Approximate In-memory computing on RERAMs

Salman Anwar Khokhar
University of Central Florida

APPROXIMATE IN-MEMORY COMPUTING ON RERAMS

by

SALMAN KHOKHAR
M.S. Computer Science, University of Central Florida, 2017
M.S. Electrical Engineering, University of Central Florida, 2014
B.Sc Electrical Engineering, University of Engineering and Technology Lahore, 2009

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2019

Major Professors: Mark Heinrich & Gary Leavens

ABSTRACT

Computing systems have seen tremendous growth over the past few decades in their capabilities, efficiency and deployment use cases. This growth has been driven by progress in lithography techniques, improvement in synthesis tools, architectures and power management. However, there is a growing disparity between computing power and the demands on modern computing systems. The standard Von-Neuman architecture has separate data storage and data processing locations. Therefore, it suffers from a memory-processor communication bottleneck, which is commonly referred to as the 'memory wall'. The relatively slower progress in memory technology compared with processing units has continued to exacerbate the memory wall problem. As feature sizes in the CMOS logic family reduce further, quantum tunneling effects are becoming more prominent. Simultaneously, chip transistor density is already so high that all transistors cannot be powered up at the same time without violating temperature constraints, a phenomenon characterized as dark-silicon. Coupled with this, there is also an increase in leakage currents with smaller feature sizes, resulting in a breakdown of 'Dennard's' scaling. All these challenges cannot be met without fundamental changes in current computing paradigms. One viable solution is in-memory computing, where computing and storage are performed alongside each other. A number of emerging memory fabrics such as ReRAMS, STT-RAMs and PCM RAMs are capable of performing logic in-memory. ReRAMs possess high storage density, have extremely low power consumption and a low cost of fabrication. These advantages are due to the simple nature of its basic constituting elements which allow nano-scale fabrication. We use flow-based computing on ReRAM crossbars for computing that exploits natural sneak paths in those crossbars.

Another concurrent development in computing is the maturation of domains that are error resilient while being highly data and power intensive. These include machine learning, pattern recognition, computer vision, image processing and networking etc. This shift in the nature of computing workloads has given weight to the idea of “approximate computing”, in which device efficiency is improved by sacrificing tolerable amounts of accuracy in computation. We present a mathematically rigorous foundation for the synthesis of approximate logic and its mapping to ReRAM crossbars using search based and graphical methods.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

A steady miniaturization of CMOS transistors over the last few decades has spear-headed the growth in computing power and pervasiveness of microelectronic devices. CMOS process technology has successfully overcome several hurdles in its course to achieve the current capability of fabricating transistors with the width of a few nanometers. We are now at a stage where the fundamental laws of quantum physics dictate an end to CMOS scaling in its current form. There is significant focus on developing other technologies and computing principles that can cater to the fast growing demand of more efficient computing in the post-Moore-era. One such area of focus is the discovery of new materials and devices for transistor designs that are not limited by quantum tunneling effects [5], [6]. New paradigms such as quantum transistors and spintronics are also being explored as computing devices [7].

In addition to device level improvements, there are advances in computing architectures. The aim of new architectures is to minimize memory latency, chip size and energy consumption. One way to overcome memory latency is to perform computation in-memory or 'near-memory' [8]. In recent years, there has been an explosion in the number of devices that gather, process and communicate data with each other. This has given way to the era of 'Internet of Things' (IoT). More and more smart devices continue to join the 'IoT' network, with the projected number being 50 billion in 2020 [9]. Most devices that lie on the periphery of an 'IoT' network perform sensing and initial data processing. These devices are known as edge devices. Common computing tasks performed on these edge devices include image processing, natural language processing or general machine learning which involve application specific algorithms. Therefore, a strong wave of interest in Application Specific Integrated Circuits (ASICs) has been regenerated in the hardware and ASIC IP communities [10]. In our work, we focus on implementing kernels used in image processing and computer vision. There are numerous edge devices that require some image and video processing on-device. They perform on-device computation before transmitting or storing the output of such computation. The resulting data usually requires less storage space and network resources. On-device computation also reduces latency in time-critical applications. Such image and video processing edge devices include smart cameras, mobile phones, security systems and vehicle-mounted cameras. For all of these devices, low energy consumption and a small die size are critical requirements. Therefore, nanoscale fabrics and devices are viable candidates for such applications [11]. Nanoscale "memristive" devices provide a desirable set of traits for smart cameras and other edge computing devices as they bring together non-volatile storage and fast switching dynamics on the same device. Memristive crossbars have been shown to be a viable platform for re-configurable computing. Because of the extremely simple nature of their basic constituting element, memristive circuits naturally have the advantages of being extremely efficient with regards to area and power metrics as opposed to CMOS devices. Two-dimensional arrays or crossbars of nanoscale memristors have been used to implement arithmetic and logical circuits using flow-based computing [12], [13]. In this approach, Boolean variables are mapped onto individual memristors, and the Boolean function is evaluated by injecting a current into the crossbar at an input nano-wire while the current at an output nano-wire determines the evaluation of the function for a certain valuation of the Boolean variables.

Approximate computing refers to the idea that some computation can be bypassed if it results in very little loss in accuracy and large gains in other metrics such as area reduction, speed or energy consumption. Computer vision and machine learning algorithms are generally computationally intensive. Hence, a lot of research goes into approximate computing of image and video processing algorithms. As we will show in later sections, our design for approximate edge detection yields very large gains in area and power saving while giving up a small amount of accuracy. Additionally, using the flow-based computing paradigm has the added advantage that the image is loaded directly onto a nanoscale memristor crossbar, facilitating the possibility of an in-memory or near-memory computing architecture. Note that, such kernels form the basis for more complicated algorithms in computer vision pipelines and are therefore present in almost all applications.

CHAPTER 2: BACKGROUND

2.1 ReRAMs

Predominantly used memory devices are DRAM, SRAM and Flash memory. DRAM construction is based on transistors and capacitors while SRAMs use latches and transistors. Based on their design and properties, DRAMs are typically used for main memory while SRAMs are used as caches. SRAMs are more expensive but have smaller size and power consumption. Both of these memories are volatile and they need a voltage source to maintain their state. DRAMs suffer from leakage current, hence they need periodic refresh and have higher power consumption. Flash memories are non-volatile but have slower read/write times and smaller storage density. These are typically used for disk storage. Another form of flash memory is called 3D-NAND or VNAND. This has much higher storage density and is a viable candidate for replacing SRAM and DRAMs as non-volatile fast response system memories and caches. Other recent types of memories that are being developed include PCM, MRAM and ReRAMs. Phase change RAMs and ReRAMs both function by altering the nature of the underlying material instead of by storing and detecting charge. PCM stands for Phase Change Memory. It is non-volatile memory with fast read times and is suitable as system memory or disk storage with much higher durability and persistance than flash memory. A electrical current is applied to change PCM cells from an amorphous to crystalline structure, allowing you to store 0s and 1s in either state while the application of low voltage can read the data back. Because the state of the material is used to store information, multiple states of the same bit can be used to store more than 1 bit in PCM ram cells.

Magnetic RAMs use electron spin states to encode information in its cells. MRAMs also have the advantage of being non-volatile but also have faster access times than Flash memory and smaller power consumption making them viable for replacing DRAM and SRAMs in cache and system memories. STT-RAMs are a popular form of MRAMs. MRAMs generally have high stability and resistance to errors due to external conditions such as radiation or temperature.

Similar to PCM and MRAMs, Resistive Random Access Memory (ReRAM) is one of a new generation of memory devices that seek to overcome the limitations of traditional SRAMs and DRAMs. The compact nature and efficiency advantages of ReRAMs stem chiefly from the extremely simple nature of it's construction. The basic structure of a ReRAM is a three layer construction of top and bottom conductive layers which sandwich a switching medium that forms a conductive filament between the top and bottom layers depending on the current flow/voltage application across them. This is known as the resistance switching effect. An illustration is shown in Figure 2.1 [14]. Because of this simple construction, ReRAM devices fabrication process is uncomplicated and can yield small process nodes and the resultant devices possess very fast switching speeds and low power consumption and small device area. It is due to these reasons that they are considered to be among the beyond CMOS devices. The resistance switching effect has been extensively studied in metal-oxide-metal fabrics such as $SiO_x, Al_2O_3, Ta_2O_5, ZrO_2, and TiO_2$ [15]. Because of the nature of the I-V curve that we see in ReRAMs, this behavior is often described as memristive behaviour and the terms ReRAMs and memristive crossbars are used interchangeably. In this work, we focus on ReRAMs as switching devices only.

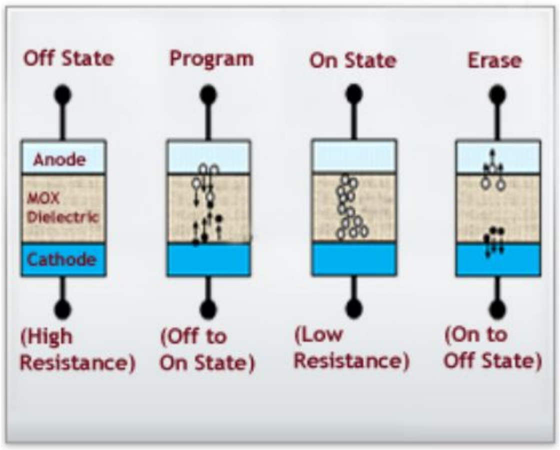


Figure 2.1: Formation of a filament in a Re-RAM cell that controls state of the cell.

Traditional ReRAM architecture is a crosspoint of top and bottom electrodes connected by resistive switching elements that can be created either using a 1 diode 1 resistor configuration (1D1R) or a 1 transistor 1 resistor configuration (1T1R). A number of systems have been developed to perform in-memory computing on ReRAMs or other memory devices [16–19]. These systems divide the memory device into separate sections for storage and processing along with additional circuitry that may be needed for computation in some techniques. In this thesis, we focus only on the computation part and assume a system for writing data into the computation part exists.

2.2 Approximate computing

Approximate computing refers to the idea that a computing processes can be allowed to provide incorrect results that are close enough to the desired output to be acceptable. In exchange for this loss of accuracy, the process gains in terms of power consumption, time or space efficiency. Approximate computing has been studied from many different perspectives such as difficulty of exact computation, saving device size, limiting power consumption and making the overall computation quicker. The choice of approximation technique depends on the nature of the task and whether it can afford to sacrifice some amount of accuracy or not. It was studied in mathematics and geometry theory for estimating complex functions. For such functions, exact computation may be complicated or time-consuming. The goal of approximating results is to approach acceptable correctness for most calculations with much smaller computation overhead. In more recent times, specifically since the arrival of computing, approximation has been introduced into a number of avenues in computing. One of the motivations has been to increase time efficiency for time-critical applications where slight loss of accuracy does not hinder user-experience such as in communication, audio-visual processing, and encoding etc. Approximate computing has achieved much greater significance in the last decade because of the emergence of a number of computing domains that are error-tolerant.

CHAPTER 3: LITERATURE REVIEW

In 1971, Leon Chua postulated the existence of a fourth fundamental circuit element that connects charge and flux, which he called the memory resistor or ‘memristor’ [20]. In 2008, Strukov et. al. showed that memristive behavior exists naturally in nanoscale devices [21]. Now, novel architectures and computing approaches [13], [22], [23] are seeking to exploit the unique properties of such devices. The behavior of a memristor is essentially that of a nanoscale variable resistor with non-volatile memory. This device can be used both as a Boolean resistive switch or an analog device with programmable resistance. The natural non-volatile behavior of a memristor makes it an ideal electronic model for a synapse. Therefore, neuromorphic computing infrastructures leverage memristive fabrics as layers of neural networks [24], [25]. However, research is still ongoing into accurate analog tuning in memristors [26], [27]. Although analog memristive chips are available, at present they are not considered viable for commercial applications. At the same time, memristive devices have been shown to be useful for general purpose computing by using them as bi-stable devices. Nanoscale Crossbars with memristive switching devices at the junctions can be used as nanoscale memory systems with efficient energy consumption [28]. ReRAMs have generated a lot of interest as high density compact memories [29], [30]. They have also been used for in-memory computing of basic logic functions [31], [32]. In a Programmable Logic-in-Memory (PLiM) computer [33], a crossbar array is used to implement majority and complement operators. The MIG-based compiler proposed in [34] allows the transformation of computer code into a sequence of majority operations that can be implemented on a PLiM. Bhattacharjee et al. [35] present a parallel architecture for in-memory computation on crossbars.

A typical computing stack has several levels at which approximation can be leveraged to improve efficiency. We discuss circuit level approximations only. Nepal *et al.* have synthesized approximate circuits by perturbing synthesis tree (AST) of behavioral description of circuits [36]. Suhail *et* *al.* have proposed approximate logic synthesis using Boolean matrix factorization [37]. Schlatcher *et al.* have proposed gate level pruning to build approximate circuits [38]. They have operated on synthesized circuits by removing gates and setting their outputs to constants (High or Low). Zdenek *et al.* have used Cartesian Genetic Programming (CGP) to design circuits within given hardware resources [39]. Soeken *et al.* have proposed approximate BDD minimization (ABM) for synthesizing approximate circuits. Given a circuit, they iteratively apply different approximation operators on its BDD and see if they lead to compact approximation within acceptable error bound [40]. The main theme of their technique is replacing cuts of a node (subgraphs in AIG) while making sure each modification keeps overall error below threshold. Chandrasekharan *et al.* have proposed And-Inverter Graphs (AIGs) rewriting for automated synthesis of approximate circuits [41]. Venkatamani *et al.* have proposed an automated synthesis technique (SALSA), they identify approximate don't care (ADC) conditions and treat them as Actual don't care conditions [42]. The same authors in their work titled 'SASIMI' search for highly co-related signals and signal pairs in the synthesized circuits and replace one with the other while removing logic behind the replaced signal [43]. Ranjan *et al.* have proposed similar idea to sequential circuits and called it ASLAN [44]. Shin *et al.* have minimized total number of literals by complementing min-term in the original Boolean function [45]. Rehman *et al.* have synthesized approximate multipliers by optimizing behavioral representation [46]. Miao *et al.* have used network optimization to identify external don't care conditions (EXDCs) that maximally approaches the target function under given error constraints [47]. While Zou *et al.* use dynamic programming to find near-optimal approximations for circuits [48].

Approximating Boolean Functions. Approximation theory is a well established topic. Approximation is necessitated by either problem complexity or limited resource availability. Mehta *et al.* [49] present a framework on the approximation of generalized Decision Trees. A study on approximation bounds for DNFs was presented by O'Donnell *et al.* [50]. In [51] Boolean functions are ε-approximated using small depth circuits. Ordered BDDs have been predominantly used for learning unknown functions using sampled executions [52]. The existing literature on reducing known functions using BDDs is limited to specific functions or theoretical studies of complexity and error bounds [53]. Herein, we limit our scope to the approximation of Boolean functions only.

CHAPTER 4: SNEAK PATH BASED IN-MEMORY COMPUTING

A ReRAM device consists of small crossbar components that have resistive switches at the crossbar junctions. A memristor is essentially a variable resistance device and ReRAMs can also be implemented with memristive elements. In the rest of this thesis, memristive device and resistive switching device are used interchangeably, since for the purpose of this work a memristor may be used a bi-stable device with either a HIGH or LOW resistance. The crossbar structure allows simultaneous data storage and processing. In the flow-based processing scheme, the state of individual memristors stores inputs in binary form. The horizontal and vertical crossbar wires are conductive. Modeling each wire as a node, an $n \times m$ crossbar can be considered to be a bi-partite graph with $n+m$ nodes and nm memristive connections between the nodes. Crossbar circuits suffer from the phenomenon of sneak-paths which occur when current is able to flow through unintended paths resulting in incorrect logic. This problem is compounded with an increase in size of a crossbar [54]. Although this is an inherent problem for ReRAM based memory storage, we exploit this property to perform in-memory computing. The principle of our sneak-path based computing is to manage sneak-paths between the input and output nanowires such that the current injected into the input nanowire flows out of the output nanowire only if the target function is $true$. Consequently, such sneak-path based computing is also known as flow-based in-memory computing [55] [56] [57] [58]. In the context of crossbars, flow-based in-memory computing and sneak-path-based in-memory computing are synonymous. The task of designing crossbars for sneak-path-based or flow-based in-memory computing is defined [56] as follows:

Definition 1 (Sneak-Paths based Synthesis of Crossbars) *Let $f : \{0,1\}^k \rightarrow \{0,1\}$ be a Boolean function with k inputs $I = \{b_1, b_2, b_3 \ldots b_k\}$, the objective is to map these k inputs or their complements on crossbar memristors such that the following two conditions are fulfilled for any input*

$x \in \{0,1\}^k$

- *If $f(x) = 1$, then there exists a sneak-path between the input and the output nanowires.*
- *If $f(x) = 0$, then there is no sneak-path between the input and the output nanowires.*

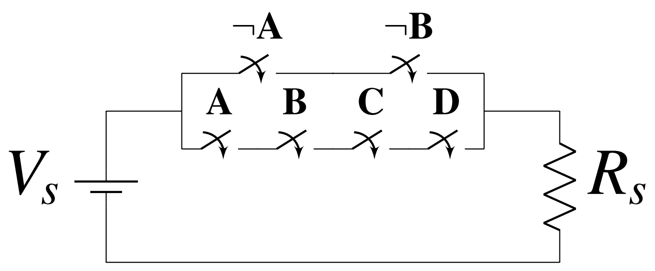


Figure 4.1: Switch based circuit for computing the Boolean formula $f = \neg A \neg B + ABCD$ using the flow of electric current.

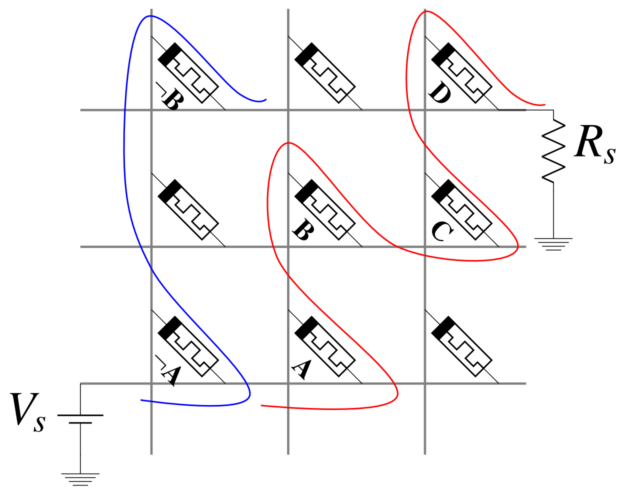


Figure 4.2: Crossbar for sneak-path based or flow-based in-memory computing of f, blue and red lines highlight the sneak-paths responsible for computing f.

In all our designs, we maintain the convention that the input is at the bottom nanowire, and the output current is sensed at the top nanowire. In Figure 4 we display the additional circuitry needed for sending output currents. Input current is driven by a the voltage source V_s, while a sense resistor R_s generates a voltage across it if an output current flows into the topmost nanowire.

When the function is $true$, low resistance of sneak-paths allow a significant amount of current to flow out of the topmost nanowire, resulting in significant voltage drop across the sense resistor R_s. When the target function is $false$, very little current flows out of the topmost nanowire, resulting in significantly smaller voltage drop across R_s.

CHAPTER 5: SEARCH BASED DESIGN SYNTHESIS

In order to synthesize approximate designs using flow-based computing in our work, we present two broad approaches. The first is to search for designs by modeling the search space and using a simulated annealing solution in a constrained search environment. In the second approach, we model Boolean functions as graphs and use graphical manipulation techniques to synthesize approximate solutions.

For the search-based design synthesis, our target applications are limited to basic image processing kernels in smart cameras.

5.1 Kernel Description

We exploit nanoscale memristor crossbars for implementing an approximate version of a compute kernel using flow-based computing. We focus on an edge detection kernel which is a basic image processing function. It is used as a pre-processing step or a preliminary feature for more complex feature computation from visual data. It is pertinent to mention here that our approach is generalized and can theoretically work for other image processing compute kernels as well. To detect edges in an image, each pixel in an image is compared to it's neighboring pixel(s). This comparison is usually performed along both axes separately. An edge is deemed to exist in the image at a pixel's location, if the difference in intensity values between itself and it's neighbor(s) is larger than a pre-determined threshold. We use κ to denote the edge detection kernel that we want to implement on a memristive cross-bar. The simplest edge-detection kernel can be mathematically expressed as follows:

$$\kappa(A,B) = (A - B) > \tau, \tag{5.1}$$

where A denotes a single pixel and B is it's neighboring pixel along either the x or y axis. Values for Grey-scale pixels are typically represented by 8-bits. Pixel value for a pixel A is calculated using it's 8 constituent bits as $\sum_{i=0}^{7} 2^i a_i$, where a_i is the i^{th} bit in pixel A. Pixel value for B is similarly calculated. Our kernel κ therefore, has 16 input Boolean variables (each bit of a pixel is one input Boolean variable) and a single Boolean output. τ is a pre-determined threshold value. In most real world applications, a threshold value is tuned using training data or chosen based on the nature of the application. In our experiments, we set it to 50. Note that the value for this parameter has no effect on the applicability of our frame work. If we change the threshold, it will simply change the truth value that is used to evaluate whether or not the current output of the crossbar is correct.

5.2 Crossbar Representation

We denote a crossbar as an $n \times m$ matrix M on which we aim to implement a simple 2 pixel edge detection kernel. Each element of M denotes the memristor connecting the horizontal and vertical nano-wires at it's indices. We want to be able to map each of our 16 Boolean input variables and their complement values onto the cross-bar. We also want to retain the ability to set a memristor value to a constant of 0 or 1. We do this using the following scheme: each element of the matrix can have a value ranging from 0 to 33. 0 denotes a memristor that is always turned off. 33 denotes a memristor that is always turned on. Values between 1 and 8 denote bits 0 to 7 of the first input pixel. Values between 9 and 16 denote bits 0 to 7 of the second pixel. Values between 17 and 24 denote complement bit values for bits 0 to 7 for the first pixel. Values between 25 and 32 denote complement bit values for bits 0 to 7 for the second pixel. The goal of our design is to map the Boolean variables onto the cross-bar in such a way that for a certain input combination of A and B, a current flows from the input nano-wire to the output nano-wire if and only if, $\kappa(A,B)$ also evaluates to 1.

5.3 Simulated Annealing

Existing attempts at finding a mapping of the input Boolean variables onto the cross-bar memristors include [59], [60]. These frameworks convert a compute kernel to a Binary Decision Diagram which is then mapped onto a cross-bar. This mapping leads to an exact solution. However, the memory and time costs for these methods, as well as the final design size grow prohibitively large as the number of Boolean variables in the compute kernel increases. We choose a relatively small cross-bar size and seek to implement an approximate version of the compute kernel on this smaller cross-bar.

Algorithm 1 Search for crossbar design for edge detection.

```
1:  procedure SEARCH-XBAR($M, \phi, \kappa, \zeta, N$)
2:      $M \leftarrow M_0$, $T \leftarrow T_0$, $\varepsilon \leftarrow \infty$, $l \leftarrow 0$
3:      while $\varepsilon > 0 \wedge l < l_{max}$ do
4:          $M' \leftarrow \Delta(M, \phi)$
5:          $\varepsilon' \leftarrow$ EVAL-XBAR($M', \zeta, N$)
6:          $X \leftarrow U(0,1)$
7:          if $\exp((\log\varepsilon - \log\varepsilon')/T) > X$ then
8:              $M \leftarrow M'$
9:              $\varepsilon \leftarrow \varepsilon'$
10:         $l \leftarrow l + 1$
11:         $T \leftarrow T \cdot \alpha$
12:     return $M$
```

Since we restrict ourselves to small cross-bar sizes, this allows us to search over the space of all possible designs. We make use of a constrained simulated annealing framework to find an optimal

Algorithm 2 Evaluation of goodness of a cross-bar design.

```
1: function EVAL-XBAR($M', \zeta, N$)
2:     $\varepsilon \leftarrow 0$
3:     for $1 \leq i \leq N$ do
4:         $p \sim \zeta$
5:         $q \sim \zeta$
6:         $\chi \leftarrow$ COMP-XBAR($M', (p, q)$)
7:         $\acute{\chi} \leftarrow \kappa(p, q)$
8:         if $\chi \neq \acute{\chi}$ then
9:             $\varepsilon \leftarrow \varepsilon + 1$
10:    return $\varepsilon$
```

solution. The constraint here is that the final solution needs to be sparse otherwise, the cross-bar will have too many paths linking the input and output nano-wires; hence, the design will yield a high number of false-positives. We enforce this constraint in the way we perturb the cross-bar at each iteration of the simulated annealing search. In each perturbation we modify some memristor mappings but also force an equal number of memristors to map to 0, hence ensuring that the total number of mapped memristors and therefore paths for current flow remain low. Our search frame work is described in **Algorithm** 1. The Δ function in our algorithm computes a perturbation for the design in each iteration. The perturbation rate ϕ, indicates how many elements of the matrix M should be changed. To evaluate the goodness of a candidate solution, we generate N pairs of pixel values. Each pixel value is independently sampled from a distribution of pixel intensities ζ learned from the BSD500 dataset [1]. We map each pair onto the current design and compare the simulated output with the exact output for κ. This evaluation function is described in **Algorithm** 2. The simulation of a cross-bar for a certain set of input variables can be done either using actual device models or by modeling the memristor connections as binary switches in a programming language based simulator. This process is described in Algorithm 3.

When performing the search for a design, we use a programming language based implementation of the cross-bar. However, once a design is found, we verify it using SPICE simulations as

Algorithm 3 Simulation procedure for crossbar matrix M given pixels A, B.

```
 1: variable definitions
 2:     M, Crossbar matrix
 3:     A,B, Pixel values
 4:     H,V, Status of horizontal and vertical wires in $M$
 5: end variable definitions
 6: function COMP-XBAR($M', (A, B)$)
 7:     Map elements of M (5.2)
 8:     Inject current at $H_1$
 9:     Simulate circuit
10:     if $H_m$ has current then
11:         return 1
12:     else
13:         return 0
```

described in subsequent sections.

5.3.1 Results with search based Framework

For our design, we have used a cross-bar size of 15 rows and 15 columns ($n = m$). Other sizes can also be used based on the nature of the kernel, which we plan to explore in future work. Our design for edge detection involving two pixels is shown in Figure 5.5. All the black colored memristors are always turned OFF while the green memristors are always turned ON. The blue memristors in the design are labeled by one of the 8 bits of the two pixels whose value (or its negation) should be loaded onto the memristor. In our simulated annealing, we set the initial temperature T_0 to 0.001 and α to 0.99.

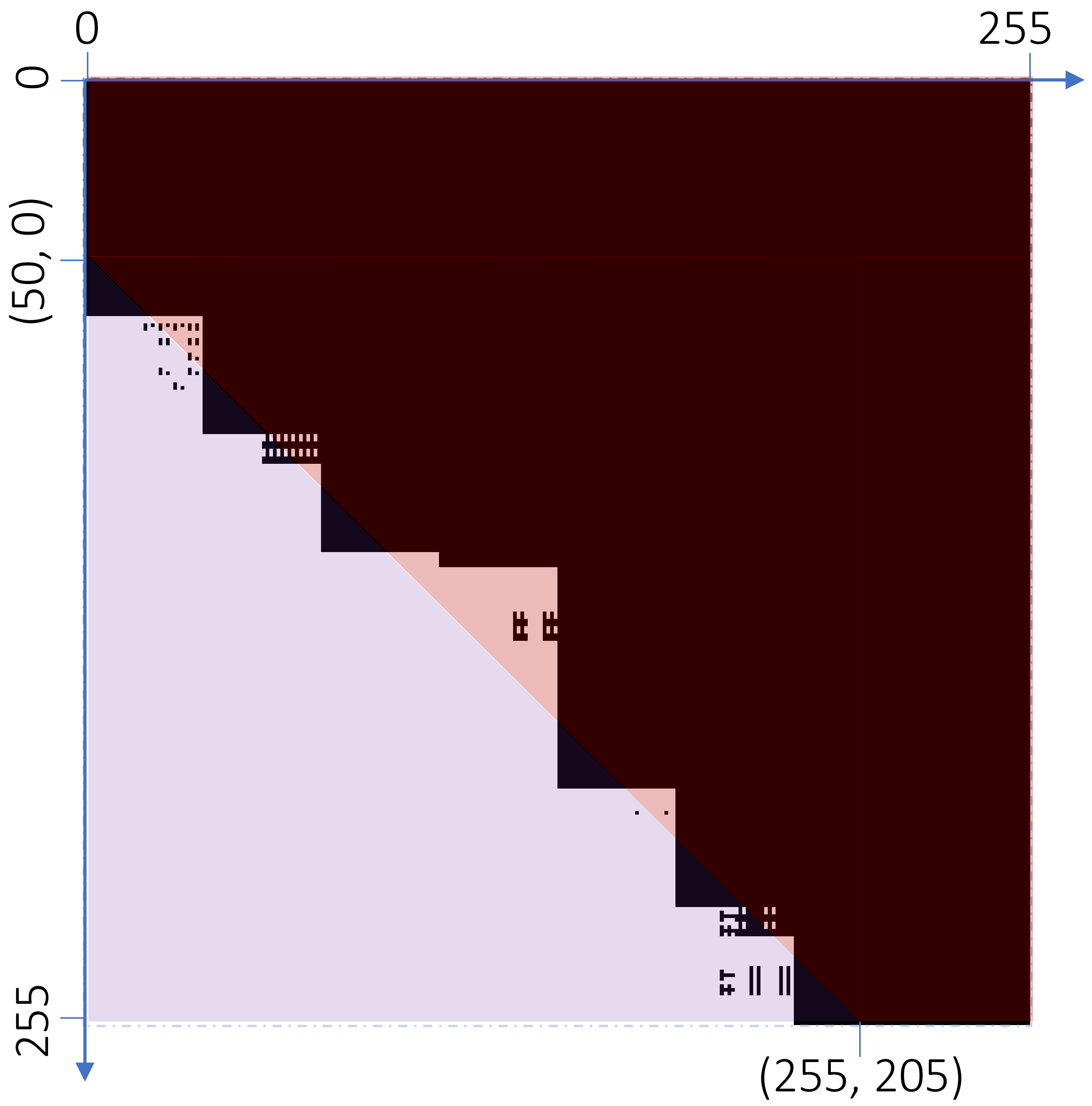


Figure 5.1: Output of our crossbar design on all possible pixel pair values.

Since our kernel takes in 2, 8-bit pixels, it is possible to map the outputs from our design on all possible inputs. In Figure 5.1, we show a 256×256 matrix. The x-axis shows values for one input pixel while the y-axis shows the values for the other input pixel. At each index (x, y) we show the result of our design. A dark pixel in this map denotes a low or ’0’ while a light pixel denotes a high or ’1’ output. We have superimposed the map of our designs output with the exact or desired output. The purple region denotes pixels that should be light while the maroon region denotes pixels that should be dark. We can see that our approximate circuit’s result closely follows the boundary between the two regions. We observe a staircase pattern around the ideal line, which is because of the inherently digital nature of our design. We can consider these errors as approximation or quantization errors in a digital circuit.

We tested our design on the Berkeley Segmentation Dataset (BSD500) [1]. This dataset contains a total of 500 images. For our experiments, we convert all images to grey scale as our crossbar designs are for single-channel images. The edge detection kernel is applied along the y-axis for this analysis. We get the error percentage on each image in the dataset. For each pixel in an image, we determine whether our method correctly identified an edge there by comparing it with an exact implementation. The results for all pixels in an image are then accumulated to get a single score for an image. The results of this analysis over all images are shown in Table 5.1.

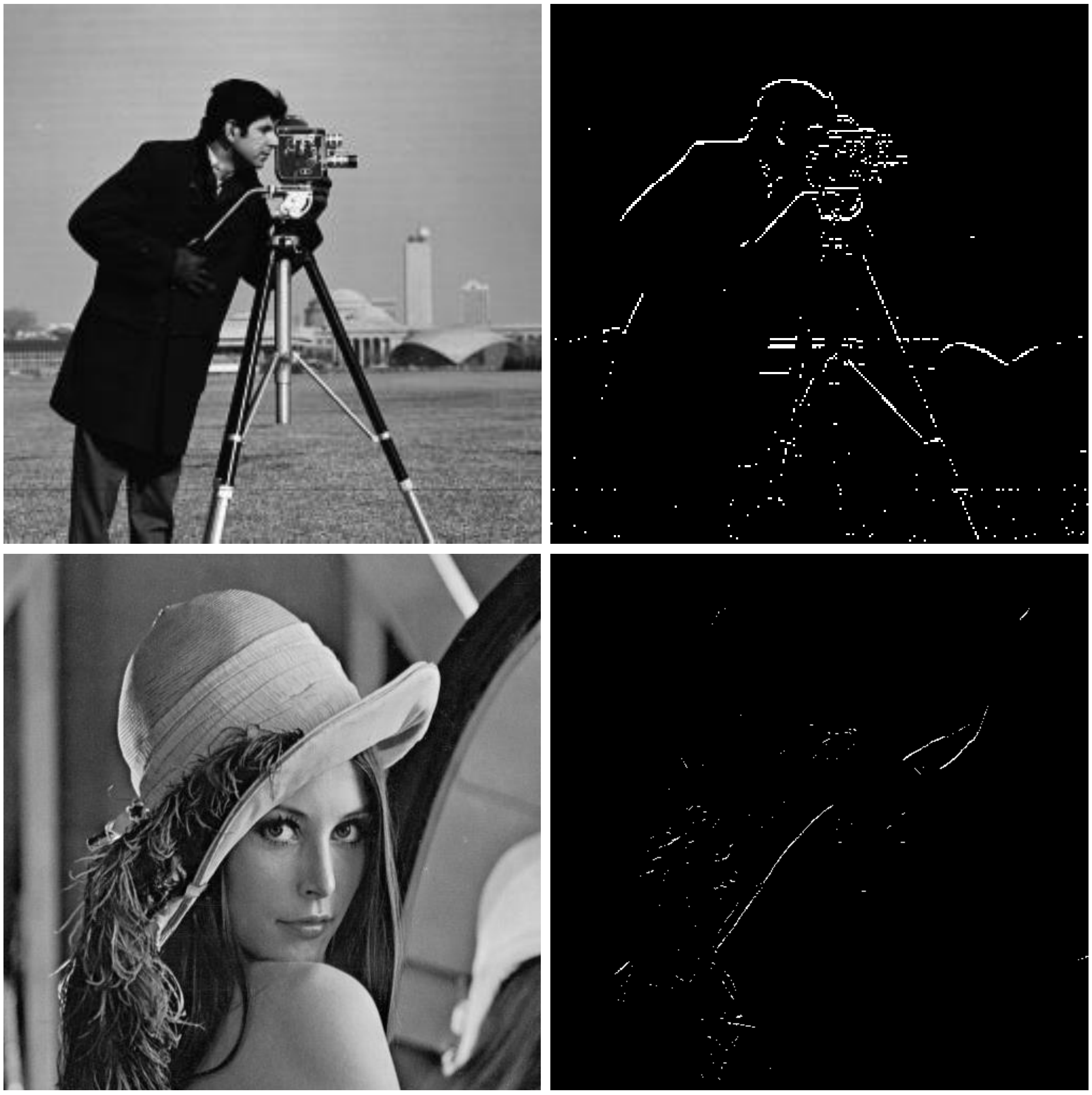
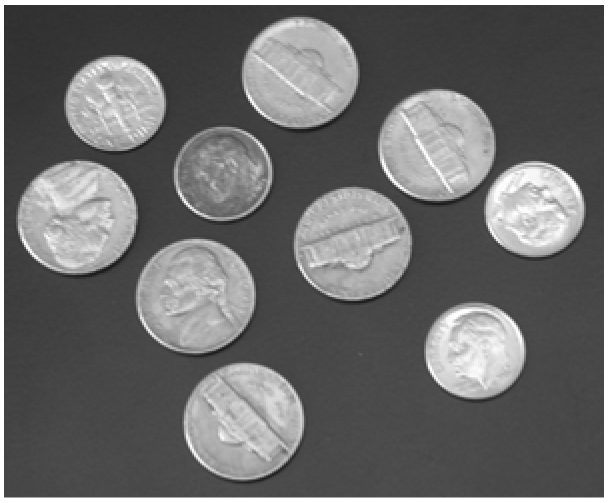


Figure 5.2: Evaluation of our crossbar design of Figure 5.5 on sample images of the BSD500 dataset.

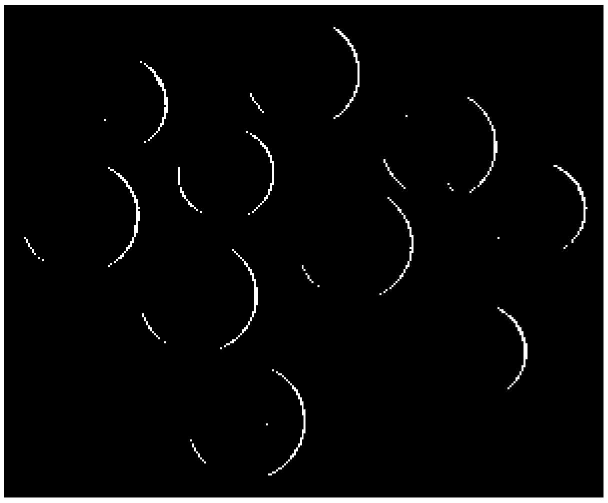
Table 5.1: The error rate (%) in edge detection for our crossbar model on the BSD500 dataset [1].

Mean Error	Max Error	Min Error
0.7 %	5.3 %	0.002 %

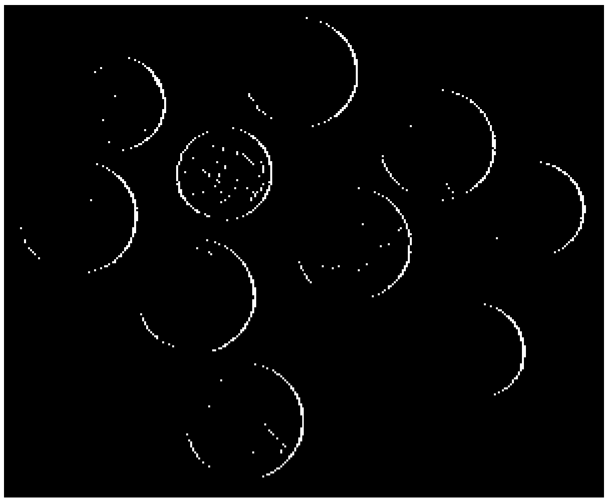
Some sample results from our crossbar design for edge detection are shown in Figure 5.2. We can see that the overall error rate of our method is extremely low, while edges obtained from the images are also visually meaningful.



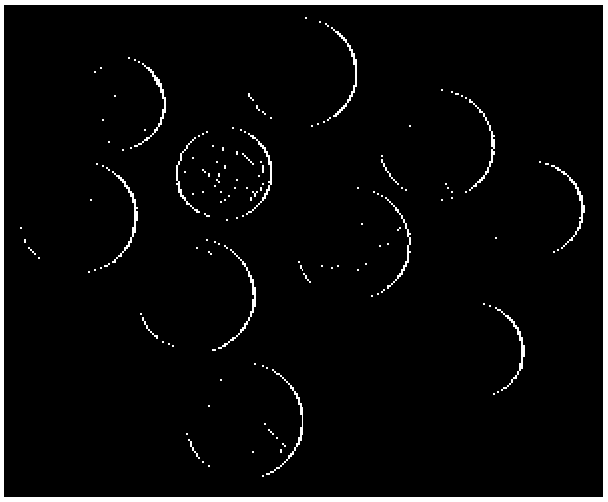
(a) Input image



(b) Result using the OpenCV library



(c) Result using code emulation



(d) Result using SPICE simulation

Figure 5.3: A comparison of our approximate edge detection using code-based emulation (c), SPICE simulation (d) and ground truth edge detection (b) on a sample image (a). For all three methods, the kernel used is [1, -1] applied along the x-axis

We have also simulated our crossbar design on HSPICE for figure 5.3a's coins image containing 300×246 pixels. For simulation, we assumed $\frac{R_{OFF}}{R_{ON}} = 10^4$, $R_{ON} = 100\Omega$, $R_{out} = 224\Omega$ and $V =$

$1000mV$. We have simulated our design on each horizontally adjacent pair of pixels of figure 5.3a. Figure 5.4 shows the histogram of the voltage values at the output for all simulated pairs of pixels. The x-axis represents the voltage across the output resistor R_{out}, where the width of each bin is $3.3nV$. The y-axis is the normalized frequency of voltage values at the output. The blue bars denote Low outputs corresponding to absence of edge. The red bars denote High outputs indicating successful detection of an edge. As shown in the graph, the output voltage was always less than 21 mV in the absence of an edge. When the simulated pair was located on an edge, the the output voltage was always greater than $156mV$. The High and Low outputs were separated by a gap of $135mV$; ensuring a perfect match between the HSPICE simulation and python emulation of the crossbar. Figure 5.3 shows a side-by-side comparison of edge detection results on a sample image obtained using the 'OpenCV' library [61], code emulation and SPICE simulation. It can be visually seen that emulation and simulation results are exactly the same whereas the 'OpenCV' result differs very slightly from our results. Our approximate methods are able to pick up all dominant edges, and show false positives on very few scattered pixels. Edge detection usually serves as the first level of features for more complicated vision algorithms down the line. Such algorithms are quite often robust from small perturbations such as the noise that we observe in our results.

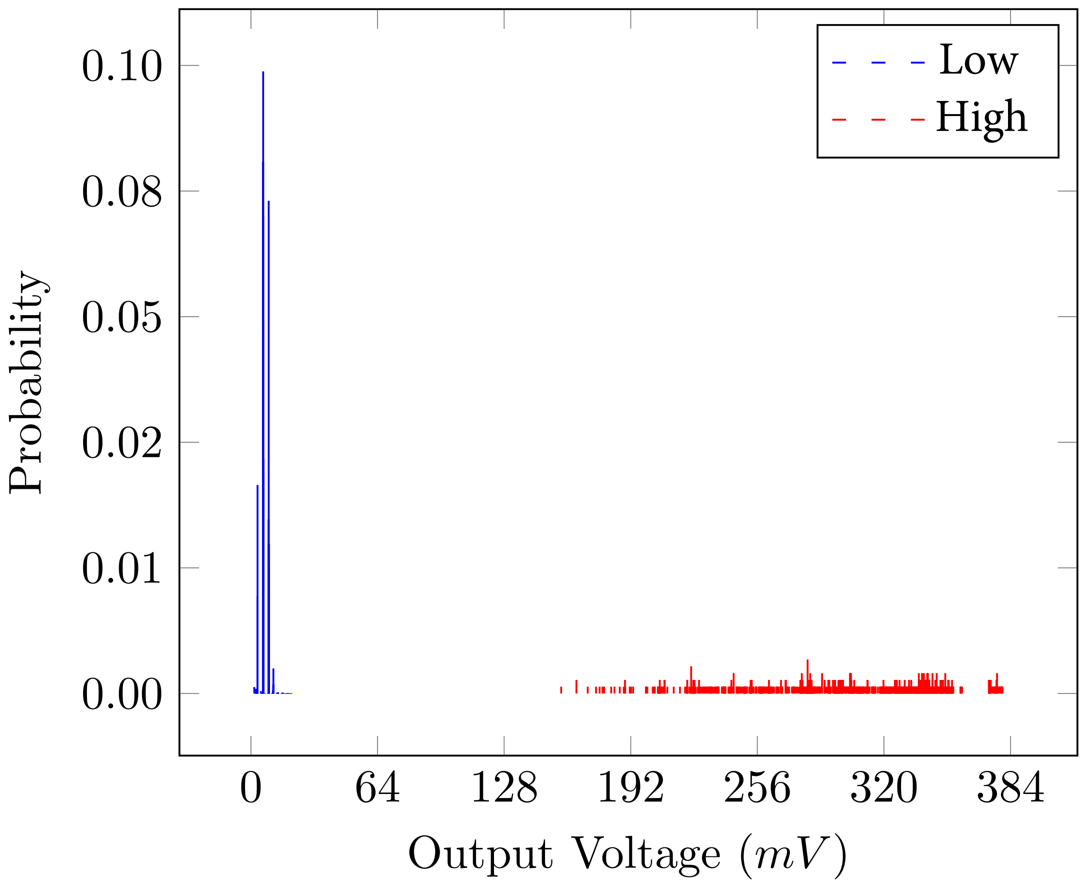


Figure 5.4: Histograms of HPSICE outputs for the coin image of Figure 5.3d. The x-axis represents the output voltage across R_{out}, while the y-axis denotes normalized frequency of the corresponding voltage values. The blue bars denote Low outputs, while red bars denote High outputs. This shows a clear separation between voltages observed for High and Low outputs.

A straightforward exact implementation of κ on a nanoscale crossbar requires the implementation of an 8-bit subtractor and comparator circuit. As discussed earlier, the generation of an exact design using existing methods such as [60], [59], [62] for such a large kernel is not trivial due to memory constraints and time cost. We were unable to generate the exact design using [60] on a standard quad-core machine. The closest result to our kernel κ is in [62] where they generate the design for a 4-bit subtractor using RGB values. Each output bit requires it's own cross-bar with the average size per bit being roughly 88×45. We were able to generate an approximate implementation of a kernel with similar size and complexity in a 15×15 crossbar, which represents a 94% gain in area savings.

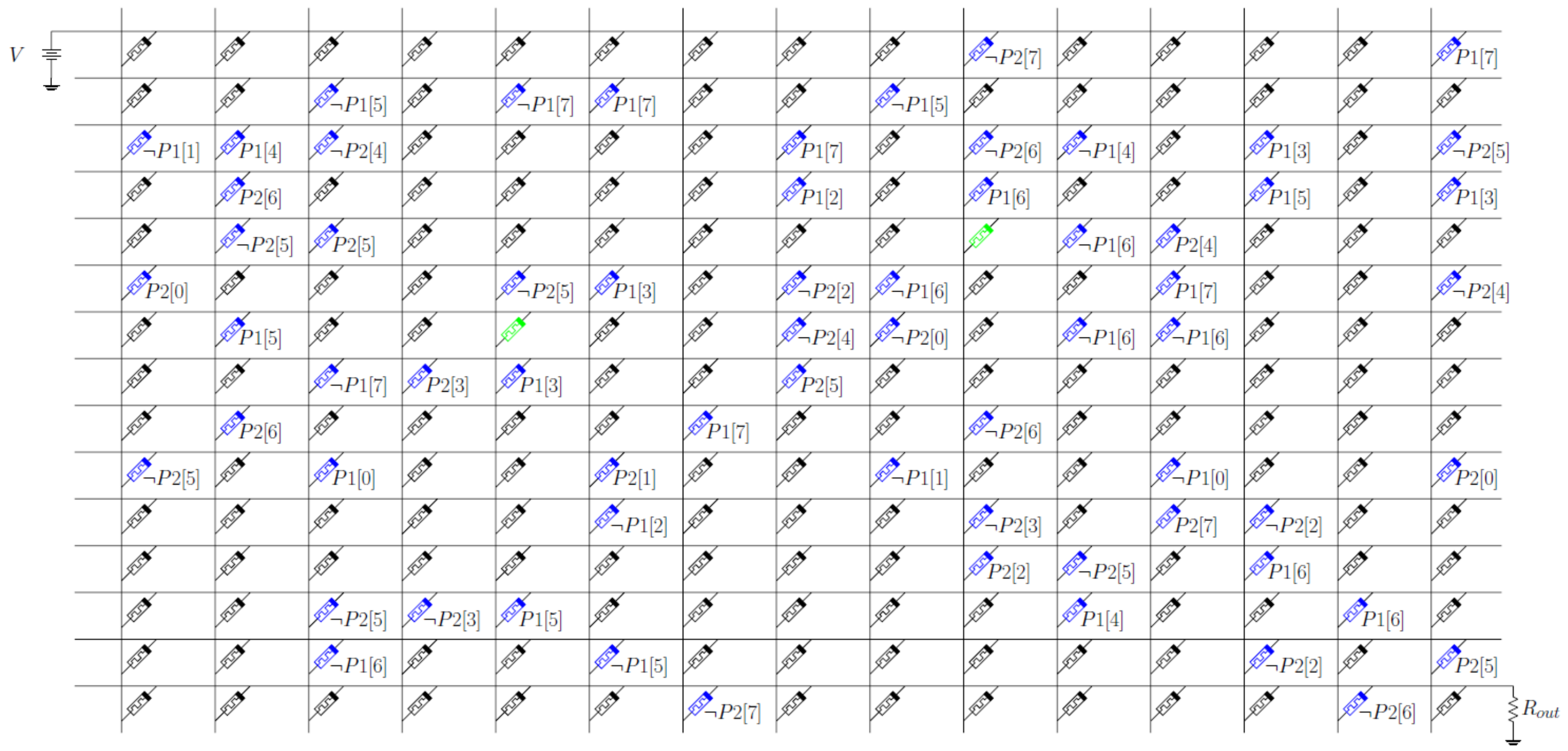


Figure 5.5: Our 15×15 crossbar design for approximate edge detection

One of the challenges still faced in computing and storage on memristive devices, is low error resilience. This is due to the presence of variation in the programmed resistance value of a memristor. To test the robustness of our design to resistance variation, we also tested our design in SPICE by artificially adding noise to our programmed resistance values of the memristors in their ON and OFF states. We increased the noise from 0% upto 16% of the resistance value and found that the gap between the output voltage was reduced only by a few μV. The flow-based computing paradigm makes use of sneak path currents to perform a computation. The combined resistance of all elements in a path from the input to the output nano-wire contribute to the final current, therefore the effect of error in any one element is negligible.

CHAPTER 6: GRAPH BASED SYNTHESIS

Our second approach to design synthesis is by graphical modeling. We use a Binary Decision Diagram representation for Boolean functions. We first provide an introduction to BDDs and then go into more details on the framework.

6.1 Binary Decision Diagrams

A BDD is a directed acyclic graph $G(V,E)$ used for compact encoding of Boolean functions [63]. A BDD has one root node, several intermediate nodes, and two terminal nodes (labelled as 0 and 1). Each node of a BDD represents a Boolean function. In that, the root node represents the original function, the terminal node-1 represents $true$, the terminal node-0 represents $false$, and non-terminal nodes represent functions derived from their predecessors through Shannon's expansion $f(x) = \neg b_i f_{b_i=0} + b_i f_{b_i=1}$. Each edge of a BDD is labelled by an input Boolean variable or its complement. Let $G_f(V,E)$ be a BDD representing the Boolean function f. Then by definition of a BDD, there exists a path between the root node and the terminal node-1 whenever its associated function f is $true$. Conversely, the root node is connected with the terminal node-0 whenever f is $false$.

6.2 FBDD Based Design Synthesis

Although Reduced Ordered BDDs (ROBDDs) are a more popular representation of BDDs due to their faster synthesis times [64], we use Free BDDs (FBDDs) in this work. Our rationale for this choice follows the fact that FBDDs allow variables to appear in any order on paths between the root node and the terminal nodes. The flexibility in variable orderings makes FBDDs more compact than ROBDDs that are required to maintain a global variable ordering along all paths between the root and the terminal nodes. Section 6.2.1 explains our approach of synthesizing *approximate FBDDs* that represent the target function f with the desired accuracy and uses them for automated synthesis of approximate crossbars.

6.2.1 Design Overview

In this section, we discuss our novel approach for synthesizing approximate FBDDs under relaxed equality constraints. Towards that, we first explain the construction of approximate FBDDs followed by the description of the overall synthesis process employing these approximate FBDDs.

Approximate FBDDs. Each branch node in a BDD represents sub-functions of it's root node. These sub-functions may or not be disjoint. We first describe how to compare two sub-functions before outlining how we use their approximation measures to synthesize BDDs that represent approximate versions of the original. A Boolean function $f : \{0,1\}^n \to \{0,1\}$ is said to $\boldsymbol{\varepsilon}$**-approximate**, another function $g : \{0,1\}^n \to \{0,1\}$ if the functions agree on at-least $(1-\varepsilon)2^n$ inputs. This can also be expressed as:

$$\Pr_{\mathbf{x}}[f(\mathbf{x}) \neq g(\mathbf{x})] < \varepsilon, \tag{6.1}$$

where, $\mathbf{x}$ is a random variable drawn from the uniform distribution $\{0,1\}^n$. In order to be able to estimate how well two nodes in a BDD graph approximate one another, we extend the definition of $\boldsymbol{\varepsilon}$**-approximation** for Boolean functions with the same input sets, to the case where the two functions are defined over unequal sets of input variables. More formally, assume a function $f : \{0,1\}^p \to \{0,1\}$, where each Boolean input variable is drawn from the set Y. The second function is defined as $g : \{0,1\}^q \to \{0,1\}$, where each Boolean input variable is drawn from the set Z. Let, $Y \neq Z$, while $|Y| = p$ may or not be equal to $|Z| = q$. Additionally, let us define $S = Y \cup Z$. Now, f is said to $\boldsymbol{\varepsilon}$**-approximate** g if:

$$\Pr_{\mathbf{w}}[f'(\mathbf{w}) \neq g'(\mathbf{w})] < \varepsilon, \tag{6.2}$$

where f', g' are modified versions of f, g, that take input from the set S. The random variable **w** can be decomposed into $(\mathbf{y}, \mathbf{y'})$ or decomposed into $(\mathbf{z}, \mathbf{z'})$, where **y** is sampled from the set Y, **z** from set Z and **y', z'** are sampled from sets '$S-Y$' and '$S-Z$' respectively. The modified functions f', g' can be evaluated as follows:

$$\begin{aligned} f'(\mathbf{y}, \mathbf{y'}) &= f(\mathbf{y}) | f(\mathbf{y'}), \\ g'(\mathbf{z}, \mathbf{z'}) &= g(\mathbf{z}) | g(\mathbf{z'}). \end{aligned} \tag{6.3}$$

Since f, g are not defined over the respective input variables from the sets $S-Y$, $S-Z$, hence those terms always evaluate to *false* and can be ignored. $\mathbf{y'}, \mathbf{z'}$ essentially serve as *don't care conditions* for f', g' respectively.

Using the above result, we can compare any two sub-functions or nodes in our BDDs. To evaluate the degree of divergence between two nodes, f and g, we essentially need to estimate the number of input combinations over which they differ, where the input permutations have to be over the union set S of the input sets Y and Z for f, g respectively. As defined earlier, **w** is the random variable consisting of Boolean variables taken from S. We can determine if the two nodes have the same behavior at sample **x** of **w** as $\Pi(f, g, \mathbf{x}) = f'(\mathbf{x}) \oplus g'(\mathbf{x})$ ($\oplus$ denotes the XOR operator). The overall divergence between f, g is then calculated as:

$$h = \sum_{\mathbf{x} \in dom(\mathbf{w})} \Pi(f, g, \mathbf{x}), \tag{6.4}$$

The two Boolean functions are approximately equal if h is less than a pre-defined threshold ε.

$$f \approx g \qquad \text{iff} \qquad h < \varepsilon, \tag{6.5}$$

where h is the fraction of $true$ bits in the output of Π, and ε is real number less than 1. For example if $\varepsilon = 0.05$, then f and g will be considered equal if their outputs differ for less than 5% of values. An important point to notice is that the notion of approximate equality in equation 6.5 does not require f and g to be defined on the same set of variables. It is still applicable even when f and g are composed of different sets of variables. This is an important point because the nodes being merge may not represent functions defined on the same set of variables. Here three corner cases are of interest, **(1)** f and g represent same Boolean formulae, then Π would always evaluate to $false$ for all inputs ($h = 0$), **(2)** f and g are complement of each other, then Π would be always be true (tautology), and h would be 1. **(3)** f and g are composed of entirely different sets of variables, i.e, $Y \cap Z = \varnothing$, then h would evaluate to 0.5.

We have employed the above proposed approximate equality for the merge operation in BDD creation. In our synthesis, two nodes are considered functionally equivalent in the approximate sense if $h(f,g)$ is less than a predefined threshold (ε). When ε is non-zero, it is obvious that more nodes will qualify to be merged together, leading to a more compact Approximate FBDD (AFBDD) at the cost of some error. By simply changing the threshold ε, we can trade-off AFBDD size and accuracy. It is important to mention that apart from representing the original function in approximate sense, AFBDD is semantically similar to any other FBDD, it has one root node, each node has functional significance, each node has two outgoing edges and all paths end on either of the two terminal nodes (0,1).

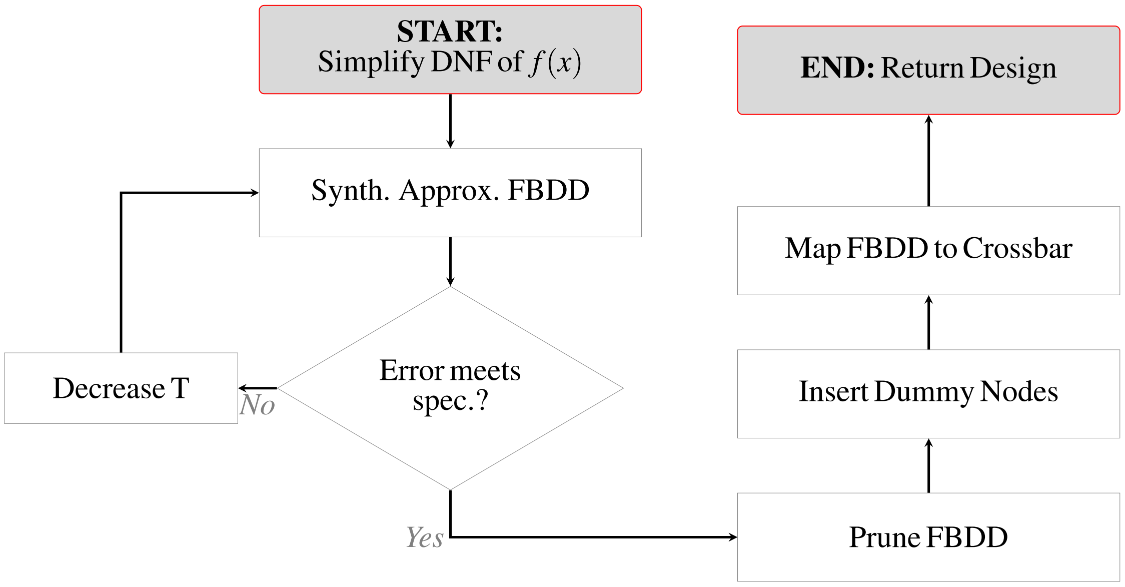


Figure 6.1: Flowchart depicting the steps of our Approximate FBDD based crossbar Synthesis.

Synthesis Framework Figure 6.1 shows our synthesis framework for generating the approximate crossbar of an arbitrary Boolean function $f(x)$ where our goal is to meet a certain accuracy specification for the crossbar. First, we simplify the disjunctive normal form (DNF) of $f(x)$. Next, we construct the approximate FBDD of this function as outlined in section (6.2.1) with a low initial threshold for equivalence T (where $T = 1 - \varepsilon$). This FBDD represents $f(x)$ in an approximate sense. We check if this FBDD represents the original function $f(x)$ with an accuracy that meets our desired specification. If it does not, we decrease the tolerance threshold T and reconstruct the approximate FBDD. This process is iterated till we obtain an FBDD with the desired accuracy.

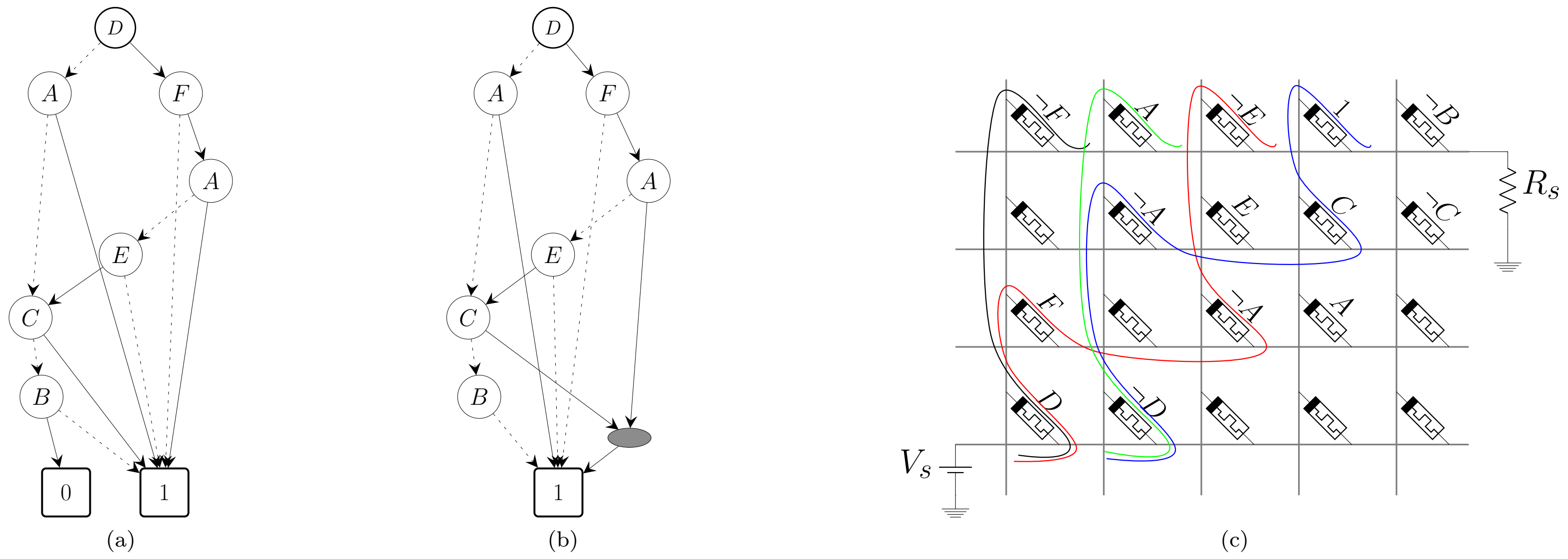


Figure 6.2: (a) Approximate FBDD for the newtag benchmark (b) Pruned Bipartite Approximate FBDD for the newtag benchmark (c) Approximate crossbar for in-memory computing of the newtag benchmark. Four (out of eight) sneak-paths (black, red, green, and blue) are highlighted. They are activated when $(D\neg F)$, $(DF\neg A\neg E)$, $(\neg DA)$, $(\neg D\neg AC)$ respectively are *true*.

Figure 6.2a shows an FBDD for representing the newtag benchmark (MCNC [3]). This FBDD represents the original function with $\approx 99\%$ accuracy. Next, we modify the FBDD for crossbar mapping. To this end, we first prune the FBDD by removing the terminal node-0 and all the edges connected with it. We remove terminal node-0 since the terminal node-1 is sufficient for flow based computing. Next, we make the pruned FBDD bipartite because a crossbar itself has a bipartite structure, i.e. no horizontal wire has a direct memristive path to another horizontal wire. The same is the case for vertical wires. We make our FBDD bipartite by inserting dummy nodes in cycles that have an odd number of edges. Removal of all odd length-ed cycles from a graph makes it bipartite. Figure 6.2b shows the pruned bipartite FBDD for the newtag function. In this figure, the grey node represents a dummy node which removes odd length-ed cycles. We can then translate the pruned bipartite FBDD to a crossbar design. We set the root node to the bottom nanowire. Nodes that are at even-numbered distances from the root are set to horizontal nanowires, and those that are at odd-numbered distances from the root are set to vertical nanowires. This completes the synthesis process. Figure 6.2c shows the crossbar obtained after mapping the pruned bipartite graph of Figure 6.2b onto the crossbar. This crossbar contains 8 sneak-paths between the bottom and the topmost nanowires. We highlight 4 sneak-paths using black, blue, red, and green lines. The black sneak-path is activated when the expression $D\neg F$ is $true$. Similarly red, green, and blue sneak-paths are activated for $DF\neg A\neg E$, $\neg DA$, $\neg D\neg AC$ respectively.

Table 6.1: Comparison of Exact and Approximate Crossbars for RevLib [2] and MCNC [3] Benchmarks

Benchmarks	Exact FBDDs	AFBDD Size	AFBDD Acc	Area Reduction
ex1_150	6 by 5	6 by 5	100%	0 %
ex2_151	9 by 6	7 by 5	90.6%	35.2%
ex3_152	6 by 6	3 by 4	93.8%	66.6 %
majority_176	8 by 6	6 by 5	96.9%	37.5%
sf_232	6 by 6	5 by 5	93.8%	30.6 %
cm152a_130	5 by 8	4 by 6	93.8%	40%
sym6_63	11 by 13	9 by 11	96.9%	30.7%
9sym6	39 by 39	29 by 21	92.2%	59.9%
life_175	21 by 21	16 by 15	94.3%	45.6 %
max46	68 by 64	44 by 45	89%	54.5 %
newill	12 by 12	8 by 9	98%	50%
newtag	7 by 8	4 by 5	99.2%	64.3 %

6.2.2 Experiments and Evaluation

We use our framework to generate approximate designs for various RevLib, MCNC benchmarks and basic image processing kernels [2, 3]. Our approach produces crossbars that are up to 80% more succinct, while maintaining $\approx$90% accuracy. Table 6.1 summarizes the performance of our approach on RevLib and MCNC benchmarks. While no direct comparison is available for approximation techniques for crossbar computing, we see that compared with general circuit approximation techniques, for a 5% approximation, we see on average a 43% reduction in area as opposed to 5.9% from [42] and 30.25% from [37]. We also synthesize crossbars for a commonly used image processing application, edge detection. Our edge detection kernel is expressed as $|p_1 - p_2| > \tau$, where 'p_1' and 'p_2' are adjacent pixels, while τ is the detection threshold. Table 6.2 compares the sizes of exact and approximate designs for different thresholds. Our best result saves 80% area while maintaining 96% accuracy. Figure 6.4 shows the results from an HSPICE simulation of one of our designs for an edge detector computing '$|p_1 - p_2| > 32$'. The size of this crossbar is 28×29 and represents the original function with an accuracy of 91.6%. Memristors with $\frac{R_{OFF}}{R_{ON}}$ as high as 10^7 have been reported in recent literature [65]. We assume a $\frac{R_{OFF}}{R_{ON}} = 10^4$. We use $V_s = 1V$, $R_s = 65\Omega$, and $R_{ON} = 50\Omega$. The X-axis plots all input combinations obtained by concatenation of input operands, while the Y-axis plots corresponding voltage across the sense resistor R_s. The green bars denote 'High' outputs, while blue bars denote 'Low' outputs. During simulation, the maximum voltage for Low output (V_{OL}) was $43mV$, while the minimum voltage for High output (V_{OH}) was $87mV$, thus resulting in a noise margin of $44mV$, which is sufficiently high to safely detect the presence of a sneak path. Figure 6.3 compares the output of an approximate design with the equivalent exact design for the 'cameraman' image. The two results are visually similar, showing that the approximate result is suitable for real world use-cases in error-tolerant applications while yielding significant increase in efficiency.

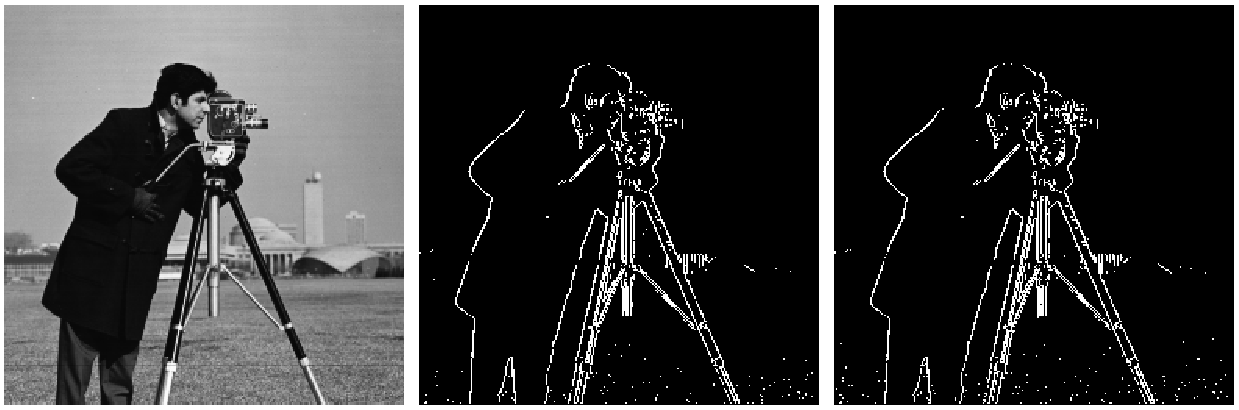

(a) Input image (b) Exact result (c) Approximate result

Figure 6.3: Performance of our approximate edge detector design on the “camera man" image compared with an exact design.

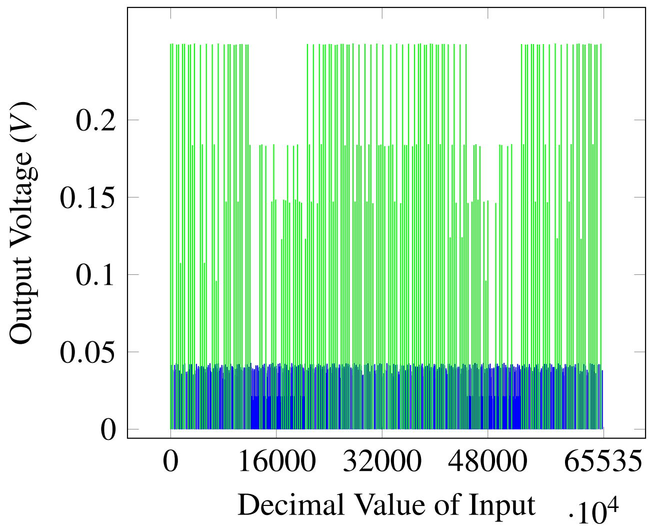


Figure 6.4: Hspice simulation of 28×29 crossbar for 8-bit edge detector($\tau = 32$). Here the X-axis shows the decimal value from concatenation of two pixels, the Y-axis denotes the voltage at the sense resistor. The Low ($false$) and High ($true$) outputs are shown by blue bars and green bars respectively.

Table 6.2: Comparison of Exact and Approximate designs of crossbars for edge detection kernels ($|p_1 - p_2| > \tau$)

Threshold	Exact designs	Approx. designs	Accuracy	Area improvement
$\tau = 32$	79 by 81	28 by 29	91.6%	87.3%
$\tau = 64$	121 by 123	73 by 73	97.4%	64.2%
$\tau = 80$	133 by 123	51 by 47	97.1%	85.3%
$\tau = 96$	113 by 115	66 by 65	94.8%	66.9%

Our results on improving area efficiency of these crossbars, validate the effectiveness of our approach. Next we use the Reduced Ordered BDD representation (ROBDD) for our synthesis.

6.3 Progressive Approximation

The sub-functions represented by branch nodes in a BDD representation depend on the variable ordering chosen for the FBDD. In order to explore further possibilities for node mergers, in our second approach, we try random variable orderings to find a better approximations. We do this incrementally starting from a small ε gradually increasing in each iteration until an approximation is found that reduces the size of the resultant crossbar while still meeting the design specification. We then start with the AFBDD obtained and repeat the same process to find a second AFBDD that is smaller than the AFBDD found after the first stage, and so on. We repeatedly cascade these AFBDDs until we find an AFBDD that meets a given accuracy specification or the maximum set iterations are reached. Finally we add a final cascade of an FBDD representation with the proposed size of circuit based ordering without adding any further approximation to ensure the final ordering results in as compact a graph as possible.

We evaluate our proposed methods on a number of large benchmarks given in Table 6.3. We can see that for almost all benchmarks we are able to obtain large area savings while retaining very high accuracy. The ‘progressive’ synthesis is generally able to achieve better results owing to the randomized variable ordering used during the search process which results in more node mergers during the AFBDD creation.

With this method, we generate several designs for edge detection with different thresholds (τ) to obtain their area-error graphs which are plotted in Figure 6.5. We can see that designs that maintain very high accuracy (close to 97%) are able to reduce the crossbar sizes by up to 80%. Since edge-detection is error-resilient, the added noise does not significantly change the final output of any algorithms that use extracted edges from an image. Figure 6.6 illustrates that edge detection output remains meaningful for close to 10% error and degrades significantly at error levels close to 25%. Table 6.2 shows crossbar sizes for some of our synthesized designs.

Table 6.3: Single Step and Progressive AFBDD-based Synthesis

	Exact designs	Approximate designs					
		Single step			Progressive		
Benchmark	Size	Size	Accuracy	Improvement	Size	Accuracy	Improvement
Mult b5	49 by 50	36 by 41	99.2%	39.7%	36 by 38	99.6%	44.16%
Mult b6	39 by 38	27 by 26	94.5%	52.6%	25 by 25	92.2%	57.8%
Mult b7	22 by 24	19 by 22	99.2%	20.83%	16 by 20	96.9%	39.39%
Mult b8	10 by 11	6 by 8	94.1%	56.36%	9 by 9	96.5%	26.36%
majority_176	8 by 6	6 by 5	96.9%	37.5%	4 by 5	96.9%	58.33%
max46	68 by 64	44 by 45	88.3%	54.5%	24 by 23	91.4%	87.31%
newill	12 by 12	8 by 9	98%	50%	2 by 1	94.5%	98.6%
newtag	7 by 8	4 by 5	99.2%	64.3%	3 by 3	97.7%	83.9%
sf_232	6 by 6	5 by 5	93.8%	30.5%	6 by 6	100%	0%
sym6_63	16 by 19	9 by 11	96.9%	30.7%	7 by 9	90.6%	55.9%
cm152_a_130	8 by 5	4 by 6	93.8%	40%	6 by 11	100%	-65%

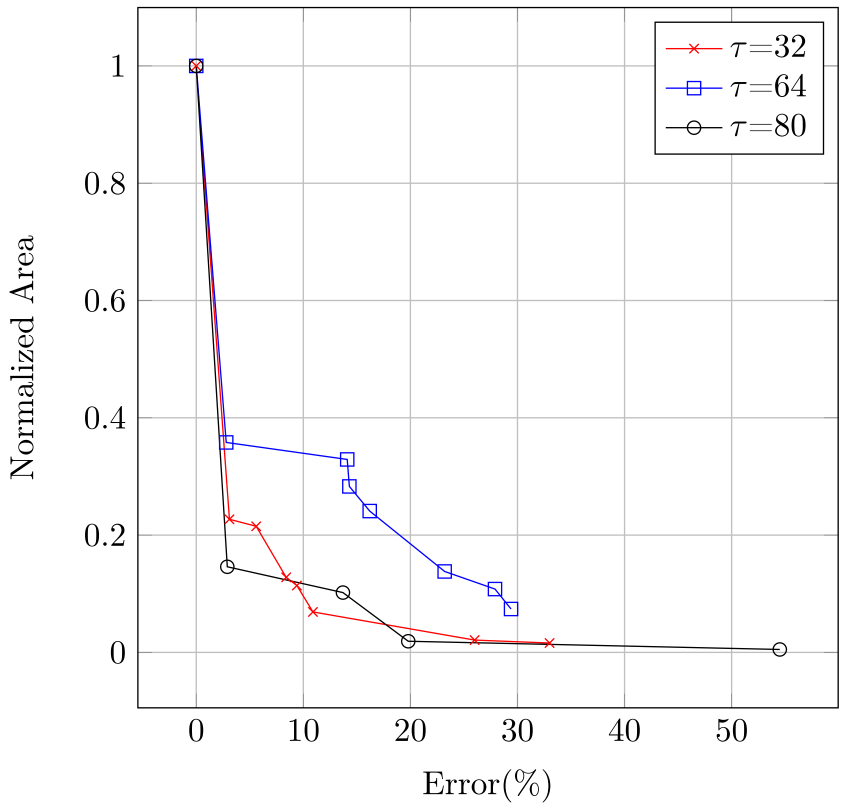
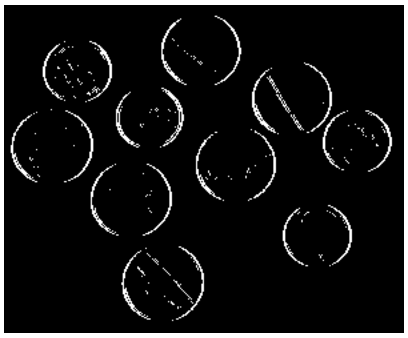
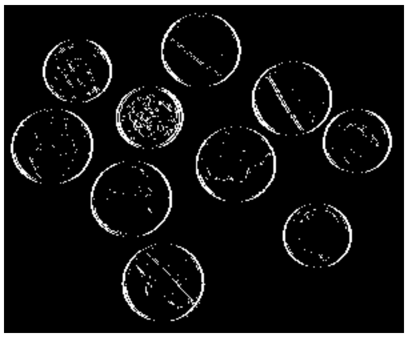
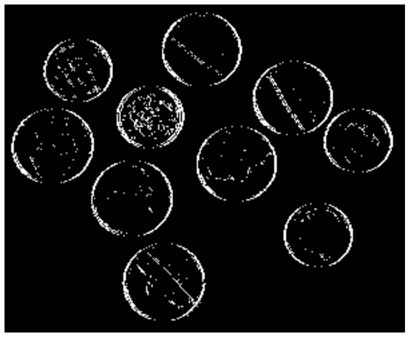


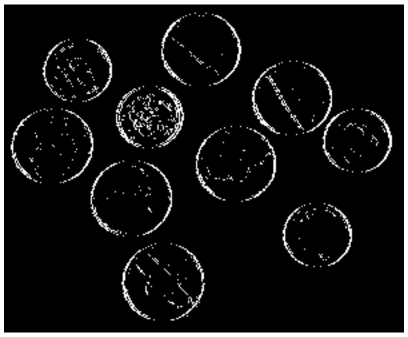
Figure 6.5: Plot of Crossbar area vs Error for edge detector with different values of threshold (τ).

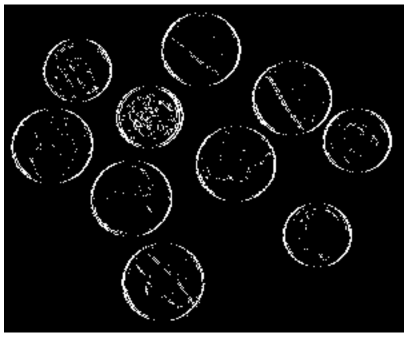
(a) Input Image

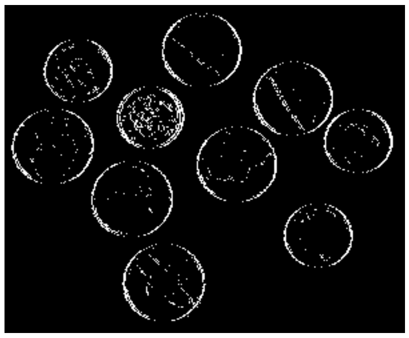

(b) Exact result

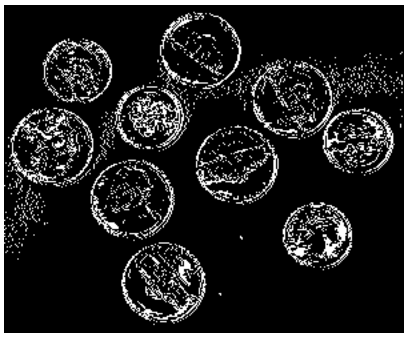

(c) Error: 3%


(d) Error: 6%


(e) Error: 8%


(f) Error: 9%


(g) Error: 11%


(h) Error: 26%

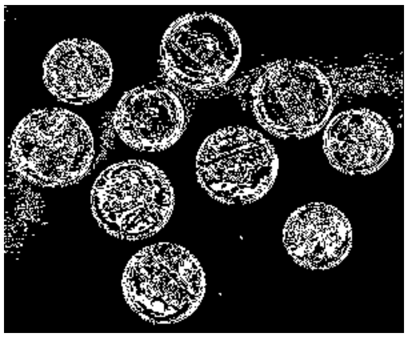

(i) Error: 34%

Figure 6.6: Performance of our approximate edge detectors (for threshold, $\tau = 32$) with varying error levels, on the "coins" image compared with an exact design.

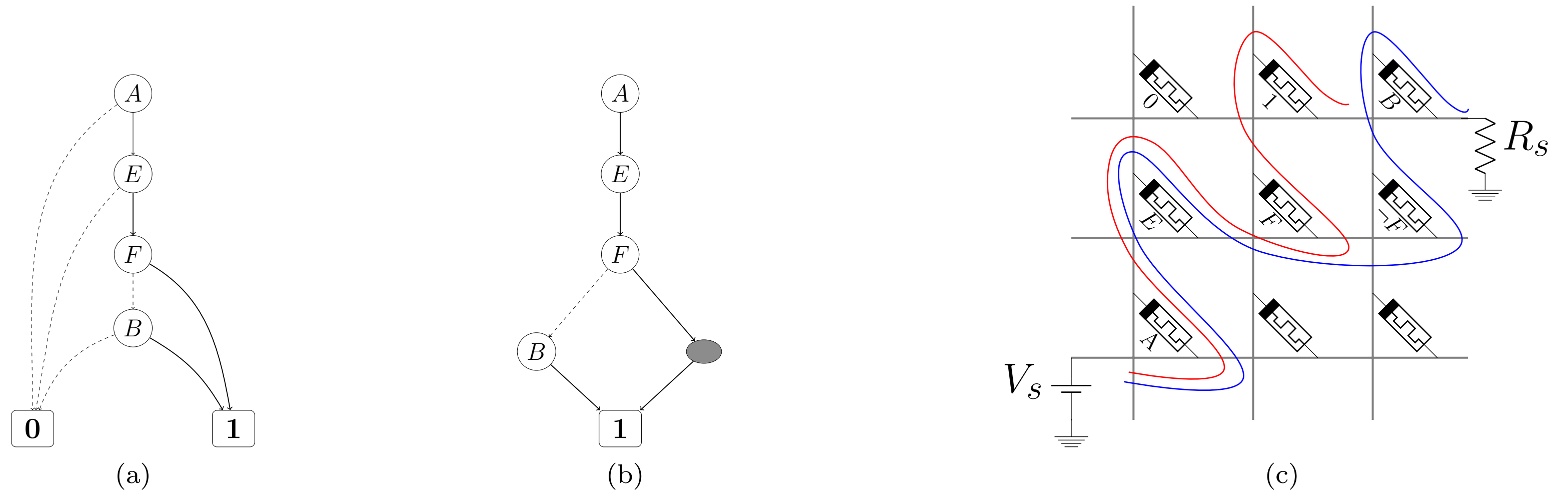


Figure 6.7: Different stages of A-ROBDD based synthesis. (a) A-ROBDD representing the MSB of a 4-bit multiplier. (b) Pruned-bipartite A-ROBDD, grey node is dummy node (c) Final design obtained after mapping pruned-bipartite A-ROBDD on a crossbar. Our approximate design is implemented on a 3×3 crossbar, while the exact design needs a 7×9 crossbar [56]

6.4 ROBDD Based Design Synthesis

We also use the more commonly used ROBDD representation of Boolean functions to synthesize our approximate designs. The overall design process is very similar to that used for FBDDs with the exception of the variable ordering in the BDD representation. Figure 6.8 shows the outline of our design process using ROBDD representation. We first construct the exact ROBDD for a target function $f(x)$. Step-2 is the approximation of ROBDDs using our threshold (T) based approximate-equality. We have elaborated the details of this step in section 6.4.1. If the accuracy of an approximate ROBDD is less than 90%, we reduce the threshold (T) of the approximate-equality and go back to Step-2. This process is repeated until the desired accuracy is achieved, which we have set at 90%. Once the desired accuracy is achieved, we prune A-ROBDD and make it bipartite. Pruning removes extraneous terminal-0 and all the edges connected with it. Such pruning does not cause any loss of functionality/accuracy. The requirement for being bipartite arises because the underlying graph of a crossbar is bipartite, where no two rows (or two columns) are connected with each other. Since a bipartite graph has no odd-length cycles, we make our A-ROBDD bipartite by inserting dummy nodes in odd-length cycles. Fig. 6.7a and 6.7b show the approximate ROBDD and its pruned bipartite version for the MSB of a 4-bit multiplier. Finally, we map the bipartite A-ROBDD onto a crossbar, which is a trivial process. The root node is mapped on the bottom nanowire, nodes which are at even numbered distance from the root are mapped onto horizontal nanowires, and nodes with odd-numbered distance from the root are mapped onto vertical nanowires. Fig. 6.7c shows the approximate crossbar for the MSB of the 4-bit multiplier, where blue and red lines highlight the sneak paths. This approximate design is implemented on a 3×3 crossbar, while the exact design needs a 7×9 crossbar [56]. Our A-ROBDD based approximate design is 85% more compact with 94% accuracy.

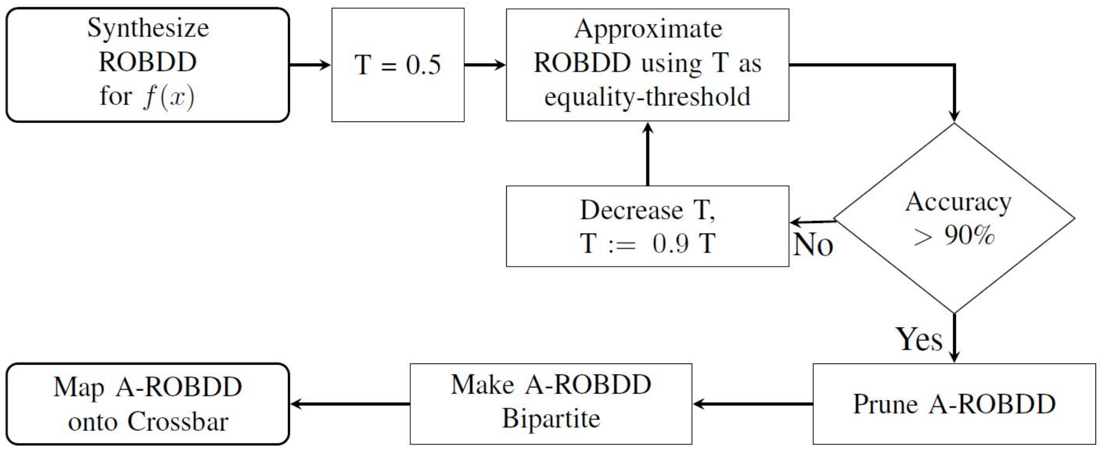

Synthesize ROBDD for $f(x)$
T = 0.5
Approximate ROBDD using T as equality-threshold
Accuracy > 90%
No
Decrease T, T := 0.9 T
Yes
Prune A-ROBDD
Make A-ROBDD Bipartite
Map A-ROBDD onto Crossbar

Figure 6.8: Flow diagram of our Approx-ROBDD based synthesis.

6.4.1 Design Overview

By definition, an ROBDD is already reduced to the point that no two nodes of an ROBDD represent the same function. In this work, we reduce an ROBDD further by merging nodes which are functionally similar under pre-specified constraint (T). To implement this concept, we first compute functional representation (f_i) of each ROBDD node (n_i) from the exact ROBDD graph. Then for each node pair (n_i, n_j); we merge n_i with n_j if their functional representations satisfy our threshold based approximate-equality. This process is repeated until no two nodes can be merged any more. When a node n_i is merged with node n_j, all incoming edges of n_i are directed towards n_j. At the end of the merge operation, all parent-less nodes are removed, which makes the resultant ROBDD more compact than the exact counterpart. We express our threshold based approximate-equality more formally as follows.

Let f_1 and f_2 be functional representations of BDD nodes n_1 and n_2 respectively. Let us assume for now that both f_1 and f_2 have same set of variables $V = \{v_1, v_2, v_3 \ldots v_k\}$. Let $\mathbf{T}_{f_1}$ and $\mathbf{T}_{f_2}$ represent the outputs of truth tables of f_1 and f_2 respectively. Let $\mathbf{H}_{12} = \mathbf{T}_{f_1} \oplus \mathbf{T}_{f_2}$ be XOR of $\mathbf{T}_{f_1}$ and $\mathbf{T}_{f_2}$, then the number of 1s in $\mathbf{H}_{12}$ (hamming distance) is a linear measure of disparity between f_1 and f_2. Fortunately, we can measure this disparity/Hamming distance without building the complete truth table. The hamming-function $f_{H_{12}}$ corresponding to $\mathbf{H}_{12}$ can be computed directly from f_1 and f_2 as follows,

$$f_{H_{12}} = XOR(f_1, f_2) = \neg f_1 f_2 + f_1 \neg f_2$$

we use the simplified form of $f_{H_{12}}$ for computing normalized Hamming distance (h). For approximate synthesis, we treat f_1 and f_2 approximately-equal if their corresponding normalized hamming distance (h) is less than a predefined threshold T.

$$f_1 \approx f_2 \quad s.t. \quad h < T$$

where $0 < T < 1$ is approximation threshold. We can change T to trade overall-accuracy with size of the synthesized crossbars. Despite its simplicity, XOR is extremely effective for measuring disparity between f_1 and f_2. Besides circumventing the need to build truth tables, XOR operation works equally well even when f_1 and f_2 are not composed of the same set of variables. Finally, the linear nature of hamming distance makes it ideal for tuning to accuracy-size trade-off.

6.4.2 Experiments and Evaluations

We have synthesized approximate crossbars for RevLib benchmarks [2], and edge-detectors for multimedia images. We have maintained accuracy above 90% for all circuits. Table 6.4 compares the sizes of our approximate crossbars with ROBDD based exact designs for a variety of functions. In this table, the first seven functions are from RevLib benchmark suite, and the last three are edge detectors with detection-thresholds of 32, 48 and 64. Our approach has successfully reduced the area requirement for RevLib benchmarks by $\approx$ 60%, and edge-detectors by $\approx$ 80%. Previously, Khokhar et al. had used simulated annealing for approximate synthesis of uni-directional edge detector '$(p_1 - p_2) > 50$' [66]. They synthesized this function on a 15×15 crossbar. In comparison, our approach synthesizes the same function on a 7×9 crossbar with $\approx$ 96.42 % accuracy, resulting in area improvement of 72% [66]. The compactness of approximate designs varies depending upon functions, e.g t481 benchmark has lent itself to only 10.6% improvement while '$|p_1 - p_2| > 32$' edge detector is compressed by $\approx$80%. As is clear from the table, approximate computing is most effective for edge-detectors.

We have also simulated our designs on HSPICE. For simulation, our designs use $V_s = 1V$, $R_s = 200\Omega$. Instead of using fixed values of R_{ON} and R_{OFF}, we have modeled them as Normal distributions ($\mathscr{N}$) centered at 50 and 500k Ω respectively, such that $R_{ON} = 50 \times \mathscr{N}(1, \sigma^2)\Omega$, $R_{OFF} = 50 \times 10^4 \times \mathscr{N}(1, \sigma^2)\Omega$, where $\sigma = 0.16$. For our simulations, $\frac{R_{OFF}}{R_{ON}} \approx 10^4$. Fig. 6.9 plots the voltage of the topmost nanowire (output voltage) of 14×15 crossbar for approximating ryy6 benchmark. Here, the X-axis displays the decimal value of the input $[x_{15} : x_0]$, the Y-axis denotes the output voltage. The red and blue bars correspond to High and Low outputs respectively. The minimum voltage for High output '$min(V_{OH})$' is consistently greater than 160 mV and the maximum voltage for Low output '$max(V_{OL})$' is consistently less than $40mV$. The voltage-differential = $min(V_{OH}) - max(V_{OL}) = 120mV$ is three times of $max(V_{OL})$, which is sufficient for error free distinction of Low and High outputs. For display, the outputs in Fig. 6.9 are selected at a sampling rate of 50. To keep the range of output-voltage consistent across different crossbars with different memristive technologies, we recommend using $R_s = \frac{k}{4} R_{ON}$, where k is the number of input bits in the target function $f(x)$.

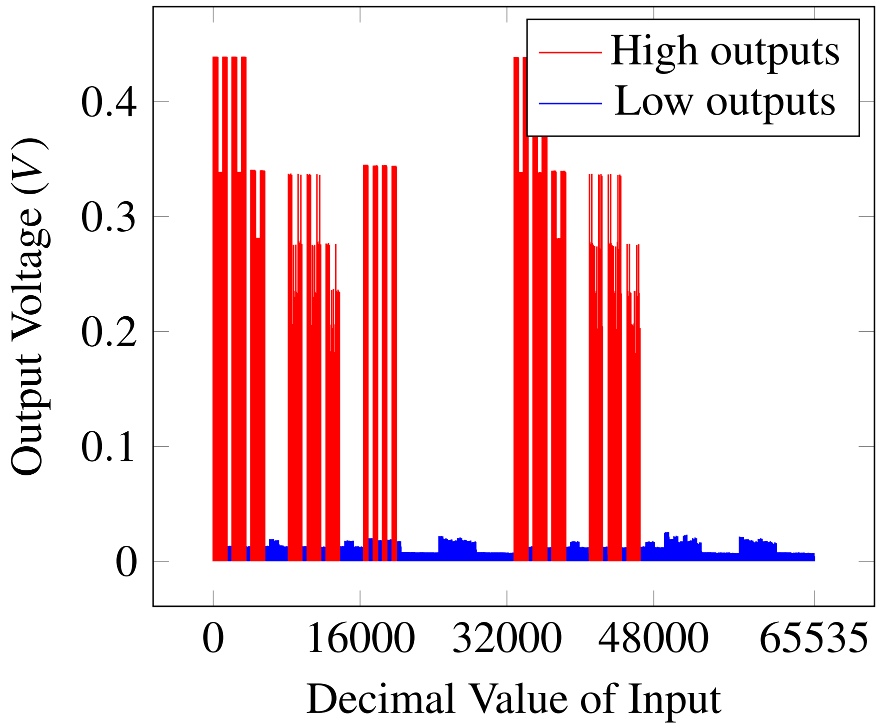


Figure 6.9: HSPICE simulation of 14×15 crossbar for ryy6 benchmark under memristor settings of $R_{ON} = 50 \times \mathcal{N}(1, 0.16)\Omega$ and $R_{OFF} = 500 \times \mathcal{N}(1, 0.16)k\Omega$.

Table 6.4: Comparison of Exact and Approximate Crossbars for different Revlib benchmarks [2].

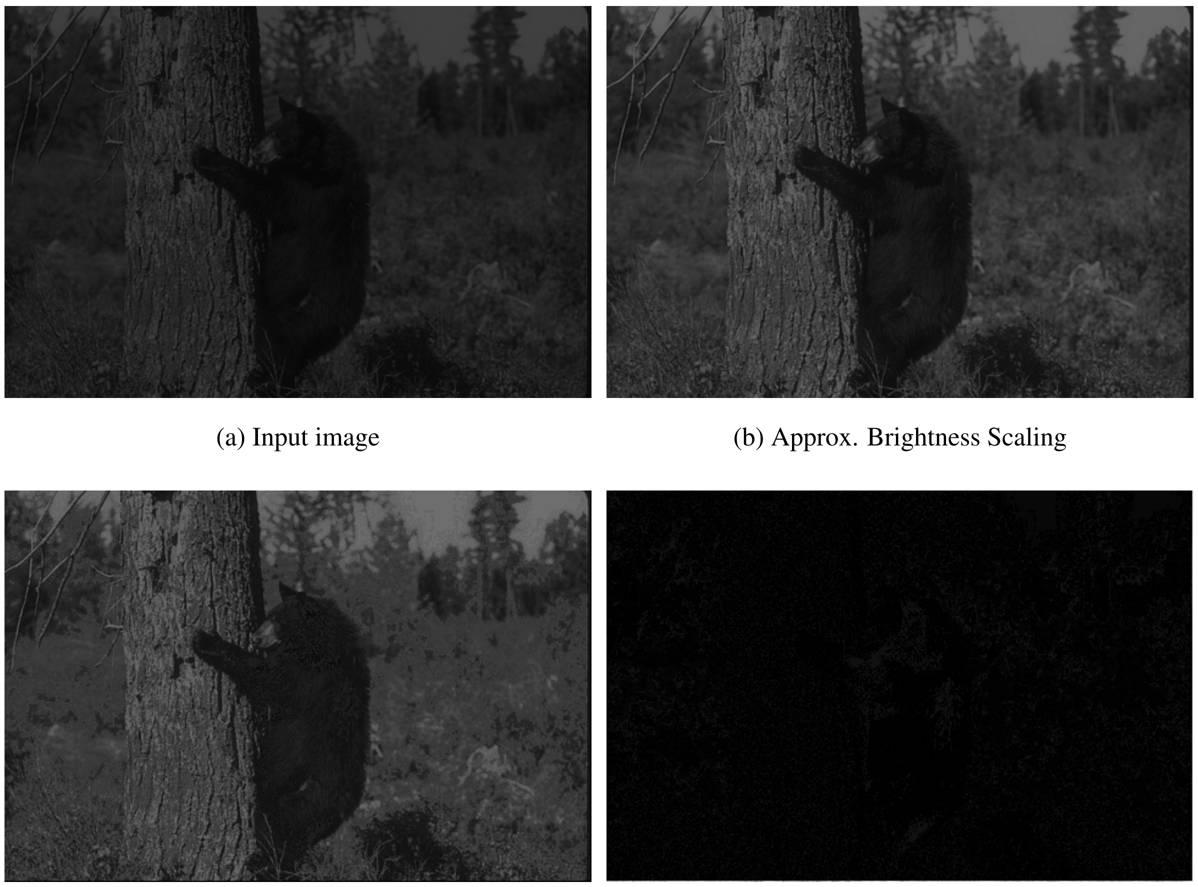
	Exact ROBDDs			
Benchmark	Size	Accuracy	Area Reduction	
newill	14 by 12	9 by 6	95.3%	67.8%
max46	58 by 55	43 by 39	97.1%	47.4%
sym9_71	18 by 17	15 by 13	93.4%	36.2%
sym10_207	20 by 20	16 by 16	95.4%	36.0%
ryy6	25 by 23	14 by 15	94.6%	63.4%
t481	28 by 26	26 by 25	92.7%	10.7%
sym6_63	9 by 9	8 by 7	90.6%	30.8%
$\lvert p1-p2\rvert > 32$	55 by 50	24 by 22	95.7%	80.8%
$\lvert p1-p2\rvert > 48$	52 by 59	26 by 37	97.1%	68.6%
$\lvert p1-p2\rvert > 64$	17 by 29	15 by 19	91.8%	42.2%

6.4.3 Brightness Scaling

We have synthesized approximate crossbars for brightness scaling and tested them on gray-scale images. The same design can be used for scaling the brightness of RGB-images by applying it on each color channel separately. Brightness scaling kernel multiplies this intensity by a constant factor α and checks if it has saturated. Let I be the intensity of a pixel in the i^{th} row and j^{th} column, then the functionality of brightness scaling can be stated as follows,

$$s(\alpha, I) = \begin{cases} \alpha I & \text{if } \alpha I \leq 255 \\ 255 & \text{if } \alpha I > 255 \end{cases}$$

The output of scaling-kernel ‘s(α, I)’ is also an 8-bit pixel. Since our approach synthesizes circuits for Boolean formulae, we first express the scaling kernel in the form of 8-bit Boolean formulae, one formula for each bit. Next we synthesize approximate crossbars for each of them. Area of scaling kernel is the sum of total area of individual circuits. Fig. 6.10 shows the performance of our scaling kernel on the ‘bear’ image. Fig. 6.10b shows the image after brightness scaling with our circuit. For comparison, Fig. 6.10c shows brightness scaling performed by the exact design, while Fig. 6.10d is the difference between the outputs of exact and approximate designs. Our circuit for brightness scaling had area of 1191 while the exact circuit needs 2230. As is clear from Figures 6.10b-6.10d, despite being 46.6% more compact, our approximate design generates a result that has almost imperceptible differences from an exact result.



(a) Input image

(b) Approx. Brightness Scaling

(c) Exact Brightness Scaling

(d) Difference b/w (b) and (c)

Figure 6.10: Performance of our approximate circuit for brightness scaling. (a) is the original image, (b) shows the result of 60% scaling with our approximate kernel, (c) is image after 60% scaling with exact kernel, (d) is the difference between the two results. Our approximate design was 46% more compact than the exact counterpart

6.4.4 Comparison with FBDD based Approximation

We now compare the performance of our FBDD and ROBDD based approximation results. These are given in Table:

Table 6.5: AFBDD vs AROBDD side-by-side comparison

Benchmark	FBDD Size	AFBDD Size	AFBDD Accuracy	Gain	ROBDD Size	AROBDD Size	AROBDD Accuracy	Gain
newill	12 by 12	8 by 9	98%	50%	14 by 12	9 by 6	95.3%	67.8%
max46	68 by 64	44 by 45	88.3%	54.5%	58 by 55	43 by 39	97.1%	47.4%
sym6_63	16 by 19	9 by 11	96.9%	30.7%	9 by 9	8 by 7	90.6%	30.8%
mult6	39 by 38	27 by 26	94.5%	52.56%	27 by 28	19 by 20	91.02%	49.7%
mult7	22 by 24	19 by 22	99.2%	20.83%	17 by 20	13 by 14	91.02%	46.5%
$\lvert p1-p2\rvert > 32$	79 by 81	37 by 39	96.9%	77.4%	55 by 50	24 by 22	95.7%	80.8%
$\lvert p1-p2\rvert > 64$	121 by 123	73 by 73	97.4%	64.2%	17 by 29	15 by 19	91.8%	42.2%
majority_176	8 by 6	6 by 5	96.9%	37.5%	6 by 5	4 by 3	94%	60%

Initial experimentation on FBDD based design synthesis for flow-based RERAM computing showed that FBDDs produced better results than ROBDDs on multiplier circuits. However this result shows that on other benchmarks including image processing benchmarks, ROBDD based synthesis performs better. More study is needed on characterization of benchmarks to understand which benchmarks should have smaller designs using ROBDDs vs FBDDs. The intuitive understanding is that because of the higher representative power of FBDDs due to unrestricted variable ordering between paths, these should give smaller designs.

6.5 Generalized Circuit Synthesis using A-ROBDDs

Our graph based synthesis methods are not specific to RERAM technology. In fact the approximation applies to any Boolean function and therefore any circuit technology. We show this by also generating results on other standard technologies. Our framework for standard cell library based synthesis is as follows: Given a Boolean function f, we first find a global variable ordering using

Dynamic weight heuristic (DWH). Then we specify an initial merge threshold ε (set to an arbitrary high value) and construct an approximate ROBDD using relaxed equality. This A-ROBDD represents the original function f in the approximate sense. If the accuracy of our approximate ROBDD does not meet our required specification, we decrease the Threshold ε in steps (say 10%) and re-synthesize approximate ROBDD under new constraint. This process is repeated until we get an A-ROBDD with the required accuracy. This process is shown in Fig. 6.11. Once an approximate ROBDD is synthesized, we convert it into PLA format and load it into Berkeley's ABC Environment [67]. We use structural hashing to convert our approximate logic to AIG format. And SAT sweep over the AIG to get the functionally reduced form of the AIG that will yield the smallest circuit design. We then synthesize our circuits using two different standard design cell libraries, 'stdcell.lib' and 'mcnc.genlib'.

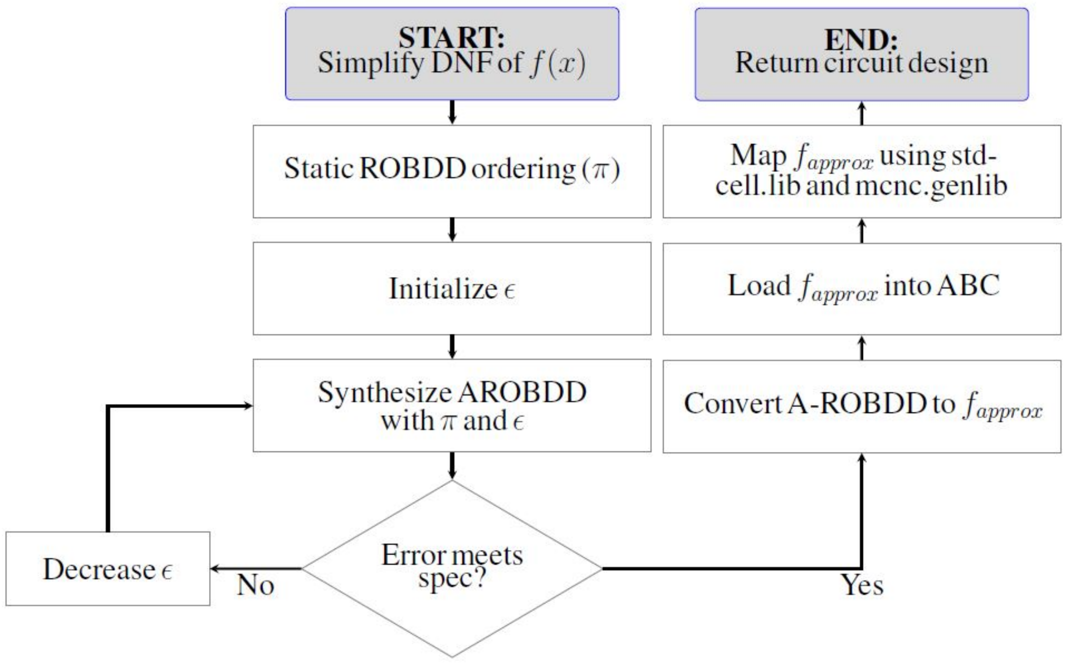
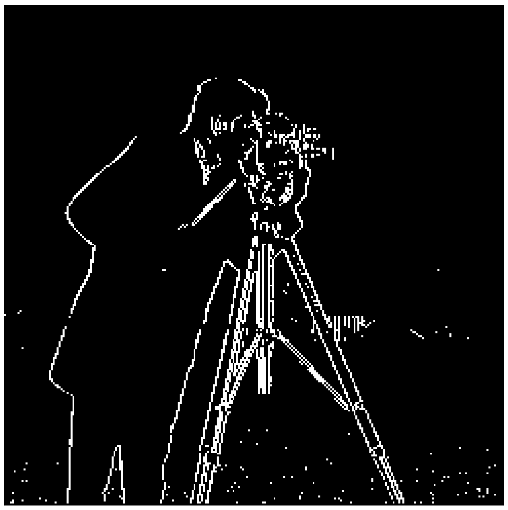
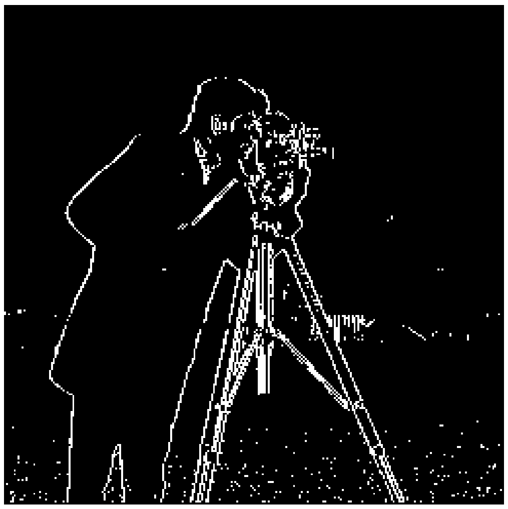

START: Simplify DNF of $f(x)$
Static ROBDD ordering (π)
Initialize ϵ
Synthesize AROBDD with π and ϵ
Error meets spec?
No
Decrease ϵ
Yes
Convert A-ROBDD to f_{approx}
Load f_{approx} into ABC
Map f_{approx} using std-cell.lib and mcnc.genlib
END: Return circuit design

Figure 6.11: Flowchart depicting the steps for synthesizing approximate circuits using ROBDDs.

(a) Gray Scale Image


(b) Exact $|p_1 - p_2| > 48$


(c) Approximate

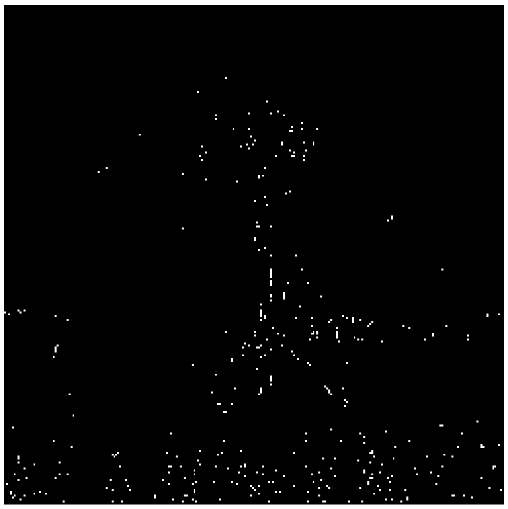

(d) Diff of Exact & Approx.

Figure 6.12: Comparison of our exact and approximate edge detection kernel on cameraman-image (a) is the original image. (b) shows result of approximate edge-detection kernel for $\tau = 48$. (c) is the edge detection for exact edge-detection kernel. (d) is the difference between exact and approximate kernels. (e) is the overall error locations. Our design of approximate edge-detector is 28% more compact as compared to exact designs for only a 4% loss in accuracy

6.5.1 Experiments and Evaluations

We have tested our approach on various elementary image processing kernels and RevLib benchmarks [2, 3]. As shown in tables 6.7 and 6.8, our approach has reduced the size of disjunctive-normal-form (DNF) by upto 90% while corresponding circuits were ≈53% more compact. The DNF-size is defined as the total number of AND-gates and OR-gates in the DNF representation of a Boolean function. It is important to mention that DNF-size is technology independent metric, while circuit area depends upon several factors such as choice of gates, their sizes and mapping technology. We have used stdcell.lib and mcnc.genlib to calculate the area of our circuits. Next we describe how we have used our approach to synthesize compact circuits for edge-detection and brightness-scaling kernels before moving onto general purpose benchmarks. We use the same kernels for edge detection and brightness-scaling as described in previous sections.

The formula for our complete edge response and thresholding kernel is given as:

$$\kappa_\tau(A,B) = \sum_{i=0}^{7} 2^i a_i - \sum_{i=0}^{7} 2^i b_i > \tau \tag{6.6}$$

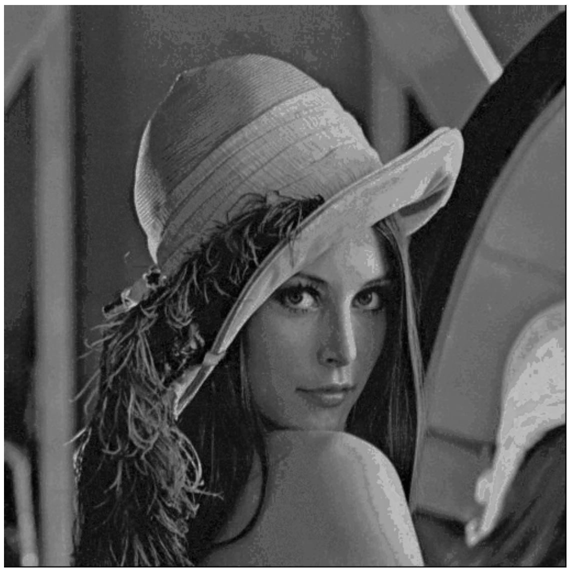
We represent equation 6.6 in disjunctive-normal-form (DNF) and synthesize its approximate circuits using our framework. Fig. 6.12 compares the result of exact edge-detector with approximate edge-detector for $\tau = 48$. Fig. 6.16 shows the regions of false positives and false negatives for this kernel. We have also designed edge detection kernels with $\tau = 16, 32, 48, 64, 80, 96, 112, 128$. Overall, we are able to maintain accuracy between 90 to 98% while reducing DNF sizes by upto 90% and the circuit sizes upto 53%. The complete results are shown in Table 6.7. We show some additional results for edge detection in Figure 6.17.

We also implement an approximate circuit for brightness scaling kernel. In this case, the kernel takes in a pixel and also outputs a pixel (8 Boolean variables) after a multiplication and a saturation operation. The saturation operation is necessary because the output pixel cannot exceed 255 as the answer must be represented in 8 bits hence any values larger than 255 are set to 255. We implement the saturation operation using the ‘min’ operator. The scaling kernel can be expressed as:

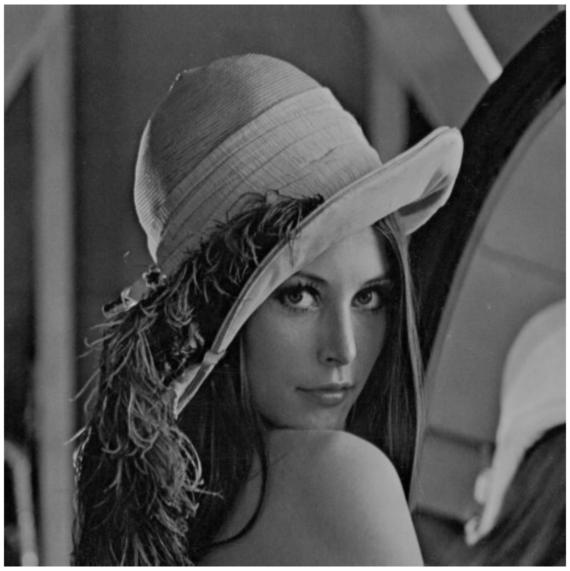
$$U = \min\{\alpha A, 255\} \tag{6.7}$$

Here, A is the input pixel, U is the output pixel and α is the scaling factor. Since our framework needs input functions in Boolean representation, we first represent each of the 8 output-bits in the form of Boolean formulae. In the next step, we create their approximations by synthesizing approximate ROBDDs and convert them into circuit representations using ABC tool. Area of brightness scaling kernel is the cumulative area of individual circuits. Overall, our approach has reduced the area of brightness scaling kernel by 30%. Fig. 6.13 compares the performance of brightness-scaling for $\alpha = 1.6$, using exact and approximate circuits on the standard Lena image. The difference between exact and approximately scaled images is shown in figure 6.13d.

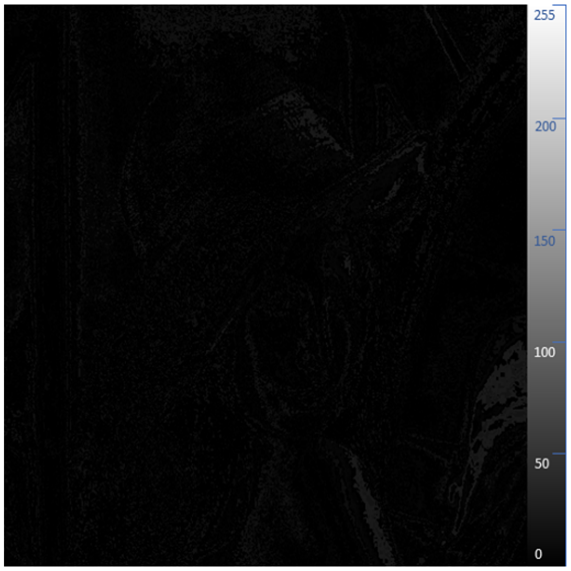
(a) Gray Scale Image



(b) Approx. Brightness Scaling



(c) Exact Brightness Scaling



(d) Diff of Exact & Approx.

Figure 6.13: Performance of approximate design of brightness-scaling circuit on the 'Lena image' (a) is the original greyscale image. (b) is the Lena image after 60% brightness scaling with our approximate circuit (c) is the image after 60% scaling with exact circuit (d) is the difference between images produced by exact and approximate scaling. Our brightness-scaling circuit was synthesized is 30% more compact for a slight loss in accuracy.

Benchmarks. Besides edge detection and brightness scaling, we have tested our approach on other benchmarks as well. The experimental setup was the same as described earlier. We first create approximation of original function by synthesizing approximate ROBDDs, then we use ABC for converting it into functionally reduced AIG which is eventually mapped as circuit using gates from stdcell.lib and mcnc.genlib.

As shown in table 6.8, besides reducing the DNF size by upto 50%, our approach has reduced the circuit area of these benchmarks by upto 40% at the cost of a small drop in accuracy. Among these benchmarks, the ryy6 presents an interesting case. Although, our approach has reduced its DNF size by more than 5 times, it does not translate into area savings for approximate circuit on stdcell.lib or mncn.genlib. Upon deeper analysis, we find that the ryy6 circuit was mapped on higher fan-in gates (such as aoi21, oai21 and nand4 etc), while the approximate circuit relied heavily on 2-input nand2 gates which are generally preferred for technology mapping. When we synthesized ryy6 using nand, nor and inverter gates only, the approximate circuit was 22% more compact than its exact counterpart.

Comparisons. Since our graph-based approximation framework is based on BDD based exact synthesis [68], I include some comparisons for that framework with other exact synthesis methods in this thesis. Table 6.6 shows a comparison of our approach with that of Xie *et al.* [4]. The 'area' in this comparison represents the number of memristors needed in the design. This includes both used and unused memristors. We can see that our approach is generally more area efficient. The computing method of IMPLY [69] uses memristive circuits for computation, however it is not directly applicable for cross-bar structures or in-memory computing. The MAGIC framework [70] can be implemented both as a memristive circuit or on cross-bars, however it presents additional challenges of sneak-paths and multiple micro-operations for each kernel increasing the computing time. For a more detailed discussion on exact synthesis, I refer the reader to the thesis of Hassen [71].

Table 6.6: Comparison of crossbar sizes with Xie *et al.* [4]

Benchmark	FBLC [4]	OFBLC [4]	Our Method
2-bit Adder	432	432	42
4-bit Adder	3948	3948	528
2-bit Mult	208	192	52
4-bit Mult	5280	4384	5260

For Approximation, we compare our method against several existing approximation tools and methods. A number of approximation methods are technology specific and cannot be applied on general circuit synthesis as our method can. We still compare the efficiency gain against them using area reduction and DNF size. In SALSA [42], the authors report an average area improvement of 5.9% for several benchmarks when they are approximated to 95% accuracy. In our experiments we report an average improvements in area of 33.89% and 34.33% for stdcell.lib and mcnc.genlib standard libraries and 64.98% reduction in DNF size for an average accuracy of 94% over our benchmarks. These results are also better than the area gain of 16.1% from [42] for an average accuracy of 75%. In BLASYS, Hashemi *et. al* [37], report an area improvement of 30.25% for 95% accuracy. However, this method only reports on multi-output kernels and synthesize circuits with common parts for various output bits. Hence the final improvement has an advantage as our current synthesis does not yet leverage common circuitry for multi-output kernels. Note that the above methods are general approximation methods, while previous methods were technology specific results for ReRAMs. Those results had a 42.9% improvement in area for a 95% accuracy. Note that the implementation for flow-based computing on ReRAMs results in nearly quadratic growth with respect to function size and therefore a small approximation results in large area gains. Despite this our result for DNF size reduction is comparable with their results.

Table 6.7: Comparison of Exact and Approximate Circuits for Edge Detection Kernels

	Exact Circuits			Approximate Circuits						
	Area using			Area using			Improvement on			
Detection Threshold (τ)	stdcell.lib	mcnc.genlib	DNF size	stdcell.lib	mcnc.genlib	DNF size	stdcell.lib	mcnc.genlib	DNF Size	Accuracy
16	274	191	1011	131	90	285	52.2%	52.9%	71.8%	95.1%
32	305	205	2189	141	100	246	53.8%	51.2%	88.8%	95.7%
48	253	188	2463	183	135	571	27.7%	28.2%	76.8%	96.8%
64	189	140	3323	105	79	271	44.4%	43.5%	91.8%	92.8%
80	207	165	3167	149	108	883	28.0%	34.5%	72.1%	92.2%
96	203	147	2879	143	107	579	29.6%	27.2%	79.9%	93.7%
112	236	161	2639	139	98	595	41.1%	39.1%	77.4%	97.2%
128	171	133	2299	106	82	347	38.0%	38.3%	84.9%	98.1%

Table 6.8: Comparison of Exact and Approximate Circuits for RevLib [2] and MCNC [3] Benchmarks

	Exact Circuits			Approximate Circuits						
	Area using			Area using			Improvement on			
Benchmark	stdcell.lib	mcnc.genlib	DNF size	stdcell.lib	mcnc.genlib	DNF size	stdcell.lib	mcnc.genlib	DNF Size	Accuracy
life_175	222	155	671	171	127	348	22.9%	18.1%	48.1%	90.6%
max46	432	324	394	271	194	254	37.3%	40.1%	35.5%	88.3%
sym10_207	271	191	1259	166	121	769	40%	36%	38.9%	95.4%
sym6_63	74	54	59	34	30	25	54%	44%	57.6%	90.6%
9sym	172	134	503	140	97	479	18.6%	27.6%	4.7%	92.2%
ryy6	33	28	623	37	28	115	-13.12%	0%	81.5%	94.61%

Approximation Analysis. In Figure 6.14 we study the error patterns from our approximate designs for the edge detection kernel. This figure is for the particular edge detection kernel for threshold set to 16. X and Y axes represent one input pixel respectively. In the ideal scenario (exact computation), the result of the kernel should be 0 (or $false$) within a narrow band along the diagonal of the 256×256 map, and 1 ($true$) everywhere outside this band. However in our approximate execution, we get errors on some pairs of pixel values. We indicate false positives with red and false negatives with blue. For an exact design, there would be no colored pixels in this map. We can see that there are a lot of erroneous outputs along the edges of the narrow band (which represent boundaries between $true$ and $false$ regions). This means that a lot of the approximation was done using the lower significance bits. However errors in other parts of the map indicate that our approximation does not simply use precision scaling but also finds other sub-functions within the original kernel that can be substituted. We see a similar pattern in the error map for the edge detection kernel with threshold set to 32 while in the case of edge detection for threshold 48, fewer errors can be found away from the diagonal. This corresponds with a smaller gain in synthesized circuit size for this kernel as shown in Table 6.7 which suggests that simple precision scaling yields smaller gains than when our method is able to find solutions that also use functional approximation.

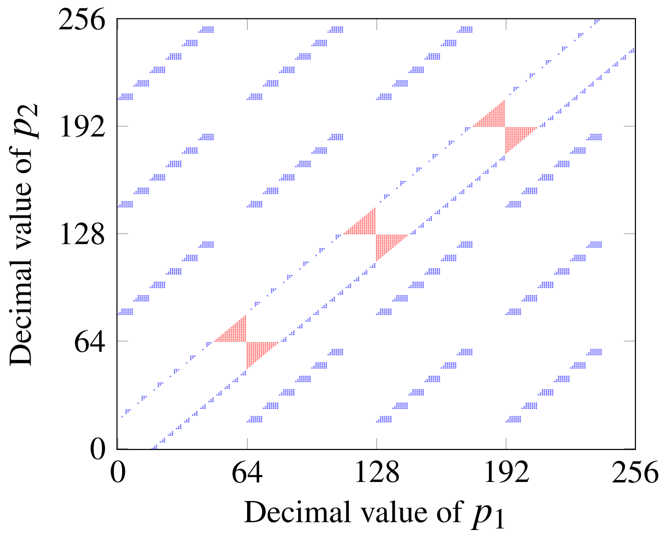


Figure 6.14: Locations of erroneous outputs for the Edge detection kernel '$|p_1 - p_2| > 16$'. X and Y axes represent the first and second pixel. For our approximate design, red squares correspond to $false$ positives and blue squares correspond to $false$ negatives. For an exact design, there should be no colored pixels.

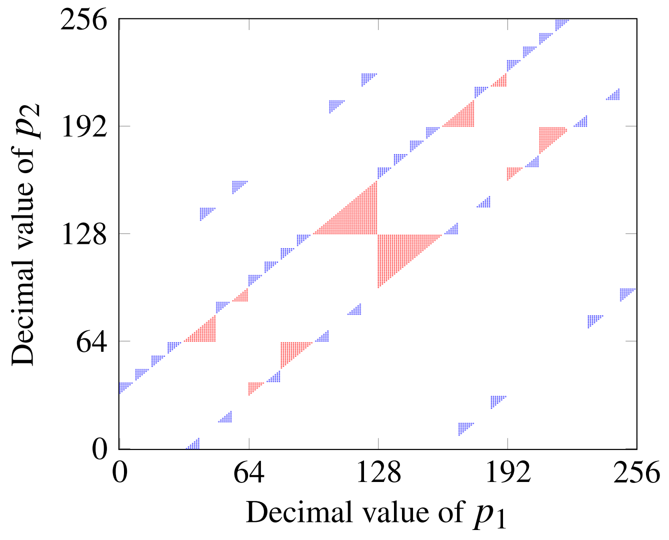


Figure 6.15: Locations of erroneous outputs for the Edge detection kernel '$|p_1 - p_2| > 32$'. The axes and color representations are the same as for Fig. 6.14

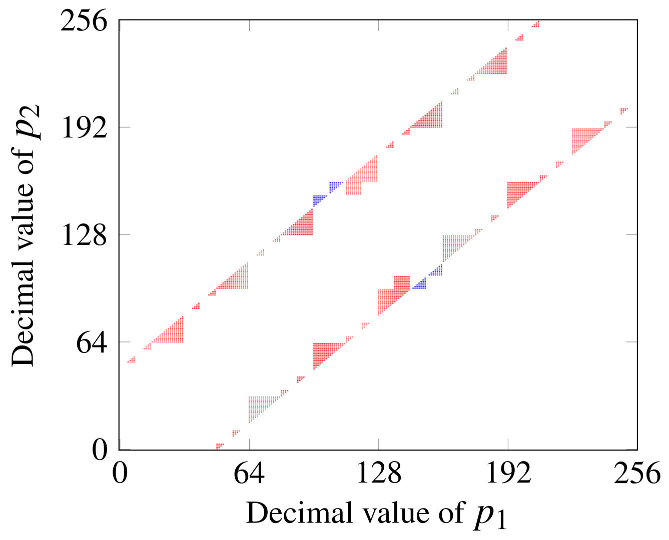


Figure 6.16: Locations of erroneous outputs for the Edge detection kernel '$|p_1 - p_2| > 48$'. The axes and color representations are the same as for Fig. 6.14. Fig. 6.12 shows the performance of this kernel on cameraman image

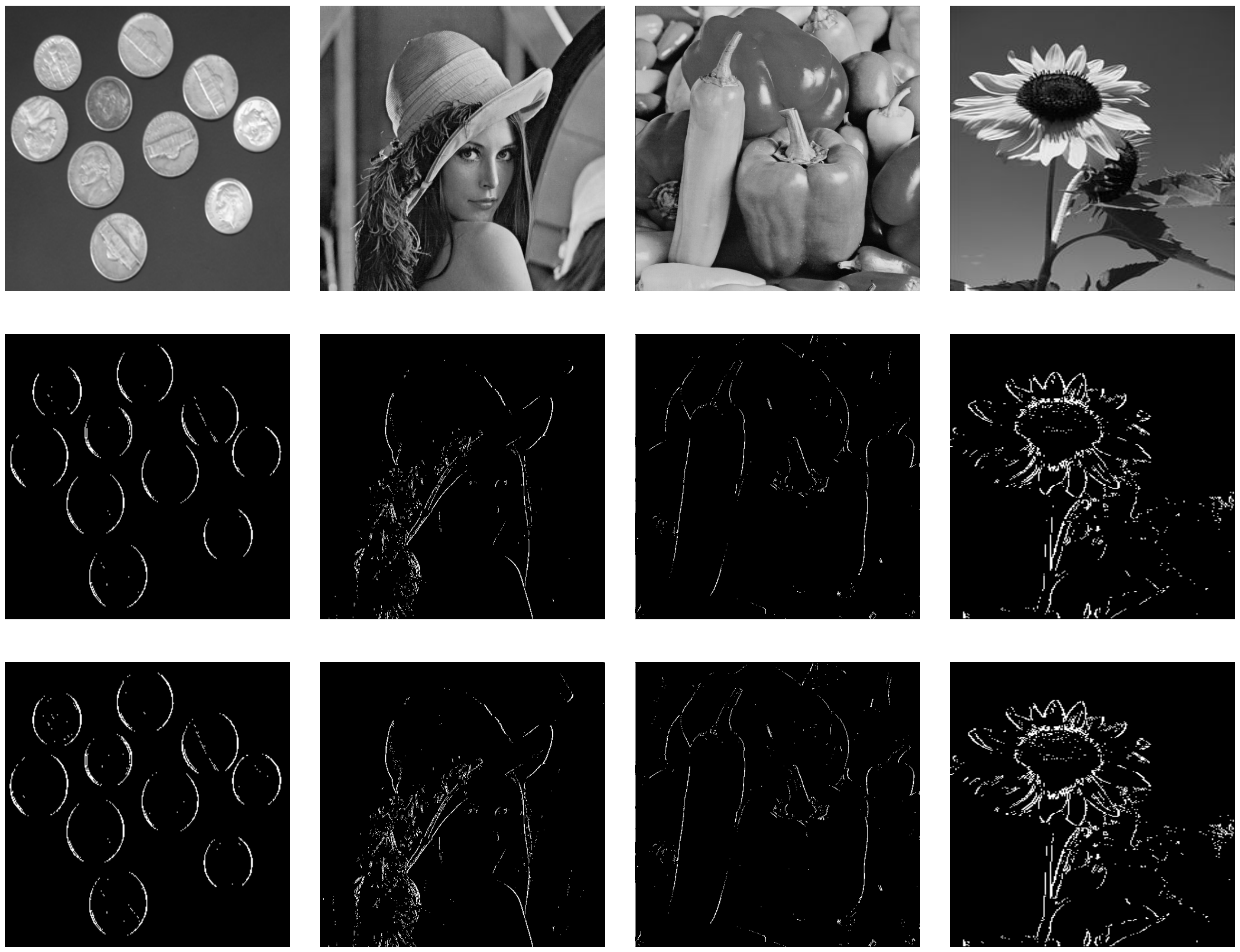


Figure 6.17: Results for edge detection kernel. Top row shows original images, middle rows show exact edge detection results and bottom row shows approximate edge detection results.

CHAPTER 7: CONCLUSION

In this thesis, we have synthesized logic circuits on compact nanoscale ReRAM crossbars for the in-memory computing of Boolean functions. We have proposed and demonstrated two broad approaches for this purpose. A search based framework models the device as a matrix that is optimized over a large search space using constrained simulated annealing. We demonstrate the effectiveness of this method by generating designs for a basic image processing kernel, called edge detection. The design is simulated on SPICE and applied to real images. The target application is surveillance cameras for this work and therefore results are restricted to image processing kernels. Image processing is an example of a modern computing workload that is tolerant to small amounts of error while being highly data intensive. It is therefore an ideal candidate for approximation of computation to obtain better device efficiency.

In our second framework we propose to model Boolean functions as Binary Decision Diagrams. We then present a mathematically sound framework to approximate Boolean functions by manipulating BDDs. We do this on FBDD as well as ROBDD representations which are mapped to RERAMs. We also show that the same approximation can also be mapped to other technologies.

We synthesize circuits that execute both the exact and approximate versions of Boolean functions. To demonstrate the effectiveness of our approaches for approximating logic functions, we also present a case study of the real world application of edge-detection and other image processing kernels such as image scaling and filtering.

7.1 Future Work

In our first framework, we search for a design using a constrained simulated annealing process. We find that the search process is sensitive to parameters such as sparsity, crossbar size and annealing temperature. It is worthwhile exploring other search algorithms for this optimization problem. Additionally, it can be explored how search for two designs may be done in tandem. If two kernels are similar, the design for one kernel may start as the initial design in the search for the second design. Keeping the difference between the two designs can also reduce storage size by getting better compression ratios.

For our second framework, we see that ROBDD based synthesis performs better than FBDD based synthesis for some kernels. The characterization of kernels is an interesting problem. If we can determine characteristics in a kernel that suggest that one of ROBDDs or FBDDs should be used for its design synthesis, it can provide valuable insight for future system design.

One of the most rapidly growing workloads in computer science is deep learning. The basic deep learning kernel is convolution. For both our frameworks, it is possible to extend the results to convolution as a first step towards larger deep learning systems. The challenges in this regard are related to computing power during the synthesis process which may be alleviated by more efficient libraries. Therefore, We plan to extend this work, first towards basic deep learning kernels such as convolution, and then to consolidated deep learning architectures and networks.

REFERENCES

[1] D. Martin, C. Fowlkes, D. Tal, J. Malik, *et al.*, “A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics,” in *Iccv*, 2001.

[2] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, “RevLib: An Online Resource for Reversible Functions and Reversible Circuits,” in *Int’l Symp. on Multi-Valued Logic*, pp. 220–225, 2008. RevLib is available at http://www.revlib.org.

[3] S. Yang, “Logic synthesis and optimization benchmarks user guide version 3.0,” 04 1999.

[4] L. Xie, H. A. Du Nguyen, M. Taouil, S. Hamdioui, and K. Bertels, “Fast boolean logic mapped on memristor crossbar,” in *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, pp. 335–342, IEEE, 2015.

[5] D. Nikonov, “Tunneling fets,” *https://nanohub.org/resources/18351*.

[6] C. Hu, “Finfet and other new transistor technologies,” *Univ. of California*, 2011.

[7] D. E. Nikonov and I. A. Young, “Overview of beyond-cmos devices and a uniform methodology for their benchmarking,” *Proceedings of the IEEE*, vol. 101, pp. 2498–2533, Dec 2013.

[8] S. Kvatinsky, “Memristor-based circuits and architectures,” *PhD thesis, Technion-Israel Institute of Technology, 2014*, 2014.

[9] D. Evans, “The internet of things how the next volution of the internet is changing everything,” *CISCO IoT IBSG white paper.*

[10] P. Sukumaran, “Enable iot asic design using platforms,”

[11] P. Narayanan, J. Kina, P. Panchapakeshan, P. Vijayakumar, K.-S. Shin, M. Rahman, M. Leuchtenburg, I. Koren, C. O. Chui, and C. A. Moritz, “Nanoscale application specific integrated circuits,” in *Proceedings of the 2011 IEEE/ACM International Symposium on Nanoscale Architectures*, pp. 99–106, IEEE Computer Society, 2011.

[12] Z. Alamgir, K. Beckmann, N. Cady, A. Velasquez, and S. K. Jha, “Flow-based computing on nanoscale crossbars: Design and implementation of full adders,” in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1870–1873, IEEE, 2016.

[13] D. Chakraborty and S. K. Jha, “Automated synthesis of compact crossbars for sneak-path based in-memory computing,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 770–775, IEEE, 2017.

[14] M. Lapedus, “Next-gen memory ramping up.” `https://semiengineering.com/next-gen-memory-ramping-up/`, 08 2018.

[15] T. Hickmott, “Low-frequency negative resistance in thin anodic oxide films,” *Journal of Applied Physics*, vol. 33, no. 9, pp. 2669–2682, 1962.

[16] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky, “Imaging: In-memory algorithms for image processing,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4258–4271, 2018.

[17] D. Bhattacharjee, Y. Tavva, A. Easwaran, and A. Chattopadhyay, “Crossbar-constrained technology mapping for reram based in-memory computing,” *arXiv preprint arXiv:1809.08195*, 2018.

[18] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, “Isaac: A convolutional neural network accelerator with in-situ

analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[19] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 27–39, IEEE Press, 2016.

[20] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on circuit theory*, vol. 18, no. 5, pp. 507–519, 1971.

[21] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *nature*, vol. 453, no. 7191, p. 80, 2008.

[22] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for rram-based in-memory computing using majority-inverter graphs," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pp. 948–953, EDA Consortium, 2016.

[23] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (imply) logic: Design principles and methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, 2014.

[24] M. Hu, H. Li, Y. Chen, Q. Wu, G. S. Rose, and R. W. Linderman, "Memristor crossbar-based neuromorphic computing system: A case study," *IEEE transactions on neural networks and learning systems*, vol. 25, no. 10, pp. 1864–1878, 2014.

[25] R. Hasan, T. M. Taha, and C. Yakopcic, "On-chip training of memristor crossbar based multi-layer neural networks," *Microelectronics journal*, vol. 66, pp. 31–40, 2017.

[26] F. Alibart, L. Gao, B. D. Hoskins, and D. B. Strukov, "High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," *Nanotechnology*, vol. 23, no. 7, p. 075201, 2012.

[27] J. Li, F. Peng, F. Yang, and X. Zeng, "A memristor crossbar-based computation scheme with high precision," *arXiv preprint arXiv:1611.03264*, 2016.

[28] W. Kim, A. Chattopadhyay, A. Siemon, E. Linn, R. Waser, and V. Rana, "Multistate memristive tantalum oxide devices for ternary arithmetic," *Scientific reports*, vol. 6, p. 36652, 2016.

[29] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal–oxide rram," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[30] Y. Ho, G. M. Huang, and P. Li, "Dynamical properties and design analysis for nonvolatile memristor memories," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 4, pp. 724–736, 2011.

[31] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'memristive'switches enable 'stateful'logic operations via material implication," *Nature*, vol. 464, no. 7290, p. 873, 2010.

[32] K.-C. Liu, W.-H. Tzeng, K.-M. Chang, Y.-C. Chan, C.-C. Kuo, and C.-W. Cheng, "The resistive switching characteristics of a ti/gd2o3/pt rram device," *Microelectronics Reliability*, vol. 50, no. 5, pp. 670–673, 2010.

[33] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (plim) computer," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 427–432, Ieee, 2016.

[34] M. Soeken, S. Shirinzadeh, P.-E. Gaillardon, L. G. Amarú, R. Drechsler, and G. De Micheli, "An mig-based compiler for programmable logic-in-memory architectures," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, Ieee, 2016.

[35] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, "Revamp: Reram based vliw architecture for in-memory computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 782–787, IEEE, 2017.

[36] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "Abacus: A technique for automated behavioral synthesis of approximate computing circuits," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, March 2014.

[37] S. Hashemi, H. Tann, and S. Reda, "BLASYS: approximate logic synthesis using boolean matrix factorization," *CoRR*, vol. abs/1805.06050, 2018.

[38] J. Schlachter, V. Camus, K. V. Palem, and C. Enz, "Design and applications of approximate circuits by gate-level pruning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, pp. 1694–1702, May 2017.

[39] Z. Vasicek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," *Evolutionary Computation, IEEE Transactions on*, vol. 19, pp. 432–444, 06 2015.

[40] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler, "Bdd minimization for approximate computing," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 474–479, Jan 2016.

[41] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Approximation-aware rewriting of aigs for error tolerant applications," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, Nov 2016.

[42] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, “Salsa: Systematic logic synthesis of approximate circuits,” in *DAC Design Automation Conference 2012*, pp. 796–801, June 2012.

[43] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, “Snnap: Approximate computing on programmable socs via neural acceleration,” in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 603–614, Feb 2015.

[44] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, “Aslan: Synthesis of approximate sequential circuits,” in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, March 2014.

[45] D. Shin and S. K. Gupta, “Approximate logic synthesis for error tolerant applications,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE ’10, (3001 Leuven, Belgium, Belgium), pp. 957–960, European Design and Automation Association, 2010.

[46] S. Rehman, W. El-Harouni, M. Shafique, A. Kumar, and J. Henkel, “Architectural-space exploration of approximate multipliers,” in *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD ’16, (New York, NY, USA), pp. 80:1–80:8, ACM, 2016.

[47] J. Miao, A. Gerstlauer, and M. Orshansky, “Multi-level approximate logic synthesis under general error constraints,” in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 504–510, Nov 2014.

[48] C. Zou, W. Qian, and J. Han, “Dpals: A dynamic programming-based algorithm for two-level approximate logic synthesis,” in *2015 IEEE 11th International Conference on ASIC (ASICON)*, pp. 1–4, Nov 2015.

[49] D. Mehta and V. Raghavan, "Decision tree approximations of boolean functions," *Theoretical Computer Science*, vol. 270, no. 1-2, pp. 609–623, 2002.

[50] R. O'Donnell and K. Wimmer, "Approximation by dnf: examples and counterexamples," in *International Colloquium on Automata, Languages, and Programming*, pp. 195–206, Springer, 2007.

[51] E. Blais and L.-Y. Tan, "Approximating boolean functions with depth-2 circuits," in *2013 IEEE Conference on Computational Complexity (CCC)*, pp. 74–85, IEEE, 2013.

[52] A. Gronemeier, "Approximating boolean functions by obdds," in *International Symposium on Mathematical Foundations of Computer Science*, pp. 251–262, Springer, 2004.

[53] K. Henshall, P. Schachte, H. Søndergaard, and L. Whiting, "An algorithm for affine approximation of binary decision diagrams," 2010.

[54] Y. Cassuto, S. Kvatinsky, and E. Yaakobi, "Information-Theoretic Sneak-Path Mitigation in Memristor Crossbar Arrays," *IEEE Transactions on Information Theory*, vol. 62, pp. 4801–4813, Sept 2016.

[55] S. K. Jha, D. E. Rodriguez, J. E. Van Nostrand, and A. Velasquez, "Computation of boolean formulas using sneak paths in crossbar computing," Apr. 19 2016. US Patent 9,319,047.

[56] A. U. Hassen, "Automated synthesis of compact multiplier circuits for in-memory computing using ROBDDs," in *2017 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pp. 141–146, July 2017.

[57] D. Chakraborty and S. K. Jha, "Automated synthesis of compact crossbars for sneak-path based in-memory computing," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 770–775, IEEE, March 2017.

[58] Z. Alamgir, K. Beckmann, N. Cady, A. Velasquez, and S. K. Jha, “Flow-based Computing on Nanoscale Crossbars: Design and Implementation of Full Adders,” in *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*, pp. 1870–1873, IEEE, 2016.

[59] A. U. Hassen, S. A. Khokhar, H. A. Butt, and S. K. Jha, “Free bdd based cad of compact memristor crossbars for in-memory computing,” in *2018 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pp. 1–7, IEEE, 2018.

[60] A. U. Hassen, S. A. Khokhar, H. A. Butt, and S. K. Jha, “Free bdd based cad of compact memristor crossbars for in-memory computing,” in *2018 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pp. 1–7, IEEE, 2018.

[61] G. Bradski, “The opencv library,” *Dr. Dobb’s Journal of Software Tools*.

[62] D. Chakraborty, S. Raj, J. C. Gutierrez, T. Thomas, and S. K. Jha, “In-memory execution of compute kernels using flow-based memristive crossbar computing,” in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–6, IEEE, 2017.

[63] S. B. Akers, “Binary Decision Diagrams,” *IEEE Transactions on Computers*, vol. C-27, pp. 509–516, June 1978.

[64] R. E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation,” *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, Aug 1986.

[65] A. Bricalli, E. Ambrosi, M. Laudato, M. Maestro, R. Rodriguez, and D. Ielmini, “Resistive Switching Device Technology Based on Silicon Oxide for Improved ON-OFF Ratio–Part II: Select Devices,” *IEEE Transactions on Electron Devices*, vol. 65, pp. 122–128, Jan 2018.

[66] S. Khokhar and A. Khalid, “nanoscale memristive crossbar circuits for approximate edge detection in smart cameras,” in *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*.

[67] A. Mishchenko, "https://people.eecs.berkeley.edu/ alanmi/abc/," in *ABC: A System for Sequential Synthesis and Verification*.

[68] A. U. Hassen, S. A. Khokhar, H. A. Butt, and S. K. Jha, "Free bdd based cad of compact memristor crossbars for in-memory computing," in *2018 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pp. 1–7, IEEE, 2018.

[69] S. Kvatinsky, G. Satat, N. Wald, E. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, pp. 2054–2066, Oct 2014.

[70] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, pp. 635–650, July 2016.

[71] A. Ul Hassen, "Automated synthesis of unconventional computing systems," 2019.