8-1-2012

Reconfigurable RRAM-based computing: A Case study for reliability enhancement

Matthew Catanzaro

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Reconfigurable RRAM-based Computing: A Case Study for Reliability Enhancement

by

Matthew M. Catanzaro

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by
Assistant Professor Dr. Dhireesha Kudithipudi
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2012

Approved by:

Dr. Dhireesha Kudithipudi, Assistant Professor
Thesis Advisor, Department of Computer Engineering

Dr. Sonia Lopez Alarcon, Assistant Professor
Committee Member, Department of Computer Engineering

Dr. Roy Melton, Senior Lecturer
Committee Member, Department of Computer Engineering
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Reconfigurable RRAM-based Computing: A Case Study for Reliability Enhancement

I, Matthew M. Catanzaro, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Matthew M. Catanzaro

Date
Abstract

Emerging hybrid-CMOS nanoscale devices and architectures offer greater degree of integration and performance capabilities. However, the high power densities, hard error frequency, process variations, and device wearout affect the overall system reliability. Reactive design techniques, such as redundancy, account for component failures by mitigating them to prevent system failures. These techniques incur high area and power overhead. This research focuses on exploring hybrid CMOS/Resistive RAM (RRAM) architectures that enhance the system reliability by performing computation in RRAM cache whenever CMOS logic units fail, essentially masking the area overhead of redundant logic when not in use. The proposed designs are validated using the Gem5 performance simulator and McPAT power simulator running single-core SPEC2006 benchmarks and multi-core PARSEC benchmarks.

The simulation results are used to evaluate the efficacy of reliability enhancement techniques using RRAM. The average runtime when using RRAM for functional unit replacement was between ~1.5 and ~2.5 times longer than the baseline for a single-core architecture, ~1.25 and ~2 times
longer for an 8-core architecture, and \( \sim 1.2 \) and \( \sim 1.5 \) times longer for a 16-core architecture. Average energy consumption when using RRAM for functional unit replacement was between \( \sim 2 \) and \( \sim 5 \) times more than the baseline for a single-core architecture, and \( \sim 1.25 \) and \( \sim 2.75 \) times more for multi-core architectures. The performance degradation and energy consumption increase is justified by the prevention of system failure and enhanced reliability. Overall, the proposed architecture shows promise for use in multi-core systems. Average performance degradation decreases as more cores are used due to more total functional units being available, preventing a slow RRAM functional unit from becoming a bottleneck.
# Contents

Abstract ..................................................................... iii

## 1 Background ......................................................... 1

1.1 Introduction ...................................................... 1
1.2 Reliability ......................................................... 3
  1.2.1 Hard Errors ............................................. 4
  1.2.2 Reliability Theory ................................. 5
  1.2.3 Majority Voter Circuits ......................... 6
  1.2.4 Standby Redundancy ......................... 7
  1.2.5 Dynamic Reliability Management (DRM) .... 8
1.3 Resistive RAM .............................................. 10
1.4 Hybrid CMOS/RRAM Architectures ................. 13
  1.4.1 Crossbar Architecture ......................... 13
  1.4.2 One Transistor and One Resistor (1T1R) Architecture 15

## 2 Related Work .................................................... 16

2.1 Memory-Based Computing .................................. 16
2.2 Spin-Torque Transfer Magnetoresistive RAM Computing 18
2.3 Reliability Metrics .......................................... 19
  2.3.1 Mean Time to Failure ......................... 19
  2.3.2 Failures in Time ................................. 21

## 3 Proposed Architecture Design .......................... 23

3.1 Test-Case Architecture Implementation .................. 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>RRAM Arrays</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>RRAM Controller</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>LUT Logic Mapping</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Integer ALU</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Integer Multiplier</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Floating-Point Adder</td>
<td>39</td>
</tr>
<tr>
<td>5</td>
<td>Simulation Platform</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>Gem5 Simulator</td>
<td>43</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Gem5 Modifications</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>McPAT Simulator</td>
<td>45</td>
</tr>
<tr>
<td>5.2.1</td>
<td>McPAT Modifications</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>SPEC2006 Benchmark Suite</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>PARSEC Benchmark Suite</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>Simulation Results and Analysis</td>
<td>49</td>
</tr>
<tr>
<td>6.1</td>
<td>Single Core Simulations</td>
<td>49</td>
</tr>
<tr>
<td>6.2</td>
<td>Multi-core Simulations</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>70</td>
</tr>
<tr>
<td>8</td>
<td>Future Work</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>74</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Characteristics of Memory Device Parameters at Nanoscale . 12
3.1 Alpha Processor Conversions . . . . . . . . . . . . . . . . . . 27
3.2 Test-Case Architecture Implementations . . . . . . . . . . . 27
3.3 RRAM Functional Data [43] . . . . . . . . . . . . . . . . . . 30
4.1 Functional Unit Decomposition Statistics . . . . . . . . . . . 35
4.2 LUT-Based Floating-Point Adder Decomposition . . . . . . 41
5.1 Functional Unit Pool Options in the O3CPU . . . . . . . . . . 44
5.2 Cache Configuration Options in the O3CPU . . . . . . . . . . 45
5.3 SPEC2006 Benchmark Attributes . . . . . . . . . . . . . . . 47
5.4 PARSEC Benchmark Attributes . . . . . . . . . . . . . . . . 48
6.1 Single Core Simulation Configurations . . . . . . . . . . . . . 51
6.2 Multi-Core Simulation Configurations . . . . . . . . . . . . . 61
List of Figures

1.1 Triple Redundant Group of Functional Units with Voter Logic 7
1.2 Standby Set of Redundant Functional Units 8
1.3 Bipolar Memristor Curve with ±2 V Threshold Switching 12
1.4 Sneak Paths Inherent in the Crossbar Architecture 14
1.5 1T1R Architecture 15

2.1 Representation of STT-MRAM Processor Pipeline [21] 19

3.1 Representation of the Proposed Hybrid CMOS/RRAM Architecture for Reliability Enhancement 24
3.2 Alpha 21264 Microarchitecture [12] 26
3.3 RRAM Controller Block Diagram 33

4.1 RRAM Array Representation for 4-Bit ALU 36
4.2 Block-Level Schematic of LUT-Based 32-Bit ALU Design 37
4.3 32-Bit LUT-Based Multiplier Design 38
4.4 32-Bit Floating-Point Addition Process 40

5.1 Representation of the Custom Tuned Simulation Framework 42

6.1 SPEC2006 Benchmark Profiles 50
6.2 SPEC2006 Benchmark Energy Usage for Different Cache Levels 52
6.3 SPEC2006 Benchmark Timing for Different Cache Levels 53
6.4 SPEC2006 Benchmark Runtime With L2 RRAM Cache and Replaced LUT-Based FUs 55
6.5 SPEC2006 Benchmark Runtime With L3 RRAM Cache and Replaced LUT-Based FUs ........................................ 56
6.6 SPEC2006 Benchmark Energy With L2 RRAM Cache and Replaced LUT-Based FUs ........................................ 57
6.7 SPEC2006 Benchmark Energy With L3 RRAM Cache and Replaced LUT-Based FUs ........................................ 58
6.8 SPEC2006 Benchmark Energy With L3 RRAM Cache and Replaced LUT-Based FUs ........................................ 60
6.9 PARSEC Benchmark Runtime for L2 SRAM and L2 RRAM Caches ............................................................... 62
6.10 PARSEC Benchmark Energy for L2 SRAM and L2 RRAM Caches ............................................................... 63
6.11 PARSEC Benchmark Runtime With 8 Cores, L2 RRAM and Replaced LUT-Based FUs ...................................... 64
6.12 PARSEC Benchmark Runtime With 16 Cores, L2 RRAM and Replaced LUT-Based FUs ...................................... 65
6.13 PARSEC Benchmark Energy With 8 Cores, L2 RRAM and Replaced LUT-Based FUs ........................................ 67
6.14 PARSEC Benchmark Energy With 16 Cores, L2 RRAM and Replaced LUT-Based FUs ........................................ 68
Chapter 1

Background

1.1 Introduction

Hard error frequency in complementary metal-oxide-semiconductor (CMOS) integrated circuits increases exponentially as CMOS technology scales, which is a growing concern for reliable system design [1][23]. The number of hard errors that lead to failures have increased by 274% from 180nm to 65nm technology [38]. CMOS scaling increases hard error rates due to smaller devices having increased power and heat densities, accelerating wear-out of the silicon. Increasing transistor counts inherently inflate the error rate as well. Significant research efforts have been made in the domain of hardware reliability, and hard error prevention and correction in CMOS [14][37][40]. Unfortunately, current reliability mechanisms incur large area and power overheads, as much as $N$ times the area and power for $N$ redundant units [2].
Recently, novel devices have been explored to serve as a possible supplement or even a replacement for CMOS technology in the near future, mainly in the field of memory. New memory devices include phase change RAM (PRAM) [20], ferroelectric RAM (FeRAM) [3], magneto-resistive RAM (MRAM) [15] and resistive RAM (RRAM) [22]. While many of these new devices suffer from problems such as slow speed, large area, high power and lack of CMOS compatibility, few show promise as a replacement memory technology. RRAM is one such technology, as it offers high density, reconfigurability, non-volatility, and CMOS compatibility. RRAM has already shown that it can be applied to many different application domains such as temperature sensing [30] and specifically for this research, reconfigurable logic [42].

Hybrid architectures using novel RRAM devices have also been explored. RRAM architectures that have been fully realized are the hybrid CMOS/RRAM architectures [9]. These architectures are advantageous because of their reconfigurability, high density ($4F^2$, where $F$ is the feature size) and low idle energy. Researchers have been focusing on the design, implementation, and realization of such hybrid architectures [41] but not much attention is given to the adoption of these architectures for fault-tolerance or enhancing the reliability of these architectures.
This research explores hybrid CMOS/RRAM architectures that can utilize RRAM as both memory and logic. CMOS functional units can be implemented in RRAM if a hard error occurs, to enhance reliability by preventing system failure. Functional units are investigated to determine which functions can be implemented as RRAM LUT-based logic, and which ones cannot. Finally, hybrid CMOS/RRAM architectures are tested and analyzed to determine the efficacy of utilizing them for reliability enhancement.

The remainder of this document is organized as follows: This chapter presents a background on hard errors, CMOS reliability techniques, and RRAM including CMOS/RRAM hybrid architectures. Chapter 2 discusses existing work relevant to this thesis. Chapter 3 presents the architectures used, while Chapter 4 presents the method of implementing computational functions in RRAM. Chapter 5 explains the simulation platform and benchmarks used to gather results, while Chapter 6 discusses and analyzes the results. Chapter 7 summarizes the results and draws conclusions, and finally Chapter 8 presents possible future work on this topic.

1.2 Reliability

A reliable processor should incorporate mechanisms for the prevention of errors, detection of errors, the correction of those errors, or a combination
of all three. Preventing errors inherently reduces the risk of failures. Error detection mechanisms allow errors to be found and isolated when they occur. Correcting errors that have occurred prevents them from causing failures. This section discusses hard errors and reliability techniques used in current CMOS systems.

1.2.1 Hard Errors

Hard errors are caused by defects or faults in the silicon of CMOS systems. They tend to be permanent because the problem lies in the physical structure of the hardware. There are two types of hard errors, extrinsic and intrinsic. Extrinsic errors are known as “early infant mortality” errors because they occur soon after operation begins. The root cause of extrinsic errors are problems in the fabrication process such as misaligned lithography masks, manifesting themselves as short circuits or open circuits. [37]

Intrinsic errors tend to manifest themselves long into the lifetime of a system, and are caused by wear-out over time. Error rates caused by wear-out can be accelerated by subjecting a system to heavy stress often or over long periods of time. Preventing and correcting these hard errors are necessary for having a reliable system. CMOS reliability techniques have been
researched and implemented in CMOS, but many have significant draw-
backs that can be avoided through the use of CMOS/RRAM architectures
such as high area overhead.

\textbf{1.2.2 Reliability Theory}

Hard error correction improves the reliability of a system by detecting hard
errors and implementing solutions to prevent total system failure. One of the
universal reliability circuits proposed by J. VonNuemann as early as 1956 [40],
the majority organ or majority logic gate, can be used to synthesize
reliable circuits from unreliable systems. The three input signal probabilities
feeding in to the majority logic gate, \(\eta_1\), \(\eta_2\), and \(\eta_3\), represent the upper
bounds for these lines to be carrying the wrong data. The upper bound of
at least two lines carrying the wrong data (which are in the same state of
excitation) is set in Equation 1.1.

\[
e = \eta_1\eta_2 + \eta_1\eta_3 + \eta_2\eta_3 - 2\eta_1\eta_2\eta_3
\]  \hspace{1cm} (1.1)

In general, the system will be reliable if \(e\) is relatively small. Such a
system works well for mitigating hard errors, when the probability of failure
of the components is less than 0.5, which is more than sufficient for present
day systems. The majority logic gate can be used in reliable architectures to
pass through the most common output, generally being the correct output. This theory is the basis for majority voter circuits.

1.2.3 Majority Voter Circuits

A majority voter circuit uses redundant functional units to perform the same logic function, where the outputs are sent to a majority logic gate, and the most common result among them is used [40]. If \( \eta \) represents the upper bound for the probability of error in a functional unit \( O \), a triple redundant group of functional units \( O^* \) with majority logic gate has probability of error \( \eta^* \), where \( \epsilon \) is the definite probability of an error, shown in Equation 1.2. The majority logic gate is often called voter logic, hence the name majority voter circuit. If a hard error occurs in one of the functional units, the redundant units still have the majority, and the correct result is relayed to the system output.

\[
\eta^* = \epsilon + (1 - 2\epsilon)(3\eta^2 - 2\eta^3)
\]  

(1.2)

Figure 1.1 demonstrates the triple redundant group of functional units \( O^* \) with voter logic labeled as \( m \). A system without redundant logic and majority voter circuitry, or other mitigation techniques, can experience total failure if a necessary/executive functional unit fails. Unfortunately, the cost
of having hard error reliable systems using redundancy is over $N$ times the amount of area and power for $N$ repeated functional units and voting logic [2].

Figure 1.1: Triple Redundant Group of Functional Units with Voter Logic

Some fault-tolerant mechanisms used to improve hard error failures are triple modular redundancy (TMR) and N-tuple modular redundancy (NMR) [14]. TMR uses three redundant components that feed into voting logic, while NMR uses $N$ redundant components, usually an odd number, that feed into voting logic. These techniques are the de-facto standard in several mission-critical applications.

1.2.4 Standby Redundancy

Another technique that is similar to majority voting is standby redundancy, as seen in Figure 1.2. Additional redundant components are implemented
and are all connected to a switching mechanism [11]. At any given moment, only one component is operational. When the currently operational component fails, the switching mechanism switches to a redundant component, which takes over the functional load. Although this technique incurs a large area overhead, it has low power overhead because redundant logic is either turned off or placed in a low power state when not in use.

1.2.5 Dynamic Reliability Management (DRM)

Hard error prevention improves the reliability of a system by extending the lifetime of hardware through stress reduction. Dynamic reliability management (DRM) is a technique that has been proven to improve reliability in under-designed systems [37]. Most processors are over-designed. That is, they are designed to operate at the worst case and can increase their performance as long as they stay within reliability margins. Under-designed
systems are built to operate at expected, or normal utilization and temperature, and have reduced overall system cost. Under-designed systems have difficulty operating at worst case, but it is assumed that the DRM mechanism has knowledge of the run-time behavior of every task, can determine when the worst case will happen, and act accordingly.

DRM leverages the knowledge of application run-time behavior to degrade performance using techniques such as voltage and frequency scaling and decreased system utilization in order to reduce system stress and prevent prolonged stress from increasing hard error rates and damaging the system. To predict the run-time system behavior accurately is a non-trivial task. Since several of the prediction mechanisms are based on statistical analysis, DRM might not be able to react in time for certain cases. Unexpected system behavior could still raise the stress on components to dangerous levels because DRM does not act. Another problem with DRM is that it does not perform any error correction. Even though DRM takes preventative measures, once a hard error occurs, it cannot fix the issue.

Redundancy-based reliability enhancement mechanisms incur large area overhead. With the new class of emerging 3D hybrid architectures, redundancy can be applied hierarchically, and only when needed, to mitigate the faults occurring from increasingly unreliable nanoscale devices. In particular, CMOS/RRAM hybrid architectures utilize the reconfigurability of
RRAM to perform both memory and logic, which leads to new opportunities in the fields of fault-tolerance and reliability.

1.3 Resistive RAM

In 1971, Leon Chua proposed a fourth basic circuit element, the memristor [10]. Memristors were realized as nanoscale thin film devices by HP Labs in 2008 [39] and have great potential to be used as non-volatile memory [22]. There are several materials with which memristive devices can be fabricated such as titanium dioxide (TiO$_2$), hafnium dioxide (HfO$_2$) and copper oxide (CuO$_x$) [17]. Memristors are one of the primary devices currently used in RRAM fabrication because of their high density ($4F^2$, where $F$ is the feature size), low read energy, low leakage power, non-volatility, high resistance ratio, and CMOS compatibility.

The memristor is a two-terminal memory device located between a top and bottom electrode, where logic storage is resistance-based. The memory device changes its resistance based on the electrical signal through it, and retains its value while no electrical signal is applied. RRAM uses memristors as the underlying device in a crossbar architecture. The difference between RRAM and other emerging memory technologies is that ferroelectricity, magnetization, and phase change of material state are not involved.
Resistance switching in RRAM devices can be either unipolar or bipolar. Unipolar devices switch based on the magnitude of an applied voltage, and are controlled by the amount of current allowed to flow. Unipolar devices generally involve Joule heating, but are difficult to control due to current compliance [25].

Bipolar devices switch based on both the magnitude and polarity of the applied voltage. A large positive voltage changes a device to be in a low resistance state, while a large negative voltage changes a device to be in a high resistance state. Bipolar devices have voltage threshold characteristics. Significant resistance switching will only occur if the magnitude of the applied voltage exceeds the magnitude of the threshold value of identical polarity [35]. In Figure 1.3, the positive and negative voltage thresholds for a bipolar memristor can be seen at approximately $\pm 2$ V.

The International Technology Roadmap for Semiconductors (ITRS) states that RRAM is comparable to other memory devices such as DRAM, SRAM and NOR Flash [1]. Table 1.1 shows the non-volatility, scalability and small feature size of RRAM compared to other memory devices. The access speeds are improving in more recent realizations of these devices, using new materials.

Peripheral and interface logic required for reading and writing data to the
RRAM restricts, to a certain extent, the high density that can be achieved [43]. To address this issue, it is possible to use multilevel RRAM memory, where multiple bits can be represented by one memristor, which increases the memory density by a factor of the number of levels [31]. Storing two bits per memristor using four resistance levels doubles the total memory density. An issue with multilevel RRAM memory is its small noise margins, where it
becomes significantly more difficult to represent a multi-bit logic state with a resistance state correctly and confidently. The state is shifted up or down undesirably with increasing ease as more levels are utilized.

Memristors are passive devices, therefore RRAM requires CMOS peripheral circuitry in order to operate. The necessity of CMOS circuitry has led to hybrid CMOS/RRAM architectures becoming a focal point for RRAM related research.

1.4 Hybrid CMOS/RRAM Architectures

Few hybrid CMOS/RRAM architectures have been proposed that have been realized at the system level [8] [45] [24] [4]. Of these, the crossbar and the one transistor and one resistor (1T1R) architecture have been widely adopted due to their practicality.

1.4.1 Crossbar Architecture

In the crossbar architecture, RRAM devices are set between two parallel arrays of nano-wires, which are placed 90° offset from one another, allowing for maximum memory density. A problem with this architecture is the presence of sneak paths as shown in Figure 1.4, which create alternative current
paths through adjacent memory locations that should not be disturbed in the current operation.

Methods have been proposed to reduce the effects of sneak paths [45] [24][28]. One such technique is $V/2$ voltage biasing, where unselected rows and columns are biased to reduce current flow through adjacent devices, minimizing disturbances. If the selected row is biased to a voltage $V$ and the selected column is biased to $Gnd$, the unselected rows and columns are biased to a voltage $V/2$. The maximum voltage drop across any unselected memristor is therefore $V/2$, below the voltage thresholds for resistance state change.

Figure 1.4: Sneak Paths Inherent in the Crossbar Architecture
1.4.2 One Transistor and One Resistor (1T1R) Architecture

The 1T1R architecture, as shown in Figure 1.5, solves the sneak path problem by integrating a CMOS transistor in series with the memristor. Each transistor requires its own individual select wire, which allows current to flow only through a selected memory device [4][29]. The problem with this architecture is the reduction in memory density, negating one of the prominent benefits of RRAM. The additional transistors and wiring for selecting each individual element result in an increased area per bit, and overall memory density is limited by CMOS scaling.

This research uses the crossbar architecture because it provides the highest density and requires less CMOS circuitry compared to the 1T1R architecture. Area, timing, and power data has been published on this architecture, and the results can be used for simulating this research [43].

![Figure 1.5: 1T1R Architecture](image-url)
Chapter 2

Related Work

Some recent work has been focused on emerging ideas and devices such as memory-based computing (MBC) and spin-torque transfer magnetoresistive RAM (STT-MRAM) computing for reliability enhancement and involve techniques relevant to this research [34][21]. In this section, a detailed description of both existing efforts and reliability metrics will be discussed.

2.1 Memory-Based Computing

Memory-based computing (MBC) is a technique that uses on-chip CMOS cache to perform look-up table (LUT) based logic functions for reliability enhancement [34]. Logic functions can be implemented in L1 or L2 SRAM cache as needed if a computational unit becomes non-functional due to a
hard error or overheats due to heavy stress. Instead of throttling the performance of a system under thermal stress by temporarily disabling an overheating functional unit, activity migration to the on-chip cache occurs. Activity migration also improves reliability by implementing and performing the operation of failed CMOS functional units in on-chip cache as LUT-based logic, preventing system failure.

A problem with using MBC in L1 and L2 SRAM cache is that the reconfiguration subspace for logic is constrained by the small cache size, although in most cases, the subspace is sufficient for logic reconfiguration. Cache that is overwritten by a functional unit implementation must be re-fetched if needed again. Therefore, the cache hit rates will be inadvertently affected, degrading average access time, when a large subspace is reconfigured. The remaining available cache space that can be used as memory is now smaller, causing data to be overwritten more frequently and increasing the miss rate. Another consideration is that on-chip SRAM cache is not located in the same area as CMOS functional units on the chip, requiring long wire lengths for getting result data to the integer and floating-point registers.
2.2 Spin-Torque Transfer Magnetoresistive RAM Computing

A special case of MBC is spin-torque transfer magnetoresistive RAM (STT-MRAM) computing. STT-MRAM is CMOS-compatible, leakage-resistant and non-volatile magnetic memory technology. Authors in [21] have proposed the use of STT-MRAM for both memory and computing. They focus on low-power, low cost, energy efficient multi-core systems using STT-MRAM to scale systems smaller than the 45nm technology node. The idea is to replace SRAM cache completely with STT-MRAM cache and CMOS functional units with STT-MRAM LUT-based logic. The near-zero leakage STT-MRAM greatly reduces the power density compared to its CMOS counterpart. Figure 2.1 shows the architectural pipeline used when implementing STT-MRAM into a processor. All CMOS logic and functional units are replaced with STT-MRAM LUT-based logic, while the caches and register files are replaced with STT-MRAM memory arrays. ALUs remain implemented in CMOS because their circuit complexity does not benefit from using multiple small input width STT-MRAM LUTs.

The STT-MRAM architecture is similar to the 1T1R RRAM architecture in that a transistor is used in series with the memory element, a magnetic tunnel junction (MTJ). Unfortunately it allows maximum memory density
of only $10F^2$. The largest problem with STT-MRAM is its low resistance ratio of $6.25k\Omega/2.5k\Omega$ ($R_{\text{high}}/R_{\text{low}}$). A high resistance ratio is required to ensure the correct value is read. If $R_{\text{high}}$ and $R_{\text{low}}$ are very close, an incorrect read is more prone to occur, making the system unreliable.

2.3 Reliability Metrics

Reliability of a system can be defined by the functional longevity of a system without failing. The most common metrics to measure reliability are the mean time to failure (MTTF) and failures in time (FIT) rate.

2.3.1 Mean Time to Failure

The expected lifetime of a processor is also known as the mean time to failure (MTTF). If a hard error occurs in a component that is not easily replaced,
the processor becomes unusable. The MTTF model consists of four primary causes, or failure mechanisms, of CMOS failures. The first failure mechanism is electromigration, which is the accumulation or depletion of metal atoms in unwanted locations due to current flow. The second is stress migration, which is the accumulation or depletion of metal atoms in unwanted locations due to mechanical stress and thermal expansion. The third failure mechanism of CMOS failure is time-dependant dielectric breakdown, which is when the gate oxide of a transistor wears down over time. The final cause in the MTTF model is thermal cycling, which is a large fluctuation in temperature due to changes to and from low-power modes or power on / power off cycles. Thermal cycles impact a processor the most at solder joints, where connections could break or short circuit. The failure mechanisms of all components can be combined to form the MTTF of an entire processor as shown in Equation 2.1, where \( j \) is the number of components, \( k \) is the list of failure mechanisms, and \( \lambda_{il} \) is the failure rate of the \( i^{th} \) component due to the \( l^{th} \) failure mechanism.

\[
MTTF_p = \frac{1}{\sum_{i=1}^{j} \sum_{l=1}^{k} \lambda_{il}} 
\]  

(2.1)
2.3.2 Failures in Time

The failures in time (FIT) rate is more commonly used in industry because it is less complex and more easily calculated via circuit level testing. The low-level details of component material, temperature, and voltage are not required. The FIT rate is simply the number of failures in $10^9$ hours. This metric can be calculated for individual components, or entire processors. The FIT rate of a processor can be calculated from the FIT rates of individual components as shown in Equation 2.2, where $j$ is the number of components, and $FITrate_i$ is the FIT rate of the $i^{th}$ component.

$$FITrate_p = \sum_{i=0}^{j} FITrate_i$$ (2.2)

These reliability metrics are only useful when low-level implementation and fabrication details are known or when systems are realized and able to be run to gather reliability statistics. They are unable to be applied to architectural simulations, although it can be assumed that FIT rates will increase if hard errors are corrected and the system continues to operate until a future failure.

The fundamental idea behind MBC, performing logic in memory using LUT-based logic, is a key concept for this research. Using MBC in RRAM instead of on-chip SRAM cache has a benefit of spatial locality, where the
replaced logic in RRAM can be physically close to the CMOS functional units if used in a 3D integrated circuit (3D-IC). Additionally, RRAM’s high density allows more memory capacity to be implemented in the same sized area. The larger the cache, the less impact implementing a functional unit in cache will have on the hit/miss rates. Integrating emerging memory devices into CMOS systems to operate as cache and logic is a key concept for this research as well. Using RRAM instead of STT-MRAM gives greater area efficiency, and RRAM also has a higher resistance ratio of $500k\Omega/10k\Omega$ ($R_{high}/R_{low}$). This research also extends the STT-MRAM concept to utilize emerging memory devices for reliability enhancement.
Chapter 3

Proposed Architecture Design

An overview of the proposed three dimensional integrated circuit (3D-IC) architecture is shown in Figure 3.1. A CMOS computational layer is used as tier one, and the RRAM is overlaid as tier two of the 3D stack. The RRAM layer is implemented as a group of equally sized arrays, which will be analyzed later in this chapter. The RRAM arrays are arranged in an H-tree structure. This structure obtains equal wire lengths to each array, maintaining equal read and write times.

The RRAM is stacked on top of the CMOS layer to improve the spatial locality of replaced functional units. RRAM arrays directly above a failed CMOS unit can be reconfigured to implement the replacement unit using short wire lengths, keeping latency to a minimum.

Each RRAM array requires its own driver and selection circuitry implemented in CMOS. Another benefit of having the RRAM as the second tier of a 3D stack is for efficient implementation of the CMOS peripheral circuitry for each array. Again, this high spatial locality of CMOS circuitry to
RRAM arrays minimizes the effects of latency due to wire length.

The RRAM layer is used as the highest level of cache because it is slower than on-chip SRAM cache, but faster than main memory accesses. The cache level that RRAM is used for has no affect on the the way functional units are implemented because the execution datapath is re-routed directly to the RRAM layer. This chapter describes the details of the architectures that are simulated, then analyzes the implementation of the RRAM arrays and finally discusses requirements for the RRAM controller.

Figure 3.1: Representation of the Proposed Hybrid CMOS/RRAM Architecture for Reliability Enhancement
3.1 Test-Case Architecture Implementation

The CMOS core used in this work is simulated as an Alpha-like processor. The Alpha 21264 microarchitecture can be seen in Figure 3.2 [12]. The pipeline provides out of order (O3) computing and can issue up to four instructions per cycle at a clock speed of 1.2 GHz. The functional units available are four integer arithmetic logic units (ALUs), one integer multiplier associated with one of the integer ALUs, one floating-point ALU unit and one floating-point multiplier. The integer and floating-point pipelines are split, each with their own issue queues and result registers. L1 cache is split into an instruction cache and a data cache, both of size 64 KB with 2-way set associativity.

The Alpha 21364 processor was designed and fabricated for 180-nm technology in 2003 and has a processing core identical to the 21264 version [32]. Additional hardware added to the 21264 core to create the new 21364 model includes a 1.75-MB 8-way set associative on-chip L2 cache with cache coherence hardware, two memory controllers, and a router to create multi-core architectures easily using this processor. The router is removed from single-core simulations because it is not utilized.

The Alpha architecture is scaled to 45 nm using International Technology
Roadmap for Semiconductors (ITRS) scaling parameters [1]. Common implementations of an integer ALU, integer multiplier and floating-point adder and their speedups were also obtained and embedded in the Alpha architecture implementation [33]. Table 3.1 shows the scaled Alpha processor.

Two architectures were used in single-core implementations. The first utilizes the 8-MB SRAM L2 cache of the scaled Alpha 21364 architecture with additional 32-MB RRAM L3 cache, while the other implements the Alpha 21264 architecture with 32-MB RRAM L2 cache. For multi-core
Table 3.1: Alpha Processor Conversions

<table>
<thead>
<tr>
<th>Item</th>
<th>Alpha 21364</th>
<th>Converted Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>180 nm</td>
<td>45 nm</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>1.2 GHz</td>
<td>4 GHz</td>
</tr>
<tr>
<td>Supply Voltage</td>
<td>1.65 V</td>
<td>1.1 V</td>
</tr>
<tr>
<td>Integer ALU</td>
<td>1 cycle</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Integer Multiplier</td>
<td>7 cycles</td>
<td>4 cycles</td>
</tr>
<tr>
<td>Floating-Point Adder</td>
<td>4 cycles</td>
<td>4 cycles</td>
</tr>
<tr>
<td>Floating-Point Divider</td>
<td>15 cycles</td>
<td>20 cycles</td>
</tr>
<tr>
<td>Floating-Point Sqrt.</td>
<td>30 cycles</td>
<td>20 cycles</td>
</tr>
<tr>
<td>Floating-Point Multiplier</td>
<td>4 cycles</td>
<td>4 cycles</td>
</tr>
<tr>
<td>L1 Cache Speed</td>
<td>4 cycles</td>
<td>4 cycles</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>64 kB</td>
<td>256 kB</td>
</tr>
<tr>
<td>L2 Cache Speed</td>
<td>15 cycles</td>
<td>10 cycles</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>1.75 MB</td>
<td>8 MB</td>
</tr>
</tbody>
</table>

simulations, both 8-core and 16-core systems are tested. The 8-core architecture has a shared 256-MB (32 MB × 8 cores) RRAM L2 cache, while the 16-core architecture has a shared 512-MB (32 MB × 16 cores) RRAM L2 cache. The differences in the simulated architectures is shown in Table 3.2. Main memory capacity for each architecture is 2 GB, and the block size is 64 KB.

Table 3.2: Test-Case Architecture Implementations

<table>
<thead>
<tr>
<th></th>
<th>RRAM L3 Cache</th>
<th>RRAM L2 Cache</th>
<th>8 Core</th>
<th>16 Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 Cache</td>
<td>8 MB SRAM</td>
<td>32 MB RRAM</td>
<td>256 MB RRAM</td>
<td>512 MB RRAM</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>32 MB RRAM</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

Portions of RRAM are reconfigured to implement LUT-based logic when hard errors occur in CMOS components. The reconfiguration is handled by the RRAM controller, explained in section 4.3. Functional units that are
practical to implement as LUT-based logic are used in this research, such as the integer ALU, the integer multiplier, and the floating-point adder. Functional units that do not map into LUT-based logic, and are not attempted in this experiment, involve operations such as shifting, division and the square root.

All of the RRAM is used as cache in normal operation, when no hard errors have occurred, to mask the area overhead of LUT-based logic. This mitigates the high area overhead incurred in traditional redundancy-based majority voting circuits and standby redundant circuits. Additionally, RRAM has low leakage power and therefore reduced power overhead compared to others. The size of RRAM cache is significantly larger than traditional SRAM caches of the same area due to its high density. The reconfigured portions of RRAM consume a small percentage of the cache, having little impact on the miss ratio compared to MBC architectures using on-chip SRAM cache.

3.2 RRAM Arrays

The maximum array size for an RRAM array is determined by the write driver current, and the voltage biasing scheme used to prevent sneak path current [43]. The $V/2$ biasing scheme is used to prevent significant sneak
path current; therefore the maximum driver current is represented in Equation 3.1, where $I_{\text{reset}}$ is the reset current, $N_r$ is the number of rows, and $I_{LRS}(V_{\text{reset}}/2)$ is the current through a device in a low-resistance state (LRS) with a $V_{\text{reset}}/2$ bias.

$$I_{\text{driver}} = I_{\text{reset}} + (N_r - 1) \times I_{LRS}(V_{\text{reset}}/2)$$  \hspace{1cm} (3.1)

Memristors are non-linear; that is, current through a memristor is not directly proportional to the voltage applied to it, because memristance (and therefore resistance) is not a constant. A coefficient of non-linearity is used for a $V/p$ biasing scheme as shown in Equation 3.2, where $K_r$ is the coefficient of non-linearity and $R_{V/p}$ and $R_V$ are static resistances of a memristor biased at $V/p$ and $V$ respectively. In the case of this research, a $p$ value of 2 is used for the $V/2$ biasing scheme.

$$K_r(p, V) = p \times \frac{R_{V/p}}{R_V}$$  \hspace{1cm} (3.2)

Using equations 3.1 and 3.2, the maximum number of rows and columns can be calculated based on the driver current when all selected rows and columns intersect at devices in the LRS. Equations 3.3 and 3.4 show this relationship, where $N_{sc}$ is the number of selected columns per row.
\[ N_r = \left( \frac{I_{\text{driver}}}{I_{\text{reset}}} - 1 \right) \times K_r(2, V_{\text{reset}}) + 1 \]  

\[ N_c = \left( \frac{I_{\text{driver}}}{I_{\text{reset}}} - N_{\text{sc}} \right) \times K_r(2, V_{\text{reset}}) + N_{\text{sc}} \]  

Using the previous equations, the maximum array size was found to be 512 bits × 512 bits (32 KB). A 32-MB RRAM memory is designed using the maximum sized arrays connected in an H-tree structure. Sensing and read/write circuitry are interfaced from the CMOS layer. The read latency for this architecture is 1.8 ns and the write latency is 200 ns as shown in Table 3.3 [43].

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read Latency (ns)</strong></td>
<td>1.773</td>
</tr>
<tr>
<td><strong>Write Latency (ns)</strong></td>
<td>200.7</td>
</tr>
<tr>
<td><strong>Read Energy (nJ)</strong></td>
<td>0.195</td>
</tr>
<tr>
<td><strong>Write Energy (nJ)</strong></td>
<td>25.81</td>
</tr>
</tbody>
</table>

The high write latency can be mitigated by using the cache inclusion protocol, where every level of cache is a subset of each subsequent level [18]. This means that a miss in the highest cache level gets written to all levels of cache after being fetched from main memory. It can be assumed that a request for the same data soon after the write would hit in a lower cache level, while the data is still being written to the higher cache level by the controller.
3.3 RRAM Controller

The controller, implemented in CMOS, handles the reading and writing to the RRAM arrays. It is assumed that data can be fetched from main memory in 64-KB blocks, common for all levels of cache in the architecture, and that reads and writes can be performed in the form of a byte (8 bits), word (16 bits), long word (32 bits) or long long word (64 bits). In addition, because the write time for RRAM is long, it is assumed that additional reads can occur in arrays that are not currently being written to.

The controller also handles hard error detection in the CMOS functional units. Error detection circuitry, such as “lazy error detection,” can detect errors one cycle after the output arrives and has an area overhead as low as 33\% of the functional unit with which it is associated [44]. Specific error detection circuitry is not within the scope of this thesis, and the assumption that error detection successfully occurs after an error manifests itself is sufficient.

After an error is detected, the CPU stalls while the error correction process occurs. First, the program counter must be rolled back to the instruction prior to where the error occurred. The controller then reconfigures the portion of the RRAM arrays to implement a failed functional unit. This reconfiguration process includes writing LUT-based logic from memory to the
arrays and setting the control logic for additional CMOS circuitry, known as “glue logic”, required to implement the failed unit. The execution datapath is rerouted by the controller to use the LUT-based logic that has been implemented in RRAM. Any data that was stored in the arrays, which are now used for LUT-based logic, is marked invalid and the controller no longer recognizes the reconfigured arrays for cache use. The program continues after the correction has been completed, and the stall that occurs during the reconfiguration process can be justified by the prevention of total system failure.

The overall controller block diagram, including read / write circuitry, glue logic, and error detection circuitry, is shown in Figure 3.3. The failed CMOS functional units are implemented in RRAM arrays as LUT-based logic functions.
Figure 3.3: RRAM Controller Block Diagram
Chapter 4

LUT Logic Mapping

Logic is mapped and implemented as LUTs using techniques such as bit-slicing to improve speed and reduce the amount of RRAM blocks required [34]. Uniformly sized RRAM blocks allow for functional units to be implemented as sequential logic by both cascading blocks and having blocks in parallel if there are no cross-block dependencies. The granularity of the RRAM blocks determines how the functional units can be implemented. Since the RRAM layer is divided into 512-bit x 512-bit arrays as determined previously, the maximum number of input bits for a LUT is determined by the output bit width. Equation 4.1 shows the calculation of maximum input bits $I_{b, Max}$, where $N_r$ is the number of rows, $N_c$ is the number of columns, and $O_b$ is the number of output bits.

$$I_{b, Max} = \lfloor (\log_2 N_r) \rfloor + \lfloor (\log_2 \frac{N_c}{O_b}) \rfloor$$  \hspace{1cm} (4.1)

Table 4.1 shows how many arrays are required to implement each functional unit when decomposed into LUT-based logic, their memory usage
in RRAM, and the number of arrays in their critical paths. The next subsections analyze the implementation details of each functional unit. The functional unit implementations were designed by including additional contributions to previous work [34].

<table>
<thead>
<tr>
<th>Arrays Used</th>
<th>Memory Used</th>
<th>Critical Path Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ALU</td>
<td>8</td>
<td>256 kB</td>
</tr>
<tr>
<td>Integer Multiplier</td>
<td>16</td>
<td>512 kB</td>
</tr>
<tr>
<td>Floating-Point Adder</td>
<td>16</td>
<td>512 kB</td>
</tr>
</tbody>
</table>

### 4.1 Integer ALU

A 32-bit ALU is implemented in the RRAM using eight arrays, where each array implements 4-bit bitwise logic functions and a 4-bit carry-select adder. A 4-bit carry-select adder requires ten output bits: four sum bits and one carry out bit, for each carry in value. Other bitwise logic requires only four output bits. Using Equation 4.1, where the maximum output width is ten bits for 4-bit carry-select addition, the maximum number of input bits possible is 14. Three bits for operation select and eight bits for operands, four from each input, are required totaling 11 bits. Therefore, a single array is sufficient for 4-bit ALU implementation.

Figure 4.1 shows how a single RRAM array can be implemented. There
are a finite number of possibilities depending on how the inputs are arranged. For viewing purposes, rows correspond to possible input vectors, while columns correspond to output selection. Ten columns of an array are used for the sum bits and carry out bits. Other columns are used for bitwise logic functions such as AND, OR, and XOR. Not all rows and columns are guaranteed to be filled in an LUT-based RRAM array, so some memory is potentially unused depending on the implemented logic function.

<table>
<thead>
<tr>
<th>B, B, B, A, A, A, A, A, A, A</th>
<th>$\bar{C}_b = 0$</th>
<th>$\bar{C}_b = 1$</th>
<th>$S_{out}$</th>
<th>$\bar{S}_{out}$</th>
<th>$A_{out}$</th>
<th>$O_{out}$</th>
<th>$X_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 0 0</td>
<td>0...0</td>
<td>1...0</td>
<td>0...0</td>
<td>0...0</td>
<td>0...0</td>
<td>0...0</td>
<td>0...0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 1 0</td>
<td>1...0</td>
<td>0...0</td>
<td>0...0</td>
<td>1...1</td>
<td>1...1</td>
<td>0...0</td>
<td>0...0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 1 0 0</td>
<td>0...0</td>
<td>1...0</td>
<td>0...0</td>
<td>0...0</td>
<td>0...0</td>
<td>0...0</td>
<td>0...0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 1 1 1</td>
<td>1...0</td>
<td>0...0</td>
<td>0...0</td>
<td>1...1</td>
<td>1...1</td>
<td>0...0</td>
<td>0...0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1 0 1 0 0 1 0 1 1 1</td>
<td>1...0</td>
<td>0...0</td>
<td>1...1</td>
<td>1...0</td>
<td>1...0</td>
<td>0...0</td>
<td>0...0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 1 1 1</td>
<td>0...0</td>
<td>1...0</td>
<td>1...1</td>
<td>1...0</td>
<td>1...0</td>
<td>0...0</td>
<td>0...0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1 1</td>
<td>0...1</td>
<td>1...1</td>
<td>1...1</td>
<td>1...0</td>
<td>1...0</td>
<td>0...0</td>
<td>0...0</td>
</tr>
</tbody>
</table>

Figure 4.1: RRAM Array Representation for 4-Bit ALU

The outputs are gathered in CMOS, particularly the RRAM controller glue logic. Specifically for integer addition, the carry selection occurs using multiplexors in CMOS. The carry bits ripple through the high-speed CMOS multiplexors, rather than in RRAM arrays. This prevents the computation from being eight sequential 4-bit ALU operations. Therefore, the RRAM
critical path traverses though a single array, when all eight arrays are performed in parallel. This process is shown in Figure 4.2. The critical path time is one RRAM array read delay and eight CMOS multiplexor delays. A 45-nm CMOS multiplexor was found to take approximately 5 ps [36]; therefore the total delay for an integer ALU implemented in RRAM is approximately 1.8 ns.

![Figure 4.2: Block-Level Schematic of LUT-Based 32-Bit ALU Design](image)

### 4.2 Integer Multiplier

A 32-bit integer multiplier is implemented in the RRAM using 16 arrays as an array multiplier. Eight arrays are implemented as 4-bit multipliers, and the other eight are implemented as 4-bit carry-select adders. A 4-bit
multiplier requires five outputs: four multiplication bits and a carry out bit. Using Equation 4.1, where the maximum output width is five bits for 4-bit multiplication, the maximum number of input bits possible is 15. Four bits for multiplication input, a carry in bit, and eight bits for operands, four from each input, are required totaling 13 bits. Therefore, a single array is sufficient for 4-bit multiplication implementation.

Figure 4.3 shows the structure of the interactions between arrays. The eight 4-bit multipliers are performed in parallel eight times, and are shifted in glue logic after each iteration. After, the eight 4-bit carry select adders are used in parallel to calculate the finalized output. Therefore, the RRAM critical path traverses through nine arrays. The critical path time is nine RRAM array read delays and eight CMOS multiplexor delays; therefore the total delay for an integer multiplication implemented in RRAM is approximately 15.3 ns.

Figure 4.3: 32-Bit LUT-Based Multiplier Design
4.3 Floating-Point Adder

Floating-point numbers in binary are represented using the IEEE 754 standard [5]. In this standard, a 32-bit signed floating-point number is divided into three parts: the 1-bit sign, the 8-bit biased exponent, and the 23-bit fraction which is used to attain the significand. Equation 4.2 demonstrates how the floating-point value is calculated using this standard.

\[ \text{Value} = (-1)^S \times 2^{e-127} \times 1.f_{22}f_{21}f_{20}...f_0 \]  

A 32-bit floating-point adder is implemented in the RRAM using 16 arrays and is performed using exponent alignment [27]. The complete floating-point addition process is shown in 4.4. Addition of two numbers first requires the difference of the exponents. This is performed using two RRAM arrays implemented as an 8-bit subtractor, two 4-bit subtractors performed in parallel. As with addition, the carry selection occurs in CMOS glue logic. Using the difference, the significands are aligned by performing shifts using multiplexors in glue logic. Selected bits from intermediate shift stages are tapped and sent to the RRAM for “sticky bit” calculation. A total of 33 bit taps are used as inputs of a combinational logic circuit, requiring one bit of output. Using Equation 4.1, where the maximum output width is one bit, the maximum number of input bits possible is 18. Therefore, the combinational
logic circuit requires two RRAM arrays, with the output of both being ORed in glue logic.

![LUT-Based RRAM Floating-Point Addition Diagram]

Figure 4.4: 32-Bit Floating-Point Addition Process

After the “sticky bit” is calculated and significands are aligned, they can be added in the RRAM layer using normal 32-bit integer addition. Like the 32-bit ALU, carry select addition allows the eight RRAM arrays to be performed in parallel, with carry selection occurring in glue logic. Next, the leading zero count is performed. First, a zero-detector is performed on the lower 15 bits of the significand in CMOS glue logic. If the result is zero, the remaining upper 13 bits are sent to RRAM for leading zero counting modulated by 15. If the result is not zero, the lower 15 bits of the significand are sent to RRAM for unmodified leading zero counting. The maximum output width for either case is six bits, representing a maximum value of 28. Using Equation 4.1, where the maximum output bit width is six bits, the maximum number of input bits possible is 15. 15 bits is sufficient for either case, therefore both cases can be implemented by one RRAM array each.
As stated previously, because a shift operation does not translate well to LUT-based logic, the leading zero shift operation is performed in glue logic using multiplexors. Finally, the result exponent normalization is performed in the RRAM layer. The aligned exponent, along with the leading zero detector output is combined to get the exponent of the result. This is done using an adder / subtractor in RRAM similar to the exponent alignment performed in the initial step using two RRAM arrays operating in parallel.

Table 4.2 shows the stages of the adder and describes how many arrays each stage requires along with the critical array path for each stage. The operation requires 16 RRAM arrays in total. The critical path traverses through five RRAM arrays and up to five shifts in CMOS glue logic for both the significand alignment and leading zero shift. Therefore, the total delay for a floating-point adder implemented in RRAM is approximately 9 ns.

Table 4.2: LUT-Based Floating-Point Adder Decomposition

<table>
<thead>
<tr>
<th>Stage</th>
<th>Arrays Used</th>
<th>Critical Path Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent Difference</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Significand Alignment</td>
<td>0*</td>
<td>0</td>
</tr>
<tr>
<td>Sticky Bit Computation</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Significand Addition</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Leading Zero Counter</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Leading Zero Shift</td>
<td>0*</td>
<td>0</td>
</tr>
<tr>
<td>Exponent Normalization</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>16</td>
<td>5</td>
</tr>
</tbody>
</table>

*Zero arrays used because operation is performed in CMOS glue logic.
Chapter 5

Simulation Platform

The proposed architecture designs were tested using the SPEC2006 benchmarks [13] for single core systems and PARSEC benchmarks for multi-core systems [6]. The Gem5 simulator [7] was used to simulate the performance of these benchmarks under various conditions, while HP’s Multi-core Power, Area, and Timing (McPAT) simulator [26] was used to generate the energy and power profiles. Various Python scripts were used to customize the simulation flow, as seen in Figure 5.1. Push-button design flow ensures that the architecture configuration and specific benchmark inputs generate the required performance/energy statistics.

Figure 5.1: Representation of the Custom Tuned Simulation Framework
5.1 Gem5 Simulator

The Gem5 simulator provides a framework for computer architecture simulation [7]. The simulator is a complex collection of predefined component models, such as CPUs (AtomicSimple and Out-of-order), caches (IL1, DL1, L2 and L3), and functional units (IntALU, IntMult and FloatAdd), written in object oriented C++. Each component has its own set of parameters and keeps statistics about its use throughout a simulation. The system uses an embedded Python interpreter to allow for easy control over the configuration of an architecture. Gem5 outputs two files. The first is the config.ini file, which contains all the architectural parameters set in the run script to ensure the simulator was setup the way it was intended. The second is the stats.txt file, which contains all the runtime and architectural activity data gathered during simulation.

Gem5 can perform simulations in syscall emulation mode or full system mode. In syscall emulation mode, the instruction set architecture is modeled in software. In full system mode, simulations are performed on a booted operating system image, which can be checkpointed. Checkpoints allow for components to be switched out mid-simulation to speed up simulation time. For multi-core systems, this is especially useful. Unimportant sequential portions of benchmarks can be performed using a simple CPU model, which
simulates very fast, but is not accurate. Once the important parallelizable portion of a benchmark begins, the out-of-order (O3CPU) model can be used, which simulates much slower, but is very accurate.

5.1.1 Gem5 Modifications

The developers of Gem5 created the default out-of-order CPU (O3CPU) model to be Alpha-like. The O3CPU model parameters were modified to represent the scaled 45-nm Alpha processor. A variety of functional unit pools were created to simulate functional units implemented as LUT-based RRAM functional units. Table 5.1 shows a list of functional unit pool variations used in simulations, which can be selected at simulation time. The number of cycles were calculated using a 4-GHz clock and the critical path delay timing determined in the previous chapter. The simulations were run completely for the case of all working CMOS functional units and the case of one broken functional unit implemented in RRAM.

<table>
<thead>
<tr>
<th></th>
<th>IntALU Cycles</th>
<th>IntMult Cycles</th>
<th>FloatAdd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default (No failed units)</td>
<td>1,1,1,1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>A Single RRAM ALU</td>
<td>8,1,1,1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>RRAM Integer Multiplier</td>
<td>1,1,1,1</td>
<td>62</td>
<td>4</td>
</tr>
<tr>
<td>RRAM Floating-Point Adder</td>
<td>1,1,1,1</td>
<td>4</td>
<td>36</td>
</tr>
</tbody>
</table>
Multiple cache configurations were also created for the different simulated architectures that were tested. Table 5.2 shows a list of the cache configuration variations used in simulations, which can also be selected at simulation time. The delay parameter implements the memory hit time (in ns) for that particular cache level.

<table>
<thead>
<tr>
<th></th>
<th>L2 Size</th>
<th>L2 Delay</th>
<th>L3 Size</th>
<th>L3 Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Single Core (No RRAM)</td>
<td>8 MB SRAM</td>
<td>2.5 ns</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Single Core with L2 RRAM</td>
<td>32 MB RRAM</td>
<td>3 ns</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Single Core with L3 RRAM</td>
<td>8 MB SRAM</td>
<td>2.5 ns</td>
<td>32 MB RRAM</td>
<td>3 ns</td>
</tr>
<tr>
<td>8 Cores with L2 RRAM</td>
<td>256 MB RRAM</td>
<td>3 ns</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16 Cores with L2 RRAM</td>
<td>512 MB RRAM</td>
<td>3 ns</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### 5.2 McPAT Simulator

HP Labs developed a multi-core power, area, and timing simulator (McPAT) [26]. It can simulate models for all components of a complete processor, including out-of-order CPUs and shared multi-level cache at many different technology nodes, from 90 nm down to 22 nm. Although McPAT can model timing, area, and energy and power, this research will use it only for energy simulation. The source code is written in object oriented C++, so the simulator is easily modified to include additional devices such as RRAM cache and LUT-based RRAM functional units.
The input file for McPAT is written as an XML text file. This interface contains both the static architectural configuration data and dynamic activity statistics, such as the config.ini file and stats.txt file output from Gem5 respectively. A Python script was developed to extract data from these two Gem5 files, create an associated XML input file, and then run that file on McPAT to get the energy and power data.

5.2.1 McPAT Modifications

RRAM cache energy calculation was implemented in McPAT as shown in Equation 5.1, where $E_{hit}$ represents the RRAM cache read energy and $E_{miss}$ represents the RRAM cache write energy taken from Table 3.3.

$$E_{cache} = (E_{hit} \times \text{Hits}) + (E_{miss} \times \text{Misses})$$  \hspace{1cm} (5.1)

The LUT-based RRAM functional unit energy calculation formula was also implemented in McPAT as shown in Equation 5.2, where $N_{arrays}$ is the number of arrays used, depending on the functional unit implemented.

$$E_{logic} = E_{hit} \times N_{arrays} \times \text{Operations}$$  \hspace{1cm} (5.2)
5.3 SPEC2006 Benchmark Suite

In 2006, the Standard Performance Evaluation Corporation released single core benchmarks (SPEC2006) to replace the SPEC2000 benchmarks [13][19]. The benchmarks represent a variety of programs using various programming languages which include C, C++, and Fortran. The benchmarks draw their tasks from real life applications, rather than artificial loops and other synthetic programs, making them ideal for obtaining authentic architectural performance data. The benchmarks were cross-compiled to operate in the Gem5 syscall emulation mode. The benchmarks cover a wide variety of sizes and applications, stressing both the memory system and the functional units. Table 5.3 shows the list of benchmarks and their attributes.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Data Representation</th>
<th>Category</th>
<th>Language</th>
<th>Memory Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>401.bzip</td>
<td>Integer</td>
<td>Compression</td>
<td>C</td>
<td>856 MB</td>
</tr>
<tr>
<td>416.gamess</td>
<td>Floating-Point</td>
<td>Chemical Computation</td>
<td>Fortran 90</td>
<td>39 MB</td>
</tr>
<tr>
<td>429.mcf</td>
<td>Integer</td>
<td>Optimization</td>
<td>C</td>
<td>844 MB</td>
</tr>
<tr>
<td>433.milc</td>
<td>Floating-Point</td>
<td>Chromodynamics</td>
<td>C</td>
<td>676 MB</td>
</tr>
<tr>
<td>435.gromacs</td>
<td>Floating-Point</td>
<td>Molecular Dynamics</td>
<td>Fortran 90 and C</td>
<td>25 MB</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>Floating-Point</td>
<td>Fluid Dynamics</td>
<td>Fortran 90</td>
<td>129 MB</td>
</tr>
<tr>
<td>444.namd</td>
<td>Floating-Point</td>
<td>Molecular Dynamics</td>
<td>C++</td>
<td>53 MB</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>Integer</td>
<td>Artificial Intelligence</td>
<td>C</td>
<td>28 MB</td>
</tr>
<tr>
<td>450.soplex</td>
<td>Floating-Point</td>
<td>Simplex Solver</td>
<td>C++</td>
<td>457 MB</td>
</tr>
<tr>
<td>453.povray</td>
<td>Floating-Point</td>
<td>Computer Vision</td>
<td>C++</td>
<td>9 MB</td>
</tr>
<tr>
<td>454.calculix</td>
<td>Floating-Point</td>
<td>Mechanics</td>
<td>Fortran 90 and C</td>
<td>216 MB</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>Integer</td>
<td>Artificial Intelligence</td>
<td>C</td>
<td>180 MB</td>
</tr>
<tr>
<td>459.GemsFDTD</td>
<td>Floating-Point</td>
<td>Electromagnetics</td>
<td>Fortran 90</td>
<td>838 MB</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>Integer</td>
<td>Physics</td>
<td>C99</td>
<td>104 MB</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>Integer</td>
<td>Video Compression</td>
<td>C</td>
<td>68 MB</td>
</tr>
<tr>
<td>470.lbm</td>
<td>Floating-Point</td>
<td>Fluid Dynamics</td>
<td>C</td>
<td>416 MB</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>Integer</td>
<td>Event Simulation</td>
<td>C++</td>
<td>121 MB</td>
</tr>
<tr>
<td>998.specrand_i</td>
<td>Integer</td>
<td>Mine Canary</td>
<td>C</td>
<td>-</td>
</tr>
<tr>
<td>999.specrand_f</td>
<td>Floating-Point</td>
<td>Mine Canary</td>
<td>C</td>
<td>-</td>
</tr>
</tbody>
</table>
5.4 PARSEC Benchmark Suite

The Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark suite was developed by a joint project between Intel and Princeton University [6]. The benchmarks are all geared towards multi-core systems, and therefore have been parallelized. The benchmarks are diverse applications from many different research areas such as computer vision, animation physics, computational finance, and data mining. The benchmarks were cross-compiled for use with an Alpha architecture on a fully booted Linux system, making them available in the Gem5 full system mode [16]. There are many different input sets that can be used including testing, small, medium, large and native sized sets. The size used in this research is the small input set, because it suitable for microarchitectural studies, but does not take an extended period of time to simulate completely in the Gem5 simulator. Table 5.4 shows the list of benchmarks and their attributes.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Application Domain</th>
<th>Parallelization Model</th>
<th>Working Set Size</th>
<th>Data Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>Financial Analysis</td>
<td>data-parallel</td>
<td>small</td>
<td>low</td>
</tr>
<tr>
<td>bodytrack</td>
<td>Computer Vision</td>
<td>data-parallel</td>
<td>medium</td>
<td>high</td>
</tr>
<tr>
<td>canneal</td>
<td>Engineering</td>
<td>unstructured</td>
<td>unbounded</td>
<td>high</td>
</tr>
<tr>
<td>facesim</td>
<td>Animation</td>
<td>data-parallel</td>
<td>large</td>
<td>low</td>
</tr>
<tr>
<td>facesim</td>
<td>Animation</td>
<td>data-parallel</td>
<td>large</td>
<td>high</td>
</tr>
<tr>
<td>ferret</td>
<td>Similarity Search</td>
<td>pipeline</td>
<td>unbounded</td>
<td>high</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>Animation</td>
<td>data-parallel</td>
<td>large</td>
<td>low</td>
</tr>
<tr>
<td>vips</td>
<td>Media Processing</td>
<td>data-parallel</td>
<td>medium</td>
<td>low</td>
</tr>
</tbody>
</table>
Chapter 6

Simulation Results and Analysis

6.1 Single Core Simulations

The SPEC2006 benchmarks were profiled in Gem5 to compare the amount of functional unit and cache operations. Figure 6.1 demonstrates that out of the four observed operations, almost all benchmarks are over 80% integer ALU or cache operations. Approximately half of the benchmarks have large amounts of floating-point addition operations while only 437.leslie3d has significant usage of the integer multiplier. The benchmarks were ran to completion for various configurations of RRAM cache, CMOS functional units, and RRAM functional units as seen in Table 6.1.

Figure 6.2 shows that the shared L2 cache miss rates for both the 32 MB RRAM and the 8 MB SRAM L2 caches are approximately the same for almost all benchmarks. The exceptions are 435.gromacs, 437.leslie3d, 444.namd, and 470.lbm, where the RRAM L2 cache has a lower miss rate due to larger size. This shows that most current applications do not benefit very much from having an L2 cache of size 32 MB as compared to 8 MB.
Even though the L2 cache capacity is four times larger for the RRAM L2 cache architecture, there are still approximately the same amount of miss induced writes for most applications. The higher write energy and also larger capacity of RRAM cache causes twice the amount of energy to be used for 32 MB of RRAM cache than a standard 8 MB SRAM cache.

The architecture having a shared 32 MB RRAM L3 cache with a shared 8 MB SRAM L2 cache uses energy in direct relation to the SRAM L2 cache miss rate. Benchmarks that have SRAM L2 cache miss rates approaching a
Table 6.1: Single Core Simulation Configurations

<table>
<thead>
<tr>
<th>Sim Name</th>
<th>L2 Cache</th>
<th>L3 Cache</th>
<th>ALUs</th>
<th>Mult</th>
<th>FPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>8 MB SRAM</td>
<td>-</td>
<td>4 CMOS</td>
<td>1 CMOS</td>
<td>1 CMOS</td>
</tr>
<tr>
<td>L2 - Cache</td>
<td>32 MB RRAM</td>
<td>-</td>
<td>4 CMOS</td>
<td>1 CMOS</td>
<td>1 CMOS</td>
</tr>
<tr>
<td>L2 - ALU</td>
<td>32 MB RRAM</td>
<td>-</td>
<td>3 CMOS/1 RRAM</td>
<td>1 CMOS</td>
<td>1 CMOS</td>
</tr>
<tr>
<td>L2 - Mult</td>
<td>32 MB RRAM</td>
<td>-</td>
<td>4 CMOS</td>
<td>1 RRAM</td>
<td>1 CMOS</td>
</tr>
<tr>
<td>L2 - FPA</td>
<td>32 MB RRAM</td>
<td>-</td>
<td>4 CMOS</td>
<td>1 CMOS</td>
<td>1 RRAM</td>
</tr>
<tr>
<td>L3 - Cache</td>
<td>8 MB SRAM</td>
<td>32 MB RRAM</td>
<td>4 CMOS</td>
<td>1 CMOS</td>
<td>1 CMOS</td>
</tr>
<tr>
<td>L3 - ALU</td>
<td>8 MB SRAM</td>
<td>32 MB RRAM</td>
<td>3 CMOS/1 RRAM</td>
<td>1 CMOS</td>
<td>1 CMOS</td>
</tr>
<tr>
<td>L3 - Mult</td>
<td>8 MB SRAM</td>
<td>32 MB RRAM</td>
<td>4 CMOS</td>
<td>1 RRAM</td>
<td>1 CMOS</td>
</tr>
<tr>
<td>L3 - FPA</td>
<td>8 MB SRAM</td>
<td>32 MB RRAM</td>
<td>4 CMOS</td>
<td>1 CMOS</td>
<td>1 RRAM</td>
</tr>
</tbody>
</table>

value of 1 demonstrate high RRAM L3 cache energy due to almost every L2 access resulting in a miss to L3, and subsequently a miss to main memory. RRAM L3 cache energy is very low for almost all other benchmarks because of low L2 miss rates resulting in minimal L3 accesses. Therefore, current applications do not take advantage of having a third cache level because 8 MB of L2 cache is usually sufficient. This architecture does take advantage of the low read energy and low idle energy of RRAM, but having the 8 MB L2 SRAM cache increases the energy usage above the baseline.

These results show that SRAM cache is more energy efficient than RRAM cache. Even though RRAM has low idle energy, cache is used frequently in all applications, and is not often idle. SRAM cache energy usage is proportional to SRAM cache idle energy and varies based more on application runtime. RRAM cache energy is proportional to hit / miss energy and varies based more on the number of RRAM cache accesses. Whether or not the L2
RRAM architecture or the L3 RRAM architecture is more energy efficient is dependant on the application being run. 68% of the SPEC 2006 benchmarks have better cache energy efficiency using the L3 RRAM architecture, while the other 32% have better cache energy efficiency for the L2 RRAM architecture.

As stated previously, most benchmarks do not benefit from having an L2 cache of size 32 MB as compared to 8 MB for the L2 RRAM architecture. Out of the four exceptions, two benchmarks saw a slight speedup of up
to 1.06 (437.leslie3d and 470.lbm) due to a decrease in miss rates when having the larger 32 MB RRAM L2 cache as seen in Figures 6.2 and 6.3. The reason more speedup is not seen is because of the slightly slower read latency of RRAM cache compared to SRAM cache. This reinforces the statement that most current applications do not benefit from having a larger than 8 MB L2 cache, but overall, the system performance is not negatively affected.

Figure 6.3: SPEC2006 Benchmark Timing for Different Cache Levels

The RRAM L3 cache architecture also demonstrated similar speedup for
the same two benchmarks, but saw slight performance degradation in three additional benchmarks. These slowdowns are due to accesses to main memory having to miss through three levels of cache, and can be seen for benchmarks 429.mcf, 445.gobmk and 458.GemsFDTD. This demonstrates that having three levels of cache does not improve the performance of current applications. Based on the cache access patterns, a third cache level can even hinder performance.

These results show that the RRAM L2 cache architecture performs approximately the same as or better than the RRAM L3 cache architecture for all cases. When taking into account both energy efficiency and performance of RRAM cache, the RRAM L2 cache architecture remains sufficient, although the RRAM L3 cache architecture can be more energy efficient.

Figures 6.4 and 6.5 show the performance of the SPEC 2006 benchmarks with CMOS functional units replaced in the RRAM layer. The graphs for both architectures are nearly identical because RRAM access time for functional unit operations is not dependent on the level at which the RRAM is used for cache.

Having an ALU implemented in RRAM reduces the completion time of all benchmarks by \(\sim 77\%\) on average. This slowdown occurs due to factors including the slower speed of the RRAM ALU, and also the dependencies that occur between ALU operations. Since almost all benchmarks
have a high percentage of ALU operations, they generally have more dependencies between instructions involving operands that are calculated using ALUs. This causes the remaining faster CMOS ALUs to stall, waiting for the RRAM ALU to achieve a result they require.

Figure 6.4: SPEC2006 Benchmark Runtime With L2 RRAM Cache and Replaced LUT-Based FUs

Benchmarks involving low percentages of integer multiplier operations and floating-point adder (FPA) operations show very minimal reduction in
performance, such as 401.bzip2, 445.gobmk and 462.libquantum. Conversely, benchmarks involving high percentages of integer multiplier operations and FPA operations show high performance degradation. Worst cases show that 437.leslie3d completes 126% slower for a replaced integer multiplier, and 444.namd completes 225% slower for a replaced FPA.

![Normalized Runtime for 32MB L3 RRAM LUT-based FUs](image)

Figure 6.5: SPEC2006 Benchmark Runtime With L3 RRAM Cache and Replaced LUT-Based FUs

The energy usage of CMOS functional units and RRAM functional units for the SPEC 2006 benchmarks is shown in Figure 6.6 for the RRAM L2 cache architecture and Figure 6.7 for the RRAM L3 cache architecture.
Again, the graphs for both architectures are nearly identical because the RRAM functional unit operate independent of the level at which the RRAM is used for cache. Few small variations are due to idle energy between the slightly different runtimes.

![Figure 6.6: SPEC2006 Benchmark Energy With L2 RRAM Cache and Replaced LUT-Based FUs]

The RRAM functional unit energy usage is normalized to the strictly CMOS functional unit energy usage. Therefore, the failed CMOS ALU / one RRAM ALU value is the total energy of all four ALU functional units (three CMOS and one RRAM). It can be seen that benchmarks require three
to five times more energy on average when using an RRAM ALU. This can be attributed to both the additional idle energy from the three CMOS ALUs due to increased runtime, and the access energy of the RRAM ALU due to frequency of use, as all benchmarks perform a significant number of ALU operations.

![Normalized Energy for 32MB L3 RRAM LUT-based FUs](image)

**Figure 6.7: SPEC2006 Benchmark Energy With L3 RRAM Cache and Replaced LUT-Based FUs**

When the CMOS integer multiplier is replaced in RRAM, energy usage is reduced for all benchmarks except 437.leslie3d. This can be attributed to the large percentage of integer multiplication operations for that benchmark,
and the number of arrays accessed for one computation. The reduction in energy usage for integer multiplication for all other benchmarks is due to the low idle energy of RRAM and minimal number of operations performed. The same sentiments can be said for the floating-point adder in RRAM. Benchmarks with higher percentages of FPA operations, over 3%, show increased energy usage when compared to the CMOS FPA. For benchmarks where the FPA is not used frequently, the RRAM implementation improves energy efficiency.

The FIT rate decreases, and therefore reliability increases, for both the RRAM L2 cache architecture and RRAM L3 cache architecture in this research because hard errors occurring in CMOS functional units are mitigated by implementing replacements in the RRAM layer, preventing system failure. The RRAM L2 cache architecture’s performance is better for all of the benchmarks compared to the RRAM L3 cache architecture, independent of the number of functional units implemented, while the RRAM L3 cache architecture’s energy efficiency is better for 68% of the benchmarks. Overall benchmark performance and energy efficiency when CMOS units are replaced in the RRAM layer depends on the benchmark instruction profile, which is application dependent, and architecture independent. Using an RRAM L2 cache is sufficient compared to using an RRAM L3 cache because both architectures improve reliability and the difference in efficacy
between them is negligible.

### 6.2 Multi-core Simulations

![Figure 6.8: SPEC2006 Benchmark Energy With L3 RRAM Cache and Replaced LUT-Based FUs](image)

The PARSEC benchmarks were profiled in Gem5 to compare the amount of functional unit and cache operations. Figure 6.8 demonstrates that out of the four observed operations, all benchmarks are over 75% integer ALU or
cache operations. All benchmarks have large amounts of floating-point addition operations while no benchmark has significant usage of the integer multiplier. The benchmarks were ran to completion for various configurations of RRAM cache, CMOS functional units, and RRAM functional units as seen in Table 6.2.

### Table 6.2: Multi-Core Simulation Configurations

<table>
<thead>
<tr>
<th>Sim Name</th>
<th>L2 Cache</th>
<th># Cores, ALUs</th>
<th># Cores, Mult</th>
<th># Cores, FPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Core Default</td>
<td>64 MB SRAM</td>
<td>All, 4 CMOS</td>
<td>All, 1 CMOS</td>
<td>All, 1 CMOS</td>
</tr>
<tr>
<td>8 Core RRAM</td>
<td>256 MB RRAM</td>
<td>Default</td>
<td>Default</td>
<td>Default</td>
</tr>
<tr>
<td>8 Core ALU</td>
<td>256 MB RRAM</td>
<td>1, 3 CMOS/1 RRAM</td>
<td>Default</td>
<td>Default</td>
</tr>
<tr>
<td>8 Core Mult</td>
<td>256 MB RRAM</td>
<td>Default</td>
<td>1, 1 RRAM</td>
<td>Default</td>
</tr>
<tr>
<td>8 Core FPA</td>
<td>256 MB RRAM</td>
<td>Default</td>
<td>Default</td>
<td>1, 1 RRAM</td>
</tr>
<tr>
<td>8 Core ALU4</td>
<td>256 MB RRAM</td>
<td>4, 3 CMOS/1 RRAM</td>
<td>Default</td>
<td>Default</td>
</tr>
<tr>
<td>8 Core Mult4</td>
<td>256 MB RRAM</td>
<td>Default</td>
<td>4, 1 RRAM</td>
<td>Default</td>
</tr>
<tr>
<td>8 Core FPA4</td>
<td>256 MB RRAM</td>
<td>Default</td>
<td>Default</td>
<td>4, 1 RRAM</td>
</tr>
<tr>
<td>16 Core Default</td>
<td>128 MB SRAM</td>
<td>Default</td>
<td>Default</td>
<td>Default</td>
</tr>
<tr>
<td>16 Core ALU</td>
<td>512 MB RRAM</td>
<td>Default</td>
<td>Default</td>
<td>Default</td>
</tr>
<tr>
<td>16 Core Mult</td>
<td>512 MB RRAM</td>
<td>1, 3 CMOS/1 RRAM</td>
<td>Default</td>
<td>Default</td>
</tr>
<tr>
<td>16 Core FPA</td>
<td>512 MB RRAM</td>
<td>Default</td>
<td>1, 1 RRAM</td>
<td>Default</td>
</tr>
<tr>
<td>16 Core ALU8</td>
<td>512 MB RRAM</td>
<td>8, 3 CMOS/1 RRAM</td>
<td>Default</td>
<td>Default</td>
</tr>
<tr>
<td>16 Core Mult8</td>
<td>512 MB RRAM</td>
<td>Default</td>
<td>8, 1 RRAM</td>
<td>Default</td>
</tr>
<tr>
<td>16 Core FPA8</td>
<td>512 MB RRAM</td>
<td>Default</td>
<td>Default</td>
<td>8, 1 RRAM</td>
</tr>
</tbody>
</table>

Note: Cores not mentioned have the default # of CMOS units and 0 RRAM units

As seen in Figure 6.9, very few benchmarks had a speedup with the larger RRAM L2 cache size, with exceptions being *ferret* and *vips* for the 8-core architectures. Every other benchmark, when having a large RRAM L2 cache, did not show much performance change at all. Again, this shows that most current applications do not benefit, or get degraded, by having a very
large L2 cache. Generally, default cache sizes are sufficient, but having a larger L2 cache can improve performance in some cases by not having to overwrite data as often as a smaller cache size would.

![Normalized Runtime for L2 SRAM and L2 RRAM Caches](image)

**Figure 6.9: PARSEC Benchmark Runtime for L2 SRAM and L2 RRAM Caches**

As expected, most benchmarks showed improvement from the 8-core to the 16-core architectures due to less word being performed per core in parallel. Benchmarks fluidanimate and vips had some slight degradation of performance when using 16-cores due to the increased communication between cores. Inter-dependant workloads were spread across too many
cores, and therefore, time was spent waiting for results from other cores rather than performing computation. Some slowdown was experienced by these benchmarks as well, which can be attributed to the higher latency of RRAM cache over SRAM cache.

![Normalized Energy for L2 SRAM and L2 RRAM Caches](image)

Figure 6.10: PARSEC Benchmark Energy for L2 SRAM and L2 RRAM Caches

As seen in Figure 6.10, more energy is used by the SRAM cache in the 16-core architecture compared to the 8-core architecture, due to the larger capacity and increased cross-core communication through the shared L2 cache. Corresponding RRAM L2 cache architectures use approximately
double the energy in all cases. These results are identical to the single core RRAM L2 cache architecture. As stated previously, this increased RRAM cache energy is caused by the high write energy of RRAM and also the increased cache capacity compared to SRAM cache architectures.

Figure 6.11: PARSEC Benchmark Runtime With 8 Cores, L2 RRAM and Replaced LUT-Based FUs

Figures 6.11 and 6.12 show the performance of the PARSEC benchmarks with CMOS functional units replaced in the RRAM layer for 8-core architectures and 16-core architectures respectively. Overall, performance degradation increases as more functional units are replaced and implemented in
the RRAM layer. One core with a failed CMOS functional unit implemented in the RRAM slows the performance of the system down less than if half the cores have the same unit failed. The amount of degradation for a specific functional unit is dependent on the application profile of the benchmark, specifically the frequency of usage of that functional unit. Benchmarks with high usage of a failed unit will have a greater slowdown due to the implemented unit in RRAM being slower, creating a bottleneck for that core.

![Normalized Runtime for RRAM LUT-based FUs for 16 Cores](image)

Figure 6.12: PARSEC Benchmark Runtime With 16 Cores, L2 RRAM and Replaced LUT-Based FUs

In general, the 16-core architecture has less performance degradation
than the 8-core architecture, and both architectures perform better than the single-core architecture with RRAM L2 cache. All three architectures can be compared because the SPEC 2006 benchmarks profiles are similar to the PARSEC benchmark profiles. This degradation decrease can be attributed to the fact that the work is spread out across more cores, meaning there are additional CMOS units available as the number of cores increases, which are faster than units implemented in RRAM. The average runtime of a single-core system with RRAM L2 cache was between \(\sim 1.5\) and \(\sim 2.5\) times longer than the baseline system, while the average runtime of an 8-core system was between \(\sim 1.25\) and \(\sim 2\) times longer than the baseline system and the average runtime of a 16-core system was between \(\sim 1.2\) and \(\sim 1.5\) times longer.

The energy usage of CMOS functional units and RRAM functional units for the PARSEC benchmarks is show in Figure 6.13 for the 8-core architectures and Figure 6.14 for the 16-core architectures. These two graphs are very similar despite the performance degradation being worse for the 8-core architectures. This is caused by the two architectures having equal ratios of CMOS to RRAM functional units. It can also be seen that, again, the energy usage is dependent on functional unit usage. Most benchmarks have high energy usage for RRAM implemented ALUs and floating-point adders because the benchmarks perform a large number of operations using those functional units. The increased amount of energy is partially due to
idle energy due to increased runtime, and also the number of RRAM arrays used for functional unit implementation. Energy usage for the multiplication units is less than the baseline in most benchmarks due to the lack of multiplication operations in the benchmarks. The more CMOS multipliers are replaced in the RRAM layer, the less total energy is used by multiplication units overall because RRAM has lower idle energy than CMOS.

Since performance degradation decreases with more cores, the increase
of functional unit energy consumption due to idle energy was not as significant as it was for a single-core system. Overall, the amount of energy consumption increased less for multi-core systems as compared to single-core systems. The average energy consumption for the ALU and floating-point adder operations for single-core systems using RRAM functional units was between $\sim2$ and $\sim5$ times more than the baseline, while the average energy usage for multi-core systems using RRAM functional units was between
∼1.25 and ∼2.75 times more. The FIT rate decreases, and therefore reliability increases, for both the 8-core and 16-core architecture in this research because hard errors occurring in CMOS functional units are mitigated by implementing replacements in the RRAM layer, preventing system failure.

Overall, using RRAM for cache can speed up a system, regardless of what level of cache it is implemented as. Using RRAM for functional unit implementation, replacing CMOS functional units that have failed, enhances the reliability of a system, at the cost of performance degradation and increased energy usage for frequently used functional units. Sparsely used functional units implemented in RRAM use less energy overall due to the low idle energy of RRAM. The magnitude of degradation and energy usage is dependent on the application profile, and not the level at which RRAM is used for cache.
Chapter 7

Conclusions

This research demonstrated that total system failure can be mitigated by dynamically reconfiguring a portion of RRAM cache to implement a failed CMOS functional unit, increasing the reliability of a system. The proposed architecture was validated using the Gem5 performance simulator and the McPAT simulator, running SPEC 2006 and PARSEC benchmarks. The benchmarks successfully completed with functional units implemented as LUT-based RRAM logic units. The large size of the RRAM cache compared to traditional CMOS cache allows for larger subspaces to be implemented in RRAM without reducing the cache hit rate. Best case performance speedup was as high 1.25 with the use of a large capacity RRAM cache without logic implementation, while the best case RRAM functional unit energy consumption was below 10% of the CMOS energy consumption for the RRAM multiplier and floating-point adder.

The average runtime when using RRAM for functional unit replacement
was between $\sim 1.5$ and $\sim 2.5$ times longer than the baseline for a single-core architecture, $\sim 1.25$ and $\sim 2$ times longer for an 8-core architecture, and $\sim 1.2$ and $\sim 1.5$ times longer for a 16-core architecture. Average energy consumption when using RRAM for functional unit replacement was between $\sim 2$ and $\sim 5$ times more than the baseline for a single-core architecture, and $\sim 1.25$ and $\sim 2.75$ times more for multi-core architectures. Performance degradation was caused by slower than CMOS access time of RRAM, the number of sequential RRAM reads required to perform an operation, and the instruction profile of a benchmark. Energy consumption increase was due to additional idle energy used during extended runtime of a benchmark and the number of arrays required to perform an operation in RRAM. The performance degradation and energy consumption increase is justified by the prevention of system failure and enhanced reliability.

Overall, the proposed architecture shows promise for use in multi-core systems. Average performance degradation decreases as more cores are used due to more total functional units being available, preventing a slow RRAM functional unit from becoming a bottleneck. This research has also created a framework for testing hybrid CMOS/RRAM architectures, with which future realizations of RRAM can be implemented and compared. As new materials for RRAM are used, improved performance and energy efficiency will demonstrate better results.
Chapter 8

Future Work

As previous research has shown, RRAM can be used for temperature sensing [30]. Hybrid CMOS/RRAM architectures implemented as 3D-ICs can passively sense temperature of the CMOS layer while performing cache operations, or actively sense temperature by accessing a specific RRAM location. If dangerously high temperatures are detected in a CMOS functional unit, the system could temporarily deactivate the component and migrate activity to the functional unit implemented in RRAM temporarily. This would degrade system performance during the time which operations are performed in the RRAM layer, but it allows the CMOS functional unit to cool, preventing permanent damage.

Another application that could be researched in the future as improved realizations of RRAM are implemented is hardware acceleration. An application’s runtime profile could be analyzed before or during runtime and additional functional units could be implemented in the RRAM layer as needed.
If a large number of independent multiplications are required, such as matrix multiplication, additional multipliers could be implemented in RRAM layer to parallelize the computation and speedup the overall application.
Bibliography


