Roy, "A process-tolerant cache architecture for improved yield in nanoscale technologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 1, pp. 27 - 38, January 2005.

[2] A. Sasan, H. Homayoun, A. Eltawil, and F. Kurdahi, "Process Variation Aware SRAM/Cache for Aggressive Voltage-Frequency Scaling," in *Design, Automation & Test in Europe Conference & Exhibition*, Nice, 2009, pp. 911 - 916.

[3] S. Ramaswamy and S. Yalamanchili, "Customizable Fault Tolerant Caches for Embedded Processors," in *International Conference on Computer Design*, San Jose, CA, 2006, pp. 108 - 113.

[4] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers, "The impact of technology scaling on lifetime reliability," in *International Conference on Dependable Systems and Networks*, Florence, 2004, pp. 177-186.

[5] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach 5th Edition*, Morgan Kaufmann, Ed. St. Louis, MO, USA: Elsevier, 2012.

[6] Lee Hyunjin, Cho Sangyeun, and Bruce R. Childers, "Performance of Graceful Degradation for Cache Faults," in *IEEE Computer Society Annual Symposium on VLSI*, Porto Alegre, 2007, pp. 409 - 415.

[7] Lee Hyunjin, Cho Sangyeun, and Bruce R. Childers, "Exploring the interplay of yield, area, and performance in processor caches," in *25th International Conference on Computer Design*, Lake Tahoe, CA, 2007, pp. 216 - 223.

[8] Premkishore Shivakumar, S.W. Keckler, C.R. Moore, and D. Burger, "Exploiting microarchitectural redundancy for defect tolerance," in *21st International Conference on Computer Design*, San Jose, CA, 2003, pp. 481 - 488.

[9] Liviu Agnola, Mircea Vladutiu, and Mihai Udrescu, "Self-Adaptive mechanism for cache memory reliability improvement," in *IEEE 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, Vienna, 2010, pp. 117 - 118.

[10] M. Augustin, M. Gossel, and R. Kraemer, "Implementation of Selective Fault Tolerance with Conventional Synthesis Tools," in *IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, Cottbus, Germany, 2011, pp. 213-218.

[11] M. Augustin, M. Gossel, and R. Kraemer, "Reducing the Area Overhead of TMR-Systems by Protecting Specific Signals," in *IEEE 16th International On-Line Testing Symposium*, Los Alamitos, CA, USA, 2010, pp. 268-273.

[12] R. E. Lyons and W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," *IBM Journal of Research and Development*, vol. 6, no. 6, pp. 200- 209, April 1962.

[13] Algirdas Avizienis, Jean-Claude Laprie, Brian Randel, and Carl Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11-33, January-March 2004.

[14] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. San Francisco, CA, USA: Elsevier, Morgan Kaufamann, 2007.

[15] Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2001.

[16] A. J. van de Goor, *Testing semiconductor memories: theory and practice*. New York, USA: John Wiley & Sons, Inc., 1991.

[17] Qaulity Concepts and Terminology, part 1: Generic Terms and Definitions, 1992, Document ISO/TC 176/SC 1 N93, February 1992.

[18] Industrial-Process Measurements and Control - Evaluation of System Properties for the Purpose of System Assesment, Part 5: Assessment of System Dependability, 1992, Draft, Publication 1069-5, International Electronical Commission (IEC) Secretariat, February 1992.

[19] M.C. Paulk, B. Curtis, M.B. Chrissis, and C.V. Weber, "Capability maturity model, version 1.1," Carnegie Mellon University, Pittsburgh, PA, Technical CMU/SEI-93-TR-24, ESC-TR-93-177, 1993.

[20] R. Chillarege et al., "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943 - 956, November 1992.

[21] Algirdas Avižienis, "Design of fault-tolerant computers," in *Proceedings of the November 14-16, 1967, fall joint computer conference AFIPS Joint Computer Conferences*, Anaheim, CA, 1967, pp. 733-743.

[22] A. Fox and D. Patterson, "Self-Repairing Computers," *Scientific American*, vol. 288, no. 6, pp. 54-61, June 2003.

[23] M. Marinescu, "Simple and Efficient Algorithms for Functional RAM Testing," in *IEEE Test Conference*, Philadelphia, 1982, pp. 236-239.

[24] R. Nair, S. M. Thatte, and J. A. Abraham, "Efficient Algorithms for Testing Semiconductor Random-Access Memories," *IEEE Transactions on Computers*, vol. 27, no. 6, pp. 572-576, June 1978.

[25] D.S. Suk and S.M. Reddy, "A March Test for Functional Faults in Semiconductor Random Access Memories," *IEEE Transactions on Computers*, vol. 30, no. 12, pp. 982 - 985, December 1981.

[26] C. A. Papachristou and N. B. Sahgal, "An Improved Method for Detecting Functional Faults in Semiconductor Random Access Memories," *IEEE*

Transactions on Computers, vol. 34, no. 2, pp. 110-116, February 1985.

[27] Magdy S. Abadir and Hassan K. Reghbati, "Functional Testing of Semiconductor Random Access Memories," *ACM Computing Surveys*, vol. 15, no. 3, pp. 175 - 198, September 1983.

[28] M.A. Breuer and A.D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, 1st ed. Maryland, USA: Computer Science Press, 1976.

[29] R. Nair, "Comments on "An Optimal Algorithm for Testing Stuck-at Faults in Random Access Memories"," *IEEE Transactions on Computers*, vol. 28, no. 3, pp. 258-261, March 1979.

[30] G. Gordon and H. Nadig, "Hexadecimal Signatures Identify Troublespots in Microprocessor Systems," *Electronics*, vol. 50, no. 5, pp. 89-96, March 1977.

[31] R. A. Frohwerk, "Signature Analysis: A New Digital Field Service Method," *Hewlett-Packard Journal*, vol. 28, no. 9, pp. 2-8, May 1977.

[32] W. W. Peterson and E. J. Weldon, *Error-Correcting Codes*. New York, USA: John Wiley & Sons, 1972.

[33] S. W. Golomb, *Shift Register Sequences*. Laguna Hills, CA, USA: Aegean Park Press, 1982.

[34] Michael L. Bushnell and Vishwani D. Agrawal, *Essentials of Electronic Testing for Digital, Memory & Mixed-Signal VLSI Circuits*. New York, NY, USA: Springer, 2000.

[35] V. D. Agrawal et al., BIST at Your Fingertips Handbook, June, 1987, AT&T.

[36] V. D. Agrawal, C.R. Kime, and K. K. Saluja, "A Tutorial on Built-In Self-Test, Part 1: Principles," *IEEE Design & Test of Computers*, vol. 10, no. 1, pp. 73-82, March 1993.

[37] V.D. Agrawal, C. R. Kime, and K. K. Saluja, "A Tutorial on Built-In Self-Test, Part 2: Applications," *IEEE Design & Test of Computers*, vol. 10, no. 2, pp. 69-77, June 1993.

[38] Daniel P. Siewiorek and Robert S. Swarz, *Reliable Computer Systems Design and Evaluation*, 3rd ed. Natick, MA, United States of America: A K Peters, 1998.

[39] R. Kraus, O. Kowarik, K. Hoffmann, and D. Oberle, "Design for Test of Mbit DRAMs," in *Proceeding of the International Test Conference*, Washington DC, USA, August, 1989, pp. 316-321.

[40] Katsuki, David Elsam, Eric S. Mann, and F. William, "Pluribus-An Operational Fault-Tolerant Multiprocessor," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1146-1159, October 1978.

[41] D.P. Siewiorek, V. Kini, H. Mashburn, S. McConnel, and M. Tsao, "A case study of C.mmp, Cm*, and C.vmp: Part I—Experiences with fault tolerance in multiprocessor systems," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1178 - 1199, October 1978.

[42] D.P. Siewiorek, V. Kini, R. Joobbani, and H. Bellis, "A case study of C.mmp, Cm*, and C.vmp: Part II—Predicting and calibrating reliability of multiprocessor systems," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1200 - 1220, October 1978.

[43] Alla R. Alameldeen, Zeshan Chishti, Chris Wilkerson, Wei Wu, and Shih-Lien Lu, "Adaptive Cache Design to Enable Reliable Low-Voltage Operation," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 50-63, January 2011.

[44] M.Y. Hsiao, D. C. Bossen, and R. T. Chien, "Orthogonal Latin Square Codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 390-394, July 1970.

[45] S. Paul, Fang Cai, Xinmiao Zhang, and S. Bhunia, "Reliability-Driven ECC Allocation for Multiple Bit Error Resilience in Processor Cache," *IEEE Transaction on Computers*, vol. 60, no. 1, pp. 20-34, January 2011.

[46] M. Mutyam et al., "Process-Variation-Aware Adaptive Cache Architecture and Management ," *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 865-877, July 2009.

[47] S. Lin and D. Costello, *Error Control Coding, second edition*.: Prentice Hall, 2004.

[48] M. Mutyam, Feng Wang, R. Krishnan, V. Narayanan, and M. Kandemir, "Process-Variation-Aware Adaptive Cache Architecture and Management," *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 865-877, July 2009.

[49] A. Agarwal, B.C. Paul, S. Mukhopadhyay, and K. Roy, "Process variation in embedded memories: failure analysis and variation aware architecture," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 9, pp. 1804-1814, September 2005.

[50] Chen Qikai, Mahmoodi Hamid, Bhunia Swarup, and Roy Kaushik, "Modeling and Testing of SRAM for New Failure Mechanisms Due to Process Variations in Nanoscale CMOS," in *23rd IEEE VLSI Test Symposium (VTS'05)*, Palm Spring, CA, USA, 2005, pp. 292-297.

[51] W. Zhang, "Replication cache: a small fully associative cache to improve data cache reliability," *IEEE Transactions on Computers*, vol. 54, no. 12, pp. 1547-1555, December 2005.

[52] Shuai Wang, Jie Hu, and Sotirios G. Ziavras, "On the Characterization and Optimization of On-Chip Cache Reliability against Soft Errors," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1171-1184, September 2009.

[53] S. Nakahara, K. Higeta, M. Kohno, T. Kawamura, and K. Kakitani, "Built-in self-test for GHz embedded SRAMs using flexible pattern generator and new repair algorithm," in *Proceedings of the International Test Conference*, Atlantic City, NJ , USA, 1999, pp. 301-310.

[54] M.H. Tehranipour, Z. Navabi, and S.M. Fakhraie, "An efficient BIST method for testing of embedded SRAMs," in *The 2001 IEEE International Symposium on Circuits and Systems*, Sydney, NSW , Australia , 2001, pp. 73-76.

[55] D. Weiss, J.J. Wuu, and V. Chin, "The on-chip 3-MB subarray-based third-level cache on an Itanium microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1523-1529, November 2002.

[56] H. Miyatake, M. Tanaka, and Y. Mori, "A design for high-speed low-power CMOS fully parallel content-addressable memory macros," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 6, pp. 956-968, June 2001.

[57] J. P. Grossman, "A systolic array for implementing LRU replacement," MIT-AI Aries Group Technical Memos, Memo 2002.

[58] Martin L. Shooman, *Reliability of Computer Systems and Networks: Fault Tolerance,Analysis,and Design*. New York, Unites States of America: John Wiley & Sons, 2002.

[59] Liviu Agnola, Mircea Vladutiu, Mihai Udrescu, and Lucian Prodan, "Improving performance of robust Self Adaptive Caches by optimizing the switching algorithm," in *IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, Cottbus, 2011, pp. 297-300.

[60] Liviu Agnola, Mircea Vladutiu, and Mihai Udrescu, "Increasing Performance and Decreasing Overhead for Self Adaptive Cache Memories," in *IEEE 17th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, Timisoara, 2011, pp. 173-178.

[61] I. Koren and Z. Koren, "Defect tolerance in VLSI circuits: techniques and yield analysis," *Proceeding of the IEEE*, vol. 86, no. 9, pp. 1819-1838, September 1998.

[62] She Xiaoxuan and P.K. Samudrala, "Selective Triple Modular Redundancy for Single Event Upset (SEU) Mitigation," in *NASA/ESA Conference on Adaptive Hardware and Systems*, Los Alamitos, CA, USA, 2009, pp. 344- 350.

[63] K. Mohanram and N. A. Touba, "Partial Error Masking to Reduce Soft Error Failure Rate in Logic Circuits," in *IEEE 18th International Symposium on Defect and Fault Tolerance in VLSI Systems*, Washington, DC, USA, 2003, pp. 433-440.

[64] O. Ruano, J. A. Maestro, and P. Reviriego, "Validation and Optimization of TMR Protections for Circuits in Radiation Environments," in *IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, Cottbus, Germany, 2011, pp. 399-400.

[65] H. Samrow et al., "Functional Enhancements of TMR for Power Efficient and Error Resilient ASIC Designs," in *IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, Cottbus, Germany, 2011, pp. 183-188.

[66] Liviu Agnola, Mircea Vladutiu, Mihai Udrescu, and Lucian Prodan, "Simplified Selective Fault Tolerance Technique for Protection of Selected Inputs via Triple Modular Redundancy Systems," in *IEEE 7th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, Timisoara,

Romania, 2012.

[67] Liviu Agnola and Ileana Ioana Cofaru, "Electronic Infrastructure Aimed at Improving Communication and Education for Business and Academia," in *International Conference on Engineering & Business Education, Innovation and Entrepreneurship (BRCEE)*, Sibiu, Romania, 2012, pp. 197-200.

[68] Shuai Wang, Jie Hu, and Sotirios G. Ziavras, "On the Characterization and Optimization of On-Chip Cache Reliability against Soft Errors," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1171-1184, September 2009.

[69] Lucian Prodan, Mihai Udrescu, and Mircea Vladutiu, "Self-Repairing Embryonic Memory Arrays," in *2004 NASA/DoD Conference on Evolution Hardware*, Seattle, Washington, USA, 2004, p. 130.

[70] P. H Bardell, W. H. McAnney, and J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques*. New York, NY, USA: John Wiley & Sons, 1987.

[71] M. Nicolaidis, "An Efficient Built-In Self-Test Scheme for Functional Test of Embedded RAMs," in *Proceeding of the IEEE Fault Tolerant Computer Systems Conference*, Ann Arbor, MI, USA, 1985, pp. 118-123.