PERFORMANCE ANALYSIS OF DYNAMIC SPARING AND ERROR CORRECTION TECHNIQUES FOR FAULT TOLERANCE IN NANOSCALE MEMORY STRUCTURES

By

CASEY MILES JEFFERY

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2004

This document is dedicated to my new wife, Mariah.

ACKNOWLEDGMENTS

It is essential that I thank my advisor, Dr. Renato Figueiredo. He has given a great deal of time and assistance in the culmination of this degree. His enthusiasm and guidance helped see me through the completion of this project.

I would also like to thank my family and friends who have supported me in this endeavor. Completion of the research and the writing of this thesis would not have been possible without their encouragement.

TABLE OF CONTENTS

page

LIST OF TABLES

Table page

LIST OF FIGURES

Figure page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

PERFORMANCE ANALYSIS OF DYNAMIC SPARING AND ERROR CORRECTION TECHNIQUES FOR FAULT TOLERANCE IN NANOSCALE MEMORY STRUCTURES

By

Casey Miles Jeffery

December 2004

Chair: Renato J. O. Figueiredo
Major Department: Electrical and Computer Engineering

Continued advances in the fabrication of molecular structures have led to speculation that nanoelectronic devices may be incorporated with CMOS technology in the realm of ultra-dense logic and memory systems. Many proposed molecular systems involve self-assembly techniques ideally suited for large, regular structures such as memory. This method of fabrication, however, is predicted to result in an increased hard error rate due to physical defects. This effect will be compounded by an increased sensitivity of the ever-smaller bit storage units to soft errors (e.g., resulting from noise and radiation effects).

Current memory devices rely on fault tolerance schemes such as modified Hamming error correcting codes and static spare swapping to cope with these errors. It is conceivable, however, that such solutions may not be sufficient in systems that are larger and built using less reliable components and/or manufacturing techniques.

This thesis describes a memory organization that supports hierarchical, dynamic fault tolerance mechanisms applicable to heterogeneous CMOS/molecular systems. It has been projected that, in such systems, programmable mapping circuitry becomes necessary when the interface between microscale and nanoscale address lines is formed in a non-deterministic manner. This mapping circuitry may double as a means of implementing more advanced forms of reconfiguration and error correction codes useful in detecting and recovering from runtime faults.

In lieu of the ability to formulate an analytical model for such complex systems, computer simulation is used to estimate the effectiveness of various fault tolerance configurations. Results show that for a given percentage of redundancy overhead, these methods may allow for substantial improvements in reliability over standard error correction codes.

CHAPTER 1
INTRODUCTION

The size of the devices used to construct electronic components has decreased at an astonishing pace over the past fifty years. It has followed the very familiar Moore's Law, which states that the number of devices doubles approximately every 18 months. As of 2004, the state of technology is still several orders of magnitude away from even the conservative limits imposed by physics (i.e., 1 bit/$Å^3$) [1], and it is in the foreseeable future when human ingenuity will allow for the integration of huge numbers of devices in a single part (i.e., $10^{12}/cm^2$) [2].

There will be many new challenges associated with this level of integration that will require complete new approaches to designing, fabricating, and testing devices. The circuits will likely require construction using self-assembly and self-alignment techniques at the molecular level, as it will almost certainly be infeasible to accurately order and connect such a large number of devices using the current lithography-based technology. The relative inexactness inherent in the self-assembly method of construction will make the technology most applicable to the manufacture of devices with large, regular patterns such as programmable logic and memory arrays at the lowest level. The arrays of nanoscale devices may then be used hierarchically as building blocks for more complex circuits [3, 4].

Even in these very regular processes, it is predicted that defect densities on the order of 5% can be expected, and the small size of the devices will make them more sensitive to external disturbances such as radiation and electromagnetic interferences [2,

5, 6]. Off-line testing to find the faults will be impractical due to the sheer number of devices. Instead, the circuits will be required to test themselves to a high degree and detect defects in a reasonable amount of time. They must also be capable of dealing with the defects by employing some form of fault tolerance.

The defects that occur at runtime are referred to as dynamic defects and are the focus of this thesis. In the past, dynamic defects have all but been ignored. Only high-end workstation and server memory currently make use of dynamic fault tolerance mechanisms in the form of error correcting codes (ECC's). The codes have typically been capable of logically correcting at most a single bit in each block of data accessed. This will not be sufficient, however, for memories with orders of magnitude higher error rates. In addition to the increase in bit error rates, an increase in the rate of row, column, chip, and other periphery-related defects is expected. Defects of this type often cannot be handled even when ECC's are implemented and ultimately lead to the failure of the entire memory.

The most likely solution to combat these problems in future memory designs will be to implement a form of fault tolerance that is more advanced than has been done in the past. This new method of fault tolerance involves more powerful ECC's, as well as dynamic reconfiguration. In dynamic reconfiguration *built-in self-test* (BIST) and *built-in self repair* (BISR) logic are used to detect and repair faults either by swapping the defective devices with spare devices or by marking the faulty devices so they will not be accessed.

In this thesis, dynamic fault tolerance configurations are introduced that implement such advanced ECC's and dynamic reconfiguration in a hierarchical fashion. A simulator

is then used to estimate the performance of the systems in terms of tolerance to faults throughout the usable lifetime of the parts. Finally, the simulation results are utilized to suggest configurations capable of providing the highest degree of synergism and correspondingly make the most efficient use of a given percentage of overhead in terms of redundant bits.

Chapter 2 gives a brief overview of the current state of memory devices, as well as an introduction to fault tolerance mechanisms and the metrics used in determining the reliability of such devices. Chapter 3 covers many of the most promising nanoscale devices and architectures that have recently been developed and the corresponding reliability concerns associated with them. Chapter 4 introduces the proposed hierarchical fault tolerance mechanisms and goes on to show how they can be applied to the architectures from the previous chapter. Chapter 5 details the simulation methodology used to model the fault tolerance mechanisms and includes the results that have been obtained. Finally, Chapter 6 summarizes the findings and gives some insight into future work that may be done in the area.

CHAPTER 2
BACKGROUND

Fault tolerance in electronic devices involves the use of redundant circuitry to detect and possibly correct faults. This is not a new notion, as it has been in place since the 1940s when it was proposed by John von Neumann. Before this discovery, it was thought that large-scale circuits (i.e., millions of devices) could not be constructed to function reliably. Von Neumann showed, however, that a very high degree of reliability can be obtained, albeit at a high cost in terms of redundancy overhead [7].

To date, von Neumann's ideas have garnered little attention in industry as circuits with hundreds of millions, and recently billions of devices have been constructed to function reliably with little or no need to implement such fault-tolerance mechanisms. These feats require extremely low error rates that are only possible with advanced silicon fabrication techniques conducted in the cleanest facilities in the world. It does seem reasonable to reevaluate the usefulness of fault tolerance, however, as device sizes approach atomic levels and the ability to build entire components with near perfection becomes too difficult and costly.

This chapter first gives a brief overview of the current state of memory device technology and fabrication. The terminology related to memory reliability is then introduced, and the basic mechanisms used to deal with defects are discussed. The topics covered in this chapter will be used as the foundation for describing more advanced fault-tolerance architectures applicable to future generations of devices.

2.1 Modern Memory Devices

It is well-known that memory devices in computer systems are used in a hierarchical fashion that can be represented in a simplified form by the pyramid structure shown in Figure 2-1. The base of the pyramid is composed of very large devices such as tape backups and networked hard-disk storage devices that have a low cost per bit, slow access times, and large data access chunks. The top of the pyramid represents the smallest, fastest devices, which are the registers in the core of the microprocessor. The registers move data in small blocks, have very fast access times, and a high cost per bit.

In recent years, there has been a continued growing and splitting up of the pyramid. The number of bits at each of the levels has grown steadily and new levels have emerged. There has also been a great deal of growth outside the pyramid. The use of non-volatile flash memories in devices such as cameras, cellular phones, and personal digital assistants is on the rise. There has also been a push towards entirely new architectures that integrate logic processing units into memory devices [8].

It is conceivable that the memories constructed from nanoscale devices may fill any of these roles, as well as others that have yet to be discovered. Because of this uncertainty, the fault tolerance architectures to be presented in this thesis are relatively generic and applicable to a wide range of memories, including random access devices in the levels that have been highlighted in Figure 2-1.

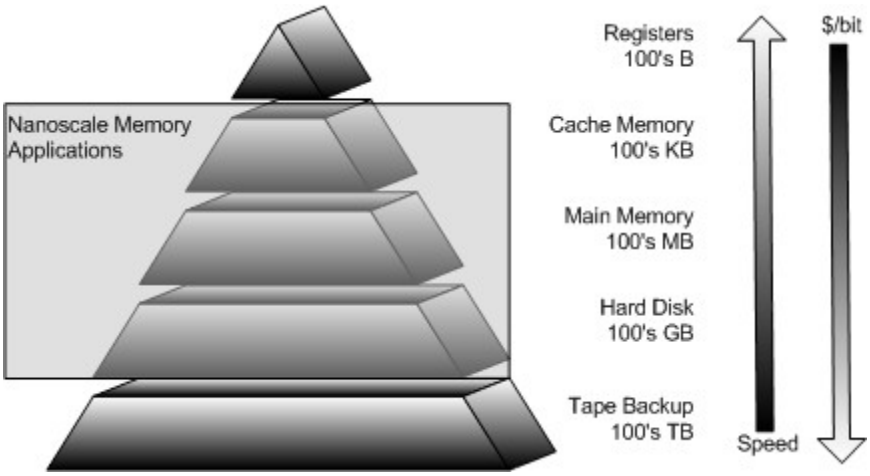


Figure 2-1. Memory hierarchy representation. Highlighting is placed over levels most applicable to the fault tolerance architectures described in this thesis

2.1.1 Memory Architecture

Modern memory devices come in many forms and are categorized based on their function and access pattern. The function is defined as whether the memory is read-only or rewritable, as well as if it is volatile or non-volatile. The access pattern of the memory classifies whether the data can be randomly accessed, as in a common DRAM device, or if the accesses must be done serially, which is the case for video memory. There are also memories that use parts of the data instead of an address for memory accesses. These are known as associative memories and are common in cache structures. An overview of a wide variety of memory devices is included in Appendix A.

The layout of most memory devices is approximately the same regardless of the type. They generally follow a square shape and make use of row and column decoders to control the selection of word and bit lines. Figure 2-2A shows the simplest example in which the decoded row signals select multiple words and the decoded column signals choose which of those words is returned. Figure 2-2B expands on this slightly by adding a block decoder that determines in which block the word line will be activated. The

design is known as *divided word line* (DWL), and the extra level of decoding has the benefit of reduced power consumption at the cost of additional decoding latency and area overhead.

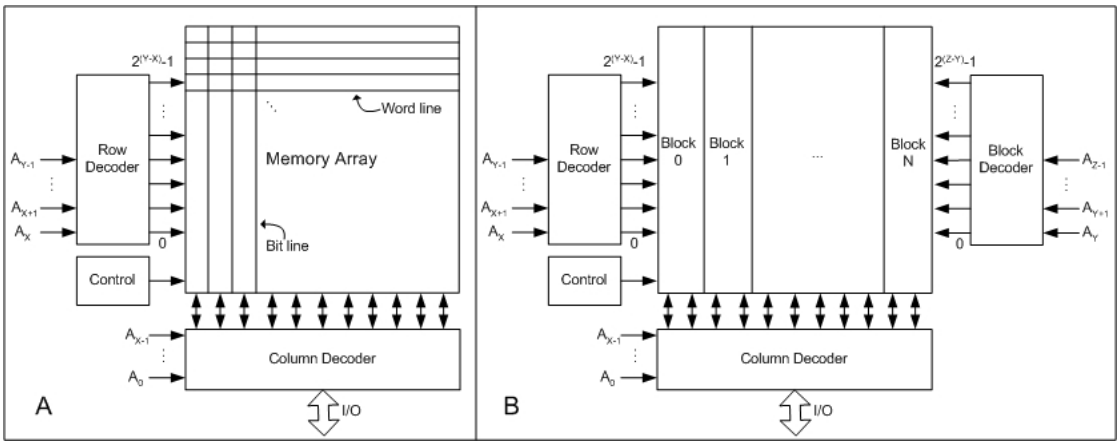


Figure 2-2. The basic memory architectures. A) The simplest example with a square memory array and row and column decoders. B) Example of a DWL structure with an extra level decoding at the block level.

2.1.2 Memory Fabrication

Virtually all modern semiconductor memory devices are manufactured on silicon wafers using fabrication processes that have been fine-tuned for decades. The fabrication starts with silicon dioxide ore (sand) which is converted through several stages before ending in the form of a doped silicon crystal ingot. The ingot is sliced into wafers, planed to be perfectly flat, and put through nearly 200 major layering, patterning, doping, and heating steps [9].

Each of these steps has some probability of introducing a defect into the device, although some are much more likely than others. The patterning process, also known as photolithography or masking, is the most critical as it define the exact size, shape, location, and orientation of the circuit devices on the wafer. It is on the bleeding edge of technology, and with current processes having more than two dozens mask steps and

device feature sizes of less than 100 nanometers, there is a tremendous possibility for the introduction of defects.

Defects are caused by four major types of contamination: particles, metallic ions, chemicals, and bacteria [9]. When particles are transferred to the wafer surface, either through the air or the growth of bacteria, they can cause opens by not allowing features to form correctly, or shorts by connecting wires together. Metallic ions cause problems by disrupting the doping and altering device electrical characteristics. Chemicals interfere in wet processing steps and can affect etching or interact to change the properties of the material.

Defects have been avoided historically by manufacturing the wafers in some of the cleanest facilities in the world. Clean room technology makes use of *high-efficiency particulate attenuation* (HEPA) filters and laminar air flow to circulate the air and filter out all particles down to sub-micron sizes. The introduction of particles is kept to a minimum by automating as much as possible and keeping people and tools out of the environment. The tools are kept in separate bays called chases and have only their interfaces exposed. When it is necessary for workers to enter the clean room, special suits are worn to reduce the release of contaminants.

There is a class rating that has been associated with the level of cleanliness that has been achieved. A Class-1 clean room has particles with a maximum size of approximately 100 nm. This is based on the rule of thumb that particles must be 10x smaller than the smallest feature size to maintain an acceptable level of physical defects. Currently, the highest density memory devices are manufactured in clean rooms with a rating of Class-0.1 or better and must be almost fully automated to avoid the

contamination brought in by humans. This is likely approaching the limit of the level of cleanliness that can be achieved, yet device feature size continues to shrink [10].

2.2 Reliability

The reliability of a device is defined by the IEEE as, "the ability of a system or component to perform its required functions under stated conditions for a specified period of time [11]." It is an important metric in memory devices since it defines the probability that a device will function correctly throughout its usable lifetime, which is typically considered to be on the order of 100,000 hours (~11 years). This section will introduce the terminology related to reliability, much of which is specific to electronic devices.

2.2.1 Characterization of Reliability

The reliability of a device is a function of its failure rate, λ. In memory devices, the failure rate is often modeled as a constant, which implies that the intervals between failures follow an exponential distribution with a probability density function of $\lambda e^{-\lambda t}$. The inverse of the failure rate gives the *mean time to failure* (MTTF), also known as *mean time between failures* (MTBF). This value is the expected time for a failure to occur. The derivation of the reliability is shown in Equation 1, and a complete summary of the terminology is provided in Table 2-1 [12, 13, 14].

$$\begin{aligned} R(t) &= 1 - \int_0^t \lambda e^{-\lambda \cdot t} \delta t \\ &= 1 - \left[1 - e^{-\lambda \cdot t}\right] \\ &= e^{-\lambda \cdot t} \end{aligned} \tag{1}$$

Table 2-1. Reliability metric summary

Metric	Function	Symbol
Failure Rate	h(t)	Λ
Mean time to failure	MTTF	$1/\lambda$
Exponential PDF	f(t)	$\lambda e^{-\lambda t}$
Exponential CDF	F(t)	$1-e^{-\lambda t}$
Reliability	R(t)	$e^{-\lambda t}$

The failure rate is not really constant over the lifetime of devices as was assumed in the derivation of reliability. It is typically represented as a collection of three failure curves that sum to form what is referred to as a bathtub curve (Figure 2-3). The relatively flat bottom of the bathtub defines the usable lifetime and allows for the assumption in the derivation.

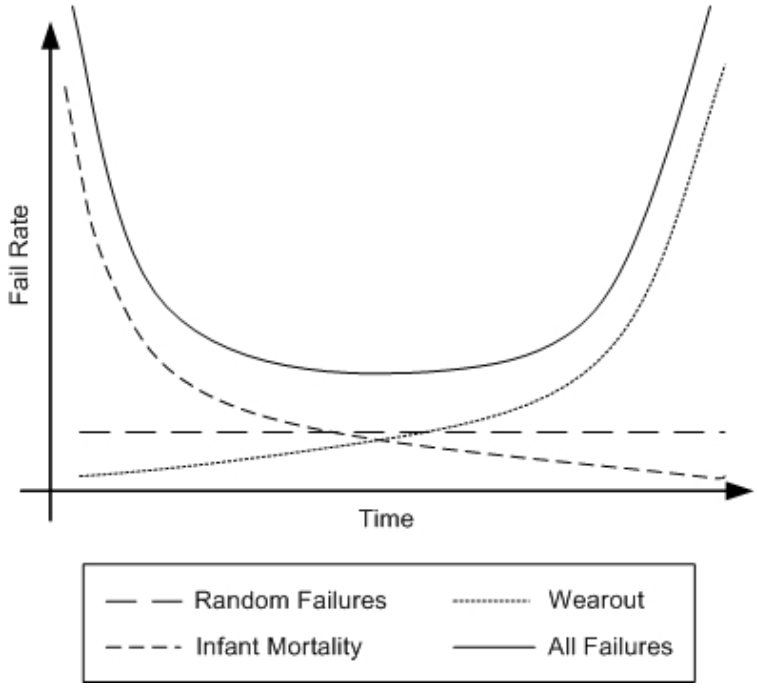


Figure 2-3. The bathtub curve used to illustrate reliability over time. The bottom of the bathtub represents the usable lifetime of components.

The steep failure rate early in the lifetime is called infant mortality and is the result of the failing of marginally functional devices. A process known as burn-in is used to accelerate the state of the devices past this point before they are distributed to customers. The second steep curve at the end of the lifetime is from the wear-out of the devices. Wear-out is of little concern as most devices are out of commission by that time. The constant curve throughout the lifetime is the effect of single, random events such as latchup or heavy ion radiation strikes.

2.2.2 Defects, Faults, and Errors

The distinction between defects, faults, and errors is not immediately clear. In this thesis the term *defect* is used to reference a physical imperfection in a device such as the opens or shorts described in the previous section. The ability of a part to work correctly when defects are present is called *defect tolerance*. If a defect occurs in a section of memory that is being accessed and results in incorrect data being returned, it is said to result in a *permanent fault*. If the fault can be corrected either by the part or through the use of software, then the part is said to have *fault tolerance*. Finally, if the fault cannot be corrected it will cause an *error* that will be seen by the user [12].

It is possible for a fault to occur without being a result of a physical defect. The extraordinarily small sizes of modern devices make them susceptible to environmental effects such as cosmic rays, alpha particles, and neutrons that can change the value of a stored bit. This type of fault is not permanent and is referred to as a *transient fault*. The next time the bit is written to it will be restored to the correct value [15].

2.2.3 Error Rates

There are typically two bit-level error rates given for a memory device, a *hard error rate* (HER) and a *soft error rate* (SER). The HER represents the frequency that

permanent faults occur, and the SER represents the rate at which transient faults arise. The rates are measured in units of *failures in time* (FIT) where 1 FIT = 1 failure/10^9 device hours [15, 16]. The device-level error rate can be calculated by simply multiplying the bit-level rate by the number of bits in the device. For example, a 1-Gbit memory with a bit-level HER of one FIT is calculated in Equation 2.

$$\mathrm{HER}_{1\mathrm{Gb}} = 10^9 \,\mathrm{bits} \times \frac{1\,\mathrm{failure}}{10^9\,\mathrm{device\ hours}} = \frac{1\,\mathrm{failure}}{\mathrm{hour}} \tag{2}$$

The HER of a device is affected somewhat by the level of maturity of the process technology, but it is most directly correlated to the reduction is size of devices [17]. The HER has increased slightly with each generation, but it has remained relatively low overall at approximately one to two percent of the total error rate [18]. This is due in large part to the precision with which silicon devices are fabricated.

Soft errors represent the majority of the errors that occur. Unlike the HER, the SER is mostly dependent on the operating voltage, charge collection efficiency, and cycle time of the device, as well as the level of radiation present in the device. The last several generations of SRAM and DRAM memories have seen reductions in charge collection efficiency and the impact of radiation through improvements in fabrication processes, both of which help to lower the SER. They have also had continuous reductions in the operating voltages and cycle times, both of which result in an increased SER. The outcome of these counteracting influences has been a slight decrease in the bit-level SER [18].

In devices such as DRAM, the effect of the cycle time on the SER must be averaged over both the refresh cycle time and the read/write cycle time. In a 16-MB DRAM chip the difference in the refresh cycle time of 15 μs and read/write cycle time of

200 ns leads to a 50x increase in SER [16]. To correctly calculate the overall SER, the ratio of each cycle time must be estimated and multiplied by the corresponding SER.

The cycle time at which the memory has the data refreshed, or scrubbed, also plays an important role in determining the overall error rate in the case that fault tolerance is implemented. If a correctable number of bits have become faulty as a result of permanent faults and are not rewritten, they have the possibility of lining up with transient faults to cause an error. The rate at which the soft errors are removed from the system is referred to as the *scrubbing interval* and must be accounted for in determining overall system reliability.

The level of radiation present has a significant impact on the SER. Experiments have shown more than a 10x increase in the SER of DRAM devices in going from sea-level to an altitude of 10,000 feet and another 10x increase in going to commercial airline flight altitude [15, 19]. The increase is the result of reduced protection from the atmosphere. For this reason, any devices that will be used at high altitudes, and especially in space, must have much more protection than those used at terrestrial levels.

At the device level, both the HER and SER have shown some cause for concern as the size of memories continues to expand at a rate of up to 60% per year [20]. Even if the bit-level error rates were to remain constant going forward, the error rate at the device level would become unacceptable. For example, an SRAM memory manufactured on 90 nm technology has a bit-level HER on the order of 10^{-5} FIT and a SER of 10^{-3} FIT at terrestrial levels. For a 1-MB array, now common as a cache in microprocessors, the device-level error rate is 8472 FIT with an acceptable MTTF of 13.5 years. If the cache size were to increase to 128-MB with the bit-level error rate unchanged, however, the

MTTF would drop to 38 days, which is clearly unacceptable and would require some form of fault tolerance [18].

2.3 Fault Tolerance in Memory

As the size of memories grew to millions of bits, it became impractical for manufacturers to fabricate fully functional parts with a high enough yield to be profitable. This led to the introduction of static redundancy where redundant rows and columns were included in the design and could be switched in to repair defects before shipping the part to the customer. In today's memories of a billion or more bits, static redundancy has become a necessity for producing functional units.

To handle the errors that occur after the part has shipped, dynamic redundancy is used. The most common form of dynamic redundancy is the use of *error correction codes* (ECC). ECC's allow errors to be detected and corrected before the data are returned from the part. These and other forms of fault tolerance will be introduced in this section and will form the basis for the hierarchical fault tolerance mechanisms that will be proposed for future memories.

2.3.1 Device Redundancy

Device redundancy is the inclusion of spare devices in the design that can be swapped with faulty devices to maintain functionality. As mentioned above, it has become an important technique for producing modern memories with an acceptable yield. Even with the precision of lithography-based fabrication, defects caused by contaminants and tiny variations in process steps have become inevitable. All types of memory from DRAM to microprocessor cache to flash are built with extra rows and/or columns that can be switched in when defects are found.

The first step in making a repair is to determine the location of the defects in the memory through the use of fault diagnosis algorithms. The specific types of defects that can occur vary by memory type, but the algorithms used are similar in all cases. Fortunately, the test cases are much simpler than those used for logic parts and work by performing a sequence of reads and writes that vary in direction, bit pattern, and timing. The goal is to locate bits that are stuck at high or low levels, lines that are bridged together as a result of a short circuit, lines that are open circuited, and bits or lines that are coupled together [12].

The patterns are applied with automated test equipment (ATE) and/or BIST circuitry integrated into the memory. ATE are very expensive tools capable of applying comprehensive sets of diagnostic test patterns to the memories at high speeds. As the size of the memories has grown, the ability to test for all faults with ATE has become too costly in many cases, and impossible in others, as some embedded memories cannot be accessed directly through the external pins. It has been estimated that for 1-Gbit memories, the test cost is roughly 85% of the manufacturing cost when done on standard ATE machines, and the ratio continues to increase [21].

To reduce the test costs to profitable levels and to gain access to embedded memories, many designs have been developed that allow for BIST. With BIST the memory can test itself by applying the necessary patterns and returning the results, which allows for much simpler and less expensive ATE. There are many techniques used for generating the patterns, but most make use of *linear feedback shift registers* (LFSR's) or microcode programming. Advance compilers have been designed to assist in the design

of the BIST architectures, which typically require less than one percent of the total chip area [22-24].

When the faults have been detected, another algorithm is used to determine which devices need to be repaired and if there is enough redundancy available. The complexity of an optimal repair algorithm has been shown to be NP-hard, so heuristic approaches are often used. The ATE is usually capable of determining optimal or near-optimal repair strategies. In cases where the repair is done on the part with BISR circuitry, however, greedy algorithms may be necessary to reduce test time and circuit complexity.

The redundant devices are swapped in by one of two reconfiguration methods. If the reconfiguration is done by the manufacturer, it is known as a *hard repair*. A hard repair is done by physically blowing fuses on the device through the use of lasers or electrical currents. If the repair is done in the field, such as in a BISR circuit, it is known as a *soft repair*. A soft repair is done by programming non-volatile devices on the parts such as EPROM's or flash cells to connect the redundant devices.

Device redundancy does have some limitations. First, when a fault occurs, the data that are in error cannot be corrected. The repair process must also be done non-concurrently, or when the memory is not in use, which means the memory must be cleared of its current contents before it can be tested. This requires that the current memory contents be copied to spare arrays before the testing if it is not to be lost. The process is also relatively slow as it is necessary to scan the entire memory, devise a repair strategy, and program in the redundant devices. Overall, device redundancy is an important means of fault tolerance, but it is not sufficient.

2.3.2 Error Correction Codes

ECC's are another important form of fault tolerance. The term is typically used to refer to linear block codes that encode redundant information about the data stored in the memory into what are referred to as codewords. The number of valid codewords is then a subset of the possible codewords. The benefit to this method is that it can be used concurrently, or when the memory is in use. It can also be integrated in many devices for a tolerable overhead in term of latency and area, and in some implementations it is capable of correcting errors.

The simplest form of ECC is the parity check code in which an extra bit is added to each word and set to give either an even (even parity) or and odd (odd parity) number of 1's. This allows for the detection of all errors in which an odd number of bits are at fault. More advanced ECC's have been developed that are capable of not only detecting errors, but also correcting them. This class of codes requires that the valid codewords be separated by a minimum number of bits given in Equation 3, where *d* is the minimum distance, and *y* is the number of errors that are correctable.

$$d \geq 2y+1 \tag{3}$$

The most common implementation is a class of Hamming code known as a *single-error correcting, double-error detecting* (SEC-DED) code. The Hamming SEC-DED code uses $2^n - 1$ total bits to encode $2^n - 1 - n$ information bits. An example is the Hamming (7, 4) code in which three parity bits are added to each 4-bit word to form a 7-bit codeword $(i_1 i_2 i_3 i_4 p_1 p_2 p_3)$. The parity bits are generated in such a way as to distinguish the location of the bit in error after decoding of the received codeword in what is referred to as the syndrome. The equations for the Hamming parity and

syndrome generation are given in Table 2-2 [12]. A complete overview of ECC's is given in [25, 26].

Table 2-2. Hamming (7,4) parity and syndrome generation equations

Parity generation	Syndrome generation
$p_1 = i_1 + i_2 + i_3$	$s_1' = p_1' + i_1' + i_2' + i_3'$
$p_2 = i_2 + i_3 + i_4$	$s_2' = p_2' + i_2' + i_3' + i_4'$
$p_3 = i_1 + i_2 + i_4$	$s_3' = p_3' + i_1' + i_2' + i_4'$

CHAPTER 3
NANOSCALE MEMORY DEVICES

The term *nanoscale* is generally used to refer to structures that have dimensions smaller than 100 nm. Nanoelectronics, or equivalently molecular electronics, then refers to any electronic device with critical dimensions of less than 100 nm. There is one exception to this; the terms are limited to novel technologies of this scale. The restriction is used to exclude CMOS technology, which is technically below the 100 nm mark in modern processes.

This chapter presents the goals, both near-term and long-term of the semiconductor industry, as well as the upcoming challenges in scaling silicon-based CMOS devices to ever smaller sizes. Several promising molecular electronic technologies are then introduced that are most applicable to memory devices and may go on to become successors to CMOS technology.

3.1 Present and Future Scaling Challenges

The International Technology Roadmap for Semiconductors (ITRS) is a document that is based on predictions of the state of semiconductor technology for approximately 15 years into the future [27]. It is compiled by a cooperative effort of organizations in every aspect of semiconductor technology from manufacturers to universities. The document tracks the device characteristics expected at each generation of technology and defines the basic requirements necessary to achieve them.

The requirements are grouped into thirteen different areas related to the physical properties, manufacturability, and testability of the devices. The document also outlines any roadblocks foreseen in each of the areas that may inhibit the realization of each generation of technology. The roadblocks are categorized as being yellow, meaning that a manufacturing solution is known but not yet implemented, or red, which means that no manufacturing solution is currently known.

The challenges in the red category form what is known as the "Red Brick Wall" that represents the major hurdles that stand in the way of continuing the exponential growth of Moore's Law. In the 2001 edition of the ITRS the "Red Brick Wall" was predicted to occur in the 2005 to 2006 timeframe. According to the update provided in 2003, some of these obstacles have been overcome, but only enough to push the wall out to about 2007. Of the dozens of near-term challenges that remain, some of the hardest are dealing with the exponential increases in leakage current, making masks for the lithography process, developing metrology tools to perform critical measurements, and building the interconnect networks on the chips.

Assuming that engineers are able to work quickly to develop the solutions necessary to maintain the growth predicted by the ITRS, the critical dimension in devices will be on the order of 20 to 30 nm in only a decade (Table 3-1). Devices of this scale will likely be fabricated quite differently than they are today. They will presumably be built using a bottom-up approach in which large, regular arrays are self-assembled rather than lithographically drawn. If lithography-based CMOS processes are still implemented, they will be integrated with nanoelectronics in what has been referred to as mixed CMOS/nano designs or NoC. NoC circuits will consist of CMOS as the

underlying structure that connects blocks of nanoscale structures together while providing only a small degree of functionality [28].

Table 3-1. ITRS DRAM production product generations

Year of Production	2004	2007	2010	2013	2016
DRAM ½ Pitch (nm)	90	65	45	32	22
Cell Area (mm^2)	0.065	0.028	0.012	0.0061	0.0025
Functions/chip (Gbits)	1.07	2.15	4.29	8.59	34.36
Gbits/cm^2 at production	0.97	2.22	5.19	10.37	24.89

3.2 Nanoscale Building Blocks

To date, there have been no large-scale memory devices constructed from molecular electronics, but many of the necessary building blocks are coming into place. There have been prototypes demonstrated of nanoscale wires, field-effect transistors, junction transistors, diodes, and switches [2]. There has also been some success in small-scale integration of these devices into working circuits. For example, Hewlett Packard demonstrated a 64-bit memory array that was constructed from a cross-bar configuration of platinum wires 40 nm in diameter. Much research is still necessary to continue refining the building blocks and discovering ways to integrate before usable products can be produced.

3.2.1 Nanoscale Wires

There first useful building block was the discovery of nanoscale wires. The two most promising are the carbon nanotube (CNT) and the semiconductor nanowire (SNW). The CNT is composed of a cylinder of carbon atoms. The wall of the cylinder can be as thin as a single monolayer, or can consist of several layers. The diameters range from a single nanometer to several nanometers, and the length can be on the order of millimeters. It is also possible for the tube to have either metallic or semiconductor properties, depending on the helicity of the carbon lattices. CNT's are typically formed

using catalyzed *chemical vapor deposition* (CVD), and unfortunately, the type of structures produced is a random event. It is not currently possible to uniquely construct metallic- or semiconductor-type nanotubes [29, 30].

SNW's refer to nanoscale wires composed of semiconductor materials in general, but the most common are Si, Ge, and GaAs. They have many of the same properties as CNT's. It is possible to construct the wires with diameters as small as 3-nm and lengths on the order of microns. The greatest benefit the SNW's have over their carbon counterparts is that, although the same method of CVD fabrication is used, it is possible to precisely control the composition, diameter, length, and doping of the structures. It is also much easier to control the parallel production of SNW's [4, 29].

3.2.2 Transistors, Diodes, and Switches

In addition to nanoscale wires, it is also necessary to have active devices as building blocks. A wide variety of transistors, diodes, and switches have been constructed, albeit on small scales. The devices have been limited to proof-of-concept using fabrication techniques far too complex for mass production, but they are the first steps towards characterizing the devices and developing necessary models.

Diodes and FET's have been constructed by using the CNT's and SNW's described above, which can be easily doped to be either p-type or n-type [29, 30]. In one implementation, a single-walled CNT or a SNW is used as the channel of a FET and is placed between two metal electrodes. The molecular FET's, benefited by ballistic transport properties, have been shown to be comparable in performance to similar CMOS devices [31]. In another example, a multi-walled CNT is side-bonded to electrodes and an FET is formed by selectively thinning portions of the tube by removing the outer shells of the CNT [30]. Both diodes and FET's have been fabricated by simply crossing

oppositely doped CNT's or SNW's to form p-n junctions at the intersections [2, 29]. Figure 3-1 graphically depicts each of these constructs.

Switches have been developed in much the same way as the diodes and the FET's. Most implementations involve crossing nanoscale wires and using the intersection as a simple two-terminal switch. The benefit of such switches is that the storage area for a bit is merely the intersection of two wires. No periphery is necessary to program or access the bit. A conventional device such as an SRAM or a DRAM requires as much as 100x larger footprint for a programmable crosspoint when built with equivalent feature sizes [2].

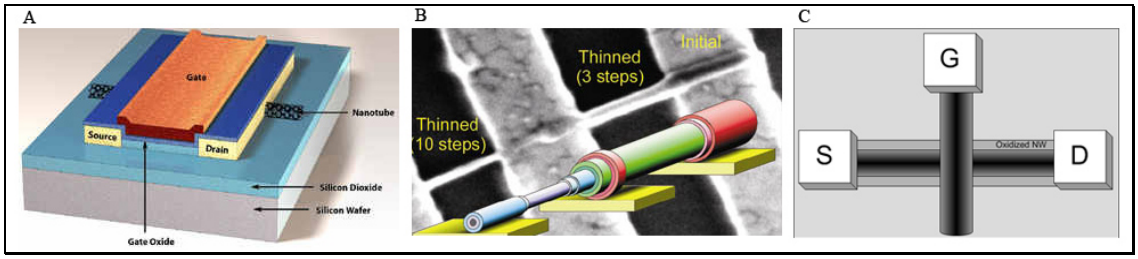


Figure 3-1. Nanoscale FET implementations. A) depicts a CNT being used as a FET channel. B) shows a multi-walled CNT with various number of shells removed. C) represents an oxidized SNW gated by a CNT.

3.3 Molecular Electronic Architectures

Currently, the greatest challenge in nanoelectronics is the ability to efficiently build and configure reliable architectures out of the core building blocks discussed in the previous section. There are many issues related to this including designing and simulating an architecture composed of billions of gates, fabricating a device that adheres to the design, and testing the circuits and repairing defects.

The design and simulation of the nanoscale circuits is beyond the scope of this thesis, but the regularity of memory devices should make this challenge manageable and certainly much simpler than it will be for logic devices. The complexity of the

fabrication process will require tradeoffs with the testability of the design. If the designs remain fully deterministic, as they are today, the testing process will remain relatively simple, although steps would have to be taken to find methods of testing such a large number of bits in a reasonable amount of time. If the designs become completely random in structure, the fabrication process will be greatly simplified and the testing process will become complex. It will be necessary to map out the device post-fabrication and determine if it is a usable part.

3.3.1 Crossbar Circuits

The crossbar architecture appears ideally suited to memory devices made from nanoscale wires. In this design a simple 2-D mesh of wires is constructed by lining up a number of wires in parallel to form a plane, rotating the plane by 90° and repeating the process. There have been a number of techniques developed for doing this, including fluid flow directed, electric field directed, biologically assisted, catalyzed growth, and nanoimprinting assembly methods, among others [2, 28, 29].

In the fluid flow and e-field methods, the wires are lined up by the flow of fluids or current, respectively, and the spacing is determined by the rate of the flow. Biologically assisted processes use complementary DNA strands as “smart glue” to selectively bind components and move them in place. In the catalyzed growth methods, a laser ablation or CVD is used to direct the growth of nanodroplets in one direction. Finally, in nanoimprinting, a mold is constructed using e-beam lithography and then wires are cast into the mold.

Nanoimprinting is the only known method of making irregular arrays of wires in a controlled fashion. The main drawback is that it has a larger pitch requirement than the

other methods. Notably, it was the method used by HP to make the prototype 64-bit memory mentioned above.

There have been several methods devised to store the information at the wire intersections of the crossbars. HP patented a method in which molecules are sandwiched between the wires. The molecules can be electrochemically modified to vary the tunneling barrier between the two wires [32]. IBM patented a similar idea except that magnetic structures are placed between the devices [33]. In a method developed at Harvard, there is nothing between the wires. The wires are suspended slightly apart from one another at the intersection. To program the device, electromechanical forces are used to attract the wires closer together. They are then held in this position by van der Waals forces at a second potential energy minimum [34].

Using crossbar circuits and storing the state of the bits at the intersections does have some drawbacks. First is the inability to create aperiodic structures, which would have to be configured into the meshes post-fabrication. Also, the diode-resistor logic available does not allow for signal gain. The lack of gain can be coped with by including molecular latches in the periphery circuitry and by keeping the size of the arrays small and using them as building blocks for larger circuits [28].

3.3.2 Micro/Nano Interface

At present, the only way to make use of the crossbar arrays is to connect them to a microscale interface at some point. This is not a trivial task because even if a microscale wire could be attached to each nanoscale wire, such a design would defeat the purpose by requiring microscale pitch between wires. Most designs have dealt with this requirement by including demultiplexor circuits in which a logarithmic number of microscale lines are used to interface with the given number of nanoscale lines.

The approach taken by HP in making a demultiplexor was to make connections between the microscale and nanoscale lines with a stochastic chemical process that has a 50% probability of making a connection. It was shown that by increasing the number of address lines necessary to address *n* data lines from $\log_2 n$ to approximately $5\log_2 n$ there is a reasonable probability that each bit can be uniquely addressed [35].

Another approached suggests that a decoder may be imprinted lithographically during fabrication across which the nanoscale wires are placed. This design also made use of a sparser decoding scheme based on two-hot coding to reduce the portions of the array that would be lost in the case of an error in the decoding circuitry [4].

A third method has also been proposed in which the decoder is actually integrated into the crossbar architecture. The outer edge of the array is fabricated to allow connections to be made to microscale devices and is programmed at higher voltages than the interior of the array. The decoder is then programmed into the periphery during the testing phase of the circuit [28]. A diagram of each of the demultiplexor designs in shown in Figure 3-2.

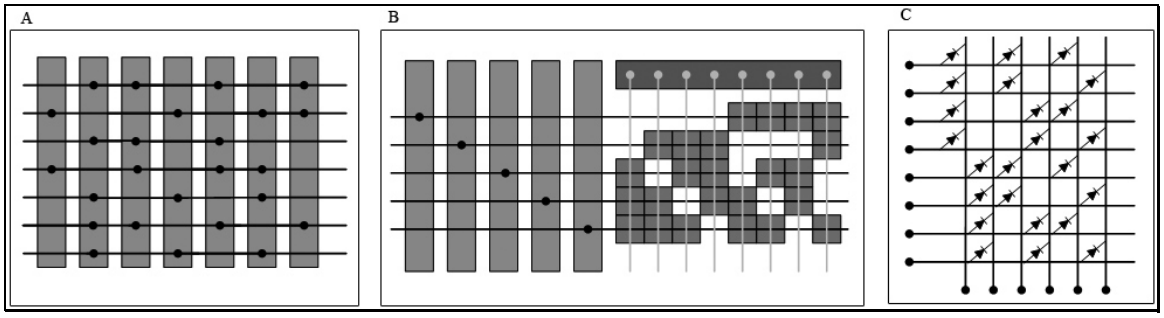


Figure 3-2. Demultiplexor designs. A) Random connections made between microscale address lines and nanoscale data lines. B) Lithographically drawn decoder pattern. C) Demux being encoded into the nanoscale wires on the periphery of the memory.

3.3.3 Hierarchical Architectures

There are a number of compelling reasons for designing the overall architecture in a hierarchical fashion where smaller arrays exhibit locality and work as independent pieces of larger constructs. As mentioned above, the size of the arrays is limited by the lack of gain in the circuits and requires that multiple smaller arrays be connected to allow for signal latching and restoration. This is also true for interconnect purposes as switching frequency continues to increase and the latency in routing the signals becomes a limiting factor. A final argument for smaller arrays is a result of the lack of control in fabricating self-assembled circuits. Smaller design should lead to greater accuracy and fewer defects.

It also makes sense to use a hierarchical architecture from a design perspective. It will simplify the process greatly to have multiple levels of abstraction. Again, this is most applicable to logic designs, but it will play a part in memories as more logic is integrated either for computational purposes or to simply perform all the necessary mapping, testing, and repair processes on the arrays.

3.3.4 New Reliability Concerns

It has been implied throughout this chapter that self-assembled molecular electronics will have a number of new concerns with regard to defects during fabrication. The first problems stemmed from the inability to control the properties of CNT's. This was exacerbated by the further inability to deterministically control the alignment and spacing of the nanoscale wires in the construction of the crossbar arrays. The limits of cleanroom technology will also come into play as devices sizes shrink to on a few nanometers. The consequence of all these factors is a defect rate expected to be on the order of 5% [2, 5, 28].

Even if the initial defects are dealt with during the manufacturing process and a working part is completed, there will inevitably be a much higher rate of both permanent and transient faults during the runtime of the devices. The extremely small size of the structures will make them more susceptible to the random single-event effects from high-energy particle strikes. The only way to create reliable parts will be to integrate fault tolerance circuits into the designs to deal with the dynamic faults.

CHAPTER 4
HIERARCHICAL FAULT TOLERANCE FOR NANOSCALE MEMORIES

This chapter introduces hierarchical memory architectures with several possible forms of fault tolerance at each of the levels. The goal is to develop generic memory architectures with new, more powerful combinations of fault tolerance mechanisms that take advantage of the intrinsic properties of molecular electronics to overcome what is expected to be a very high defect rate. Several possible implementations will also be suggested based on previous work in the area.

4.1 Benefits of a Hierarchical Approach

It makes sense to use a fault tolerance scheme that is hierarchical given that the underlying memory structure is expected to be hierarchical. In fact, it has been found to be beneficial to do so. The defects that occur in a memory device have been related to the "birthday surprise" problem [36]. The problem calculates the expected number of independent trials necessary to have one of a set of equally likely results occur twice. In the case of birthdays, the outcome is two persons sharing the same birthday. The expected value in this case is a surprisingly low 24 people. The value is also low for memories if only single level fault tolerance is used. It was shown in [37] that a 16-Mbit memory chip with 1,000 random failures has greater than a 98% chance of having two errors line up.

The benefit to having a second level of fault tolerance is that it effectively removes the first sets of "matches." For example, if a SEC-DED code is used as a first-level mechanism and row redundancy as a second, then when two bits line up in a word to

cause a failure the row can be replaced by the second level of fault tolerance. It is as if the persons with matching birthdays were taken out of the room. The effects of each fault tolerance mechanism work constructive to perform better together than either would individually. This was termed synergism in [37] and is the basis for the configurations to be presented in the remainder of the chapter.

4.2 Proposed Memory Organization

The architecture used in this thesis is based on the specific set of conditions applicable to molecular electronics that were specified in Chapter 3, and which can be summarized as follows.

- The memories will be built hierarchically from relatively small, regular arrays of nanoscale wires.
- The arrays will be self-assembled with a high degree of randomness and an error rate on the order of 5% [2, 5, 28]. The arrays must function even with defects present.
- Locality is important; the fault tolerance mechanisms should work independently at the array level and only signal higher-level structures when local resources are exhausted.
- A high degree of configuration and testing will be necessary post-fabrication that will necessitate logic structures in the memory arrays.

4.2.1 Architecture Overview

The basic structure of the memory organization to be considered is shown in Figure 4-1. Much as the DWL architecture added an extra level of word line decoding, this hierarchical architecture extends it further to allow for more sublevels. The lowest level, which will be referred to as level-0, consists of small, regular arrays of nanoscale wires such as CNT's or SNW's. There is also a logic component that will be called a *processing node*–PN0 in this case. The processing nodes are used to map out the

structure of the arrays during the testing and configuration phases, and later they are used to implement the fault tolerance mechanisms in the array.

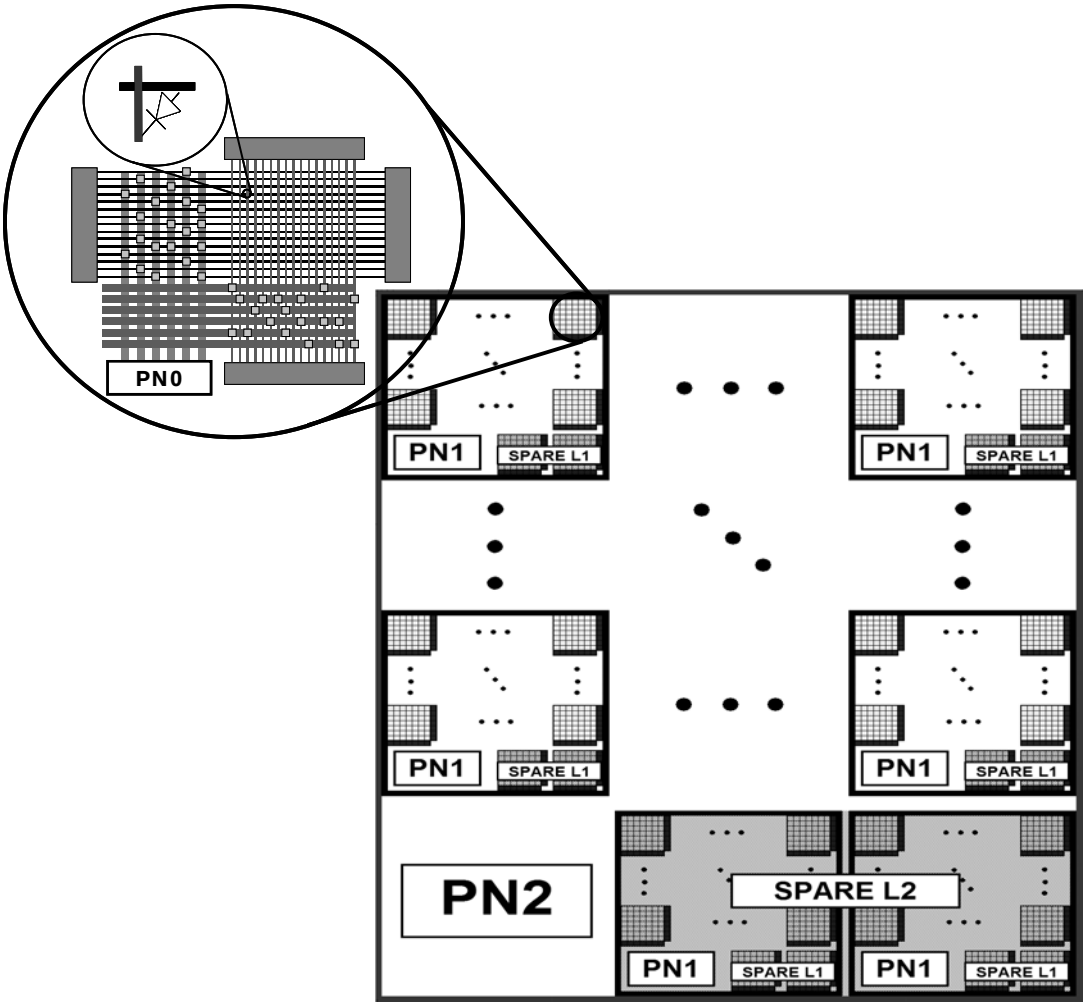


Figure 4-1. Proposed hierarchical memory architecture. The lowest level consists of arrays of nanoscale wires with a processing node used for both static testing and dynamic fault tolerance. An arbitrary number of higher levels can exist, each consisting of arrays of lower level structures and a processing node.

The next higher level, level-1, is composed of an array of level-0 structures. Again a processing node, PN1, is present to carry out the mapping necessary during the testing and configuration of the part. It is then used at runtime to implement fault tolerance mechanisms throughout the life of the part. If there were another level present, it would be referred to as level-2 and would consist of arrays of level-1 structures. The hierarchy

can continue in this fashion, which is similar to previous hierarchical memory designs [38, 39]. It has been shown that in terms of timing latency the optimal number of levels in the hierarchy increases slightly with memory size and inversely with minimum feature size, but is on the order of five to ten [40].

4.2.2 Possible Implementation Strategy

As mentioned in Chapter 3, the higher level structures need not be implemented with nanoscale devices. It may be the case, at least initially, that NoC architectures are employed and lithographically-based CMOS structures are used in conjunction with nanoscale arrays. The CMOS structure would provide the logic functionality necessary for the mapping and fault tolerance, as well as basic infrastructure such as power, ground, and the clock signal.

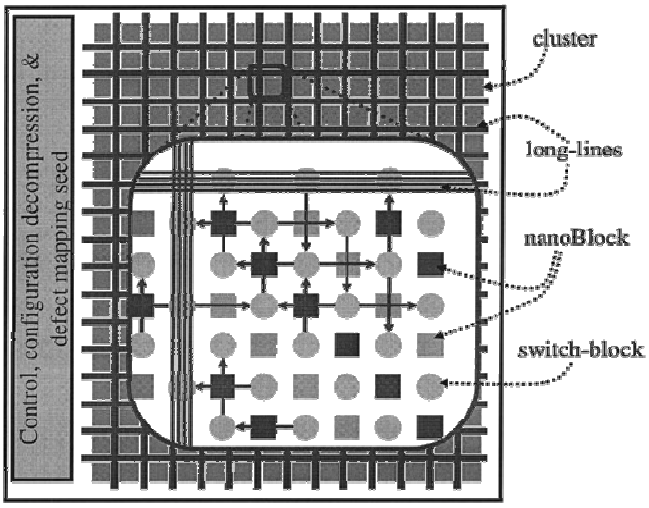


Figure 4-2. A diagram of the nanoFabric architecture

Examples of such a NoC approach are the nanoFabric architecture [3], shown in Figure 4-2, and the similar array-based approach in [4]. In these designs, the lowest level consists of arrays of nanoscale wires. The next level then connects these arrays into a larger array called a cluster. The top level is an array of clusters with long nanoscale wires connecting the clusters. Inside each cluster, the sub-arrays allow for both

programming and routing to be done entirely at the nanoscale. Through simple routing techniques, arrays can not only be used as a memory device but can be configured to implement full logic functionality.

4.3. Fault Tolerance Mechanisms

The basic static and dynamic fault tolerance mechanisms in use today were introduced in Chapter 2. It is assumed, however, that these will not be sufficient for nanoscale memories. More advanced techniques are available and are presented in this section.

4.3.1 Hierarchical Built-in Self-Test and Self-Repair

The need for BIST and BISR circuits is suggested by all of the conditions outlined in the previous section. The underlying assumptions that molecular electronic circuits will be very large, have high bit error rates, and require a great deal of functionality to be configured post-fabrication suggest that ATE alone will not be capable of testing and configuring the structures. The integrated mechanisms for test and repair are also necessary to contend with the dynamic defects that occur throughout the lifetime of the devices. They function synergistically with ECC's and other forms of fault tolerance by replacing defective devices that in many cases do not cause faults individually, but put a portion of memory at high risk of faults by lining up with future defects.

In the proposed design, the reconfiguration processes are similar both during the device testing and at runtime. The BIST/BISR circuitry used to map out the functionality of the arrays is integrated into the PN's and continues to be employed at runtime when defects are detected either through error detection circuits or at scheduled servicing intervals.

The reconfiguration options of the repair circuits are limited by the speed of the interconnections in the devices. It is not feasible to swap in a spare row from a distant array, for example, as it would require a decrease in the system clock rate. The hierarchical approach handles this by allocating spare devices to each array and allowing the local PN to conduct the sparing process. It is not until the local PN has exhausted the available resources that a signal is sent to the next level up in the hierarchy. When this occurs, the entire lower-level structure may be swapped with a spare.

4.3.2 Advanced Error Correction Codes

ECC's were introduced as a method of dynamic fault tolerance that has been implemented in some modern, high-reliability memories. The most common codes are capable of either detecting errors or possibly correcting one error. There are much more powerful linear block codes possible, but because of the complexity of the encoding process double-error-correcting and triple-error-detecting is the most powerful one considered. A summary of the original codes, as well as several additional codes that are considered in this thesis is given in Table 4-1.

There are other forms of ECC's that do not fall into the category of linear block codes, such as Reed-Solomon, Golay, Turbo, and RAID codes. Normally, these forms of error correction are not applied to memory devices. The one exception is RAID parity, which has been used by IBM in high-reliability Chipkill server memory [43, 44]. The benefit of RAID is that, unlike typical ECC's, it is designed for inter-block redundancy. When used in conjunction with ECC's, it allows for one more than the number of bits correctable by the ECC to be tolerated. The memory is configured with an additional level-1 array in each row or column of level-1 arrays. The additional array is used to store the parity of each bit in the other arrays. In this way, even if an entire array were to

fail, the data in error could be reconstructed by taking the parity of the other arrays – so long as the other arrays do not contain uncorrectable errors at the same relative location.

Since the RAID parity is the only alternate form of dynamic redundancy typically applied to random access memory devices, it is the only one that will be included in this thesis. The additional requirement for the use of RAID redundancy is knowledge of more than the number of correctable bits being in error. This requires not only that an ECC be implemented in addition to the RAID code, but that the ECC is capable of detecting at least one more error than it is able to correct. For this reason, only the codes in Table 4-1 that are marked with an asterisk* can be used in conjunction with the RAID parity.

Table 4-1. ECC summary for the proposed architecture.

Code	# Redundant Bits
Parity*	1
SEC	$\log_2 N + 1$
SEC-DED*	$\log_2 N + 2$
DEC	$2\log_2 N + 2$
DEC-TED*	$2\log_2 N + 3$

CHAPTER 5
MEMORY SIMULATION AND RESULTS

The complexity of the fault tolerance mechanisms implemented in the proposed architecture ruled out the derivation of analytical solutions to describe the performance of the systems. The size of the memories involved also made the Monte-Carlo simulation of the systems non-trivial. This chapter describes how the simulations of the various architectures have been accomplished and validated. The results of the simulations are included and analyzed with the goal of providing insight into optimal memory configurations in terms of reliability for a given amount of redundancy overhead.

5.1 Simulator Architecture

The concept of developing simulation tools for analyzing the performance of memory architectures is not new. A variety of software packages are available for running many types of simulations. These packages are limited, however, in their support for simulating the states of very large memories. The requirement that multi-gigabit and even terabit sized memories need to be simulated in a reasonable amount of time, not once, but for thousands of iterations, rules out the generic simulation packages that are designed more for modeling the states of tools in a factory than the states of billions of nodes.

To obtain the level of efficiency necessary, a very specialized simulator that implements only the desired functionality has been developed. The Perl programming language was chosen for the development environment since it offers the benefit of efficient, easy-to-use data structures, while maintaining adequate speed at runtime. Perl

also provides a means of easily implementing a graphical interface with the Tk expansion module and eliminates the need to port and/or recompile the simulator code to various platforms that might execute the simulations. The individual components are described in detail in the following sections, and the source code has been included for reference in Appendix B.

5.1.1 Simulator Components

The process of simulating the memories can be broken down into three steps. The first is the generation of memory configuration parameters for the simulator, including the size and arrangement of the memory, the hard and soft error rates, and the fault tolerance mechanisms to be supported. The next step is the simulation of the memory for a reasonable lifetime, which is typically considered to be 100,000 hours. The final step is the analysis of the simulation results.

The bulk of the complexity lies in the simulator itself, which can be further broken into three steps. The first of these is the generation of failures throughout the lifetime of the memory. It would be intractable to store the complete state of all bits being simulated. Instead, a form of statistics called order statistics is used to generate, in order, the occurrence of failures for the entire lifetime of the part before any simulation takes place. The second step is the process of simulating and recording the actions taken by the memory controller to each of the failures. The final step is integrated into the simulation process and involves the check-pointing of the state of the memory at predefined polling intervals for off-line analysis during the results analysis.

The simulator is constructed in a modular fashion based on the categories just described. The programs for input parameter generation, simulation, and analysis are implemented as shown in Figure 5-1. Each module is a standalone program and

communication occurs through the use of files, which allows for the output of each to be saved and reused in subsequent analyses.

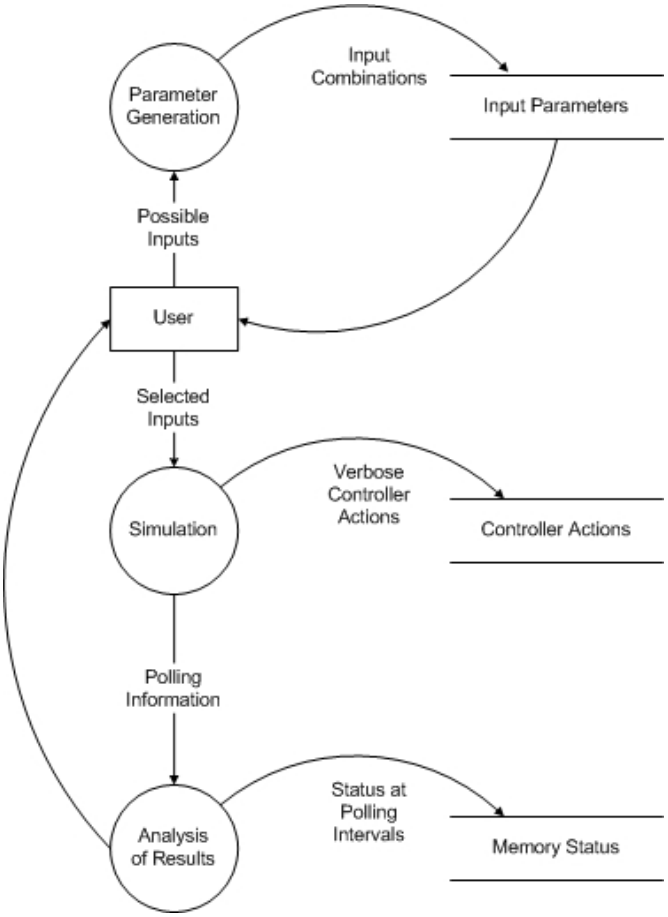


Figure 5-1. High-level simulator data flow diagram.

5.1.2 Simulation Interface

There are two interfaces provided to the simulator. One is a command line version available for use in running batch processes of several configurations for which input parameters have already been chosen and inserted into a batch file. The other is a graphical version that provides tools for configuring the parameters, executing the simulation, and analyzing the results.

A screenshot of the graphical interface is shown in Figure 5-2. The top section of the interface provides a text area that is used for displaying results of simulations and

configuring the batch execution scripts. Basic editing and searching capabilities are supported, as well as a tabbed notebook feature that allows for multiple files to be opened.

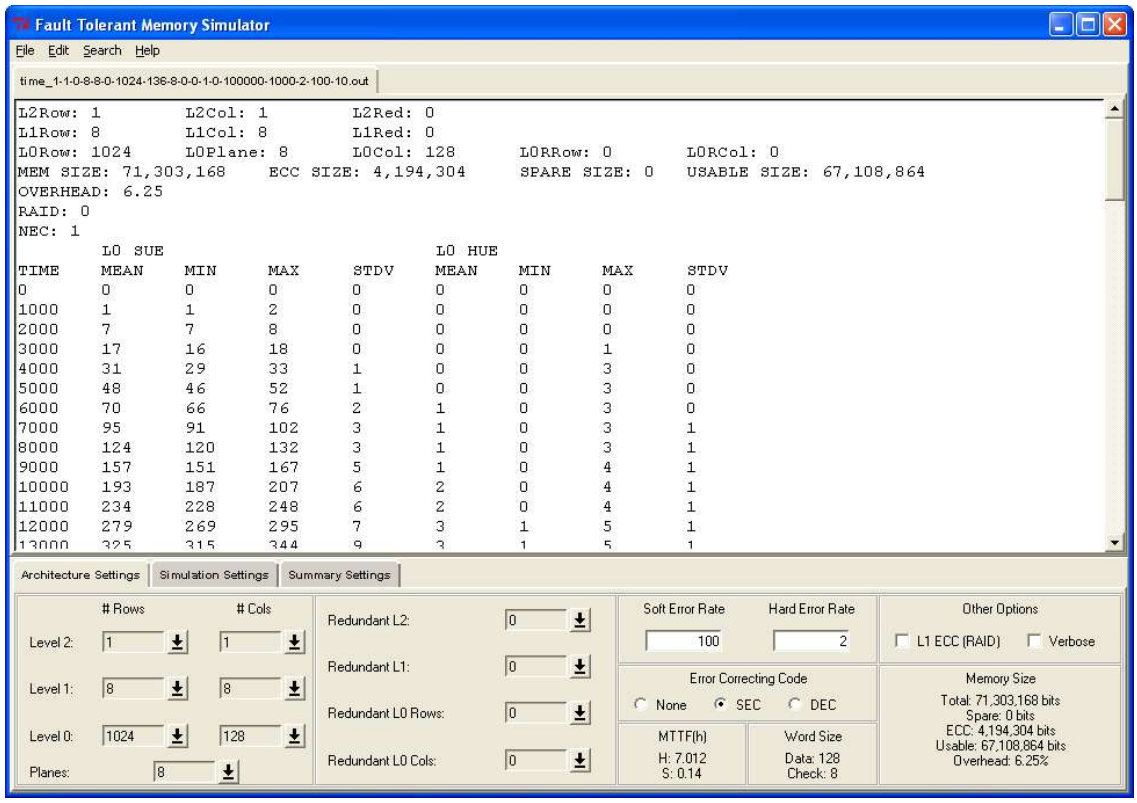


Figure 5-2. Simulation interface with sample results

The lower portion of the interface is also tabbed. The first tab contains settings widgets used for configuring the architecture of the memory to be simulated, which includes all of the input parameters given in Table 5-1 (see 4.2.1 for the Architecture Overview). There is an additional option for enabling verbose output, which provides an increased level of detail during the simulation. There are also a range of labels that are dynamically updated to reflect the overall memory structure described by the input parameters. The labels display the distribution of bits in the memory, as well as the memory-level MTTF based on the hard and soft error rates that have been defined. The tab for simulation settings is much simpler; the only inputs are for the lifetime of the

memory, the polling interval, and the number of complete trials to be executed. The final tab is used for displaying summary information in the form of statistical box-plots, which are useful for comparing the results of multiple different configurations.

5.2 Simulation Component Details

This section covers each of the simulation components in greater detail, including the algorithms implemented and the inputs and outputs.

5.2.1 Parameter Generation

The first program in the simulator is used for input parameter generation. It was designed to ease in the generation of the many input parameter values to the memory simulator. It takes as input a listing of possible simulator inputs from Table 5-1 along with a range of values for each. Then, based on criteria such as percentage of redundancy overhead and possible fault tolerance mechanisms, the program generates all possible input configurations. The algorithm simply iterates through all combinations of the values for all parameters. Then, for each combination found that is within the search boundaries, a line is written to the output file that contains all the parameter information.

The output file from the program shows all possible combinations of the parameter values that fit the search criteria. The file is typically quite large with a high degree of redundancy and so requires some manual filtering by the user to narrow down the list of possibilities to a representative few. The file is composed of tab-separated values so it can be imported into a spreadsheet program such as Excel and sorted and filtered to assist in choosing the test cases. The final selections are then copied into a batch file for use in automated executions.

Table 5-1. Simulator input parameters

Input Parameter	Description
Level 2 Row	The number of rows of level-2 devices
Level 2 Column	The number of columns of level-2 devices
Level 2 Redundancy	The number of spare level-2 devices
Level 1 Row	The number of rows of level-1 devices
Level 1 Column	The number of columns of level-1 devices
Level 1 Redundancy	The number of spare level-1 devices
Level 0 Row	The number of rows of level-0 devices (bit level)
Level 0 'k' Value	The number of columns representing a word (bit level)
Level 0 Plane	The number of words in a row
Level 0 Row Red.	The number of spare level-0 rows
Level 0 Column Red.	The number of spare level-0 columns
Errors Correctable	The number of errors correctable with ECC's
RAID/Chipkill	Whether or not parity implemented across level-1 rows
Usable Lifetime	The lifetime for which to simulate (hours)
Poll Interval	The interval at which to record the memory state (hours)
Hard Error Rate	The bit-level hard error rate (FIT)
Soft Error Rate	The bit-level soft error rate (FIT)

5.2.2 Simulation

The second program performs the simulations of the device configurations developed by the parameter generation. The simulation program is quite complex and can be broken down into three subcategories. The first of these deals with the generation of the failure times, types, and locations throughout the lifetime of the memory. Once generated, these failures are fed into the memory controller simulation which records the actions taken by the controller in response to each of the failure events. While the simulation is in progress, the memory state is recorded at predetermined polling intervals.

Failure generation. In the initial versions of the simulator, an attempted was made to model the memory at a bit level. The hard and soft failures for each bit were generated and ordered in a linked-list structure. This method required the entire memory state to be kept and a sorting algorithm be used to ensure that the errors were processed in order. As the total number of bits increased to over one million, this method was soon determined to be impractical. Several alternative methods of simulation were considered, such as those based on Markov models and order statistics [45-47].

The current failure generation module makes use of order statistics to generate the hard bit failure times for the entire lifetime of the memory at the start of the simulation. The equation for the i^{th} failure time, T_i, is given in (4), where $F^{-1}(x)$ is the percent point function of the time to fail, U is a sequence of uniformly distributed random variables on the interval (0,1), n is the size of the memory, and r is the desired number of ordered failures to be generated [47].

$$T_i = F^{-1}\left[1 - \prod_{j=1}^{i} U_j^{\frac{1}{n-j+1}}\right]_{j=1,..,r} \tag{4}$$

Each of the failure times is randomly associated with a failure type based on the probability distribution of failures in the type of memory being simulated. Each failure is also assigned to a random location in memory. The result is a list of failures with the location parameters for each of the levels defined by the number of the row or the column. A zero is used to represent an entire row, column, or device failure. This is illustrated in Figure 5-3.

Index	Type	Fail Time	Physical Location						
			Level 2		Level 1		Level 0		
			Row	Col	Row	Col	Row	Plane	Col
1	B	755.73910	1	1	1	1	4	1	1
2	R	1587.40267	1	1	1	1	7	0	0
3	C	2619.78800	1	1	1	1	0	3	2
…									

Figure 5-3. Sample failure generation output

The occurrence of soft bit failures need not be simulated at all as they can be estimated from the number of hard uncorrectable errors in the system at each of the polling intervals. The equation for the SUE is given in (5), where α is the SER, *t* is the time in hours, G is the number of good bits in words with the number of correctable bits in error, and $B_{nec}(t)$ is the number of words in the memory with the number of correctable bits in error at time *t*. The equation finds the probability that the soft errors in the system will affect one of the good bits in a word that already contains *nec* bad bits.

$$SUE(t) = \frac{\alpha t}{10^9} \times G \times B_{nec}(t) \tag{5}$$

Simulation. The memory is simulated by iterating through each of the failures in the fail list and determining whether the fault tolerance mechanisms in the memory are sufficient to avoid uncorrectable errors of the data were to be accessed. This requires the interactions of the failures be determined to ascertain when and how the defects line up.

It is also necessary during the simulation process to store information about the system. This is done in the data structure shown in Figure 5-4. The top level of the structure is a hash with the key determined by the level-2 row and level-2 column values. The only information stored at this level is the status of the level-2 device, such as whether it is a spare that has been swapped in or if it is has been swapped out and is considered dead. The next level consists of two hashes, the first has a key derived from the level-1 row and level-1 column and the other is just the level-1 row. In the case of the

row and column, information local to each of the level-1 arrays is stored, which includes the status of the array, the number of spare rows and columns that have been used, the number of words with 1 to *nec* bad bits, and the number of uncorrectable errors. The entry that stores the start of the chain indexes the first failure in the fail list for the array and will be discussed in more detail shortly. The hash that is indexed by only the level-1 row is used only in the case that RAID parity is implemented across level-1 arrays and stores additional information necessary for calculating the number of HUE's.

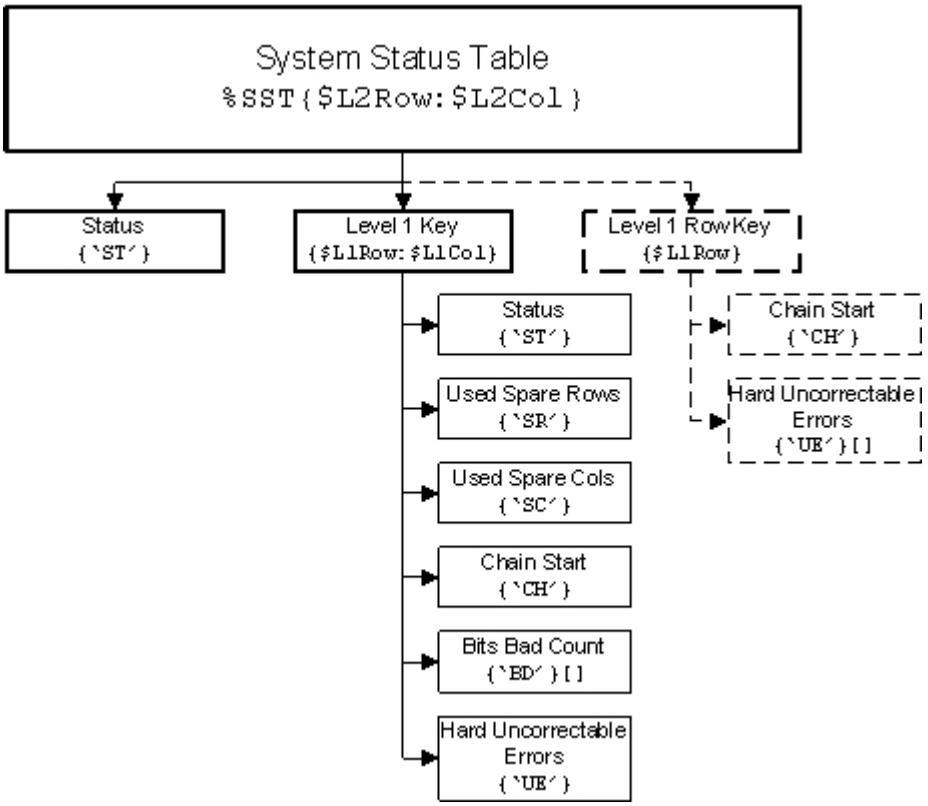
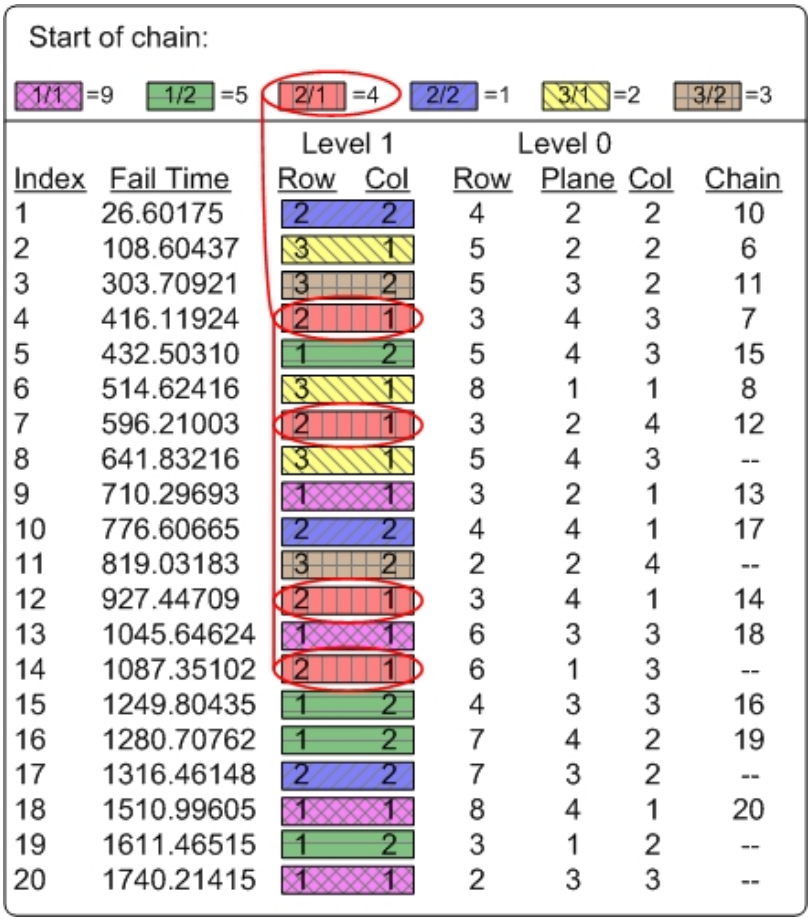


Figure 5-4. Hierarchical hash data structure used in the simulator

The interaction of the failing bits is modeled by "chaining" them together by level-1 array. The chaining process, which is similar to that used in Libson and Harvey [48], avoids having to search through all previous failures in the fail list for each new failure. Chaining is done by first accessing the start of the chain in the simulation data structure to determine if there has been a previous failure in the same level-1 array as the current

failure. If the current failure is the first one in the array, then its index is recorded as the start of the chain for the array and it is examined to see if it results in a HUE. If the current failure is not the first failure in the array, then the index for the first failure is retrieved and the two failures are checked to see if they line up to cause a HUE. The index for the current failure is also entered into the chain entry in the fail list at the first failure. If at a later time, another failure occurs in the same level-1 array, the index to the first failure is retrieved from the data structure and the second failure is retrieved by looking at the first failure's chain entry. The current failure's index is then inserted into the second failure's chain entry and the three failures are checked for the possibility of lining up to cause a HUE. The process continues in this fashion for all failures with each only being processed with those in the same level-1 array without the need to ever scan the entire fail list.

An example is given in Figure 5-5, which represents a somewhat simplified failure list. In this case, the memory is configured with eight rows with four words of four bits each at level-0. The level-1 is composed of an array of three rows and two columns of level-0 arrays. The level-2 has been reduced to a single instance for simplicity and all errors only affect a single bit. Highlighting has been used throughout the list to identify the level-1 array that each error affects. It has been repeated at the top of the figure to represent the start of the chain that would be stored in the simulator's data structure. A chaining column has also been inserted to represent how the failures are connected. In this case, there is only a single case in which more than one error occurs in a word. It is in the level-1 array at the location (2,1) in row three and word four; it has been designated by circling and connecting the failures in the figure.



Index	Fail Time	Level 1 Row	Level 1 Col	Level 0 Row	Level 0 Plane	Level 0 Col	Chain
1	26.60175	2	2	4	2	2	10
2	108.60437	3	1	5	2	2	6
3	303.70921	3	2	5	3	2	11
4	416.11924	2	1	3	4	3	7
5	432.50310	1	2	5	4	3	15
6	514.62416	3	1	8	1	1	8
7	596.21003	2	1	3	2	4	12
8	641.83216	3	1	5	4	3	--
9	710.29693	1	1	3	2	1	13
10	776.60665	2	2	4	4	1	17
11	819.03183	3	2	2	2	4	--
12	927.44709	2	1	3	4	1	14
13	1045.64624	1	1	6	3	3	18
14	1087.35102	2	1	6	1	3	--
15	1249.80435	1	2	4	3	3	16
16	1280.70762	1	2	7	4	2	19
17	1316.46148	2	2	7	3	2	--
18	1510.99605	1	1	8	4	1	20
19	1611.46515	1	2	3	1	2	--
20	1740.21415	1	1	2	3	3	--

Figure 5-5. Example of failure chaining process in simplified fail list. Highlighted chain shows two bits in the same word lining up to be in error.

The simulation process becomes somewhat more complicated in the case that RAID redundancy is enabled across level-1 arrays. The first difference is in the determination of the HUE's within an array. In the previous case, the HUE's could be determined locally within each level-1 array. When RAID is enabled, however, the failures that line up within the arrays are tracked but not recorded as HUE's. Instead, there is a second chaining process that chains together the low-level failures by level-1 row.

The chaining process used in the case of RAID is the same as that used for the bit-level errors except the failures are compared on a word level to see if multiple words are uncorrectable in the same relative position across a row of level-1 arrays. It is only when

two uncorrectable words line up that the RAID parity cannot be used to regenerate the words in error and HUE's are recorded. Only the last failure to line up to cause a HUE in each word needs be chained together, and there are two HUE's for the first occurrence and then each subsequent word that lines up results in another single HUE. The HUE information is tracked in the hash structure indexed by the level-1 row and supersedes the information in the other hash when calculating the number of HUE's for the system.

Memory state recording. The recording process logs the number of hard and soft uncorrectable errors in the memory. The result of the logging is an overview of the number of hard and soft uncorrectable errors present throughout the lifetime of the memory, as well as a log of the number and type of spares used.

The logging is done by iterating through the data structure that maintains the number of hard uncorrectable errors in the system and calculating a total for the entire memory. It is also during this polling process that the number of soft uncorrectable errors is estimated using (5). When multiple iterations are run, which is necessary to obtain accurate results, statistical measures such as the mean, minimum, maximum, and standard deviation of the UE values is calculated and recorded. The goal for this thesis was to run 5,000 iterations of each simulation to assure statistical significance based on the law of large numbers [47, 48].

5.2.3 Results Analysis

The results of the simulations are derived from the log file kept during the polling process, as well as optional verbose logs kept during the simulation. The log file kept during the polling process gives values for the number of hard and soft uncorrectable errors in the system, and optionally the number and type of spares used. A portion of a sample log file is displayed in Figure 5-2. The name of the file is prefixed with "time_"

followed by the values for the parameters that define the simulation and as described in Table 5-1. In the example shown, the first lines of the file give the parameter values for the memory in an easy-to-read format. This is followed by the results for the number of HUE's and SUE's in the system at every 1,000 hours up to the memory lifetime of 100,000 hours. The mean, minimum, maximum, and standard deviation are given for cases that the simulation is executed for multiple trials. When the graphical version of the simulator is used, it is possible to plot the statistical summaries in a box-plot format.

5.3 Simulator Validation

It is not possible to obtain or derive analytical models for the proposed memory architecture model in its full complexity, so the validation of the simulator has been approached as follows. First, a memory configuration that implements ECC without sparing has been evaluated against an analytical model for memories with SEC [14]. The results in Table 5-2 are averaged over 5,000 trials and show small differences in the MTTF for small memory sizes for which the analytical model was initially used.

Table 5-2. Simulator validation to SEC analytical model for small memory sizes

Memory Size (b)	Simulator MTTF	Analytical MTTF	Difference (%)
256x39	2.025×10^{12}	2.010×10^{12}	0.739
1024x39	1.014×10^{12}	1.000×10^{12}	1.37
256x72	1.107×10^{12}	1.100×10^{12}	0.657
1024x72	5.455×10^{11}	5.490×10^{11}	0.640

A comparison was also made against the analytical model when extended to larger memory sizes. These results are shown in Figure 5-6, and although there is a larger percentage difference in the results, it is clear that the simulator closely tracks the model.

It should also be noted that the model was admittedly not entirely accurate for large memory sizes and should not be used for memories over 1-Gbit [14].

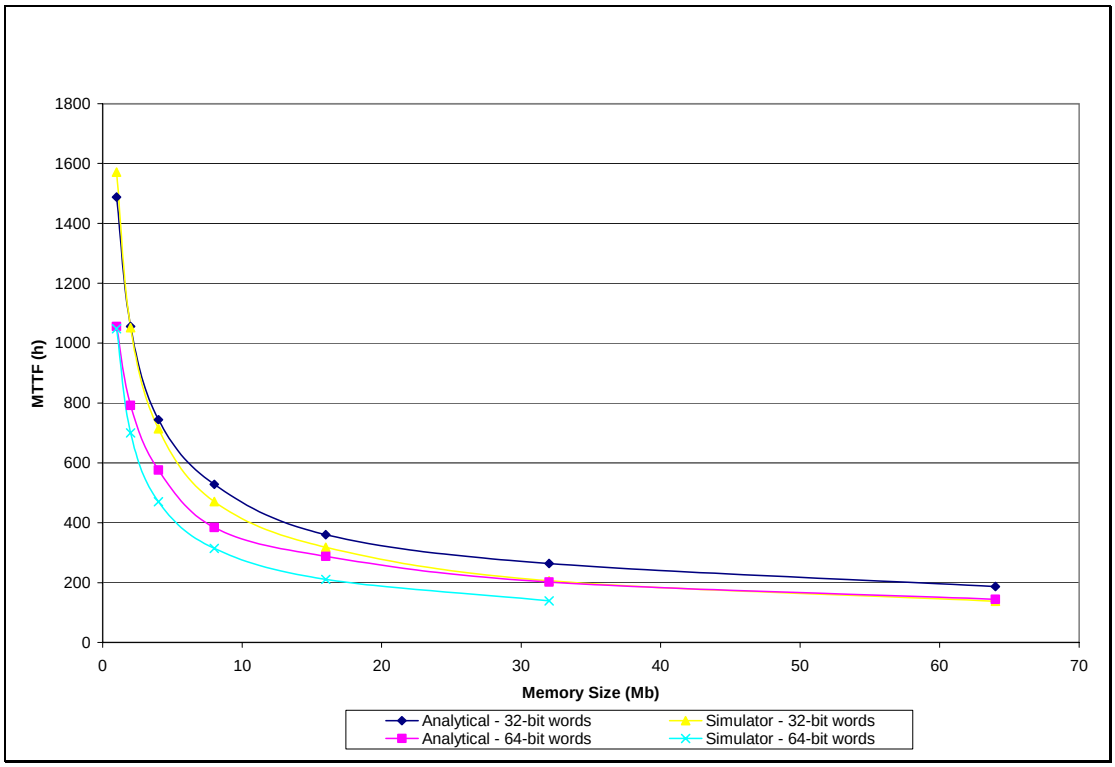


Figure 5-6. Simulator validation to SEC analytical model for large memory sizes

Configurations with ECC and sparing have also been evaluated though the analysis of simulation trace outputs and by comparison against a previously developed simulator that implemented different algorithms, data structures (event-driven vs. chaining), random number generation (per-bit Poisson distribution vs. ordered statistics), and programming language (C vs. Perl). The results for a 1-Mb memory configuration implementing SEC and DEC run in each of the simulators showed an average difference of 6.64 and 8.94%, respectively, over 5,000 trials.

5.4 Simulation Results

The results shown in Figures 5-7 to 5-9 are from a set of experiments used to compare the performance of various fault tolerance mechanisms based on the percentage

of bits used for redundancy. The simulations are based on multi-level memory organizations with 1-Gbit of usable memory size, which are composed of 1-Mbit modules at the lowest level. The results are the average of 5,000 trials and show the number of HUE's in the system after 100,000 hours (~11.4 years) of system life.

It is immediately apparent that without the use of a Hamming ECC code, the performance is relatively poor, even if RAID and spare devices are used. It can be seen that for small word sizes SEC with spares has better performance than DEC initially, but the benefit is lost later in the memory lifetime as the spares are exhausted. It is also notable that the SEC implementations using RAID have a significant performance advantage for a given amount of overhead, although this is at the tradeoff in complexity and possible latency increase of implementing inter-block redundancy.

The only configurations to protect the memory from all errors regardless of word size are DEC with a small number of spares, which requires a significant overhead. The next closest configuration in terms of performance is a combination of SEC with both RAID and spare devices implemented. Again, this configuration requires a high degree of overhead but it is less than that needed by DEC. This provides a strong case for the use of hierarchical fault tolerance mechanisms as a way of optimizing the reliability of devices at the lowest cost in terms of the percentage of devices used for redundancy. The synergism allows for multiple, simpler mechanisms to outperform more complex techniques.

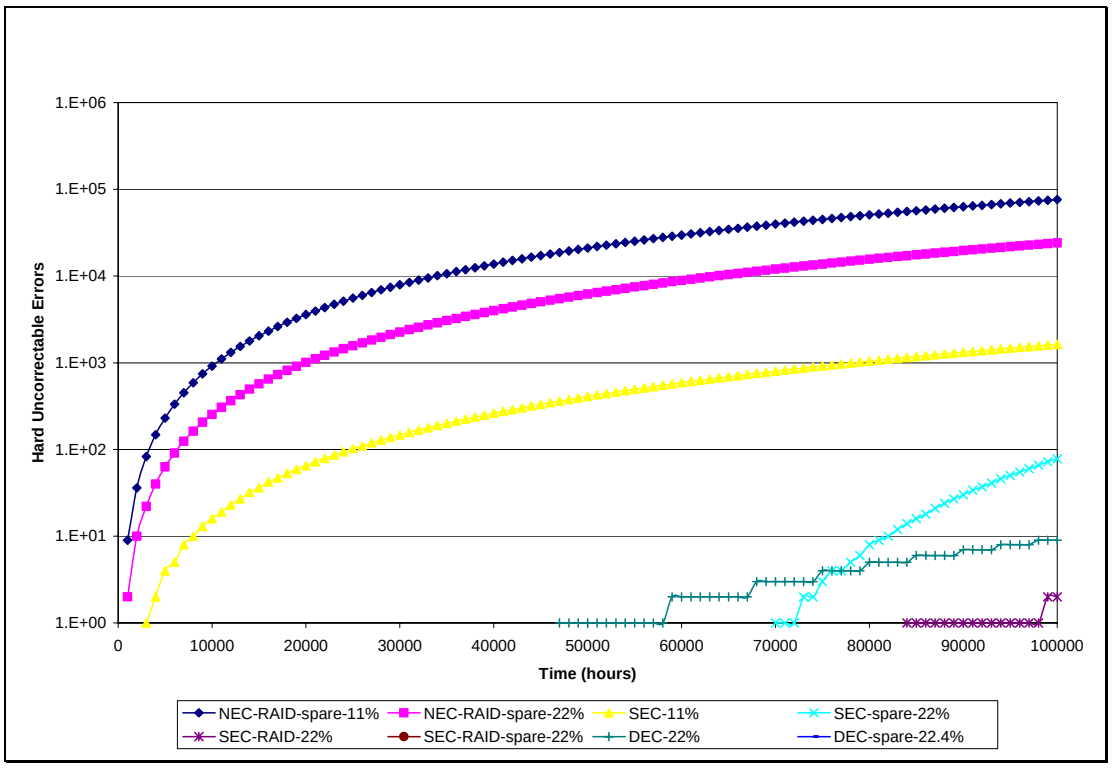


Figure 5-7. The HUE's for 1-Gbit memory with 64-bit word size using various percentages of fault tolerance

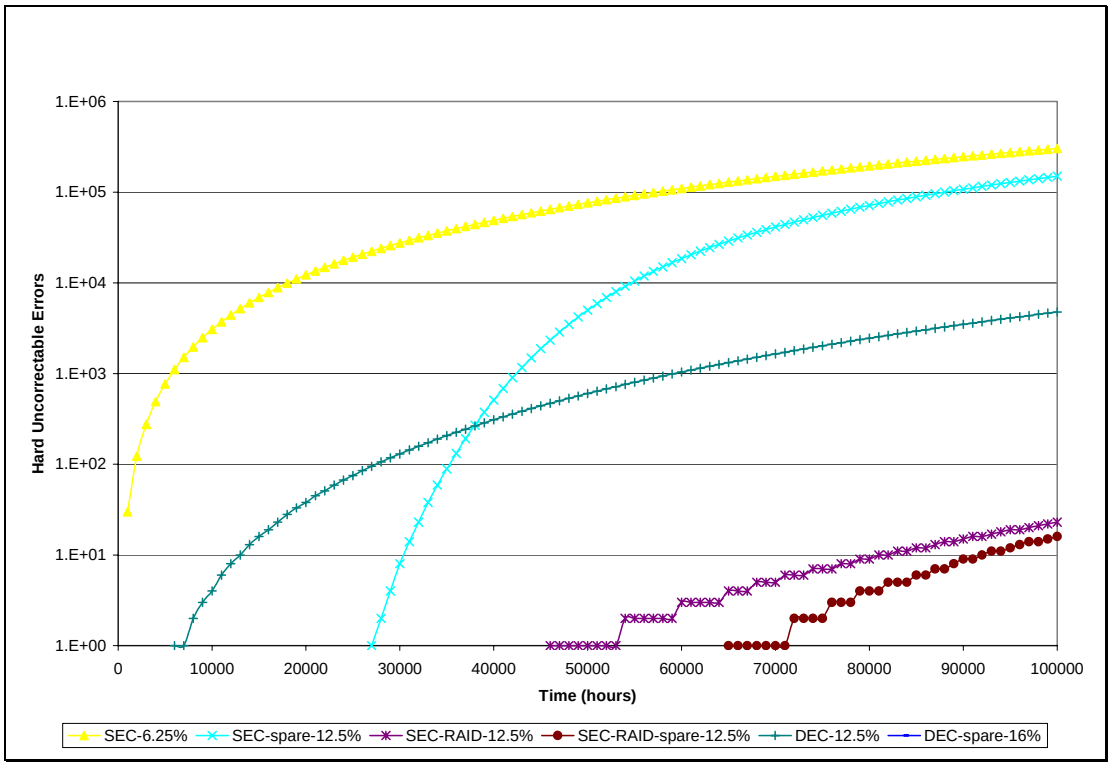


Figure 5-8. The HUE's for 1-Gbit memory with 128-bit word size using various percentages of fault tolerance

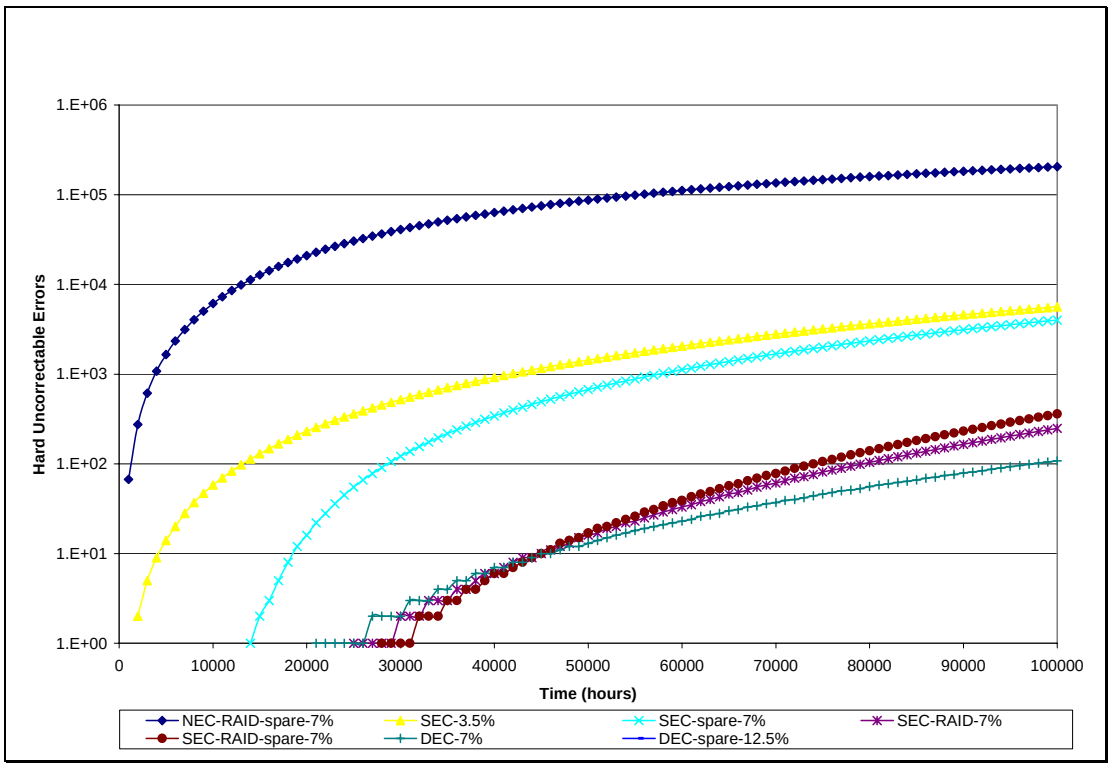


Figure 5-9. The HUE's for 1-Gbit memory with 256-bit word size, using various percentages of fault tolerance

CHAPTER 6
CONCLUSION AND FUTURE WORK

6.1 Conclusions

Continued advances in the fabrication of molecular structures have led to speculation that molecular electronic devices may be incorporated with or even supplant CMOS technology in the realm of ultra-dense logic and memory systems. These systems will be manufactured using self-assembly techniques, which will require entirely new methods of design, simulation, fabrication, and testing. One of the consequences of the self-assembly process is an increased hard error rate due to physical defects and soft error rate from an enhanced sensitivity to noise and radiation effects.

In this thesis a generic, hierarchical fault tolerance architecture has been proposed as a method of dealing with the increased error rates. The architecture incorporates many of the attributes predicted to be inherent in molecular electronics, such as a hierarchical design, randomness in device assembly and interface, and the need for locality of communication. The key concepts from this architecture are as follows:

- Reuse at runtime the logic circuitry necessary for static address mapping
- Perform dynamic reconfiguration at multiple levels in a localized, hierarchical fashion
- Incorporate intra- and inter-device ECC's with reconfiguration

The performance of the architecture has been modeled against existing architectures through a specially constructed simulation tool. The results of the analysis

show that there is an opportunity for the advancement of fault tolerance techniques through the synergistic effect of multiple, hierarchical mechanisms.

6.2 Future Work

There are a number of possibilities for future work in this area molecular electronics. One of the first steps that must be accomplished is the development of accurate models from the device level through the architectural level of abstraction. As models are developed and validated by empirical data, then the requirements in terms of fault tolerance can be fully understood.

There are some additional metrics necessary to completely model the performance of these new fault tolerance mechanisms that will only become available with the fabrication of larger systems and the development of accurate models. One of the most important is the overhead of the mechanisms in terms of latency. Unless molecular electronics ushers in a major shift in the computer architecture paradigm, this will remain a critical factor as there is an ever increasing performance gap between memory and logic devices [20].

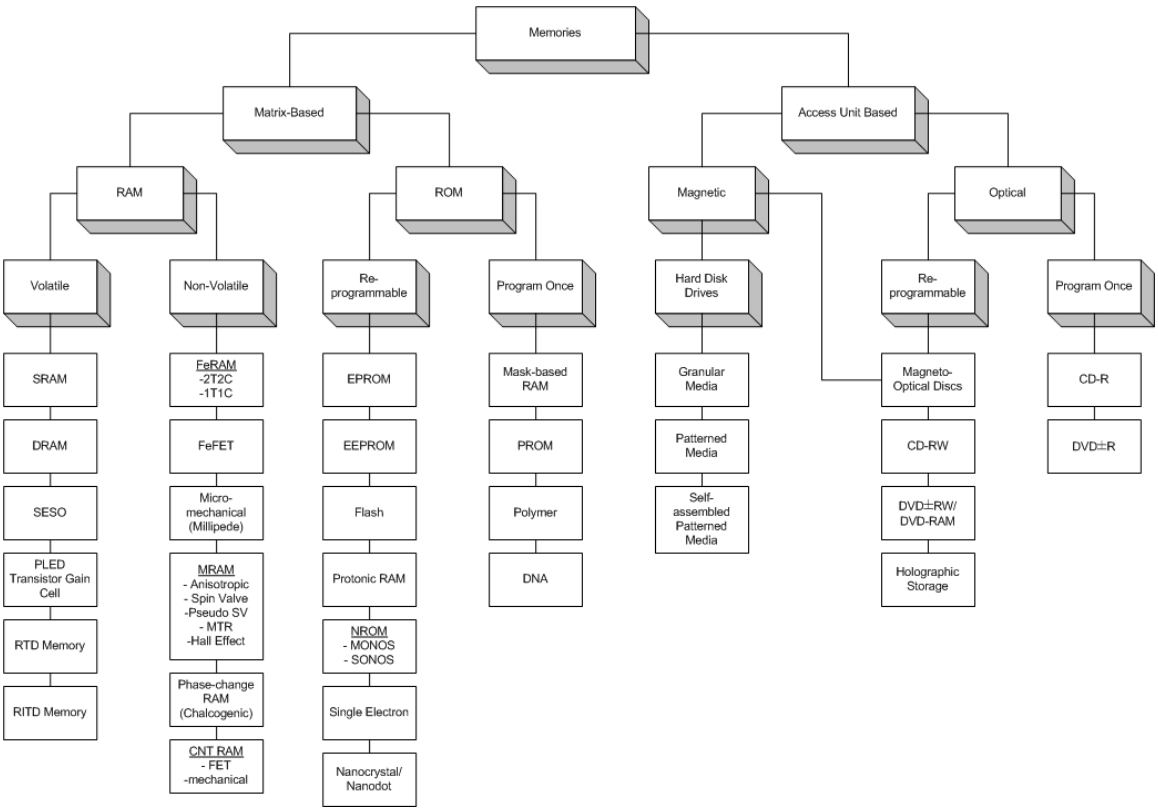
It will also be necessary to examine more complex fault models than are currently implemented in the simulator. Examples include fault clustering [49], infant mortality, and vintage learning [47, 48]. It is also useful to model the devices from a software perspective. If the faults occur in memory but are never accessed or are overwritten with new data before being accessed, then they are not observed as errors [50].

There are several other forms of fault tolerance that have not been covered in this thesis that may prove to be beneficial. One of the most researched is based on the redundancy methods originally proposed by von Neumann. In his work, von Neumann suggested a multiplexing technique in which each wire is replaced with a bundle of wires

and a majority vote is used to determine the value of the wire [6, 7]. This concept has been expanded upon to allow for multiple, redundant components in what is termed *R-fold modular redundancy*. A special case is *Triple Modular Redundancy* (TMR), which uses three redundant components [7]. Again, a majority vote is used to determine the final value. Other techniques involve the use of asynchronous cellular arrays [51], fault alignment exclusion [52], error masking [53], and layout modification [54].

Another avenue of exploration is the area of defect tolerance, or the ability to create functional systems in the presence of defects. The location and type of all the defects in the system is determined through a testing process and the system is designed to avoid using those devices. The most famous implementation of defect tolerance is the HP Teramac supercomputer, which is fully functional with a 3% defect rate [41, 42]. The logic for the entire computer system is encoded in memory and functions through the use of simple *look-up tables* (LUT). The LUT's are connected hierarchically in a fat-tree topology. The fat-tree allows for a very high communication bandwidth and gives the compiler the ability to easily map a wide variety of custom architectures onto the system, including a BIST architecture that can map around defects.

APPENDIX A
MODERN MEMORY DEVICES



APPENDIX B
SIMULATOR SOURCE CODE

B.1 Initialization and Variable Declaration

```
use Cwd;              #provide cross-platform file access
use Fcntl;
use GD::Graph::boxplot;
use Math::Random qw(random_uniform random_uniform_integer);
use Math::Round qw(nhimult nearest);
use Statistics::Descriptive;
use strict;
use Tk;               #use Tk GUI tools
use Tk::BrowseEntry;  #use Tk Combobox widgets
use Tk::Dialog;       #use dialog boxes
use Tk::NoteBook;     #make use of tabbed notebook widget
use Tk::TextUndo;     #use textboxes with undo/redo

############################################################
# Variable declaration
############################################################
my ($currentPage, $execCnt, $fileNum, $fName);
my ($b1Acc, $b2Acc, $crAcc, $raAcc, $sueAcc, $ueAcc);
my (@fileName, @files_t);
my (%pageName);

# Common variables for simulator
my ($her, $kVal, $L0Col, $L0Plane, $L0Row, $L0RedCol);
my ($L0RedRow, $L1Col, $L1Key, $L2Key, $L1Red, $L1Row);
my ($L2Col, $L2Red, $L2Row, $lifetime, $maxTrials);
my ($nec, $path, $pollInterval, $raid, $rVal, $ser);
my ($verbose);

# Variables reset for each simulation
my (%SST);           #used to track array status
my ($count);         #counts number of failures generated
my (@fails);         #array with all failure info

# Statistic arrays for determining when error rates
# are calculated to desired degree of accuracy
my (@herL0StatData, @serL0StatData);
my (@herL1StatData, @serL1StatData);
my (@herL0PlotData, @serL0PlotData);
my (@herL1PlotData, @serL1PlotData);

$path = cwd;
# End variable declaration
```

```
# Default values
$L2Row = 1;  #rows of L2 devices (cards)
$L2Col = 1;  #columns of L2 devices (cards)
$L2Red = 0;  #number of spare L2 devices (cards)
$L1Row = 8;  #rows of L1 devices (chips)
$L1Col = 8;  #columns of L1 devices (chips)
$L1Red = 0;  #number of spare L1 devices (chips)
$L0Row = 1024;  #rows of L0 devices per plane (cells)
$kVal = 128;  #cols of L0 devices per plane (word size in bits)
$L0Plane = 8;  #number of words the L0 arrays are split in to
$L0RedRow = 0;  #number of spare rows of L0 devices (cells)
$L0RedCol = 0;  #number of spare columns of L0 devices (cells)
$nec = 1;  #how many bits are correctable with ECC
$raid = 0;  #extra col of arrays added for RAID across blocks
$lifetime = 1e5;  #how much system life should be generated
$pollInterval = 1e3;  #freq the state of the memory is taken
$her = 2;  #hard error rate in FIT/bit
$ser = 100;  #soft error rate in FIT/bit
$maxTrials = 1;
$execCnt = 0;  #number of complete executions that have completed

# Determine number of bits used for error correcting codes
# 'k' data bits and 'r' check bits are calculated
$rVal = $nec * (log($kVal)/log(2) + 1);
# add one more check bit for RAID with NEC or SEC
($nec <= 2 and $raid ? $rVal += 1: 0);
$L0Col = $kVal + $rVal;

# Total memory size is size of all usable and spare memory
my $L0UsableSize = $kVal * $L0Row * $L0Plane;
my $L0ECCSize = $rVal * $L0Row * $L0Plane;
my $L0SpareSize = ($L0RedRow * $L0Col * $L0Plane)
                + ($L0RedCol * $L0Row);
my $L0MemSize = $L0UsableSize + $L0ECCSize + $L0SpareSize;

my $L1UsableSize = $L0UsableSize * $L1Row * $L1Col;
my $L1ECCSize = ($L0ECCSize * $L1Row * $L1Col)
              + ($raid * $L0MemSize * $L1Row);
my $L1SpareSize = ($L0SpareSize * $L1Row * $L1Col)
                + ($L0MemSize * $L1Red);
my $L1MemSize = $L1UsableSize + $L1ECCSize + $L1SpareSize;

my $totalUsableSize = $L1UsableSize * $L2Row * $L2Col;
my $totalECCSize = $L1ECCSize * $L2Row * $L2Col;
my $totalSpareSize = ($L1SpareSize * $L2Row * $L2Col)
                   + ($L1MemSize * $L2Red);
my $totalMemSize = $totalUsableSize +
                   $totalECCSize +
                   $totalSpareSize;

#Calculate MTTF from hard and soft errors
my $herMTTFMem = nearest(0.001, (1e9 /
                                 ($her * $totalMemSize)));
my $serMTTFMem = nearest(0.001, (1e9 /
                                 ($ser * $totalMemSize)));
my $herMTTFBit = 1e9 / $her;
$verbose = 0;
```

B.2 Graphical User Interface

```
# Toplevel window initialization for GUI
my $top_w = MainWindow->new(); #create new toplevel
my $sw = $top_w->screenwidth;
my $sh = $top_w->screenheight;
#center toplevel on screen
$top_w->geometry("965x650+" . int($sw/2 - 965/2) . "+" .
      int($sh/2 - 675/2));
$top_w->title("Fault Tolerant Memory Simulator"); #title it
$top_w->raise();
my $top_f = $top_w->Frame();

## Create frames for top and bottom tabbed notebooks
#the top one will contain open files and the bottom
#will contain coverage metrics and other information
#also create frame for script selection buttons
my $files_f = $top_f->Frame(-relief => "raised",
                            -borderwidth => 2)
    ->pack(-fill => 'both', -expand => 1);
my $main_f = $top_f->Frame(-relief => "raised",
                           -borderwidth => 2)
    ->pack(-fill => 'both');

##Create the top tabbed notebook
my $files_nb = $files_f->NoteBook()
    ->pack(-fill => 'both', -expand => 1);

##Create the bottom tabbed notebook
my $main_nb = $main_f->NoteBook()
    ->pack(-fill => 'both', -expand => 1);

##Create pages for information about settings and batches
my $archSettings_pg = $main_nb
    ->add(0,-label => 'Architecture Settings');
my $simSettings_pg = $main_nb
    ->add(1, -label => 'Simulation Settings');
my $summSettings_pg = $main_nb
    ->add(2, -label => 'Summary Settings');

## Create the frames for the "Main" settings
###########################################
# Architecture settings frames
my $archSettings_f = $archSettings_pg->Frame(-relief => "groove",
                                             -borderwidth => 2)
    ->pack(-fill => 'both', -expand => 1);
my $archSettingsLeft_f = $archSettings_f->Frame()
    ->pack(-side=> 'left', -fill => 'both', -expand => 1);
my $archSettingsMiddle_f = $archSettings_f->Frame(-relief =>'groove',
                                                  -borderwidth => 2)
    ->pack(-side=> 'left', -fill => 'both', -expand => 1);
my $archSettingsRight_f = $archSettings_f->Frame()
    ->pack(-side=> 'left', -fill => 'both', -expand => 1);
my $dimenTop_f = $archSettingsLeft_f->Frame(-relief =>'groove',
                                            -borderwidth => 2)
    ->pack(-side => 'top', -fill => 'both', -expand => 1);
```

```
my $plane_f = $dimenTop_f->Frame()
    ->pack(-fill => 'both', -side => 'bottom');
my $colSize_f = $dimenTop_f->Frame(-label => '# Cols')
    ->pack(-fill => 'both', -expand => 1, -side => 'right');
my $rowSize_f = $dimenTop_f->Frame(-label => '         # Rows')
    ->pack(-fill => 'both', -expand => 1, -side => 'left');
my $L2Col_f = $colSize_f->Frame()
    ->pack(-fill => 'both', -expand => 1, -side => 'top');
my $L1Col_f = $colSize_f->Frame()
    ->pack(-fill => 'both', -expand => 1, -side => 'top');
my $L0Col_f = $colSize_f->Frame()
    ->pack(-fill => 'both', -expand => 1, -side => 'top');
my $L2Row_f = $rowSize_f->Frame()
    ->pack(-fill => 'both', -expand => 1, -side => 'top');
my $L1Row_f = $rowSize_f->Frame()
    ->pack(-fill => 'both', -expand => 1, -side => 'top');
my $L0Row_f = $rowSize_f->Frame()
    ->pack(-fill => 'both', -expand => 1, -side => 'top');
my $redunL2_f = $archSettingsMiddle_f->Frame()
    ->pack(-fill => 'both', -expand => 1, -side => 'top');
my $redunL1_f = $archSettingsMiddle_f->Frame()
    ->pack(-fill => 'both', -expand => 1, -side => 'top');
my $redunL0Row_f = $archSettingsMiddle_f->Frame()
    ->pack(-fill => 'both', -expand => 1, -side => 'top');
my $redunL0Col_f = $archSettingsMiddle_f->Frame()
    ->pack(-fill => 'both', -expand => 1, -side => 'top');
my $memSpecs_f = $archSettingsRight_f->Frame()
    ->pack(-fill => 'both', -expand => 1);
my $memSpecsLeft_f = $memSpecs_f->Frame()
    ->pack(-side => 'left', -fill => 'both', -expand => 1);
my $memSpecsRight_f = $memSpecs_f->Frame()
    ->pack(-side => 'right', -fill => 'both', -expand => 1);
my $errorRate_f = $memSpecsLeft_f->Frame(-relief =>'groove',
                                          -borderwidth => 2)
    ->pack(-side => 'top', -fill => 'both', -expand => 1);
my $ecc_f = $memSpecsLeft_f->Frame(-relief => 'groove',
                                   -borderwidth => 2,
                                   -label => 'Error Correcting Code')
    ->pack(-side => 'top', -fill => 'both', -expand => 1);
my $info_f = $memSpecsLeft_f->Frame()
    ->pack(-side => 'top', -fill => 'both', -expand => 1);
my $algoSel_f = $memSpecsRight_f->Frame(-relief => 'groove',
                                        -borderwidth => 2,
                                        -label => 'Other Options')
    ->pack(-side => 'top', -fill => 'both', -expand => 1);
my $memSize_f = $memSpecsRight_f->Frame(-relief => 'groove',
                                        -borderwidth => 2,
                                        -label => 'Memory Size')
    ->pack(-side => 'top', -fill => 'both', -expand => 1);
my $ser_f = $errorRate_f->Frame(-label => 'Soft Error Rate')
    ->pack(-side=> 'left', -fill => 'both', -expand => 1);
```

```
my $her_f = $errorRate_f->Frame(-label => 'Hard Error Rate')
    ->pack(-side=> 'right', -fill => 'both', -expand => 1);
my $mttf_f = $info_f->Frame(-label => 'MTTF(h)',
                            -relief => 'groove',
                            -borderwidth => 2)
    ->pack(-side=> 'left', -fill => 'both', -expand => 1);
my $chkBits_f = $info_f->Frame(-label => 'Word Size',
                               -relief => 'groove',
                               -borderwidth => 2)
    ->pack(-side=> 'right', -fill => 'both', -expand => 1);

# Create updating text box with memory size info
my $spareSizePrnt = commify($totalSpareSize);
my $memSizePrnt = commify($totalMemSize);
my $usableSizePrnt = commify($totalUsableSize);
my $eccSizePrnt = commify($totalECCSize);
my $herMTTFMemPrnt = commify($herMTTFMem);
my $serMTTFMemPrnt = commify($serMTTFMem);
my $overheadPrnt = commify(nearest(0.01,($totalMemSize
                       - $totalUsableSize)
                       / $totalUsableSize * 100));
my $memSize_l = $memSize_f->Label(-text => "Total: $memSizePrnt bits\n
                                  Spare: $spareSizePrnt bits\n
                                  ECC: $eccSizePrnt bits\n
                                  Usable: $usableSizePrnt bits\n
                                  Overhead: $overheadPrnt%",
                                  -width => 30)->pack();
my $mttf_l = $mttf_f->Label(-text => "H: $herMTTFMemPrnt\n
                                   S: $serMTTFMemPrnt",
                            -width => 15)->pack();
my $chkBits_l = $chkBits_f->Label(-text => "Data: $kVal\nCheck: $rVal",
                                  -width => 15)->pack();

# Create comboboxes
my @L2Choices = qw(1 2 4 8 16 32 64 128 256 512);
my @L1Choices = qw(1 2 4 8 12 16 22 32 42 46 64 128 256 512 1024);
my @L0RChoices = qw(32 64 128 256 512 1024 2048 4096 8192);
my @L0CChoices = qw(32 64 128 256 512);
my @L0PlaneChoices = qw(1 2 4 8 16 32 64);

my $L2_l = $L2Row_f->Label(-text => "  Level 2:")
    ->pack(-side => 'left');
my $L2Col_cb = $L2Col_f->BrowseEntry(-listwidth => 25,
                                     -variable => \$L2Col,
                                     -choices => \@L2Choices,
                                     -width => 8,
                                     -browsecmd => \&updateMemSize)
    ->pack(-side => 'right', -fill => 'both', -padx => 5);
my $L2Row_cb = $L2Row_f->BrowseEntry(-listwidth => 25,
                                     -variable => \$L2Row,
                                     -choices => \@L2Choices,
                                     -width => 8,
                                     -browsecmd => \&updateMemSize)
    ->pack(-side => 'right', -fill => 'both', -padx => 5);
my $L1_l = $L1Row_f->Label(-text => "  Level 1:")
    ->pack(-side => 'left');
```

```
my $L1Col_cb = $L1Col_f->BrowseEntry(-listwidth => 25,
                                     -variable => \$L1Col,
                                     -choices => \@L1Choices,
                                     -width => 8,
                                     -browsecmd => \&updateMemSize)
    ->pack(-side => 'right', -fill => 'both', -padx => 5);
my $L1Row_cb = $L1Row_f->BrowseEntry(-listwidth => 25,
                                     -variable => \$L1Row,
                                     -choices => \@L1Choices,
                                     -width => 8,
                                     -browsecmd => \&updateMemSize)
    ->pack(-side => 'right', -fill => 'both', -padx => 5);
my $L0_l = $L0Row_f->Label(-text => "  Level 0:")
    ->pack(-side => 'left');
my $L0Col_cb = $L0Col_f->BrowseEntry(-listwidth => 25,
                                     -variable => \$kVal,
                                     -choices => \@L0CChoices,
                                     -width => 8,
                                     -browsecmd => \&updateMemSize)
    ->pack(-side => 'right', -fill => 'both', -padx => 5);
my $L0Row_cb = $L0Row_f->BrowseEntry(-listwidth => 25,
                                     -variable => \$L0Row,
                                     -choices => \@L0RChoices,
                                     -width => 8,
                                     -browsecmd => \&updateMemSize)
    ->pack(-side => 'right', -fill => 'both', -padx => 5);
my $L0P_l = $plane_f->Label(-text => "  Planes:")
    ->pack(-side => 'left');
my $L0Plane_cb = $plane_f->BrowseEntry(-listwidth => 25,
                                       -variable => \$L0Plane,
                                       -choices => \@L0PlaneChoices,
                                       -width => 8,
                                       -browsecmd => \&updateMemSize)
    ->pack(-side => 'bottom', -fill => 'y', -padx => 5);

my @redunChoices = qw(0 1 2 4 8 16 32 64 104 128 256
                      512 1024 2048 4096 8192 16384);
my $redunL2_l = $redunL2_f->Label(-text => "  Redundant L2: ")
    ->pack(-side => 'left');
my $redunL2_cb = $redunL2_f->BrowseEntry(-listwidth => 25,
                                         -variable => \$L2Red,
                                         -choices => \@redunChoices,
                                         -width => 8,
                                         -browsecmd => \&updateMemSize)
    ->pack(-side => 'right', -fill => 'both', -padx => 20);
my $redunL1_l = $redunL1_f->Label(-text => "  Redundant L1: ")
    ->pack(-side => 'left');
my $redunL1_cb = $redunL1_f->BrowseEntry(-listwidth => 25,
                                         -variable => \$L1Red,
                                         -choices => \@redunChoices,
                                         -width => 8,
                                         -browsecmd => \&updateMemSize)
    ->pack(-side => 'right', -fill => 'both', -padx => 20);
my $redunL0Row_l = $redunL0Row_f->Label(-text => "  Redundant L0 Rows:
")
    ->pack(-side => 'left');
```

```
my $redunL0Row_cb = $redunL0Row_f->BrowseEntry(-listwidth => 25,
                                        -variable => \$L0RedRow,
                                        -choices => \@redunChoices,
                                        -width => 8,
                                        -browsecmd => \&updateMemSize)
    ->pack(-side => 'right', -fill => 'both', -padx => 20);
my $redunL0Col_l = $redunL0Col_f->Label(-text => "Redundant L0 Cols: ")
    ->pack(-side => 'left');
my $redunL0Col_cb = $redunL0Col_f->BrowseEntry(-listwidth => 25,
                                        -variable => \$L0RedCol,
                                        -choices => \@redunChoices,
                                        -width => 8,
                                        -browsecmd => \&updateMemSize)
    ->pack(-side => 'right', -fill => 'both', -padx => 20);

# Create HER/SER entry boxes
my $ser_en = $ser_f->Entry(-validate => 'focusout',
                           -validatecommand => \&updateMemSize,
                           -textvariable => \$ser,
                           -justify => 'right',
                           -width => 10)
    ->pack(-padx => 20, -pady => 9);
my $her_en = $her_f->Entry(-validate => 'focusout',
                           -validatecommand => \&updateMemSize,
                           -textvariable => \$her,
                           -justify => 'right',
                           -textvariable => \$her,
                           -width => 10, )
    ->pack(-padx => 20, -pady => 9);

# Create error correction selection buttons
my $eccNEC = $ecc_f->Radiobutton(-text => 'None',
                                 -value => 0,
                                 -variable => \$nec,
                                 -command => \&updateMemSize)
    ->pack(-side => 'left', -padx => 5);
my $eccSEC = $ecc_f->Radiobutton(-text => 'SEC',
                                 -value => 1,
                                 -variable => \$nec,
                                 -command => \&updateMemSize)
    ->pack(-side => 'left', -padx => 5);
my $eccDEC = $ecc_f->Radiobutton(-text => 'DEC',
                                 -value => 2,
                                 -variable => \$nec,
                                 -command => \&updateMemSize)
    ->pack(-side => 'left', -padx => 5);

# Create fault tolerance algorithm selection buttons
my $L1ECCSel = $algoSel_f->Checkbutton(-text => 'L1 ECC (RAID)',
                                       -variable => \$raid,
                                       -command => \&updateMemSize)
    ->pack(-side => 'left', -padx => 5);
my $verbSel = $algoSel_f->Checkbutton(-text => 'Verbose',
                                      -variable => \$verbose)
    ->pack(-side => 'left', -padx => 5);
```

```
# Simulation settings frames
my $simSettings_f =  $simSettings_pg->Frame(-relief => 'groove',
                                            -borderwidth => 2)
    ->pack(-fill => 'both', -expand => 1);
my $simSettingsLeft_f =  $simSettings_f->Frame(-label => 'Time (h)',
                                               -relief => 'groove',
                                               -borderwidth => 2)
    ->pack(-side=> 'left', -fill => 'both', -expand => 1);
my $simSettingsRight_f =  $simSettings_f->Frame(-label => 'Execute',
                                                -relief =>'groove',
                                                -borderwidth => 2)
    ->pack(-side=> 'left', -fill => 'both', -expand => 1);
my $pollTime_f = $simSettingsLeft_f->Frame(-relief => 'groove',
                                           -borderwidth => 2)
    ->pack(-fill => 'both', -expand => 1);
my $endTime_f = $simSettingsLeft_f->Frame(-relief => 'groove',
                                          -borderwidth => 2)
    ->pack(-fill => 'both', -expand => 1);
my $executeSim_f = $simSettingsRight_f->Frame(-relief =>'groove',
                                              -borderwidth => 2)
    ->pack(-fill => 'both', -expand => 1);
my $maxTrials_f = $executeSim_f->Frame()
    ->pack(-fill => 'x', -expand => 1);

my $pollTime_l = $pollTime_f->Label(-text => "Polling Interval(h): ")
    ->pack(-side => 'left');
my $pollTime_en = $pollTime_f->Entry(-justify => 'right',
                                     -textvariable => \$pollInterval,
                                     -width => 10)
    ->pack(-side => 'left');
my $endTime_l = $endTime_f->Label(-text => "Max Run Time(h): ")
    ->pack(-side => 'left');
my $endTime_en = $endTime_f->Entry(-justify => 'right',
                                   -textvariable => \$lifetime,
                                   -width => 10)
    ->pack(-side => 'left');

my $maxTrials_l = $maxTrials_f->Label(-text => "Maximum Trials: ")
    ->pack(-side => 'left');
my $maxTrials_en = $maxTrials_f->Entry(-justify => 'right',
                                       -textvariable => \$maxTrials,
                                       -width => 10)
    ->pack(-side => 'left', -pady => 5);
my $executeSim_b = $executeSim_f->Button(-text => 'Execute 0',
                                         -width => 20,
                                         -command => \&runSimulation)
    ->pack(-pady => 5);
my $plotSim_b = $executeSim_f->Button(-text => 'Plot',
                                      -width => 20,
                                      -command => \&plotResults)
    ->pack(-pady => 5);

# Summary settings frames
my $summSettings_f =  $summSettings_pg->Frame(-relief => "raised",
                                              -borderwidth => 2)
    ->pack(-fill => 'both', -expand => 1);
```

```
my $summSettingsLeft_f =  $summSettings_f->Frame(-relief => 'groove',
                                                 -borderwidth => 2)
    ->pack(-side=> 'left', -fill => 'both', -expand => 1);
my $summSettingsMiddle_f =  $summSettings_f->Frame(-relief => 'groove',
                                                   -borderwidth => 2)
    ->pack(-side=> 'left', -fill => 'both', -expand => 1);
my $summSettingsRight_f =  $summSettings_f->Frame(-relief =>'groove',
                                                  -borderwidth => 2)
    ->pack(-side=> 'left', -fill => 'both', -expand => 1);

# Create menubar
$top_w->configure(-menu => my $menubar_mb = $top_w->Menu);
# Create menu items in menubar
map {$menubar_mb->cascade(-label => '~' . $_->[0],
                          -menuitems => $_->[1])}
    ['File', &file_menuitems],
    ['Edit', &edit_menuitems],
    ['Search', &search_menuitems],
    ['Help', &help_menuitems];

## Pack all main frames to toplevel
$top_f->pack(-side=>'top', -fill=>'both', -expand=>1);
$files_f->pack(-side => "top", -fill => 'both', -expand => 1);
$archSettings_f->pack(-side=>"top", -fill=>'both');

MainLoop();  #Display Tk widgets
```

B.3 Simulation Subroutines

B.3.1 Generate Failures

```
sub generate
{
    # Variable declarations
    my ($failType, $failL0Col, $failL0Plane, $failL0Row, $failL1Col);
    my ($failL1Row, $failL1RowCalc, $failL2Col, $failL2Row);
    my ($failL2RowCalc, $index, $inverse, $minTime, $power);
    my ($rand, $Uprod);

    $Uprod = 1;  #initialize Uprod to 1 since it will be mult by power
    while ($minTime <= $lifetime)
    {
        #use IBM FTMS equations to calculate failure times in order
        $count += 1;
        $rand = random_uniform();
        $power = $rand ** (1 / ($totalMemSize - $count + 1));
        $Uprod *= $power;
        $inverse = -1 * $herMTTFBit * log($Uprod);
        $minTime = $inverse;

        ##determine type and location of each failure
        #generation of fail type
        #$rand = random_uniform();
```

```
        if ($rand < 0.35)
        {
            $failType = 'B';
            $failL0Row = random_uniform_integer(1, 1, ($L0Row +
                                                       $L0RedRow));
            #if error is in spare row, set the flag
            if ($failL0Row > $L0Row)
            {
                #failure can't be in both a spare row and a spare col
                $failL0Col = random_uniform_integer(1, 1, $L0Col);
                $failL0Plane = random_uniform_integer(1, 1, $L0Plane);
            }
        }
        elsif ($rand < 0.67)
        {
            $failType = 'R';
            $failL0Col = 0;
            $failL0Plane = 0;
            $failL0Row = random_uniform_integer(1, 1, ($L0Row +
                                                       $L0RedRow));
        }
        elsif ($rand < 0.85)
        {
            $failType = 'C';
            $failL0Row = 0;
           #choose one of the cols including the spares, then determine
           #col/plane recieved the error-spares are in "extra" plane
           $failL0Col = random_uniform_integer(1, 1,
                        (($L0Col * $L0Plane) + $L0RedCol));
           $failL0Plane = nhimult(1, $failL0Col / $L0Col);
           $failL0Col = $failL0Col % $L0Col;
        }
        else
        {
            $failType = 'A';
            $failL0Row = 0;
            $failL0Col = 0;
        }

        $failL1Row = random_uniform_integer(1, 1, $L1Row);
        $failL1Col = random_uniform_integer(1, 1, ($L1Col + $raid));
        $failL2Row = random_uniform_integer(1, 1, $L2Row);
        $failL2Col = random_uniform_integer(1, 1, $L2Col);

        @{$fails[$count]} = ($count, $failType, $minTime, $failL2Row,
                             $failL2Col, $failL1Row, $failL1Col,
                             $failL0Row, $failL0Plane, $failL0Col);
    }
    print "COUNT: $count\n";
}
```

B.3.2 Simulate

```
#-----------------------------------------------------------------------#
# KEY TO LABELS:
# 'A'   =>  Label for L0 CHIP (entire array) error
# 'B'   =>  Label for L0 BIT error
# 'BD'  =>  Track BAD COUNT for words with < NEC bad bits
# 'C'   =>  Label for L0 COLUMN error
# 'CH'  =>  CHAIN of L0 or L1 fails linking those in similar devices
# 'DD'  =>  Marks L0 array as DEAD
# 'DE'  =>  Track L0 rows that have been DEALLOCATED
# 'R'   =>  Label for L0 ROW error
# 'SC'  =>  Tracks number of SPARE COLUMNS used
# 'SR'  =>  Tracks number of SPARE ROWS used
# 'ST'  =>  STATUS information for array
# 'UE'  =>  Track count of UNCORRECTABLE ERRORS
# 'UU'  =>  Mark L0 spare array as UNUSED
#-----------------------------------------------------------------------#

sub simulate
{
  # Variable declarations
  my ($errorCnt, $failTime, $flag, $goodBits);
  my ($index, $key, $keyC, $keyR);
  my ($L0ColKey, $L0PlaneKey, $L0RowKey, $L1ColKey, $L1RowKey);
  my ($L0TempColKey, $L0TempPlaneKey, $L0TempRowKey, $L1TempColKey);
  my ($numWords, $redunSuccess, $tempIndex, $tempL1Key, $time);
  my ($totalL0SUE, $totalL1SUE, $totalL0UE, $totalL1UE);
  my (@colChain, @rowChain, @totalBadBit, @totalBadWord);
  my (%bitErrors, %wordErrors);
  # End variables

  #read fail info in from file and put in a 2D array
  $time = $pollInterval;
  #calculate parameters used in the estimation of number of SUEs
  #total number of L1 devices in memory
  $numWords = $L2Row * $L2Col * $L1Row *
              $L1Col * $L0Row * $L0Plane;
  #fraction of good bits in words with NEC bad bits
  #these are the ones succeptable to errors
  $goodBits = $L0Col - $nec;

  #go through each failure generated by order stats
  FAILLOOP: while ($index <= $count) {
    $index += 1;  #go on to next fail in generated list
    $flag = 0;  #set to only go through hash once
    $failTime = $fails[$index][2];
    while ($time <= $failTime)
    {
      if ($flag == 0)
      {
        $flag = 1;
        #reset variables
        @totalBadBit = @totalBadWord = ();
        $totalL0UE = $totalL0SUE = 0;
        $totalL1UE = $totalL1SUE = 0;
```

```
          #go through all L0 arrays
          foreach $key (keys %SST)
          {
            #only count the errors if they are in the main memory:
            #they must not be marked 'dead' and must either not be from
            #a spare device or be a spare device that is marked 'used'
            if ($SST{$key}{'ST'} ne 'DD' and
               ($key !~ /S/ or $SST{$key}{'ST'} eq 'UU'))
            {
              #go through each row key and, if it is a number then
              #go through all corresponding columns associated with it
              foreach $keyR (keys %{$SST{$key}})
              {
                if ($keyR =~ /\d+/)
                {
                  #key is for L0 failure info
                  if ($keyR =~ /-/)
                  {
                    if ($SST{$key}{$keyR}{'ST'} ne 'DD' and
                        ($key !~ /S/ or $SST{$key}{$keyR}{'ST'} eq 'UU')
)
                    {
                      $totalBadBit[1] += $SST{$key}{$keyR}{'BD'}[1];
                      $totalBadBit[2] += $SST{$key}{$keyR}{'BD'}[2];
                      $totalL0UE += $SST{$key}{$keyR}{'UE'};
                      #print "totalL0UE: $totalL0UE\n";
                    }
                  }
                  #if RAID enabled, then key is for L1 failure info
                  elsif ($raid)
                  {
                    #record total num rows with 2 to L0Col word failures
                    #as well as the overall number of L1 (word) HUEs
                    for my $i (2..($L1Col + 1))
                    {
                      if ($SST{$key}{$keyR}{'UE'}[$i])
                      {
                        $totalBadWord[$i] += $SST{$key}{$keyR}{'UE'}[$i];
                        $totalL1UE += $SST{$key}{$keyR}{'UE'}[$i] * $i;
                      }
                    }
                  }
                }
              }
            }
          }
          #Calc SUE by multiplying SER by the num good bits in a word with
          #nec bad bits x total words w/ nec bad bits and finally by time
          #div by 1e9 to account for FIT given as fails in 1e9 bit-hours.
          if ($nec > 0)
          {
            $totalL0SUE = ($ser / 1e9 * $time) * $goodBits *
                          $totalBadBit[$nec];
          }
```

```
  else
  {
    $totalL0SUE = ($ser / 1e9 * $time) * $goodBits * $numWords;
  }

  #calculate L1 SER if RAID is enabled
  if ($raid)
  {
    #when RAID is enabled, the total L1SUE is the total number of
    #L0 soft errors multiplied by the ratio of words susceptible to
    #lining up to cause a word error this needs to be done for the
    #number of rows with k = 1..(L0Col -1) word errors the number
    #of words susceptible in each case is (L0Col - k) and the case
    #of k == 1 needs to be mult. by 2 to account for two new UEs

    #the total number of rows with 1 UE in them multiplied by the
    #number of words susceptible a soft UE lining up with these
    #would result in two UEs
    $totalL1SUE = ($totalL0UE - $totalL1UE) * ($L1Col - 1) * 2;
    foreach my $i (2 .. ($L1Col - 1))
    {
      $totalL1SUE += $totalBadWord[$i] * ($L1Col - $i);
    }
    #currently have the total number of words susceptible to
    #a soft error lining up with them divide these by total
    #number of words to get a ratio and multiply by the total
    #number of soft errors in the system
    $totalL1SUE = ($totalL1SUE / $numWords) * $totalL0SUE;
  }
  #add each failure amount to an array to be used later
  #in a statistical analysis of multiple trials
  push @{$serL0StatData[$time]}, $totalL0SUE;
  push @{$herL0StatData[$time]}, $totalL0UE;
  if ($raid)
  {
    push @{$serL1StatData[$time]}, $totalL1SUE;
    push @{$herL1StatData[$time]}, $totalL1UE;
  }
  $time += $pollInterval;
}
#generate key values from fail locations
$L2Key = $fails[$index][3].'-'.$fails[$index][4];
$L1Key = $fails[$index][5].'-'.$fails[$index][6];
$L1RowKey = $fails[$index][5];
$L1ColKey = $fails[$index][6];
$L0RowKey = $fails[$index][7];
$L0PlaneKey = $fails[$index][8];
$L0ColKey = $fails[$index][9];

#Don't process failure if it is in a part of memory that has
#already been killed or if it is in a L0 row that has been
#deallocated
if ($SST{$L2Key}{'ST'} eq 'DD' or
    $SST{$L2Key}{$L1Key}{'ST'} eq 'DD')
{
  next FAILLOOP;
}
```

```
elsif ($L0RowKey > $L0Row)
{
  #keep track of all spare rows that are no longer available
  $SST{$L2Key}{$L1Key}{'SR'} += 1;
  next FAILLOOP;
}

#check to see if failure is of a bit in an L1 Device (chip)
elsif ($fails[$index][1] eq 'B')
{
  #initialize bit errors to the current error
  %bitErrors = ();
  $bitErrors{$L0Col} = 1;
  #reset arrays to be used to track all bit errors in the same
  #row/col as the current one these failure indexes will be
  #disabled if the row/col is replaced with a spare.
  @colChain = ();
  @rowChain = ();
  #FIXME: put back in when spare columns used
  #push @colChain, $index;
  push @rowChain, $index;

  #check to see if key is present in system status table
  #the SST points to the start of each chain of bit failures
  if ($SST{$L2Key}{$L1Key}{'CH'} == 0)
  {
    #SST row entry was blank, so set SST row to current index value
    $SST{$L2Key}{$L1Key}{'CH'} = $index;
  }
  else
  {
    #look through previous failures in same L0 array matrix
    #set the index to the start of the chain
    $tempIndex = $SST{$L2Key}{$L1Key}{'CH'};

    #cycle through all errors chained together
    while(1)
    {
      $L0TempRowKey = $fails[$tempIndex][7];
      $L0TempPlaneKey = $fails[$tempIndex][8];
      $L0TempColKey = $fails[$tempIndex][9];
      #bit causes error if it is in the same row and plane but a
      #different column also check that the index has not been
      #disabled by swapping in a spare
      if ($fails[$tempIndex][12] != 1 and
          $L0RowKey == $L0TempRowKey)
      {
        #this is the same row as the current bit failure, so keep
        #track of it in case this row is replaced with a spare
        push @rowChain, $tempIndex;
        if ($L0PlaneKey == $L0TempPlaneKey)
        {
          #only count this error if it is a new error
          #and it is in a word that hasn't been deallocated;
          #otherwise go on to the next failure
```

```
            if ($L0ColKey == $L0TempColKey)
            {
              next FAILLOOP;
            }
            elsif ($fails[$tempIndex][12] == 2)
            {
              next FAILLOOP;
            }
            else
            {
              #this is the same plane as the current failure, as long
              #as it is a different col (i.e., not the same bit), it
              #lines up with another failure
              $bitErrors{$L0TempColKey} = 1;
            }
          }
        }
        #if failure entry in the chain is itself chained, then
        #continue with the next link
        if ($fails[$tempIndex][10] > 0)
        {
          #set the index variable to next fail index in the chain
          $tempIndex = $fails[$tempIndex][10];
        }
        #otherwise, add the current failure to the chain
        else
        {
          $fails[$tempIndex][10] = $index;
          #last error for this row, so break from the while() loop
          last;
        }
      }
    }
  }
}
#count the number of unique bit errors that lined up in the word
$errorCnt = keys(%bitErrors);
($verbose > 1 ? print SIMFILE "ERRORCNT: $errorCnt\n" : 0);
#if the number of errors is less than NEC, they won't be corrected
#so they need to be tracked to see how they line up with soft
#errors to create SUE's
if ($errorCnt < $nec)
{
  $SST{$L2Key}{$L1Key}{'BD'}[$errorCnt] += 1;
  #the next lowest value is reduced
  if ($nec > 1)
  {
    $SST{$L2Key}{$L1Key}{'BD'}[($errorCnt - 1)] -= 1;
  }
}
#if the total number of bit errors in the row is equal to the
#number that can be corrected, then need to replace this row with a
#spare if possible
elsif ($errorCnt == $nec)
{
  #need to replace this row if possible
  #if length of array is less than total spares, then there is a
  #spare L0 row available
```

```
  if ($SST{$L2Key}{$L1Key}{'SR'} < $L0RedRow)
  {
    $SST{$L2Key}{$L1Key}{'SR'} += 1;
    #disable all index is current chain with the same row,
    #since this entire row was replaced
    foreach $keyR (@rowChain)
    {
      #set disable flag
      $fails[$keyR][12] = 1;
    }
  }
  #need to swap in a spare at the L1 level (chip)
  else
  {
    $SST{$L2Key}{$L1Key}{'BD'}[$errorCnt] += 1;
    #the next lowest value is reduced
    if ($nec > 1)
    {
      $SST{$L2Key}{$L1Key}{'BD'}[($errorCnt - 1)] -= 1;
    }
  }
}
#if the number of errors is greater than the number that are
#correctable, then there wasn't a spare available to replace the
#device previously so UE's occur if ECC is not available across the
#L1 blocks.  If ECC is available, then need to track the words with
#UE's to find when two of them line up to cause a L1 UE
elsif ($errorCnt > $nec)
{
  #if RAID not implemented, a UE occurs
  $SST{$L2Key}{$L1Key}{'UE'} += 1;
  #deallocate this word - if another bit error occurs in this word,
  #then it will see this word in the chain and know to go on to the
  #next error
  $fails[$index][12] = 2;
  if (!$raid)
  {
    #if error correction is avail., then all spares have been used
    #and any errors are permanent UE's the L0 rows containing these
    #errors will be deallocated
    if ($nec)
    {
      ##deallocate row
      #$SST{$L2Key}{$L1Key}{'DE'}{$L0RowKey} = 1;
      ##record the uncorrectable error
    }
    else
    {
      #if no error correction, then make use of spares even though
      #a UE occurs this will reduce the amount of deallocation that
      #occurs.  ECC allows for spares to be used up before
      #bitErrors > nec if length of array is less than total
      #spares, then there is a spare L0 row available
```

```
    if ($SST{$L2Key}{$L1Key}{'SR'} < $L0RedRow)
    {
      $SST{$L2Key}{$L1Key}{'SR'} += 1;

      #disable all index is current chain with the same row,
      #since this entire row was replaced
      foreach $keyR (@rowChain)
      {
        #set disable flag
        $fails[$keyR][12] = 1;
      }
    }
  }
}
else
{
  #if RAID ECC is implemented across blocks, then for a given L1
  #row, go through L1 Col's.  Then for each of these L1 arrays,
  #look through all other errors that were uncorrectable at the
  #L0 level.  If any of these are in the same L0 row as the
  #current failure, then an L1 uncorrectable error occurs.

  #if no previous L0 UE's in this L1 row, insert current index
  #value and go on
  if($SST{$L2Key}{$L1RowKey}{'CH'} == 0)
  {
    $SST{$L2Key}{$L1RowKey}{'CH'} = $index;
  }
  else
  {
    #reset array each time chain is traversed
    %wordErrors = ();
    #record current error
    $wordErrors{$L1ColKey} = 1;

    #set tempIndex to the index of the first failure that caused
    #an L0 UE.  This index is used to determine which word
    #(plane) had the failure.
    $tempIndex = $SST{$L2Key}{$L1RowKey}{'CH'};
    while(1)
    {
      $tempL1Key = $fails[$tempIndex][5].'-'.
        $fails[$tempIndex][6];
      $L1TempColKey = $fails[$tempIndex][6];
      $L0TempRowKey = $fails[$tempIndex][7];
      $L0TempPlaneKey = $fails[$tempIndex][8];
      #word failure must be in the same L0 row and L0 plane
      #(aka relative word location)
      #and be in a different L1 device
      if ($L0RowKey == $L0TempRowKey and
          $L0PlaneKey == $L0TempPlaneKey)
      {
        if ($L1ColKey != $L1TempColKey)
        {
```

```
                    if ($SST{$L2Key}{$tempL1Key}{'ST'} ne 'DD')
                    {
                      #if L1 chip status is not dead, then count it
                      $wordErrors{$L1TempColKey} = 1;
                      ($verbose > 1 ? print SIMFILE "WORD ERROR IN SAME
                       RELATIVE LOCATION IN DIFFERENT WORD!\n" : 0);
                    }
                    else
                    {
                      ($verbose > 1 ? print SIMFILE "NO ERROR\n" : 0);
                    }
                  }
                  else
                  {
                    #current error is in the same word as a previous error
                    #don't chain this failure; just go on to next failure
                    next FAILLOOP;
                  }
                }
                else
                {
                  ($verbose > 1 ? print SIMFILE "NO ERROR\n" : 0);
                }
                #if failure entry in the chain2 is itself chained,
                #then continue with the next link
                if ($fails[$tempIndex][11] > 0)
                {
                  #set the index variable to next fail index in the chain2
                  $tempIndex = $fails[$tempIndex][11];
                }
                #otherwise, add the current failure to the chain
                else
                {
                  $fails[$tempIndex][11] = $index;
                  #last error in current chain2, so break from loop
                  last;
                }
              }
              #if there was an uncorrectable word error, then record it
              $errorCnt = keys (%wordErrors);
              ($verbose ? print SIMFILE "ERROR CNT: $errorCnt\n" : 0);
              #if two or more errors, then that num of UEs occur
              if ($errorCnt >= 2)
              {
                $SST{$L2Key}{$L1RowKey}{'UE'}[$errorCnt] += 1;
                #the next lowest value is reduced
                if ($errorCnt > 2)
                {
                  $SST{$L2Key}{$L1RowKey}{'UE'}[($errorCnt - 1)] -= 1;
                }
              }
            }
          }
        }
      }
    }
  }
}
```

B.3.3 Device Redundancy

```
sub deviceRedundancy
{
  #check all the possible spares for each key and find the first one
  #that isn't swapped in or marked for being bad from a chip failure
  foreach my $i (1 .. $L1Red)
  {
    #if a spare is found that is usable, mark it and the current
    #bad L1 device (chip) as swapped
    if ($SST{$L2Key}{"S-$i"}{'ST'} ne 'UU')
    {
      $SST{$L2Key}{"S-$i"}{'ST'} = 'UU';
      $SST{$L2Key}{$L1Key}{'ST'} = 'DD'; # probably redundant
      #return success at swapping in spare L1 device
      return 1;
    }
  }
  #need to swap in spare at a higher level (entire L2 card)
  #in future, might let errors build up before doing this
  foreach my $i (1 .. $L2Red)
  {
    #if a spare is found that is usable, mark it
    #and the current bad L2 device (card) as swapped
    if ($SST{"S-$i"}{'ST'} ne 'UU')
    {
      $SST{"S-$i"}{'ST'} = 'UU';
      $SST{$L2Key}{'ST'} = 'DD';
      #return success at swapping in spare L2 device
      return 2;
    }
  }
  return 0;
}
```

B.3.4 Run Simulation

```
sub runSimulation
{
  my ($diffTime1, $diffTime2, $endTime1, $endTime2, $startTime);
  #variables for calculating accuracy
  #my ($acc, $average, $stdev);
  my $filename = join '-', $L2Row, $L2Col, $L2Red, $L1Row, $L1Col,
                           $L1Red, $L0Row, $L0Col, $L0Plane, $L0RedRow,
                           $L0RedCol, $nec, $raid, $lifetime,
                           $pollInterval, $her, $ser, $maxTrials;

  updateMemSize();  #make sure all parameters are updated
  #insert commas intoall numbers before printing them out
  my $spareSizePrnt = commify($totalSpareSize);
  my $memSizePrnt = commify($totalMemSize);
  my $usableSizePrnt = commify($totalUsableSize);
  my $eccSizePrnt = commify($totalECCSize);
  $herMTTFMemPrnt = commify($herMTTFMem);
  $serMTTFMemPrnt = commify($serMTTFMem);
```

```
  #reset variables on each execution
  @herL0StatData = ();
  @serL0StatData = ();
  @herL1StatData = ();
  @serL1StatData = ();

  foreach my $i (1..$maxTrials)
  {
    $startTime = time();
    #reset variables
    %SST = ();
    $count = 0;   #counts number of failures generated
    @fails = ();  #array with all failure info

    #generate all failure times and locations
    sysopen(GENFILE, "gen_$filename.out", O_RDWR | O_TRUNC | O_CREAT)
        or die "Can not open the file gen.out in $path\n";
    generate();

    #record end time to calc amount of time necessary to generate fails
    $endTime1 = time();

    #output for first simulation if verbose variable set selected
    #verbose==1 gives one level of output
    #verbose==2 gives more detailed
    if ($verbose and $i == 1)
    {
      sysopen(SIMFILE, "sim_$filename.out", O_RDWR | O_TRUNC | O_CREAT)
          or die "Can not open the file sim_$filename.out in $path\n";
    }
    simulate();
    if ($verbose and $i == 1)
    {
      close SIMFILE or die "Closing: $!";
    }
    close GENFILE or die "Closing: $!";
  }

  #at given intervals print current failure data to a file
  sysopen(TIMEFILE, "time_$filename.out", O_RDWR | O_TRUNC | O_CREAT)
          or die "Can not open the file time_$filename.out in $path\n";
  print TIMEFILE "L2Row: $L2Row\tL2Col: $L2Col\tL2Red: $L2Red\n";
  print TIMEFILE "L1Row: $L1Row\tL1Col: $L1Col\tL1Red: $L1Red\n";
  print TIMEFILE "L0Row: $L0Row\tL0Plane: $L0Plane\tL0Col: $kVal\t";
  print TIMEFILE "L0RRow: $L0RedRow\tL0RCol: $L0RedCol\n";
  print TIMEFILE "MEM SIZE: $memSizePrnt\tECC SIZE: $eccSizePrnt\t";
  print TIMEFILE "SPARE SIZE: $spareSizePrnt\tUSABLE SIZE:
$usableSizePrnt\n";
  print TIMEFILE "OVERHEAD: $overheadPrnt\nRAID: $raid\nNEC: $nec\n";

  if (!$raid)
  {
    print TIMEFILE "\tL0 SUE\t\t\t\tL0 HUE\n";
    print TIMEFILE
"TIME\tMEAN\tMIN\tMAX\tSTDV\tMEAN\tMIN\tMAX\tSTDV\n";
  }
```

```
else
  {
    print TIMEFILE "\tL0 SUE\t\t\tL0 HUE\t\t\tL1 SUE\t\t\tL1 HUE\n";
    print TIMEFILE "TIME\tMEAN\tMN\tMAX\tSTDV\tMEAN\tMN\tMAX\tSTDV\t";
    print TIMEFILE "MEAN\tMN\tMAX\tSTDV\tMEAN\tMN\tMAX\tSTDV\n";
  }

  #add failures at each polling interval, for all memory trials to a
  #stats data structure this will allow the statistical results of all
  #trials to be printed to the TIMEFILE
  for (my $i = 0; $i <= $lifetime; $i += $pollInterval)
  {
    if (!$raid)
    {
      my $herL0Stat = Statistics::Descriptive::Sparse->new;
      my $serL0Stat = Statistics::Descriptive::Sparse->new;
      $herL0Stat->add_data(@{$herL0StatData[$i]});
      $serL0Stat->add_data(@{$serL0StatData[$i]});
      #print stat results for the total series of executions to a file
      printf TIMEFILE "$i\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
        $serL0Stat->mean, $serL0Stat->min, $serL0Stat->max,
        $serL0Stat->standard_deviation,
        $herL0Stat->mean, $herL0Stat->min, $herL0Stat->max,
        $herL0Stat->standard_deviation;
    }
    else
    {
      my $herL0Stat = Statistics::Descriptive::Sparse->new;
      my $serL0Stat = Statistics::Descriptive::Sparse->new;
      my $herL1Stat = Statistics::Descriptive::Sparse->new;
      my $serL1Stat = Statistics::Descriptive::Sparse->new;
      $herL0Stat->add_data(@{$herL0StatData[$i]});
      $serL0Stat->add_data(@{$serL0StatData[$i]});
      $herL1Stat->add_data(@{$herL1StatData[$i]});
      $serL1Stat->add_data(@{$serL1StatData[$i]});

      #print stat results for the total series of executions to a file
      printf TIMEFILE "$i\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t
                       %d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
        $serL0Stat->mean, $serL0Stat->min, $serL0Stat->max,
          $serL0Stat->standard_deviation,
        $herL0Stat->mean, $herL0Stat->min, $herL0Stat->max,
          $herL0Stat->standard_deviation,
        $serL1Stat->mean, $serL1Stat->min, $serL1Stat->max,
          $serL1Stat->standard_deviation,
        $herL1Stat->mean, $herL1Stat->min, $herL1Stat->max,
          $herL1Stat->standard_deviation;
    }
  }
  #Generate data structure for plotting with GDGraph::boxplot
  #The first line is the label for each of the memory configurations
  #Each memory configurations will have been executing maxTrials
  #num times and the statistical results will be plotted
  push @{$serL0PlotData[0]}, "Trial $execCnt";
  push @{$herL0PlotData[0]}, "Trial $execCnt";
  push @{$serL1PlotData[0]}, "Trial $execCnt";
  push @{$herL1PlotData[0]}, "Trial $execCnt";
```

```
  #copy error counts for trials at a given time in the memory lifetime
  for my $i (0..$maxTrials)
  {
    push @{$serL0PlotData[1][$execCnt]}, $serL0StatData[$lifetime][$i];
    push @{$herL0PlotData[1][$execCnt]}, $herL0StatData[$lifetime][$i];
    push @{$serL1PlotData[1][$execCnt]}, $serL1StatData[$lifetime][$i];
    push @{$herL1PlotData[1][$execCnt]}, $herL1StatData[$lifetime][$i];
  }
  #track number of memory configs executed since the script was started
  $execCnt += 1;
  #update the Execute button to represent the num configs executed
  $executeSim_b->configure(-text => "Execute $execCnt");
  close TIMEFILE or die "Closing: $!";
  openFile("time_$filename.out");
}
```

B.4 Graphical User Interface Subroutines

B.4.1 Commify

```
sub commify
{
  my $text = reverse $_[0];
  $text =~ s/(\d\d\d)(?=\d)(?!\d*\.)/$1,/g;
  return scalar reverse $text;
}
```

B.4.2 Plot Results

```
sub plotResults
{
  my $serL0graph = new GD::Graph::boxplot();
  my $herL0graph = new GD::Graph::boxplot();
  my $serL1graph = new GD::Graph::boxplot();
  my $herL1graph = new GD::Graph::boxplot();
  $serL0graph->set(
      x_label           => 'Memory Configuration',
      y_label           => 'Number of L0 Soft Uncorrectable Errors',
      title             => 'Memory Simulation L0 SUE Results',
  );
  $herL0graph->set(
      x_label           => 'Memory Configuration',
      y_label           => 'Number of L0 Hard Uncorrectable Errors',
      title             => 'Memory Simulation L0 HUE Results',
  );
  $serL1graph->set(
      x_label           => 'Memory Configuration',
      y_label           => 'Number of L1 Soft Uncorrectable Errors',
      title             => 'Memory Simulation L1 SUE Results',
  );
  $herL1graph->set(
      x_label           => 'Memory Configuration',
      y_label           => 'Number of L1 Hard Uncorrectable Errors',
      title             => 'Memory Simulation L1 HUE Results',
  );
```

```
  my $serL0gd = $serL0graph->plot( \@serL0PlotData );
  my $herL0gd = $herL0graph->plot( \@herL0PlotData );
  my $serL1gd = $serL1graph->plot( \@serL1PlotData );
  my $herL1gd = $herL1graph->plot( \@herL1PlotData );

  if ($serL0gd)
  {
    open(IMG, '>serL0.png') or die $!;
    binmode IMG;
    print IMG $serL0gd->png;
    close IMG;
  }
  else
  {
    print "EMTPY SET: SER L0\n";
  }

  if ($herL0gd)
  {
    open(IMG, '>herL0.png') or die $!;
    binmode IMG;
    print IMG $herL0gd->png;
    close IMG;
  }
  else
  {
    print "EMTPY SET: HER L0\n";
  }

  if ($serL1gd)
  {
    open(IMG, '>serL1.png') or die $!;
    binmode IMG;
    print IMG $serL1gd->png;
    close IMG;
  }
  else
  {
    print "EMTPY SET: SER L1\n";
  }

  if ($herL1gd)
  {
    open(IMG, '>herL1.png') or die $!;
    binmode IMG;
    print IMG $herL1gd->png;
    close IMG;
  }
  else
  {
    print "EMTPY SET: HER L1\n";
  }
}
```

B.4.3 Update Memory Size

```
sub updateMemSize
{
  # Determine number of bits used for error correcting codes
  # 'k' data bits and 'r' check bits are calculated
  $rVal = $nec * (log($kVal)/log(2) + 1);
  # add one more check bit for RAID with NEC or SEC
  ($nec <= 2 and $raid ? $rVal += 1: 0);
  $L0Col = $kVal + $rVal;

  # Total memory size is size of all usable and spare memory
  $L0UsableSize = $kVal * $L0Row * $L0Plane;
  $L0ECCSize = $rVal * $L0Row * $L0Plane;
  $L0SpareSize = ($L0RedRow * $L0Plane * $L0Col) +
                 ($L0RedCol * $L0Row);
  $L0MemSize = $L0UsableSize + $L0ECCSize + $L0SpareSize;

  $L1UsableSize = $L0UsableSize * $L1Row * $L1Col;
  $L1ECCSize = ($L0ECCSize * $L1Row * $L1Col) +
               ($raid * $L0MemSize * $L1Row);
  $L1SpareSize = ($L0SpareSize * $L1Row * $L1Col) +
                 ($L0MemSize * $L1Red);
  $L1MemSize = $L1UsableSize + $L1ECCSize + $L1SpareSize;

  $totalUsableSize = $L1UsableSize * $L2Row * $L2Col;
  $totalECCSize = $L1ECCSize * $L2Row * $L2Col;
  $totalSpareSize = ($L1SpareSize * $L2Row * $L2Col) +
                    ($L1MemSize * $L2Red);
  $totalMemSize = $totalUsableSize + $totalECCSize + $totalSpareSize;

  $spareSizePrnt = commify($totalSpareSize);
  $memSizePrnt = commify($totalMemSize);
  $usableSizePrnt = commify($totalUsableSize);
  $eccSizePrnt = commify($totalECCSize);
  $overheadPrnt = commify(nearest(0.01, ($totalMemSize -
                                         $totalUsableSize) /
                                  $totalUsableSize * 100));

  #Calculate MTTF from hard and soft errors
  #print "HER: $her\tSER: $ser\tTOTALMEM: $totalMemSize\n";
  $herMTTFMem = nearest(0.001, (1e9 / ($her * $totalMemSize)));
  $serMTTFMem = nearest(0.001, (1e9 / ($ser * $totalMemSize)));
  $herMTTFBit = 1e9 / $her;

  $memSize_l->configure(-text => "Total: $memSizePrnt bits\n
                        Spare: $spareSizePrnt bits\n
                        ECC: $eccSizePrnt bits\n
                        Usable: $usableSizePrnt bits\n
                        Overhead: $overheadPrnt%");
  $mttf_l->configure(-text => "H: $herMTTFMem\nS: $serMTTFMem");
  $chkBits_l->configure(-text => "Data: $kVal\nCheck: $rVal");
  return 1;
}
```

B.4.4 Menubar Subroutines

```
#create EDIT menu item in menubar
sub edit_menuitems
{
  [
    [qw/command Undo -accelerator Ctrl-Z -command/ => \&tUndo],
    [qw/command Redo -accelerator Ctrl-Y -command/ => \&tRedo],
    '',
    [qw/command Copy -accelerator Ctrl-C -command/ => \&tCopy],
    [qw/command Cut -accelerator Ctrl-X -command/ => \&tCut],
    [qw/command Paste -accelerator Ctrl-V -command/ => \&tPaste],
    '',
    [qw/command/,'Select All',qw/-accelerator Ctrl-A
     -command/ => \&tSelectAll],
    [qw/command/,'Unselect All',qw/-command/ => \&tUnSelectAll],
  ]
}
#create FILE menu item in menubar
sub file_menuitems
{
  [
    [qw/command New -command/ => \&newFile],
    [qw/command Open -command/ => \&openFile],
    '',
    [qw/command Save -command/ => \&saveFile],
    [qw/command/,'Save As',qw/-command/ => \&saveAs],
    [qw/command/,'Save All',qw/-command/ => \&saveAll],
    '',
    [qw/command Close -command/ => \&closeFile],
    [qw/command/,'Close All',qw/-command/ => \&closeAll],
    '',
    [qw/command Exit -command/ => \&exitProg]
  ];
}
#create HELP menu item in menubar
sub help_menuitems
{
  [
    [qw/command/,'Help Topics',qw/-command/ => sub {print "Help
Topics\n"}],
    '',
    [qw/command About -command/ => \&showAbout],
  ];
}
#create SEARCH menu item in menubar
sub search_menuitems
{
  [
    [qw/command Find -command/ => \&tFind],
    [qw/command/,'Find Next',qw/-command/ => \&tNext],
    [qw/command/,'Find Previous',qw/-command/ => \&tPrev],
    [qw/command/,'Replace',qw/-command/ => \&tReplace],
  ];
}
```

```
#close all programs, saving if necessary
sub closeAll
{
  #Save all pages before closing
  for (sort keys %pageName)
  {
    if($pageName{$_})
    {
      $files_nb->raise($pageName{$_});
      &closeFile();
    }
  }
}

#standard CLOSE FILE function
sub closeFile
{
  $currentPage = $files_nb->raised();
  my $page2delete = $currentPage;
  $files_nb->delete($pageName{$currentPage});
  delete $pageName{$page2delete};
}

#close all files and exit program
sub exitProg
{
  &closeAll;
  exit 1;
}

#create a new notebook page with text box and insert data into it
sub newFile
{
  #Are there any open pages?
  if(($files_nb->raised()) eq "")
  {
    $fileNum = 1;
  }
  else
  {
    $fileNum += 1;
  }
  #Create a new page
  #if a filename was defined by open file, use it
  #otherwise, name Untitled #
  if($fName)
  {
    $fileName[$fileNum] = $fName;
    $fName = "";
  }
  else
  {
    $fileName[$fileNum] = "Untitled $fileNum";
  }
  $pageName{$fileNum} = $fileNum;
  my $files_pg = $files_nb->add($pageName{$fileNum},
                                -label=> $fileName[$fileNum]);
```

```
  $files_nb->Resize();
  # Create a scrolled text widget and place inside the notebook page
  $files_t[$fileNum] = $files_pg->Scrolled("Text", -height => 8,
                                           -width => 70,
                                           -wrap => 'none',
                                           -background => 'white',
                                           -scrollbars => "osoe",
                                           -font => 'Courier 11');
  $files_t[$fileNum]->pack(-side => 'bottom',
                           -fill => 'both',
                           -expand => 1);
  $files_nb->raise($pageName{$fileNum});
  $top_w->update();
}

#open a file and then send it to a new notebook page
sub openFile
{
  #if parameter was passed to open, open that file;
  #otherwise, give open dialog box
  if(@_)
  {
    $fName = shift(@_);
  }
  else
  {
    #define acceptable types of files to be read in
     my $types = [
        ['All Files',        '*',    ],
        ['Text Files',   ['.out', '.txt', 'TEXT']],
     ];
     #return user selection to filename
     $fName = $top_f->getOpenFile(-filetypes=>$types,
                                  -initialdir => $path);
  }
  if ($fName =~ /(.*)[\/\\](.+\.?.*)/)
  {
    $path = $1;
    $fName = $2;
    chdir "$path" or die "Can't cd to $path: $!\n";
  }

  #Open the file in a new page if one was defined and the file is
  #not already opened
  #if same file is open, close current version and reopen new version
  for (sort keys %pageName)
  {
    if($fileName[$pageName{$_}] eq $fName)
    {
      $files_nb->raise($pageName{$_});
      &closeFile();
    }
  }
  if($fName)
  {
    &newFile;
  }
```

```
  else
  {
    return 0;
  }
  sysopen(INFILE, "$fileName[$fileNum]", O_RDONLY)
      or die "Can not open the file $fileName[$fileNum] in $path\n";

  #use filename to title notebook tab
  $files_nb->pageconfigure($pageName{$fileNum},
                           -label=> "$fileName[$fileNum]");
  while (<INFILE>)
  {
    $files_t[$fileNum]->insert('end', $_);
  }
  close INFILE or die "Closing: $!";
}

#standard SAVE ALL function
sub saveAll
{
  #save current page that is raised
  $currentPage = $files_nb->raised();
  #loop through each page to save them
  for (sort keys %pageName)
  {
    $files_nb->raise($pageName{$_});
    &saveFile();
  }
  $files_nb->raise($pageName{$currentPage});
}

#standard SAVE AS function
sub saveAs
{
  $currentPage = $files_nb->raised();
  $files_t[$currentPage]->FileSaveAsPopup();
  #retrieve filename of saved file
  my $saveAsName = $files_t[$currentPage]->FileName();
  if($saveAsName eq "") #canceled save
  {
    return 0;
  }
  else
  {
    if($saveAsName =~ /.*[\/\\](.*)/)
    {
      $fileName[$currentPage] = $1;
    }
    else
    {
      $fileName[$currentPage] = $saveAsName;
    }
    # Show the file name in the page tab
    $files_nb->pageconfigure($pageName{$currentPage},
                             -label=>$fileName[$currentPage]);
  }
}
```

```
#standard SAVE function
sub saveFile
{
  $currentPage = $files_nb->raised();
  my $infoBox = $top_f->messageBox(-title => 'Are You Sure?',
          -message => "Do you want to overwrite the existing file(s)?",
          -type => 'YesNo',
          -icon => 'question',
          -default => 'no'
      );
  if($infoBox =~ /yes/i)
  {
    #display Save dialog box
    $files_t[$currentPage]->Save($fileName[$currentPage]);
    # Retrieve filename
    my $saveName = $files_t[$currentPage]->FileName();
    if($saveName eq "") #canceled save
    {
      return 0;
    }
    else
    {
      #only capture the filename and not the path
      if($saveName =~ /.*[\/\\](.*)/)
      {
        $fileName[$currentPage] = $1;
      }
      else
      {
        $fileName[$currentPage] = $saveName;
      }
      # Set label of page to filename.
      $files_nb->pageconfigure($pageName{$currentPage},
                       -label=>$fileName[$currentPage]);
    }
  }
}

#check to see if file should be saved before close if it has been
changed
sub shouldSave
{
  $currentPage = $files_nb->raised();
  my $file = $files_nb->pagecget($pageName{$currentPage}, -label);
  my $save_d = $top_f->Dialog(-title=>"Save File?",
                              -text=>" Do you want to save $file? ",
                              -bitmap=>'question',
                              -default_button=>'No',
                              -buttons=>[qw/Yes No Cancel/]);
  return $save_d->Show();
}
```

B.4.5 Text Editor Subroutines

```
#copy text selection
sub tCopy
{
  $currentPage = $files_nb->raised();
  #$files_t[$currentPage]->clipboardColumnCopy();
  $files_t[$currentPage]->insertControlCode();
}

#cut text selection
sub tCut
{
  $currentPage = $files_nb->raised();
  $files_t[$currentPage]->clipboardCut();
}

#paste from clipboard
sub tPaste
{
  $currentPage = $files_nb->raised();
  $files_t[$currentPage]->clipboardPaste();
}

#undo last action
sub tUndo
{
  $currentPage = $files_nb->raised();
  $files_t[$currentPage]->undo();
}

#redo last action
sub tRedo
{
  $currentPage = $files_nb->raised();
  $files_t[$currentPage]->redo();
}

#select all text
sub tSelectAll
{
  $currentPage = $files_nb->raised();
  $files_t[$currentPage]->selectAll();
}

#unselect all text
sub tUnSelectAll
{
  $currentPage = $files_nb->raised();
  $files_t[$currentPage]->unselectAll();
}
```

```
#find text
sub tFind
{
  $currentPage = $files_nb->raised();
  $files_t[$currentPage]->FindPopUp();
}

#find next match
sub tNext
{
  $currentPage = $files_nb->raised();
  $files_t[$currentPage]->FindSelectionNext();
}

#find prev match
sub tPrev
{
  $currentPage = $files_nb->raised();
  $files_t[$currentPage]->FindSelectionPrevious();
}

#search and replace
sub tReplace
{
  $currentPage = $files_nb->raised();
  $files_t[$currentPage]->findandreplacepopup();
}

#show About information box
sub showAbout{
  my $aboutString = "Fault Tolerant Memory Simulator v0.5\n\n";
  $top_w->messageBox(-title => 'About',
                     -message => "$aboutString", -type => 'OK');
```

LIST OF REFERENCES

1. M. P. Frank, "Reversibility for efficient computing," Ph.D. dissertation, Dept. Elect. Eng. and Comp. Sci., Massachusetts Institute of Technology, Cambridge, MA, 1999.

2. M. Butts, A. DeHon, and S. C. Goldstein, "Molecular electronics: devices, systems and tools for gigagate, gigabit chips," in *Proc. IEEE/ACM International Conference on Computer Aided Design*, pp. 433-440, Nov. 2002.

3. S. C. Goldstein and M. Budiu, "NanoFabrics: spatial computing using molecular electronics," in *Proc. 28th Annual International Symposium on Computer Architecture*, pp. 178-189, July 2001.

4. A. DeHon, "Array-based architecture for FET-based nanoscale electronics," *IEEE Trans. on Nanotechnology*, vol. 2, no. 1, March 2003.

5. M. Mishra and S. C. Goldstein, "Scalable defect tolerance for molecular electronics," in *Workshop on Non-Silicon Computing, 8th International Symposium on High-Performance Computer Architecture*, (Cambridge, MA), Feb. 2002.

6. K. Nikolić, A. Sadek, and M. Forshaw, "Fault-tolerant techniques for nanocomputers," *Nanotechnology*, vol. 13, pp. 357-362, May 2002.

7. J. von Neumann, "Probabilistic logic and the synthesis of reliable organisms from unreliable components," in *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton, NJ: Princeton University Press, 1956.

8. C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable processors in the billion-transistor era: IRAM," *Computer*, vol. 30, no. 9, pp. 75-78, Sept. 1997.

9. P. Van Zant, *Microchip Fabrication: A Practical Guide to Semiconductor Processing.* 4th ed., New York, NY: McGraw-Hill, 2000, pp. 49-110.

10. H. Kitajima and Y. Shiramizu, "Requirements for contamination control in the gigabit era," *IEEE Trans. on Semiconductor Manufacturing*, vol. 10, no. 2, pp. 267-272, May 1997.

11. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* New York, NY: 1990.

12. K. Chakraborty and P. Mazumder, *Fault-Tolerance and Reliability Techniques for High-Density Random-Access Memories*. Upper Saddle River, NJ: Prentice Hall, 2002, pp. 2-18.

13. D. J. W. Noorlag, L. M. Terman, and A. G. Konheim, "The effect of alpha-particle-induced soft errors on memory systems with error correction," *IEEE Journal of Solid-State Circuits*, vol SC-15, no. 3, pp. 319-325, June 1980.

14. A. M. Saleh, J. J. Serrano, and J. H. Patel, "Reliability of scrubbing recovery-techniques for memory systems," *IEEE Trans. on Reliab.*, vol. 39, no. 1, pp. 114-122, April 1998.

15. P. Hazucha, C. Svensson, and S. A. Wender, "Cosmic-ray soft error rate characterization of a standard 0.6-μm CMOS process," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 10, pp. 1422-1429, Oct. 2000.

16. S. Schaefer, "DRAM Soft Error Rate Calculations," Micron Design Line, vol. 3, no. 1, 1994.

17. A. H. Johnston, G. M. Swift, and D. C. Shaw, "Impact of CMOS scaling on single-event hard errors in space systems," in *Proc. IEEE Symposium on Low Power Electronics*, pp. 88-89, Oct. 1995.

18. R. Baumann, "The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction," *IEEE International Electron Devices Meeting*, pp. 329-332, Dec. 2002.

19. J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, et al., "IBM experiments in soft fails in computer electronics (1978-1994)," *IBM J. Res. Develop.*, vol. 40, no. 1, pp. 3-17, Jan. 1996.

20. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. 3rd ed., New York, NY: Morgan Kaufmann, 2003, pp. 11-56.

21. N. Sakashita, F. Okuda, K. Shimomura, H. Shimano, M. Hamada, T. Tada, S. Komori, K. Kyuma, A. Yasuoka, and H. Abe, "A built-in self-test circuit with timing margin test function in a 1Gbit synchronous DRAM," in *Proc. International Test Conference*, pp. 319-324, 1996.

22. K. Chakraborty, S. Kulkarni, M. Bhattacharya, P. Mazumder, and A. Gupta, "A physical design tool for built-in self-repairable RAMs," *IEEE Trans. on VLSI Systems*, vol. 9, no. 2, pp. 352-364, April 2001.

23. K. Zarrineh and S. J. Upadhyaya, "On programmable memory built-in self test architectures," in *Proc. Design, Automation and Test*, pp. 708-713, March 1999.

24. R. Treuer and V. K. Agarwal, "Built-in self-diagnosis for repairable embedded RAMs," *IEEE Design & Test of Computers*, vol 10, no. 2, pp. 24-33, June 1993.

25. C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: a state-of-the-art review," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 124-134, March 1984.

26. K. Furutani, K. Arimoto, H. Miyamoto, T. Kobayashi, K. Yasuda, and K. Mashiko, "A built-in hamming code ECC Circuit for DRAM's," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 1, pp. 50-56, Feb. 1999.

27. Sematech. (July 1, 2003) International Technology Roadmap for Semiconductors. [Online] Available: http://public.itrs.net

28. M. R. Stan, P. D. Franzon, S. C. Goldstein, J. C. Lach, and M. M. Ziegler, "Molecular elctronics: from devices and interconnect to circuits and architecture," in *Proc. of the IEEE*, vol. 91, no. 11, pp. 1940-1957, Nov. 2003.

29. X. Duan, Y. Huang, and C. Lieber, "Nanowire nanoelectronics assembled from the bottom-up," in *Molecular Nanoelectronics*, M. A. Reed and T. Lee, Eds. Stevenson Ranch, CA: American Scientific Publishers, 2003, pp. 199-227.

30. P. Avouris, J. Appenzeller, R. Martel, and S. J. Wind, "Carbon nanotube electronics," in *Proc. of the IEEE*, vol 91, no. 11, pp. 1772-1784, Nov. 2003.

31. J. Guo, M. Lundstrom, and S. Datta, "Performance projections for ballistic carbon nanotube field-effect transistors," *Applied Physics Letters*, vol. 80, no. 17, pp. 3192-3194, April 2002.

32. J. R. Heath, R. S. Williams, and P. J. Kuekes, "Chemically synthesized and assembled electronics devices," U.S. Patent 6 459 095, Oct. 1, 2002.

33. W. J. Gallagher, J. H. Kaufman, S. S. Papworth, and R. E. Scheuerlein, "Magnetic memory array using magnetic tunnel junction devices in the memory cells," U.S. Patent 5 640 343, June 17, 1997.

34. T. Rueckes, K. Kim, E. Joselevich, G. Y. Tseng, C. Cheung, and C. M. Lieber, "Carbon nanotube-based nonvolatile random access memory for molecular computing," *Science*, vol. 289, pp. 94-97, July 2000.

35. P. J. Kuekes and R. S. Williams, "Demultiplexer for a molecular wire crossbar network," U.S. Patent 6 256 767, July 3, 2001.

36. G. Yang and T. Fuja, "The reliability of systems with two levels of fault tolerance: the return of the 'birthday surprise'," *IEEE Trans. on Computers*, vol. 41, no. 11, pp. 1490-1496, Nov. 1992.

37. C. H. Stapper and H. S. Lee, "Synergistic fault-tolerance for memory chips," *IEEE Trans. on Computers*, vol. 41, no. 9, pp. 1078-1086, Sept. 1992.

38. T. Chen and G. Sunada, "Design of a self-testing and self-repairing structure for highly hierarchical ultra-large capacity memory chips," *IEEE Trans. on VLSI Systems*, vol. 1, no. 2, pp. 88-97, June 1993.

39. T. Hirose, H. Kuriyama, S. Murakami, K. Yuzuriha, T. Mukai, K. Tsutsumi, Y. Nishimura, Y. Kohno, and K. Anami, "A 20-ns 4-Mb CMOS SRAM with hierarchical word decoding architecture," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, pp. 1068-1074, Oct. 1990.

40. T. Chen, D. Louderback, and G. Sunada, "Optimization of the number of levels of hierarchy in large-scale hierarchical memory systems," in *Proc. IEEE Symposium on Circuits and Systems*, 1992, vol. 2, pp. 2104-2107.

41. W. B. Culbertson, R. Amerson, R. J. Carter, P. Kuekes, and G. Snider, "Defect tolerance on the Teramac custom computer," in *Proc. 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 116-123.

42. J. R. Heath, P. J. Kuekes, G. S. Snider, and R. S. Williams, "A defect-tolerant architecture: opportunities for nanotechnology," *Science*, vol. 280, pp. 1716-1721, June 1998.

43. T. J. Dell, "A white paper on the benefits of Chipkill-correct ECC for PC server main memory," IBM Microelectronics Division, Nov. 1997.

44. D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *ACM SIGMOD Record*, vol. 17, no. 3, pp. 109-116, 1988.

45. K. A. Mehdi and J. M. Kontoleon, "Design and analysis of a self-testing-and-self-repairing random access memory 'STAR-RAM' with error correction," *Microelectron. Reliab.*, vol. 38, no. 4, pp. 605-617, 1998.

46. K. A. Mehdi and J. M. Kontoleon, "A rapid SMART approach for reliability and failure mode analysis of memory and large systems," *Microelectron. Reliab.*, vol. 36, no. 10, pp. 1547-1556, 1996.

47. C. L. Chen and R. A. Rutledge, "Fault-tolerant memory simulator," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 184-194, March 1984.

48. M. R. Libson and H. E. Harvey, "A general-purpose memory reliability simulator," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 196-205, March 1984.

49. D. M. Blough, "Performance evaluation of a reconfiguration-algorithm for memory arrays containing clustered faults," *IEEE Trans. on Reliab.*, vol. 45, no. 2, pp. 274-284, June 1996.

50. A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. Lie, D. D. Mannaru, A. Riska, D. Milojicic, "Susceptibility of modern systems and software to soft errors," *HP Labs Tech. J.*, vol. 43, March 2001.

51. F. Peper, J. Lee, F. Abo, T. Isokawa, S. Adachi, N. Matsui, and S. Mashiko, "Fault-tolerance in nanocomputers: a cellular array approach," *IEEE Trans. on Nanotechnology*, vol. 3, no. 1, pp, 187-201, March 2004.

52. D. C. Bossen, C. L. Chen, and M. Y. Hsiao, "Fault alignment exclusion for memory using address permutation," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 170-176, March 1984.

53. P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proc. IEEE International Conference on Dependable Systems and Networks*, pp. 389-398, June 2002.

54. I. Koren and Z. Koren, "Defect tolerance in VLSI circuits: techniques and yield analysis," in *Proc. of the IEEE*, vol. 86, no. 9, pp. 1819-1836, Sept. 1998.

BIOGRAPHICAL SKETCH

Casey Jeffery is a graduate student in the Electrical and Computer Engineering Department at the University of Florida. He will graduate in December 2004 with a Master of Science degree in computer engineering and will continue working toward a Ph.D. degree with a focus on computer architecture. Casey completed his Bachelor of Science degree in the same area at the South Dakota School of Mines and Technology.

At present, Casey has completed four internships with Intel Corporation and is working on his fifth for a total of more than two-and-a-half years with the company. He has worked at four different sites throughout the United States in a variety of areas, including microprocessor architecture R&D, product engineering, and software engineering. He also worked as a teaching assistant for junior- and senior-level undergraduate digital design laboratories.