Improvements and Enhancements for Self Adaptive Cache Memories

PhD Student: Liviu Agnola

PhD Advisor: Prof. Dr. Ing. Mircea Vladutiu

Second PhD Report

PhD Student: Liviu Agnola

PhD Advisor: Prof. Dr. Ing. Mircea Vladutiu

Table of Contents

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 an increase in capacity, the probability of hard, permanent faults increases, especially in SRAM cells [1]. Due to this fact the usual method: 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].

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.

We 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 this benefits will be at only a small cost in performance, due to the fact that it is a case of graceful degradation [5] [6] [7]. 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 [8]. 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 [8].

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 [9]. 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 [9]. 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 [9]. 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 [9]. A service failure is a transition from correct service to incorrect service [9]. A service outage is the period of delivering an incorrect service, a service restoration is the transition from incorrect service to a correct service [9].

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 [9]. 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 [9]. 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 [9].

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 [9]. The specification may identify several such modes, for example: limited service, slow service, emergency service, and others [9].

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.

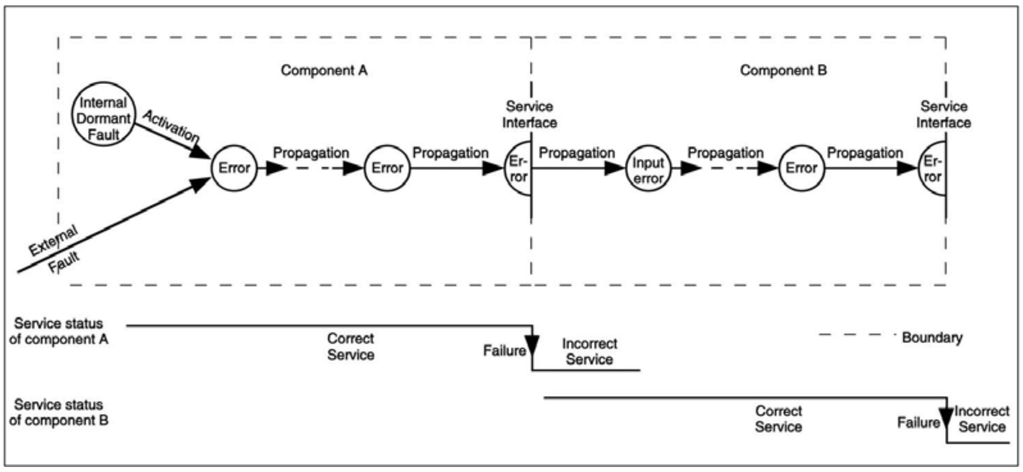


Figure 2.1: Error propagation, from [9].

$$\begin{tikzcd} \cdots \arrow[r] & \text{fault} \arrow[r, "activation"] & \text{error} \arrow[r, "propagation"] & \text{failure} \arrow[r, "causation"] & \text{fault} \arrow[r] & \cdots \end{tikzcd}$$

Figure 2.2: The fundamental chain of dependability and security threads, from [9].

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 [9]:

- 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 [9] 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 [9] the dependability of a system is an integrating concept that includes the following attributes:

- Availability
- Reliability
- Safety
- Integrity
- Maintainability

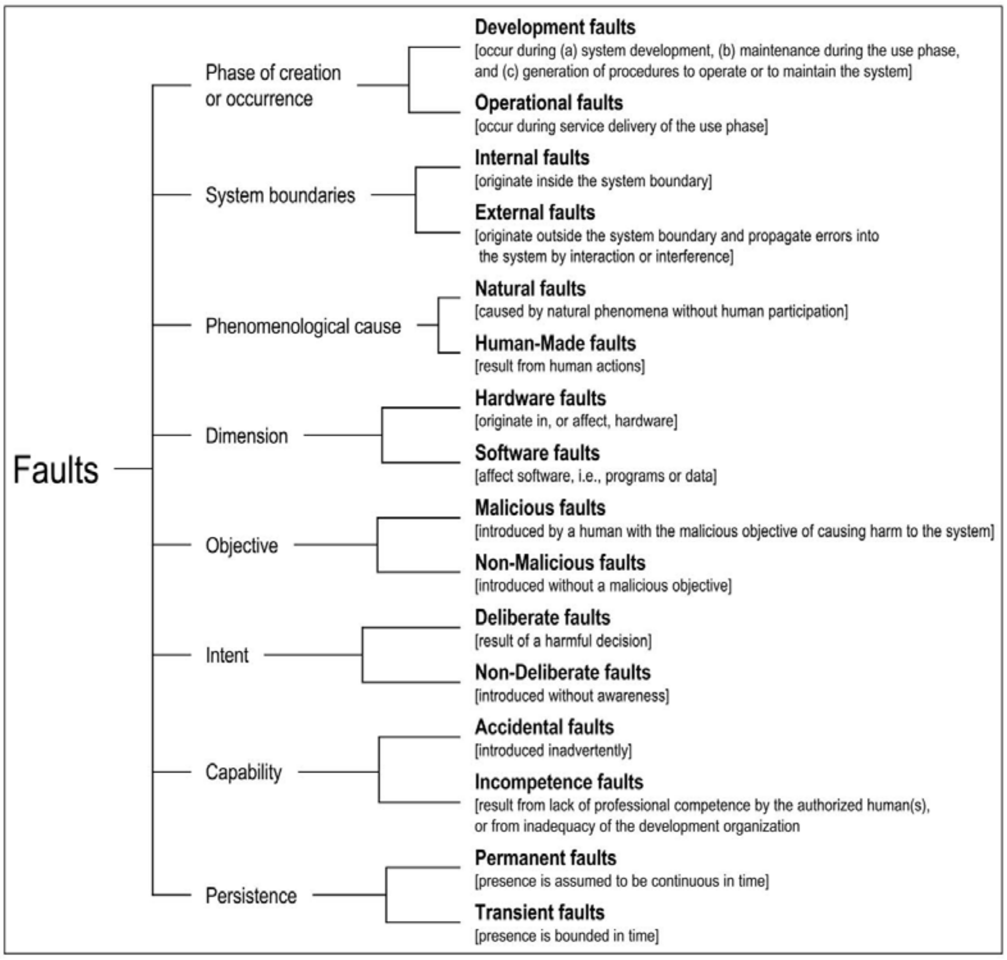

Faults
Phase of creation or occurrence
Development faults
[occur during (a) system development, (b) maintenance during the use phase, and (c) generation of procedures to operate or to maintain the system]
Operational faults
[occur during service delivery of the use phase]
System boundaries
Internal faults
[originate inside the system boundary]
External faults
[originate outside the system boundary and propagate errors into the system by interaction or interference]
Phenomenological cause
Natural faults
[caused by natural phenomena without human participation]
Human-Made faults
[result from human actions]
Dimension
Hardware faults
[originate in, or affect, hardware]
Software faults
[affect software, i.e., programs or data]
Objective
Malicious faults
[introduced by a human with the malicious objective of causing harm to the system]
Non-Malicious faults
[introduced without a malicious objective]
Intent
Deliberate faults
[result of a harmful decision]
Non-Deliberate faults
[introduced without awareness]
Capability
Accidental faults
[introduced inadvertently]
Incompetence faults
[result from lack of professional competence by the authorized human(s), or from inadequacy of the development organization
Persistence
Permanent faults
[presence is assumed to be continuous in time]
Transient faults
[presence is bounded in time]

Figure 2.3: The elementary fault classes, from [9].

In the followings we will present the definition of security as illustrated in [9]. 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.

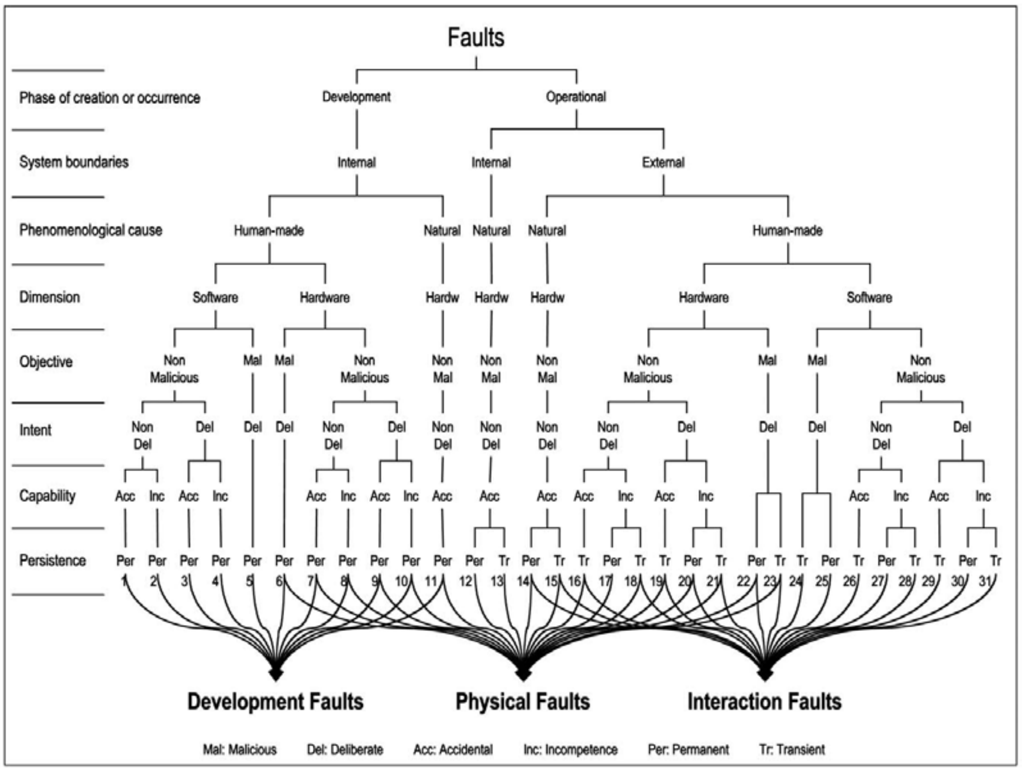


Figure 2.4: The classes of combined faults, from [9].

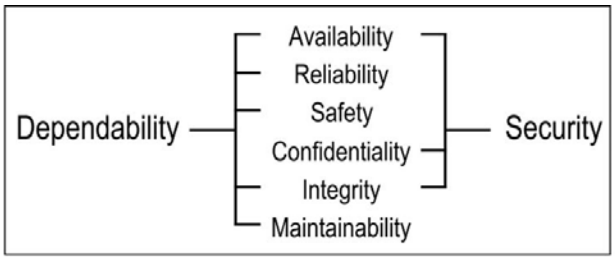


Figure 2.5: Dependability and security attributes, from [9].

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 [10]: 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 [11]: 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 [9]. Due to the unavoidable presence of faults, no system is totally available, safe, secure, or reliable [9].

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 [9], 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 [12] [13].

Fault tolerance, which purpose is to avoid failures of the system, is implemented via error detection or correction and through system recovery [9] [14]. 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 [15] 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 [9]. 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.

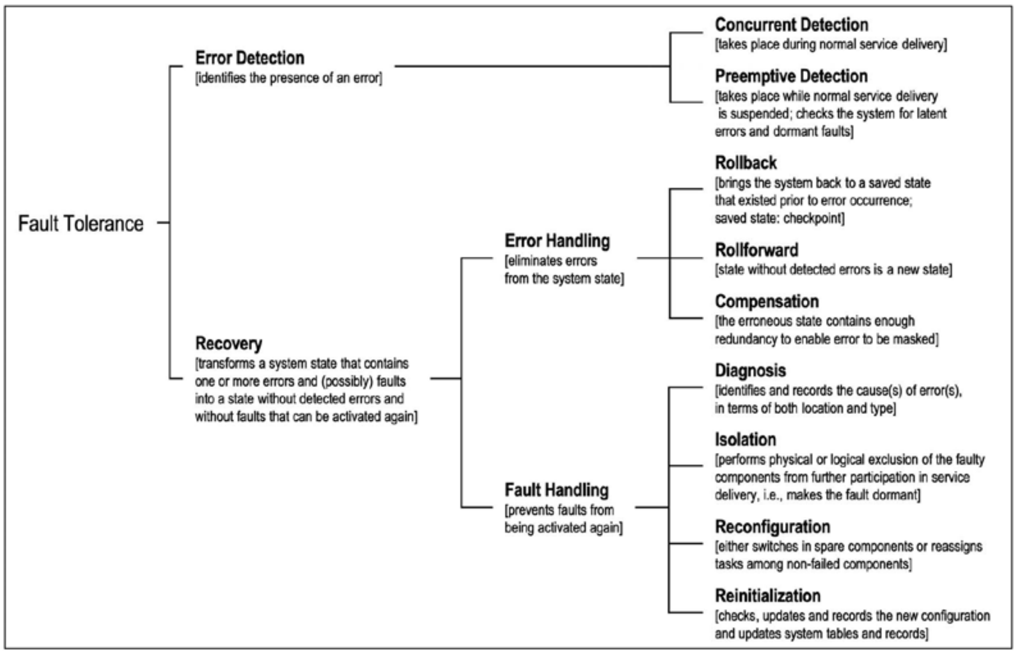

Fault Tolerance
Error Detection
[identifies the presence of an error]
Concurrent Detection
[takes place during normal service delivery]
Preemptive Detection
[takes place while normal service delivery is suspended; checks the system for latent errors and dormant faults]
Recovery
[transforms a system state that contains one or more errors and (possibly) faults into a state without detected errors and without faults that can be activated again]
Error Handling
[eliminates errors from the system state]
Rollback
[brings the system back to a saved state that existed prior to error occurrence; saved state: checkpoint]
Rollforward
[state without detected errors is a new state]
Compensation
[the erroneous state contains enough redundancy to enable error to be masked]
Fault Handling
[prevents faults from being activated again]
Diagnosis
[identifies and records the cause(s) of error(s), in terms of both location and type]
Isolation
[performs physical or logical exclusion of the faulty components from further participation in service delivery, i.e., makes the fault dormant]
Reconfiguration
[either switches in spare components or reassigns tasks among non-failed components]
Reinitialization
[checks, updates and records the new configuration and updates system tables and records]

Figure 2.6: Fault tolerance techniques, from [9].

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 [9].

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 [16]. 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.

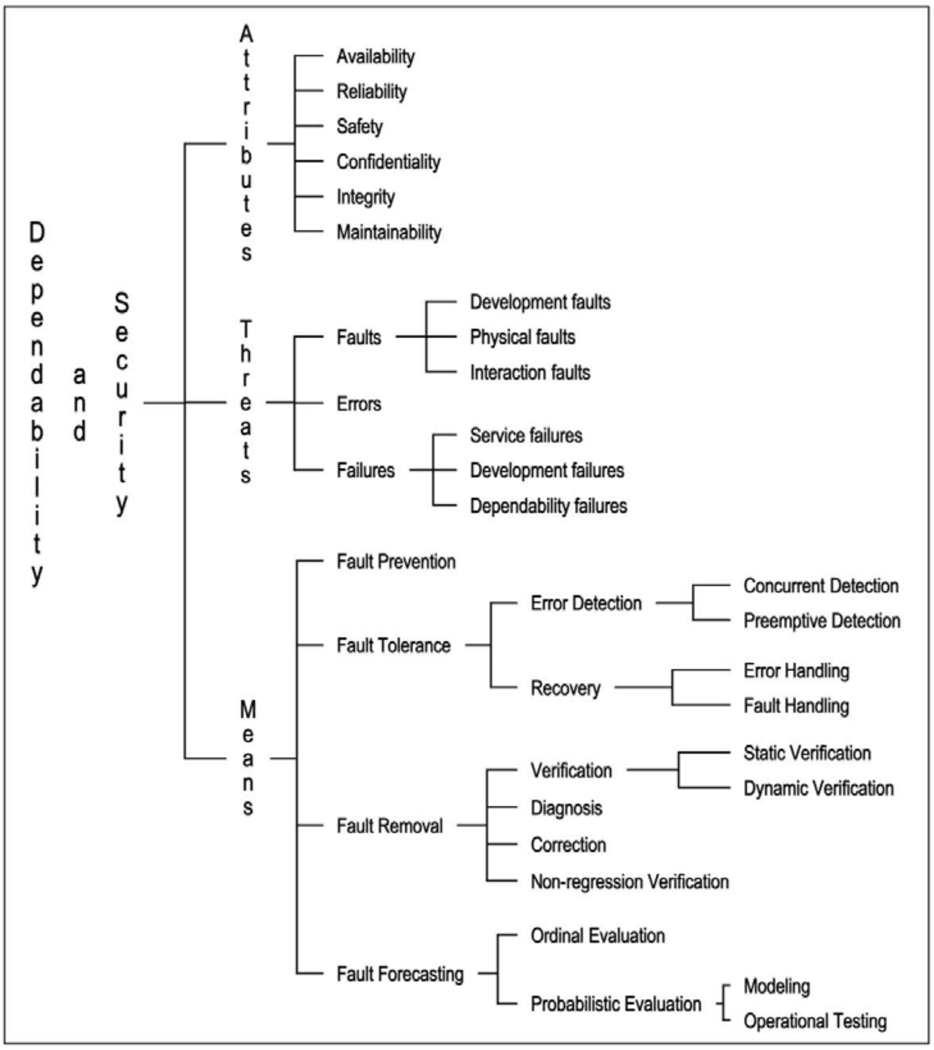


Figure 2.7: A refined dependability and security tree, from [9].

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.

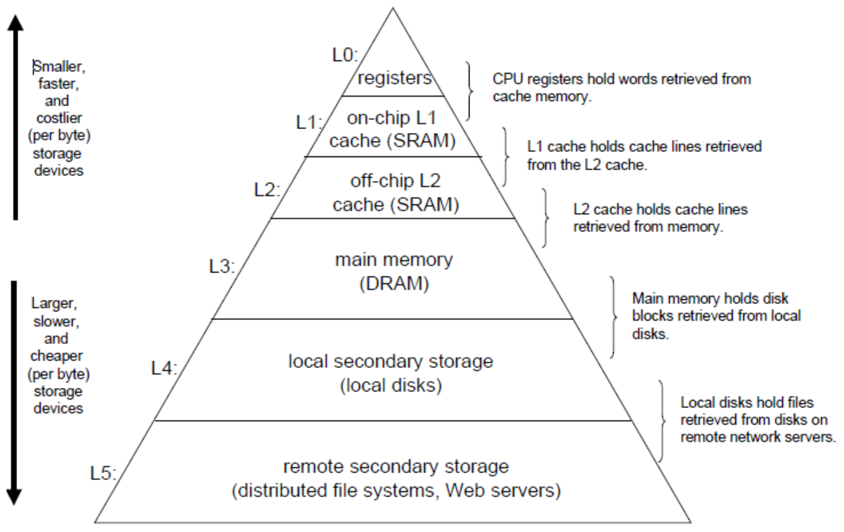

Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0: registers
L1: on-chip L1 cache (SRAM)
L2: off-chip L2 cache (SRAM)
L3: main memory (DRAM)
L4: local secondary storage (local disks)
L5: remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network servers.

Figure 2.8: The memory hierarchy, from [16]

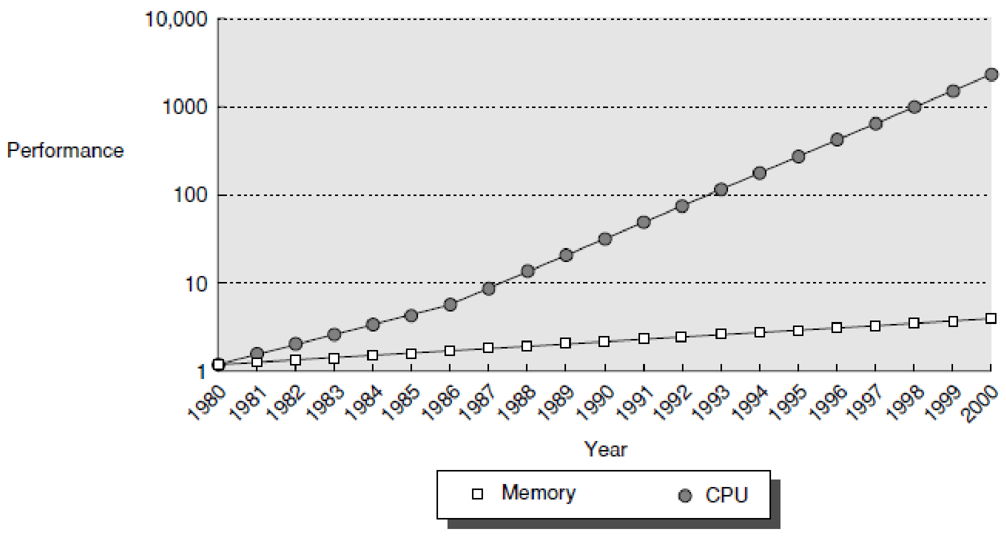


Figure 2.9: The gap in performance between memory and CPU, from [17]

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 [16]. Figure 2.10 shows a typical structure for a computing system with a two level cache system.

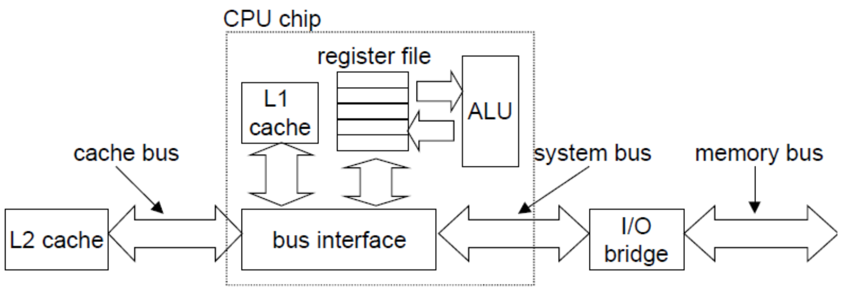


Figure 2.10: Typical structure for two level cache, from [16].

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 [16]. An example of such a cache memory is illustrated in Figure 2.11.

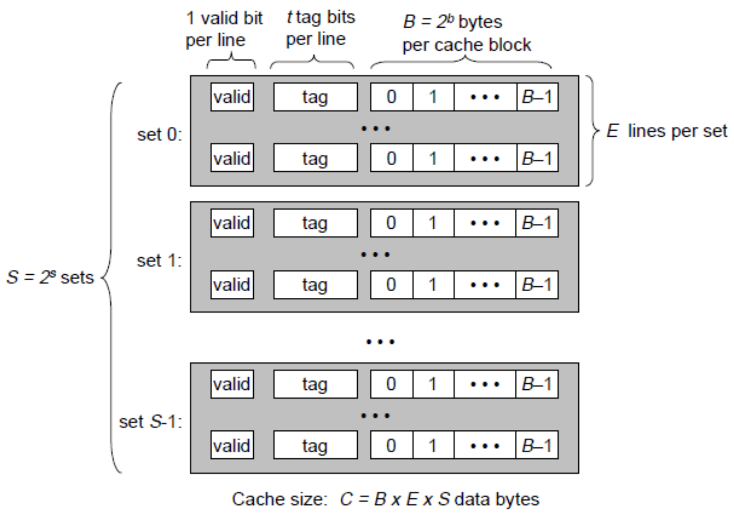


Figure 2.11: General organization of a cache memory, from [16].

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.

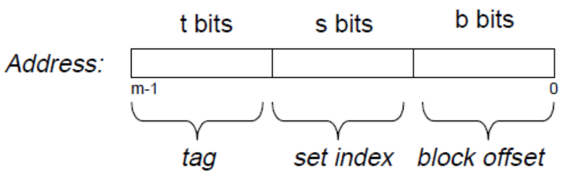
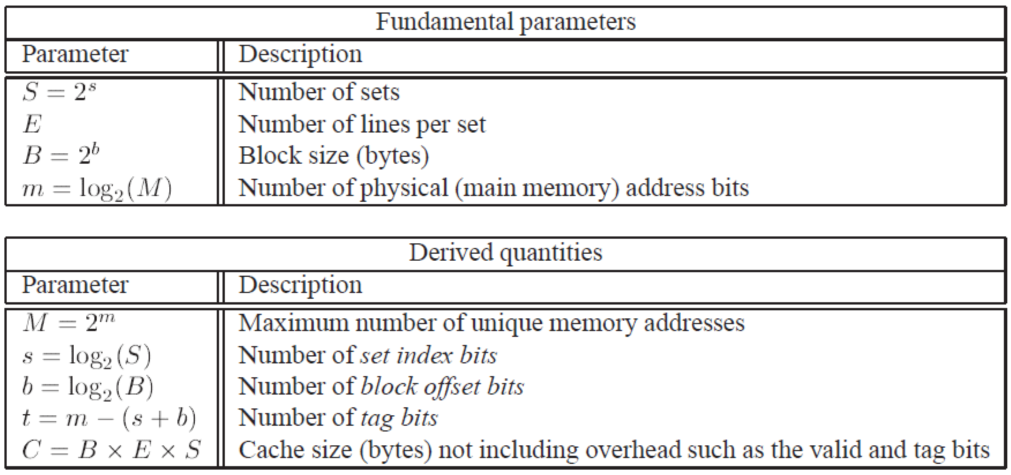


Figure 2.12: Address organization of a cache memory, from [16].

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 [16].

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 [16] [17].

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 [16] [17].

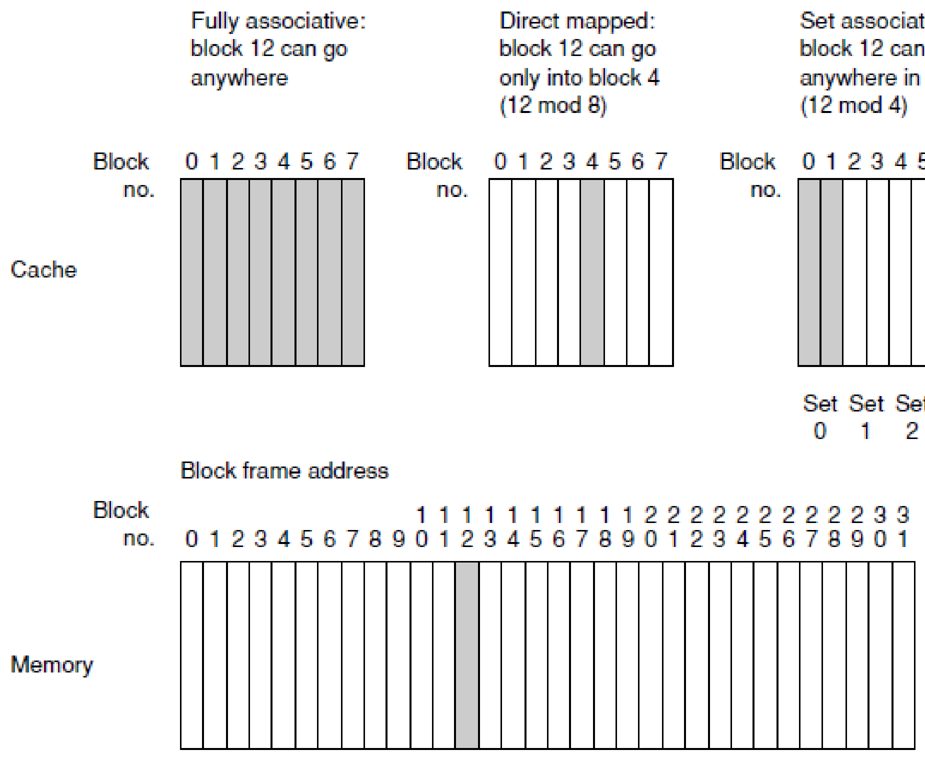

Fully associative: block 12 can go anywhere
Direct mapped: block 12 can go only into block 4 (12 mod 8)
Set associative: block 12 can go anywhere in set 0 (12 mod 4)
Block no.
0 1 2 3 4 5 6 7
Cache
Set 0
Set 1
Set 2
Set 3
Block frame address
Memory
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Figure 2.14: Mapping differences between groups of caches, from [17].

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 [16].

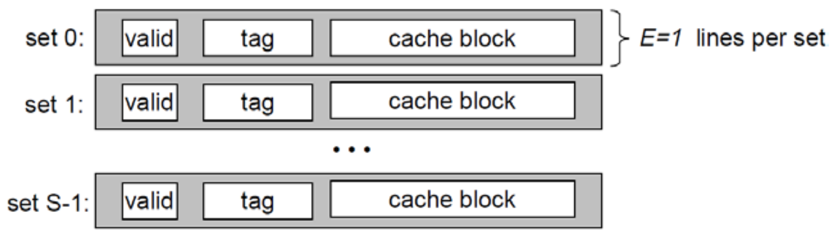


Figure 2.15: Direct mapped cache, from [16].

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.

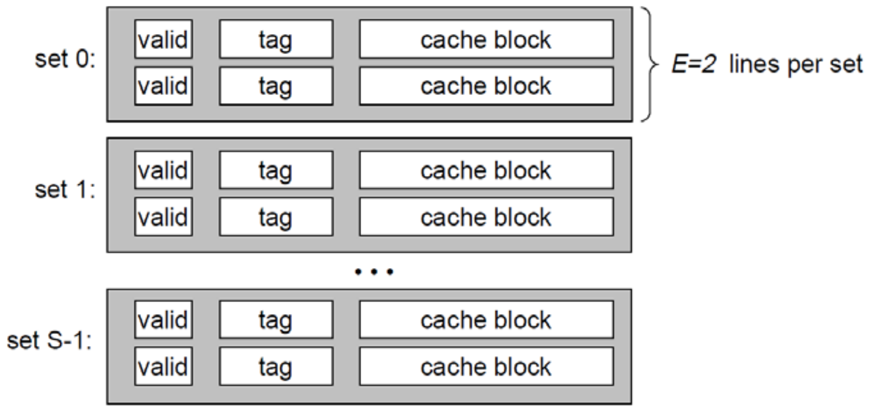


Figure 2.16: Organization of a 2-way set associative cache memory, from [16].

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.

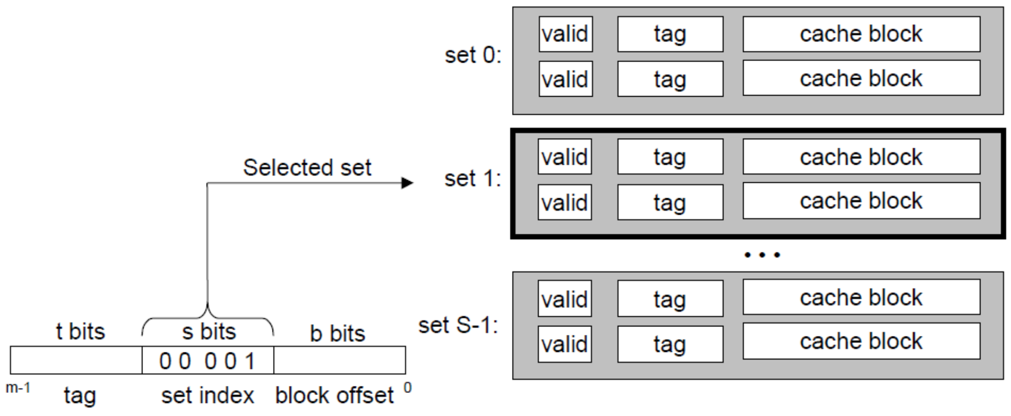


Figure 2.17: Set selection in a set associative cache memory, from [16].

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.

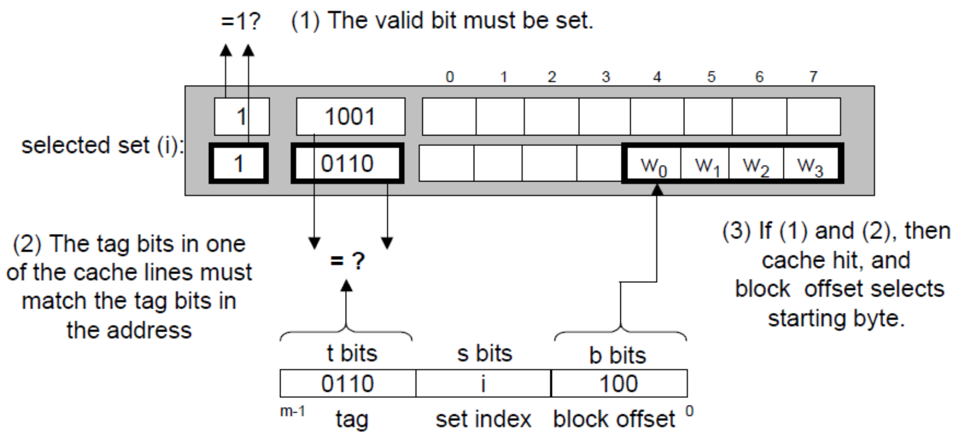

=1?
(1) The valid bit must be set.
0 1 2 3 4 5 6 7
selected set (i):
1
1001
1
0110
w0 w1 w2 w3
(2) The tag bits in one of the cache lines must match the tag bits in the address
= ?
(3) If (1) and (2), then cache hit, and block offset selects starting byte.
t bits
s bits
b bits
0110
i
100
m-1
tag
set index
block offset
0

Figure 2.18: Line matching and set selection in a set associative cache memory, from [16].

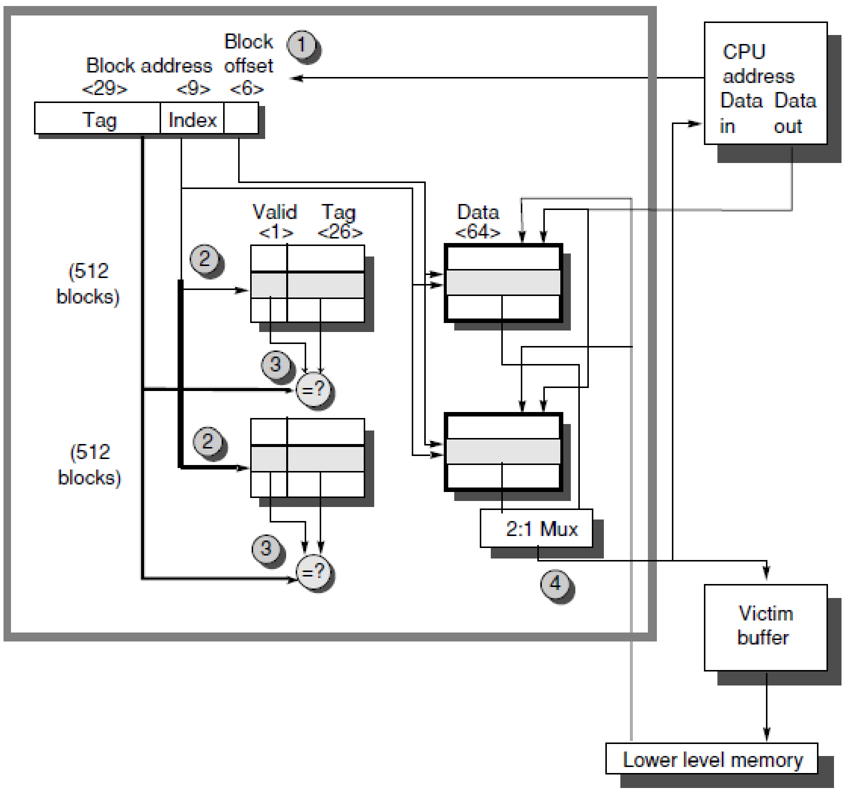


Figure 2.19: The organization of the cache in Alpha 21264 microprocessor, from [17].

2.3 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.

2.3.1 Functional RAM chip models and faults

We will first present the functional model for a RAM memory with all of the main components, Figure 2.20 illustrates this.

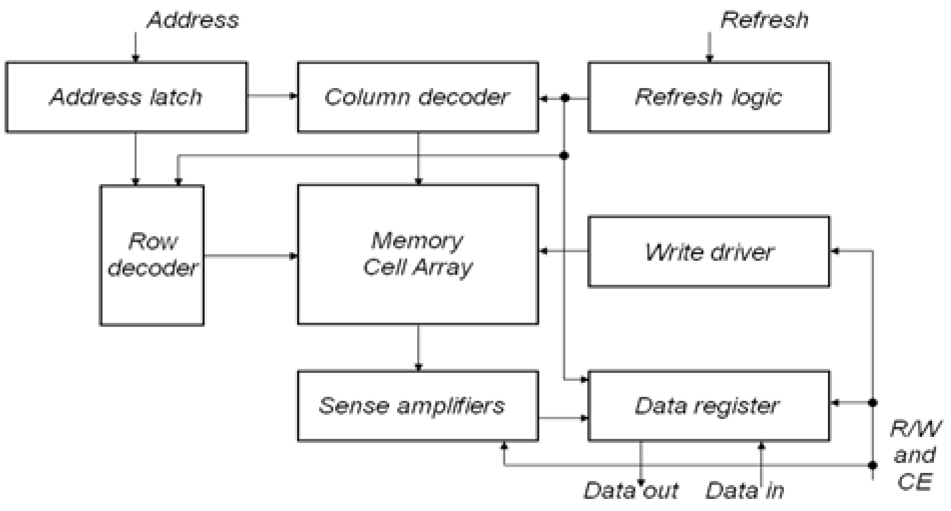


Figure 2.20: DRAM memory model, from [18].

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 2.21 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 2.1, 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 2.1, 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 2.22. As can be seen in Figure 2.22 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 2.22 generated the following types of errors: stuck-at faults, transition faults, coupling faults and neighborhood pattern sensitive faults. Table 2.2 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.

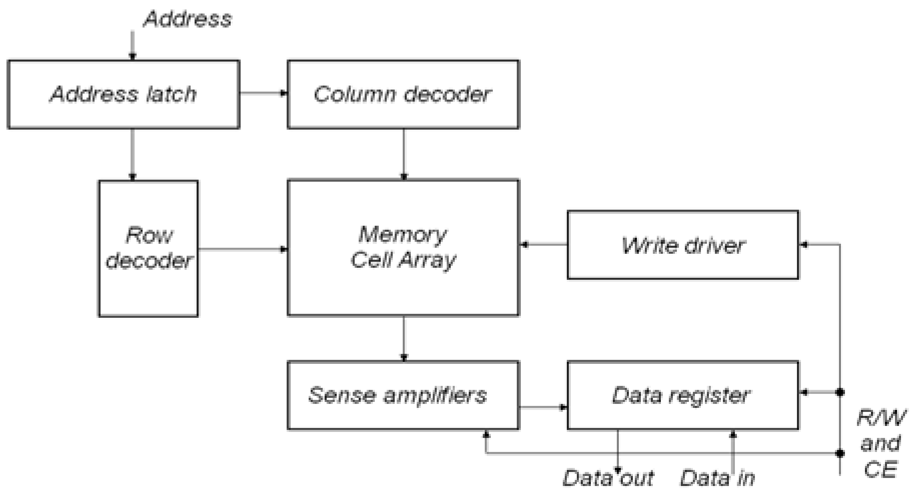


Figure 2.21: SRAM memory model, from [18].

Table 2.1: RAM functional faults, from [18].

	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

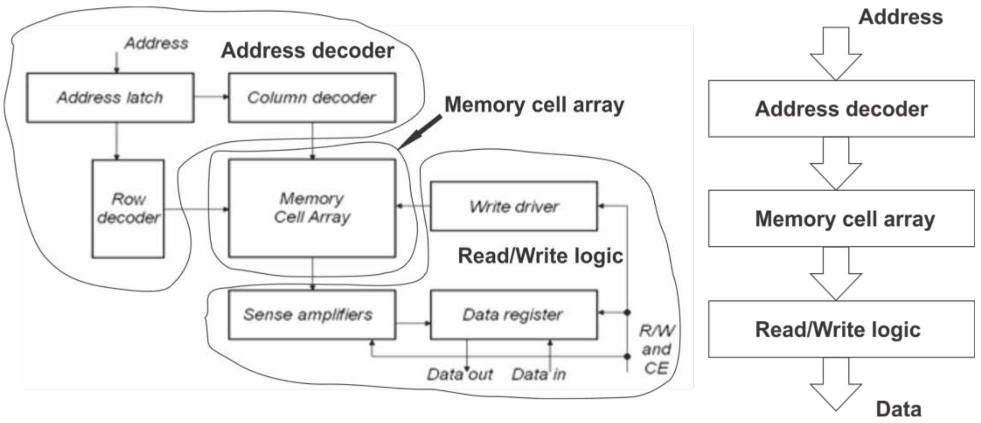


Figure 2.22: Reduced functional model, from [18].

Table 2.2: Reduced functional faults, from [18].

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 2.2 into three categories: faults involving one cell, faults involving two cells, and faults involving *n* cells. The classification is as follows [18]:

- 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 2.3 describes the standard notations used when describing faults and types of faults as presented in [18].

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 2.3: Standard fault notations, from [18].

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 $x \in \{0,1\}$
$\uparrow$	denotes a write 0 operation to a cell containing 1
$\downarrow$	denotes a write 0 operation to a cell containing 1
$\updownarrow$	denotes a write $\bar{x}$ operation to a cell containing an *x*
$\rightarrow$	denotes a write 0 operation to a cell containing an 0
$\rightarrow$	denotes a write 1 operation to a cell containing an 1
$\Rightarrow$	denotes a write *x* operation to a cell containing an *x*
$\forall$	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

2.3.2 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) [18]. 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 [18].

Figure 2.23a shows a state diagram for a healthy memory cell. In Figure 2.23b and Figure 2.23c 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.

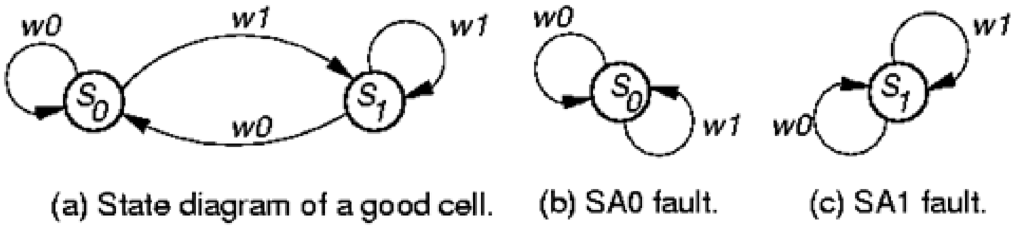


Figure 2.23: State diagram for SAF, from [18].

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 [18]. The notation for the up TF, as shown in [18] 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 [18].

Example

Figure 2.24 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.

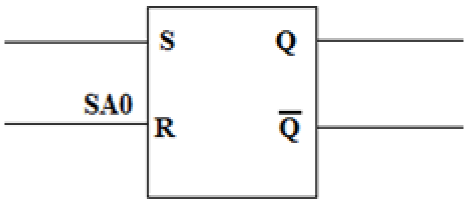


Figure 2.24: A flip-flop as a model for a transition fault, adapted from [18].

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 [18]: 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 2.25. The notations are the same as for the stuck-at faults.

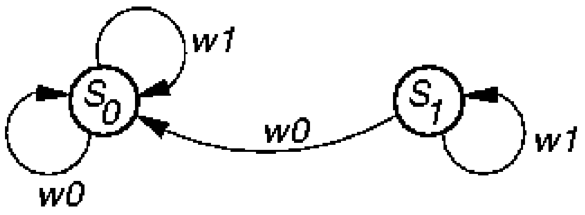


Figure 2.25: State diagram for TF, from [18].

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 [18] [19] [20] [21], 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 [20]. If there is no restriction on the placement of the *k* cells the *k*-coupling fault is very complicated to test for [22].

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 [18].

The inversion coupling faults (CFin) has the following definition: a ↓ (or ↑) transition in one cell inverts the contents of a second cell [18].

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)” [18].

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 [18].

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” [18].

As a conclusion to the state coupling faults Figure 2.26 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 2.27 the NPSF terminology, as presented and used in [18], 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 [18].

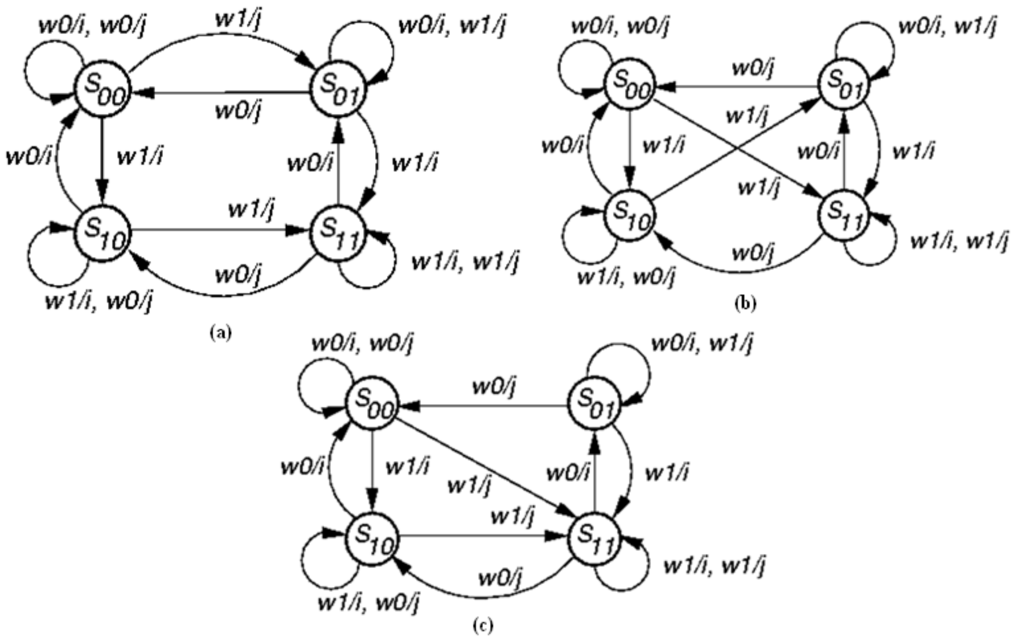


Figure 2.26: State diagrams involving two cells, from [18].

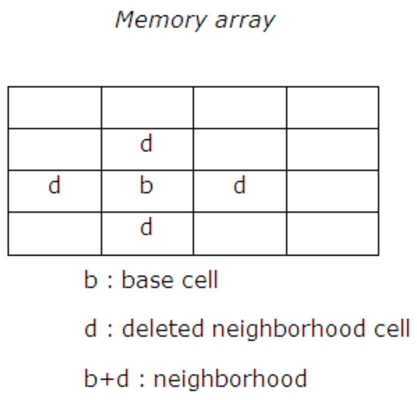


Figure 2.27: NPSF terminology, from [18].

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” [18].

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” [18].

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” [18].

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.

2.3.3 Traditional and March Tests

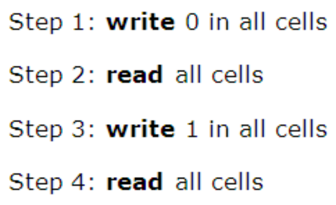
In this subsection of our thesis we provide our reader with a brief description of the traditional test: zero-one, checkerboard, GALPAT and Walking 1/0, sliding diagonal, and butterfly. Also we will provide a short description of the march test MATS and MATS+, concluding this subsection with a comparison between the traditional tests and a couple of march tests. Table 2.4 summarizes the notation used throughout this subsection.

Table 2.4: 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 2.28. This algorithm is also known as MSCAN (Memory Scan) [18] [23].



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 2.28: Zero-One test algorithm, from [18].

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)* [18].

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 2.29. Figure 2.30 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)* [18].

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

Figure 2.29: Cell numbering for checkerboard algorithm, from [18].

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 2.30: Checkerboard algorithm, from [18].

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 2.31. The fault coverage for both these test, according with [18] 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 *$O(n^2)$* [18] [24].

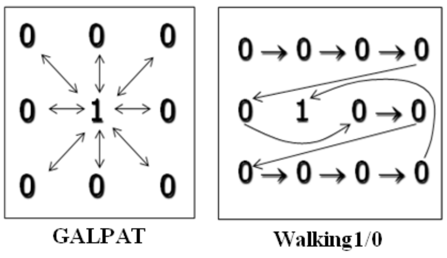


Figure 2.31: Read actions for GALPAT and Walking 1/0, from [18].

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 2.32 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 $O(n^{3/2})$ [18].

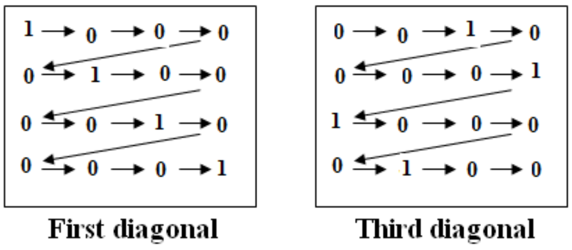


Figure 2.32: Read actions for sliding diagonal, from [18].

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 [18]. 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)* [18].

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 [18] 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 [18], it detects all SAF. This test requires a number of $4n$ operations, having the test time order *O(n)*. The basic scheme of the MATS test is illustrated in Figure 2.33.

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

Figure 2.33: MATS test scheme, from [18].

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 [18].

MATS+

MATS+ is a special version of the MATS test, used when the technology of the memory chip is unknown [18] [25]. 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 2.34.

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

Figure 2.34: MATS+ test scheme, from [18].

We will conclude this section with a summary of the tests described in this section alongside with some other march tests described in [18]. This summary is presented in Table 2.5. As can be observed from Table 2.5 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 2.5: Comparison of memory test algorithms, from [18].

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 Built-In Self-Testing and Graceful Degradation

Throughout 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.

3.1 Memory Built-In Self-Test

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

3.1.1 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 the 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 [26]. 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 [27] 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 built BISTs for integrated circuits he applied the work of Peterson and Weldon [28] and Golomb [29] on error correcting codes and shift registers [30].

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 [30]. A way to ensure these restrictions is to specify a test as one of the system functions, so it becomes a self-test [30]. Many of the digital systems designed at AT&T around 1987 had self-tests, usually implemented in the software [30] [31]. Although this approach provided flexibility and its fault coverage and diagnosis weren't as high as expected [30]. This led to the building of the self-test function into the hardware [32] [33]. 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 [18] [30] [34].

3.1.2 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 [30].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 [18]. The area overhead for memory DFT for a 4Mb DRAM is 1% [35]. The area overhead for memory BIST can be expected at around 2% [30].

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 [18]. For DFT parts of a test are implemented on chip, these are the ones that provide with 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 [18]. This is why the test times are in favor of the BIST when compared to DFT [35].

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 [18]); 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 [18]; 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.

4 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.

4.1 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] [2]. The hard errors can appear due to process variation [1] [3] and aging [4].

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 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 [5] [6] [7]. 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 fault locations.

4.1.1 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.

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 of the cell it 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 of the product. 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 “line” is faulty. We refer to a line as all of the cache locations in which a main memory location can be mapped (see 5.1.2).
- Taking the preemptive measures in order to assure that no more than one location per line will be faulty (5.2.1).
- If there is no way that we can avoid more than one faulty location per line, then we have to decrease the set associativity of the cache. See 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

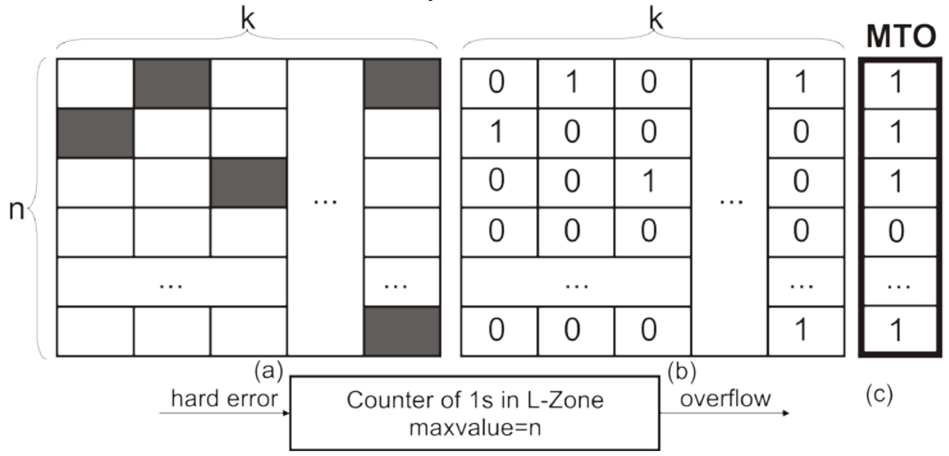


Figure 4.1: SAM description. (a) Memory cell array; (b) L-Zone; (c) MTO column and counter, from [8].

4.1.2 “More Than One” column

If we have only a faulty cell on a line that means that the set associativity of that line is reduced by one, and we will encounter a problem if in some lines all the locations work correctly while in other lines we face two or more faulty locations. 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 line 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 locations per set, *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 line becomes '1' when an error is found on that line, 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.

4.2 Modifications of the Set Associativity

4.2.1 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 example 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 5.2.

We will focus on the "modify_address_to_first_not_0_in MTO_column" instruction for this we will use an example. Considering the situation from Figure 5.3 (a) and we have a new uncorrectable error in line two set two. The memory contents will look like in Figure 5.3 (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 5.3 (c).

```
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 4.2: SAM algorithm, from [8].

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 5.3 (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 C explains the way to implement the switch.

4.2.2 Reducing the set associativity

If we encounter a number of *m* faulty locations, where *m* is a multiple of the number of locations per set, n (i.e. $m=n\cdot l$, $l\in\{1, \ldots ,k\}$, where *k* is the number of sets), 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 paper 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.

One of the implementations of the LRU algorithm is depicted in Figure 5.4 (a). The main idea is to maintain a list of cache set indices sorted from LRU to MRU (Most Recently Used) [8]. 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.

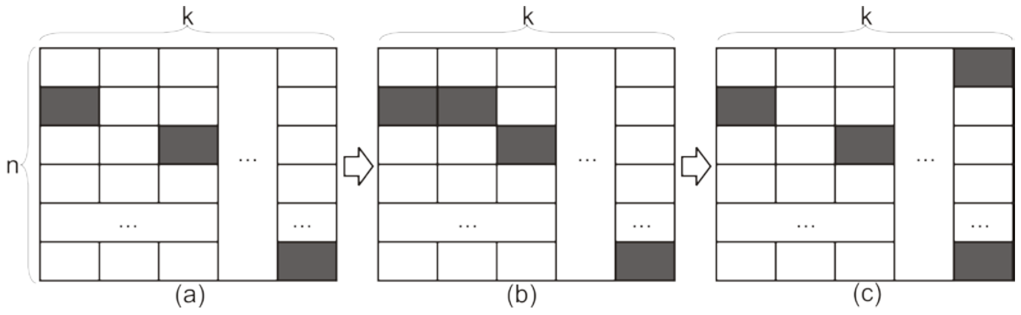


Figure 4.3: SAM remapping. (a) initial memory; (b) unacceptable error distribution;

(c) acceptable error distribution, from [8].

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 5.4 (b).

4.2.3 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 which is taken from the first line in the memory cell array with the MTO-bit equal to '0'. See Figure 5.5, which is a simple example of cache memory with faulty locations.

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.

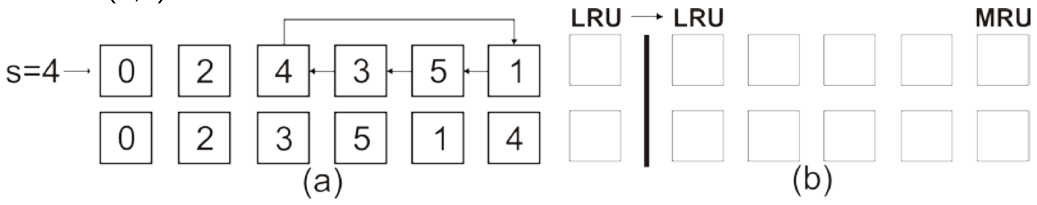


Figure 4.4: (a) LRU algorithm (b) reducing the set associativity, from [8].

4.3 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.

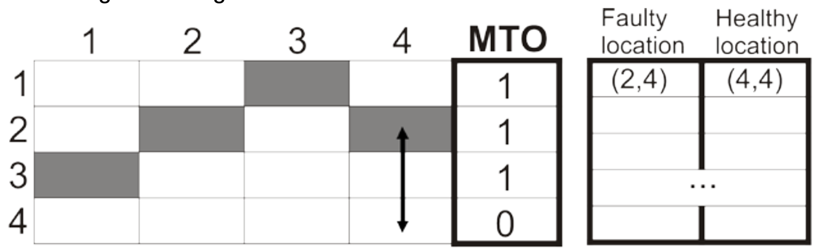


Figure 4.5: Switching table, from [8].

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.

If we have a memory like the one in Figure 5.6, 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 would have to be equal: $possibleH=possibleF$ which implies that:

$$x=(n(k+1))/(2\cdot k) \qquad (5.1)$$

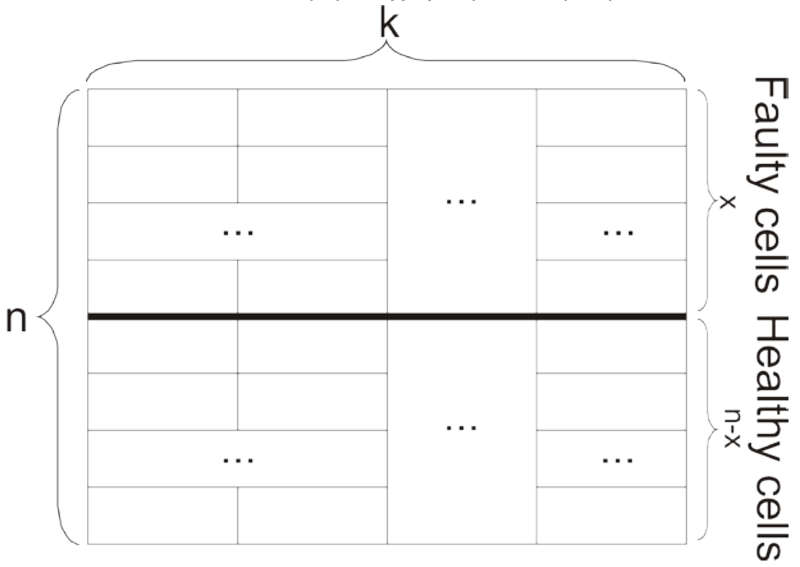


Figure 4.6: Faulty/healthy cells memory organization, from [8].

Example. Consider a L2 cache, 2MB 8-way associative, with 256B block size, as described in [5]. We will calculate the overhead for this memory, for the case of the most probable 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 the switching table equal to 2·13=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 to a direct mapping. There are 448+439+427+410+384+342+256=2706 locations necessary in the switching table, thus making its size 2706·26=70356 bits, see Table 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, see Figure 5.7.

Table 4.1: Numbers of locations required in the switching table, from [8].

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 [8], where if a whole row becomes faulty, the yield will be decreased, SAM can maintain a cache memory working even if a whole line becomes faulty; this is done by the use of the switching table.

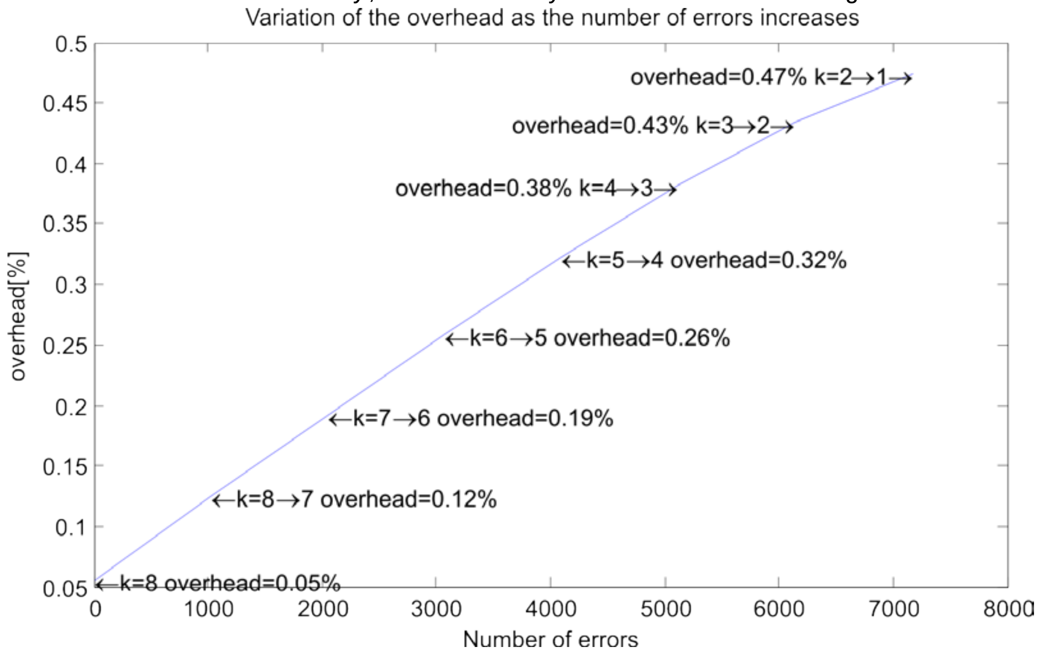


Figure 4.7: Overhead for each reduction of the set associativity, from [8].

4.4 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 [36], 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 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 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 paper. 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.

5 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.

5.1 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.

5.1.1 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 6.1. 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 6.1 summarizes the four functioning states along with a short description.

5.1.2 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 5.2).

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.

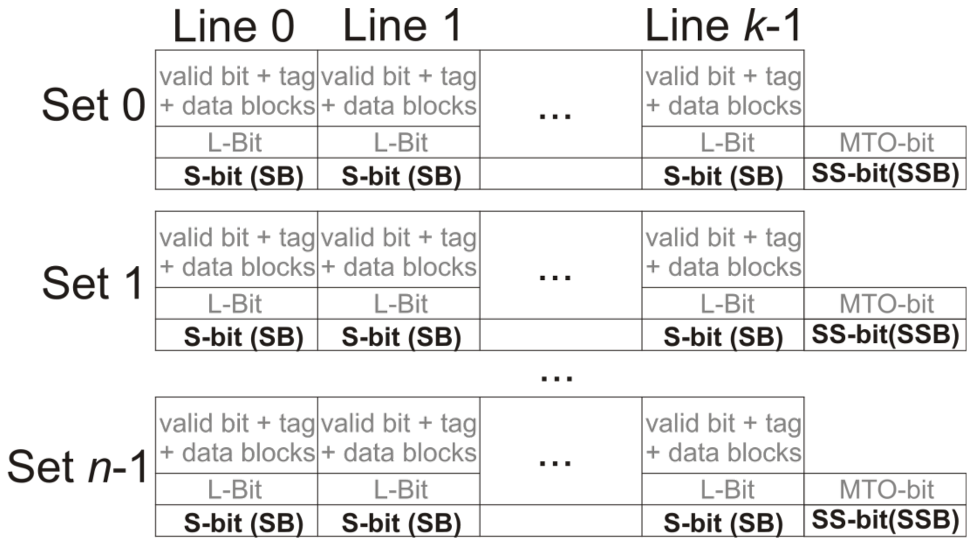


Figure 5.1: Cache memory, after introducing the switching bits

Table 5.1: Description of Switching Bits

	Bits value	Description
Set States	MTO=0 SSB=0	Healthy set
Set States	MTO=0 SSB=1	Switched set (healthy set that has to be used to maintain performance in a faulty set)
Set States	MTO=1 SSB=0	Faulty set (at least one cell in that set is set is faulty)
Set States	MTO=1 SSB=1	Set with a double switched cell
Line States	LB=0 SB=0	Healthy cell
Line States	LB=0 SB=1	Switched cell (faulty cell that has to be maintained functional)
Line States	LB=1 SB=0	Faulty cell (it is usually the first faulty cell encountered when the MTO of the set is 0)
Line States	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, a situation that can be seen in Figure 5.3. 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 [8]. 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.

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 [8]. 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. 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 a time penalty. 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 5.1.5. 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.

5.1.3 Algorithms after the Switching Bits

In this subsection we present the algorithms described in Section 5.1.2 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 5.4.

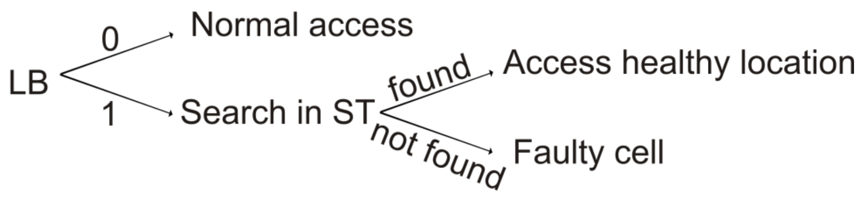


Figure 5.2

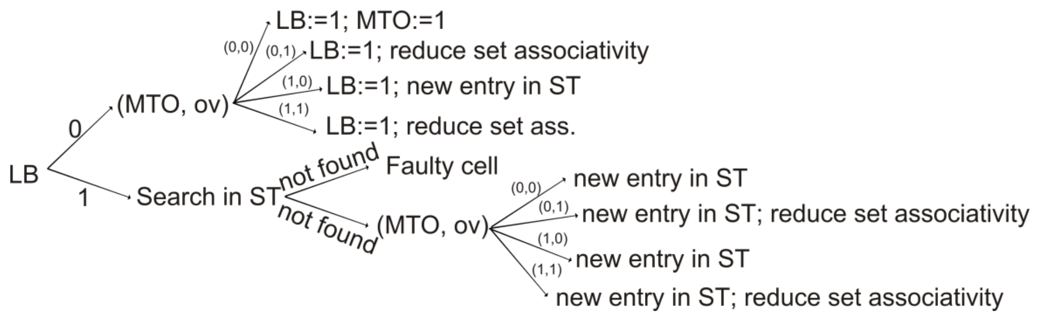


Figure 5.3

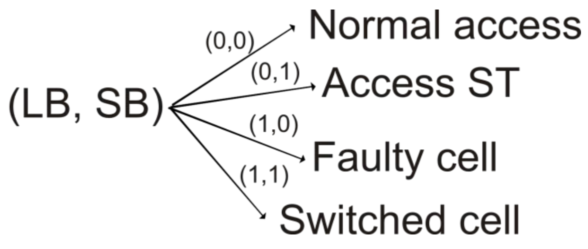


Figure 5.4

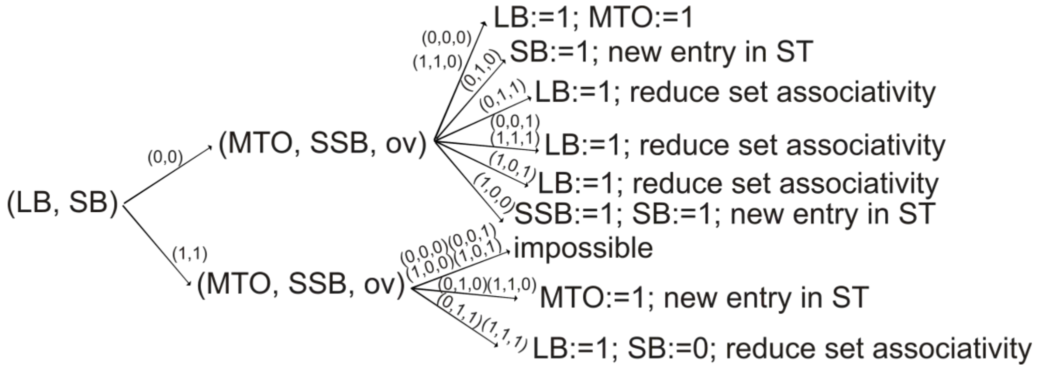


Figure 5.5

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 5.1, 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 5.1), 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 5.1), 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 it 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 $p^2 \cdot (k-r)/(n \cdot k-l)$, 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 5.5 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 5.5. 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, as described in [8], is performed. 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 [8]. 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 [8].

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 [8].

5.1.4 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 in section III.C. 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 simulation results that are presented in Sections 5.1.5 and 5.1.6.

5.1.5 Theoretical results

We start this subsection with some very 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 Equation 5.1. 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 Equation 5.2.

$$LB_{after}\ OR\ SB\ =\ LB_{before} \qquad \text{Equation 5.1}$$

$$MTO - bit_{after}\ OR\ SSB\ =\ MTO - bit_{before} \qquad \text{Equation 5.2}$$

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 5.2 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 5.2, includes cases 1A and 2A, and case 2B, from Table 5.2, includes cases 3A and 4A. Table 5.3 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 5.4 presents the probabilities of the SAM method situations without the switching bits, while Table 5.5 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 5.4 and Table 5.5 into three parts, and – in order to get the overall probability – we just need to multiply the probabilities from the three parts. Throughout Table 5.4 and Table 5.5 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 5.2: Cases for new entries in the switching table

	(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 5.3: Cases for new entries in switching table

	LB	SB	MTO-bit	SSB	ov.
1B	0	0	1	1	0
2B	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 Equation 5.3. These equations are deducted from the changes that were made in the algorithm presented throughout the sections from 5.1.1 to 5.1.4.

$$\frac{after}{before} = \frac{ST_i \cdot ST_{total} + (nk - l - ST_{total})\left(l - nr - \frac{ST_i^2}{n \cdot k}\right)}{nk(l - rn)} \qquad \text{Equation 5.3}$$

The ratio between the accesses in the switching table before and after adding the switching bits is presented in Equation 5.4.

$$\frac{after}{before} = \frac{ST_{total}}{l + ST_{total}} \cdot \left(1 + \frac{ST_i^2}{n^2 \cdot k}\right) \qquad \text{Equation 5.4}$$

5.1.6 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 [8] and [5]. 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 Equation 5.3. On the other hand, Figure 5.6 presents the comparative analysis of the overhead before and after the switching bits were added.

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 5.4: 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 5.5: 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 5.8. In Figure 5.8 the performance is plotted in terms of accesses in the switching table for the SAM method, with and without the switching bits. Figure 5.7 presents the gains both in terms of overhead and performance percentagewise.

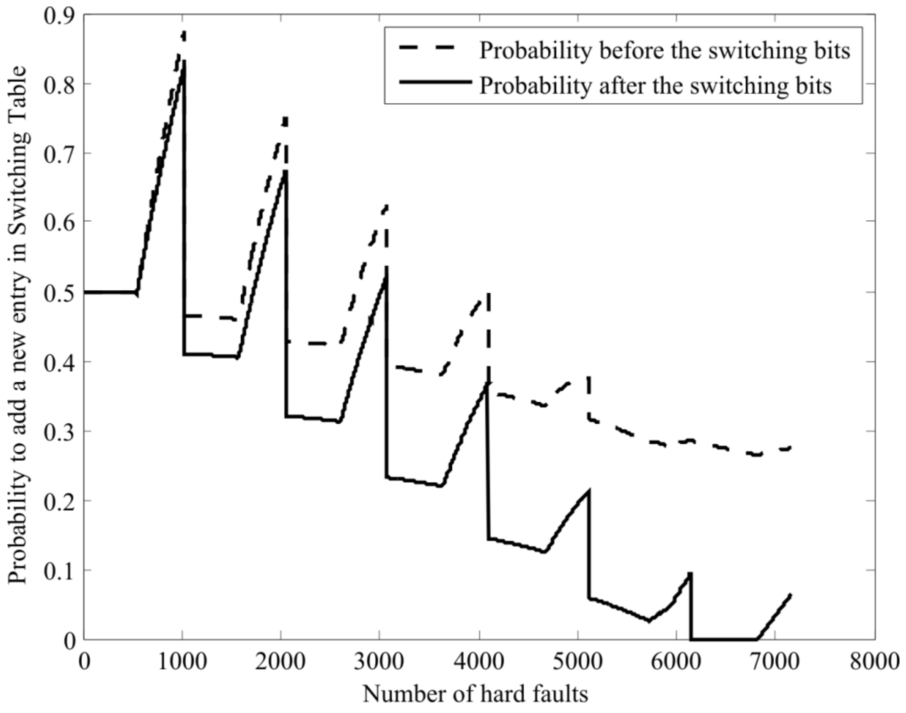


Figure 5.6: Probability for new entry in switching table

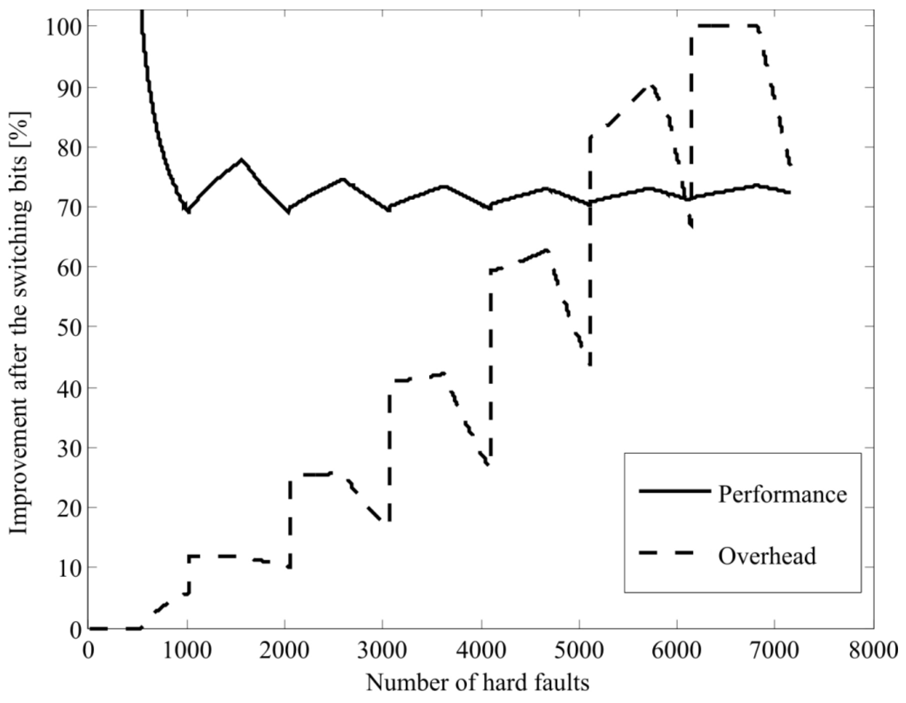


Figure 5.7: Improvement from original SAM

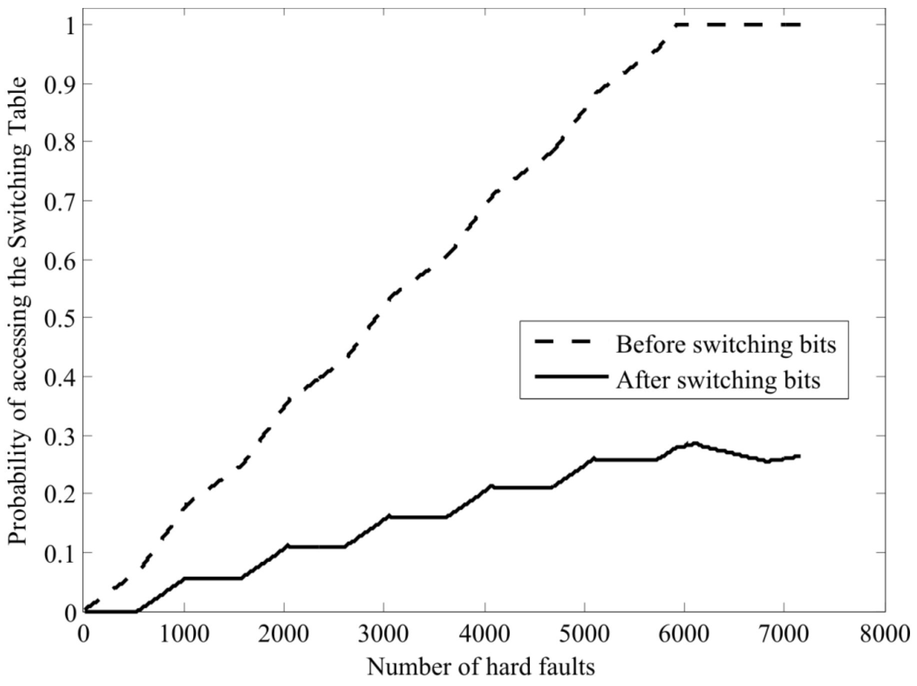


Figure 5.8: Probability for accessing the switching table

5.1.7 Conclusions

The first goal of this first section of this chapter was to perform an analysis of the SAM method [8], in terms of overhead and performance. This analysis was performed throughout sections II, III and IV. 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 5.1.5 and 5.1.6 and Figure 5.6. 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 5.7 and Figure 5.8, and for some cases they can be completely eliminated, as seen from section 5.1.4. 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.

5.2 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.

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.

5.2.1 First case

For case *a.* from section 5.2 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 5.9, while the algorithm used in case *a* is depicted in Figure 5.10.

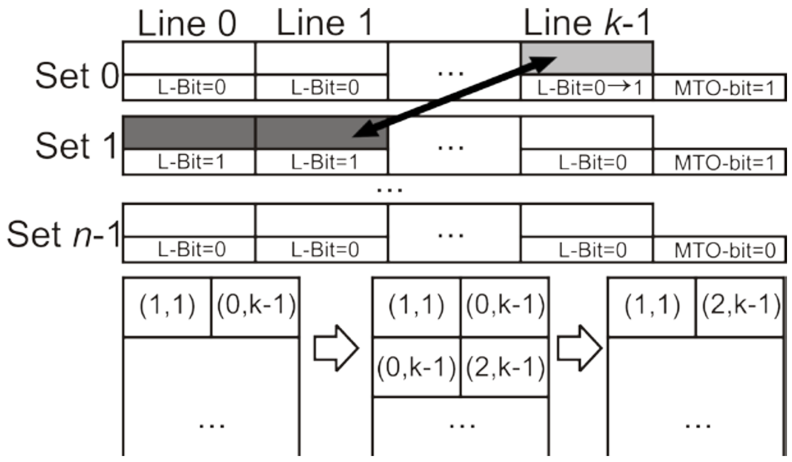


Figure 5.9: First case example

```
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 5.10: First case algorithm

5.2.2 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 the reorganizing the switching table is depicted in Figure 5.12, together with a corresponding example, Figure 5.11.

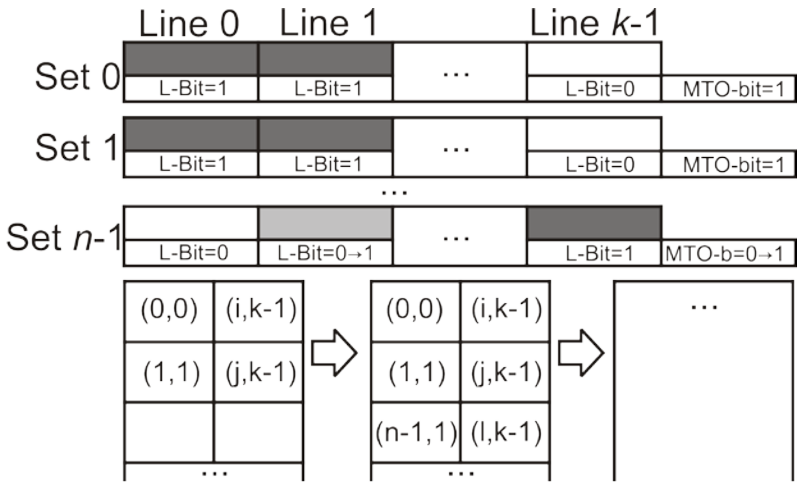


Figure 5.11: Second case example

```
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 5.12: Second case algorithm

5.2.3 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 [1] 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 5.13, along with the employed algorithm in Figure 5.14.

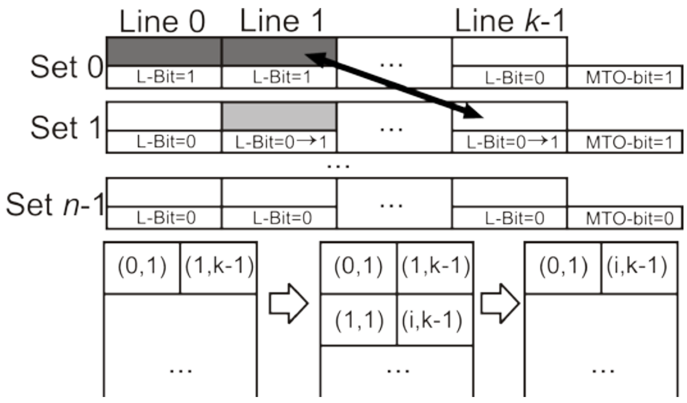


Figure 5.13: Third case example

```
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 5.14: Third case algorithm

5.2.4 Improvements

Table 5.6 provides the number of locations in the switching table, determined probabilistically, as it was performed in [8]. 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 [8] and [5].

Note that case *a.* from section III refers only to a cell in a set, and case *c.* from section III 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 5.6: Results obtained using the described improvements

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

5.2.5 Overhead Gains

As presented in Section 5.2, 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 [8]. 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* as the number of sets with faulty lines, as in Equation 5.5.

$$x = \frac{n\,k + l}{2\,k} \qquad \text{Equation 5.5}$$

In Equation 5.5 *n* is the number of sets in the cache memory, *k* is the number of lines per set, or the set associativity. 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 5.15. After applying Equation 5.5 to this cache memory, we obtain the results depicted in Table 5.6. Figure 5.16 illustrates a comparison in terms of overhead between the original SAM [8] and its improved version which is described in this paper.

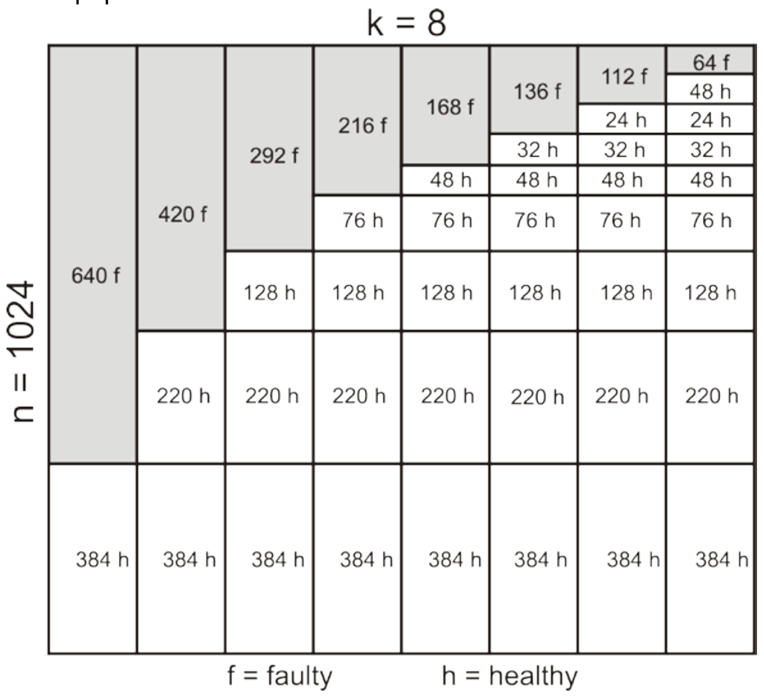


Figure 5.15: Example of fault distribution

5.2.6 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 5.6. Table 5.6 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.

$$ank\tau\xi \qquad \text{Equation 5.6}$$

$$a'nk\tau'\xi \qquad \text{Equation 5.7}$$

Equation 5.6 shows the time penalty of the switching table before improving SAM, while Equation 5.7 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 5.6 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 5.17 illustrates the two time penalties; before and after our improvements; due to the reduction of switching table size, the improvements can be observed even from the first errors.

In Figure 5.18 we have summarized the improvements brought by our method percentage-wise, both in overhead and in performance. As it can be seen in Figure 5.18, 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%.

5.2.7 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 is can be maintained as it is and 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 [37] 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 mapped 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 section II'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.

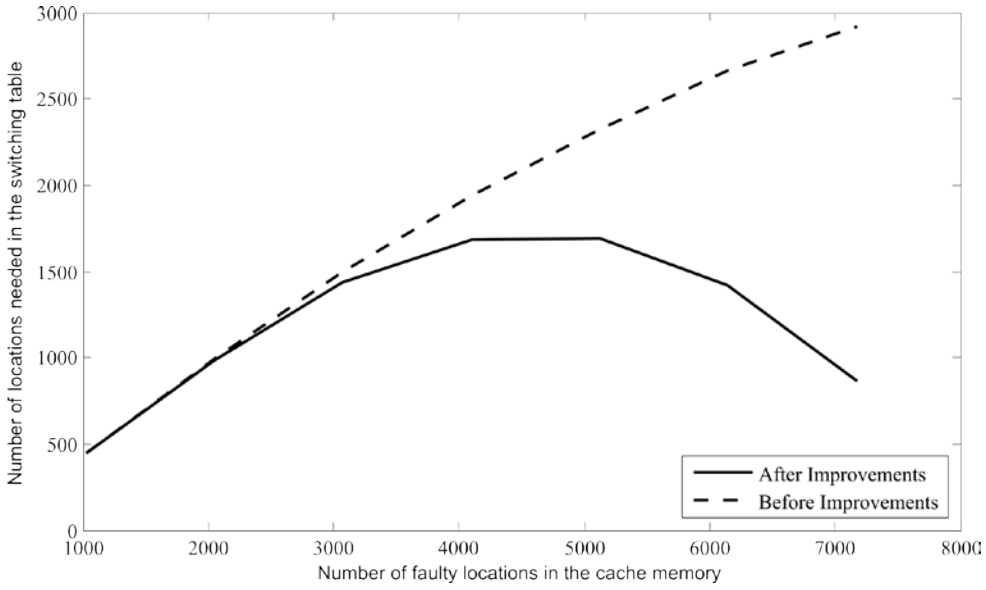


Figure 5.16: Overhead improvement

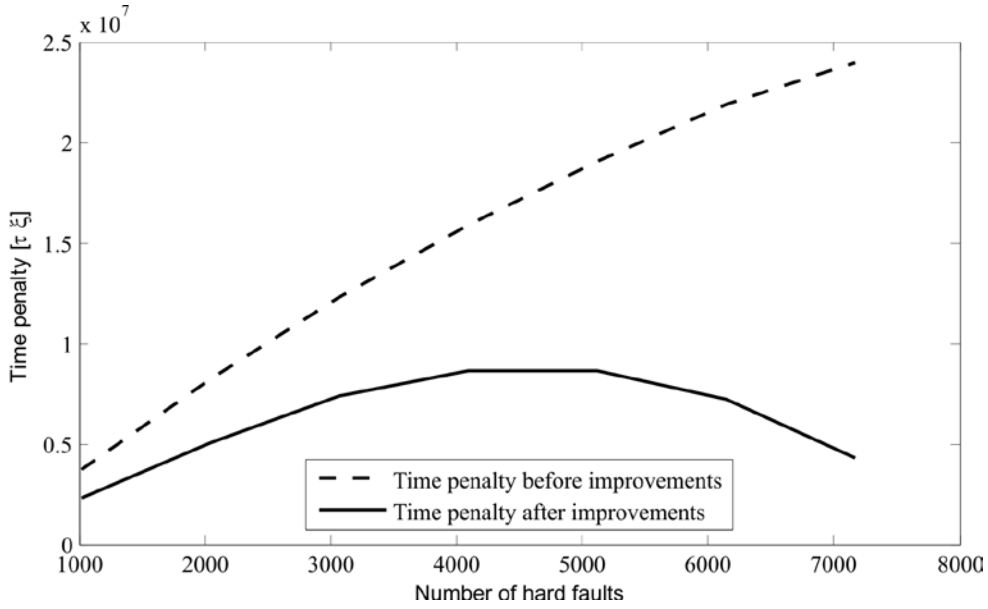


Figure 5.17: Performance improvement

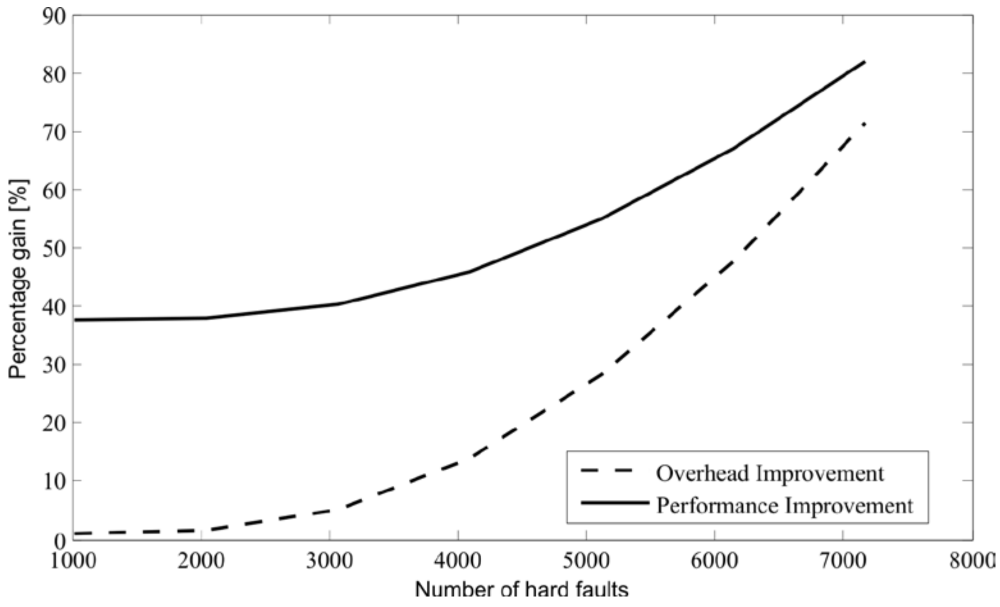


Figure 5.18: Improvements obtained

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. 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 a 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 [8] and [1], that for a cache memory as the one described in section IV.C, 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 quantity). 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 5.2.5, as can be seen by inspecting Figure 5.16, 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 [37].

5.2.8 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 [8].

The reliability improvement of SAM with these modifications of the switching table remains the same as in [8]. As described in [36] 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 $R=1-p^{(k-1)\cdot n+1}$ [8], 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.

6 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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

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

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

[28] W. W. Peterson and E. J. Weldon, *Error-Correcting Codes*. New York, USA: John

Wiley & Sons, 1972.

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

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

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.