APPROXIMATE BINARY DECISION DIAGRAMS FOR HIGH-PERFORMANCE COMPUTING

by

ANAGHA SIVAKUMAR
B.Tech. University of Kerala, 2016

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

Spring Term
2018

Major Professor: Sumit Kumar Jha

ABSTRACT

Many soft applications such as machine learning and probabilistic computational modeling can benefit from approximate but high-performance implementations. In this thesis, we study how Binary decision diagrams (BDDs) can be used to synthesize approximate high-performance implementations from high-level specifications such as program kernels written in a C-like language. We demonstrate the potential of our approach by designing nanoscale crossbars from such approximate Boolean decision diagrams. Our work may be useful in designing massively-parallel approximate crossbar computing systems for application-specific domains such as probabilistic computational modeling.

ACKNOWLEDGMENTS

I would like to acknowledge my advisor, Dr. Sumit Kumar Jha, for his continuous encouragement and involvement throughout the duration of my thesis. A special note of gratitude is noted towards Dr. Gary Leavens and Dr. Sharma Thankachan, for graciously offering their time, and agreeing to be a part of my thesis committee.

For technical contributions, I would also like to thank fellow researchers, Mr. Dwaipayan Chakraborty, Mr. Sunny Raj, and Mr. Salman Khokhar. Their valued input and help through their work and publications in the field have played a major role in the inspiration behind my thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Over the last few decades, the computing architecture was governed by three basic concepts. In 1945, a computer architecture based on the Von Neumann model, known as the Von Neumann model and Princeton architecture, or simply as the Von Neumann architecture, was designed. Named after the mathematician and physicist, Jon Von Neumann, this design involves an input unit, an output unit, a central processing unit, and a memory unit.

According to the Von Neumann model, data and instructions are stored separately in two different units; while a single bus is used to access both instructions an data. So, operations like an instruction fetch and an arithmetic operation cannot be executed simultaneously because they require the same bus. Thus, machines with this architecture run in a sequential and linear order since they implement stored-program concept.

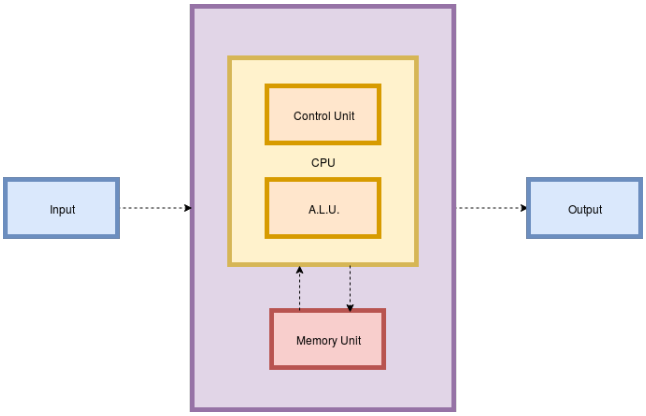


Figure 1.1: Von Neumann architecture

In 1965, Gordon Moore, the co-founder of Intel, observed the pace at which the number of transistors increased on a chip of a particular area. He stated that the number of transistors on a chip

doubled every year, whose pace, he then later changed to doubling every two years in 1975.

Similarly, in 1974, Robert H. Dennard postulated that the performance per watt in computing is also increasing at an exponential rate similar to that of Moore's law. This meant that even when the number of transistors on a chip doubled, the power density would stay constant and thus draw higher performance over the years.

As performance and cost are considered as two of the most important factors in the commercial aspect of technological developments, Moore's law and Dennard scaling proved to enhance the computing world into a more inexpensive and powerful one. This trend was seen to continue until in 2005, major computer processor companies like Intel and AMD began focusing more on increasing the number of CPU cores and on enhancing singe-threaded CPU performance.

The following years displayed a gradual move away from both Moore's law and Dennard's law. This has led to an an escalated need to explore further in the study and research of emerging devices like memristors.

The present era of technology has seen a gradual move towards computing methodologies that go beyond Von-Neumann architectures. Memristors are electrical components, which have the unique property of being able to pass electric current through them, as well as, a non-volatile memory. In this way, memristors can act as both a memory element and resistor.

Binary Decision Diagrams or BDDs are directed acyclic graphs (DAGs) that have the ability to represent boolean functions. These data structures were developed from the idea of Shannon expansion, which is splitting up a function into two sub-functions such that it follows an if-then-else condition, by C.Y. Lee. Further, the concept of binary decision diagrams were established and extensively utilized in the field of efficient algorithms by Randal E. Bryant[2]. If the manner in which the data is stored onto the crossbar is accurate, then only if the given Boolean formula eval-

uates to true based on the input loaded into the crossbar, would there be a current flow from one nanoscale wire to another. Deriving a smaller BDD can result in a smaller memristor crossbar network, which may result in low power consumption while the power density stays the same. This calls for the need to approximate binary decision diagrams, which in turn, leads to optimized crossbar networks, as well.

One method of incorporating this optimization is using a search algorithm, known as siulated annealing. Simulated annealing (SA) is a probabilistic approach of approximating the global optimum of a given function. This approach is particularly suitable when the search state space is large and discrete, and the cost of computation is valued more than the precision of the result. That is, a global optimum value is preferred over a local optimum value. The methodology behind it, can be depicted simply as follows:

1: Set initial temperature variable, T to 1.
2: Generate a random initial solution.
3: Loop until T reaches minimum value.
4: Find a neighbor solution.
5: Compare the cost of the neighbor solution with the cost of the initial solution.
6. Calculate the acceptance probability, and decide whether or not to move to the neighbor solution.
6: Decrease temp variable, T by 0.9999.
7: Continue looping.

For single-instruction multiple-data (SIMD) parallelism, flow-based computing using memristor crossbar networks form a perfect fit. We can write a compute kernel in a C-like language and then transform this kernel into a memristor crossbar, and thus, reducing the need to learn another programming model. Many applications in probabilistic computational modeling, computer vision, and machine learning can benefit greatly from this approximated computing technique.

This thesis combines many far-reaching approaches, which have a large contribution in the computer architecture society. It combines concepts such as approximation using search algorithms and flow-based computing through memristor crossbar networks to provide the following contributions:

1. We present a methodology for mapping C kernels to memristor crossbar networks,particularly focusing on a simple polynomial computation program.

2. We present a simulated annealing approach for generating approximate memristor crossbars for implementing Boolean computations.

CHAPTER 2: LITERATURE REVIEW

Vital aspects in the field of computer architecture technology, that have massively been adopted in my research, are nanoscale memristor crossbar networks and approximation computing. The field of nanoscale memristors is crucial in the minimization of the size and maximization of the power density of circuits. The extent to which the technology of memristor crossbar networks can be utilized is still vast, and an active field to be researched upon. The maximum utilization of this fourth fundamental passive electrical component may only be reached by the 2020s. According to the Allied Market Research, the value of the memristor market is reportedly intended to increase from 3.2 to a whopping 79 million dollar market by 2022, over the span of seven years.

One of the most far-reaching aspects of memristor crossbar networks is its inadvertent ability to nullify the Von Neumann bottleneck in computer architecture. The Von Neumann bottleneck is the hindrance in throughput, that is produced in stored-program computers.

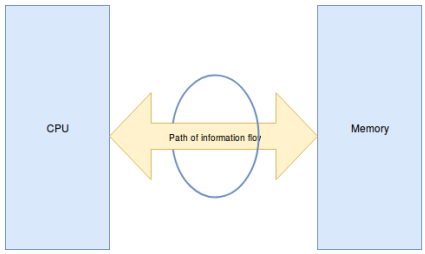


Figure 2.1: Von Neumann Bottleneck

Over the last few decades, we have seen modifications to the processor, as well as memory, to increase their independent efficiencies. Particularly, the increase in processor speed and the increase in density of memory have created an even larger gap in the data transfer between the two components. To overcome this impediment in computer architecture, new technologies or extensive

research in existing tools have surfaced. Different solutions including caching, prefetching, multi-threading, and other methods were utilized, out of which in-memory computing yielded extensive applications in different fields.

Memristors have the unique ability to behave like a resistor and change state, depending on the current that flows through it, as well as, like a non-volatile memory by storing the value of the charge it had flowed through it previously. This makes it an electrical component that is capable of performing in-memory computation.

A memristor crossbar is in the form of a network of memristors where these memristors are located at the intersections of the rows and columns of the network, as shown in Fig.2.1. If a memristor is switched ON, the memristance will be of low memristance, that is, it lets current pass though, if the magintude of the current is high enough. If a memristor is switched OFF, it implies that the memristance is high, meaning the blockage of flow of current through it. Thus, current passes through a desired path through the crossbar by turning the corresponding memristors ON.

$$\mathbb{M} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

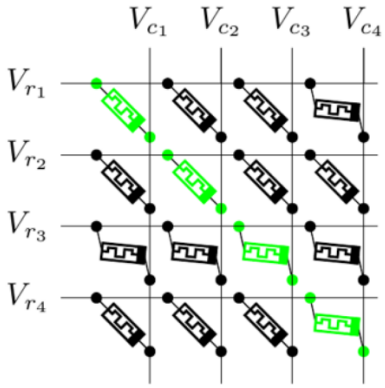


Figure 2.2: Matrix and equivalent circuit representation of a memristor crossbar network

However, a disadvantage of memristors is the presence of sneakpaths in the circuit. A sneakpath is an unintended path for current to pass through an ON memristor (of low resistance), which runs parallel to a specified path that passes through the required OFF memory element(of high resistance). Recent publications have shown an increased interest in the sneakpath constraints in memristors [12]. In the study of nullifying the Von Neumann bottleneck using memristive crossbar networks, Dr.Sumit Jha and Dr.Alvaro Velasquez have described the working and advantages of using the sneakpaths in memristors to their advantage, instead of as an impediment, for Boolean computations[9].

Flow-based crossbar computing enables information to be stacked onto a memristor crossbar in a well-outlined structural pattern with the aim that the flow of current through the crossbar can be utilized to perform high-level programs[3]. Memristive crossbar networks have been used to implement high-level image processing, such as edge detection, through flow-based computing[13].

In the pathway of attempting to keep up with the pace of Moore's law, increasing number of devices are trying to be accommodated on the same chip. Look-up Table (LUT) based Field Programmable Gate Arrays (FPGAs) is a domain which benefits greatly from adopting methods that place accuracy second to power consumption. Approximate computing enhances applications that can withstand minor decrease in accuracy for increased performance. LUT-based FPGA is one such field that has applications that would benefit greatly from greater power density at the cost of precision of results. This circuit optimization is done using BDD approximation, that is, generating a binary decision diagram, which is smaller in size than the original BDD, but at a low input hamming distance. Thus, the original BDD and approximated BDD only differ from each other in terms of a few input assignments and size. The accuracy is measured by checking the number of variable assignments that satisfy the XOR of the original BDD and approximate BDD[17].

A search-based optimization algorithm is generally used in BDD approximation, such as genetic programming,simulated annealing, etc. In the work done on LUT-based FPGAs, the search algorithm used is Cartesian Genetic Programming (CGP). However, the comparison of genetic programming and simulated annealing, based on their application in the traveling salesman problem, had shown interesting results. Genetic programming was seen to show better solution accuracy, while simulated annealing displayed better runtime and speed[19].

CHAPTER 3: METHODOLOGY

To analyze the aftermath of approximating binary decision diagrams on performance factors including precision, time, and space, we have taken a polynomial computation program, written in C++, as the input. The polynomial function used is f(x) = 5 * x +11.

The research approach involves converting this high-level C++ code to LLVM intermediate representation(IR) using the LLVM compilation framework. The resultant LLVM IR is then mapped onto a Boolean vector(an array of BDD elements) using the BuDDy package. For a more comprehensive graphical representation of the resultant BDD, we use the dot tool of graphviz package to display the BDD structure in graphical form.

Python contains many packages, such as, networkx and numpy, that make it a suitable language to perform BDD manipulation and approximation. We map the resultant BDD to a directed graph in Python using the networkx package. Using simulated annealing as the optimization algorithm, we can approximate the resultant BDD to a BDD of a smaller size, while preserving an appropriate accuracy threshold value.

3.1 Compute C kernel

The language used to write the code kernel may be C or C++. The code kernel implemented takes variable and integer inputs, and performs polynomial computation on them (See Appendix A).

A simple depiction of the structure and flow of our code kernel, that computes a polynomial,y=ax+b where a and b are integer constants is shown in Fig 3.1.

```
struct input
{
        int a, int x, int b;
} input I;

struct output
{
        int y;
} output O;

void polynomial_compute_kernel()
{
        O.y = I.a * I.x + T.b;
}
```

Figure 3.1: Pseudocode of polynomial computation

3.2 Convert C kernel to LLVM IR

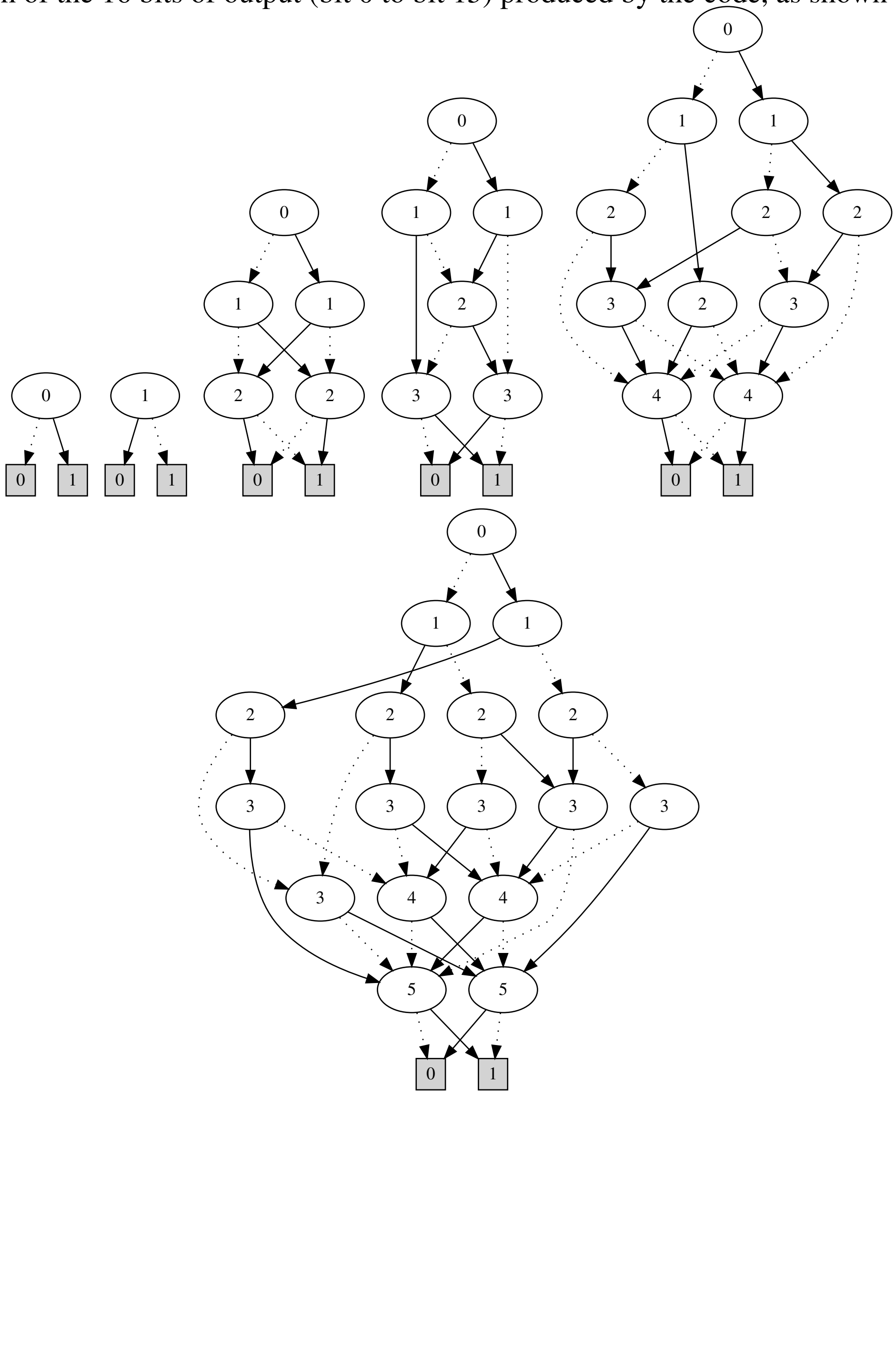
The high level program is first converted to its corresponding LLVM intermediate representation by using the LLVM compilation framework. Thus, an assembly version of the high level code is obtained, which is easier to map onto a BDD. The LLVM IR is intermediate to high level and more advanced than machine-level language, thus, making it easier to be comprehended or mapped to both levels of languages (See Appendix B).

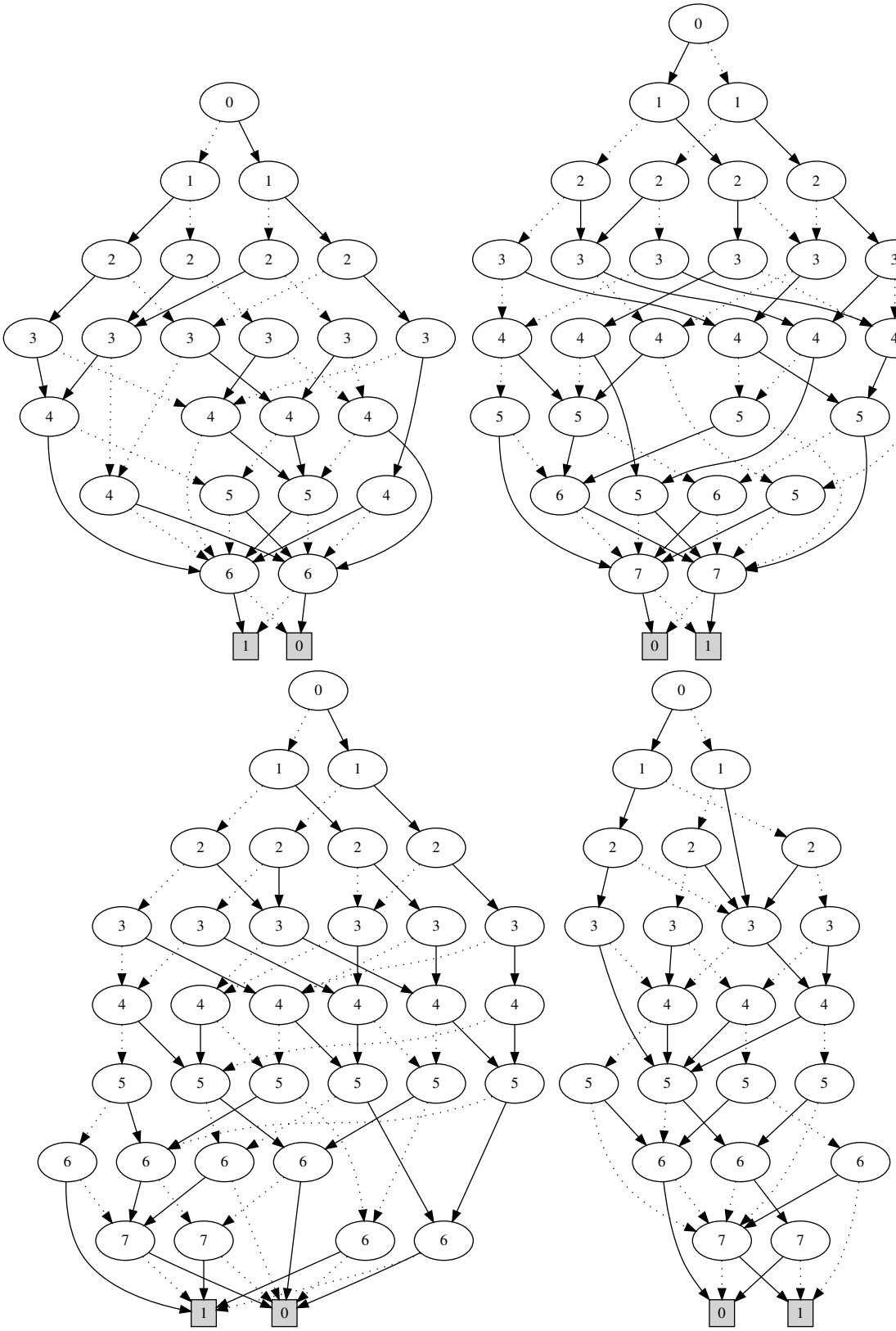
3.3 Map LLVM IR to BDD

Next, the LLVM IR is transformed to a boolean vector using the BuDDy package, which is an open source BDD package (See Appendix C).

BuDDy has many internal classes and methods that make it easy to convert the code to a BDD or

an array of BDDs. Using the dot tool of the graphviz package, we can plot the binary decision diagrams of each of the 16 bits of output (bit 0 to bit 15) produced by the code, as shown below.




0
1 1
2 2 2 2
3 3 3 3 3 3
4 4 4 4 4 4
5 5 5 5
6 5 6 5
7 7
0 1
0
1 1
2 2 2 2
3 3 3 3 3 3
4 4 4 4 4 4
5 5 5 5
6 5 6 5
7 7
0 1
0
1 1
2 2 2 2
3 3 3 3 3 3
4 4 4 4 4 4
5 5 5 5 5 5
6 6 6 6
7 7 6 6
1 0
0
1 1
2 2 2
3 3 3 3
4 4 4
5 5 5 5
6 6 6
7 7
0 1

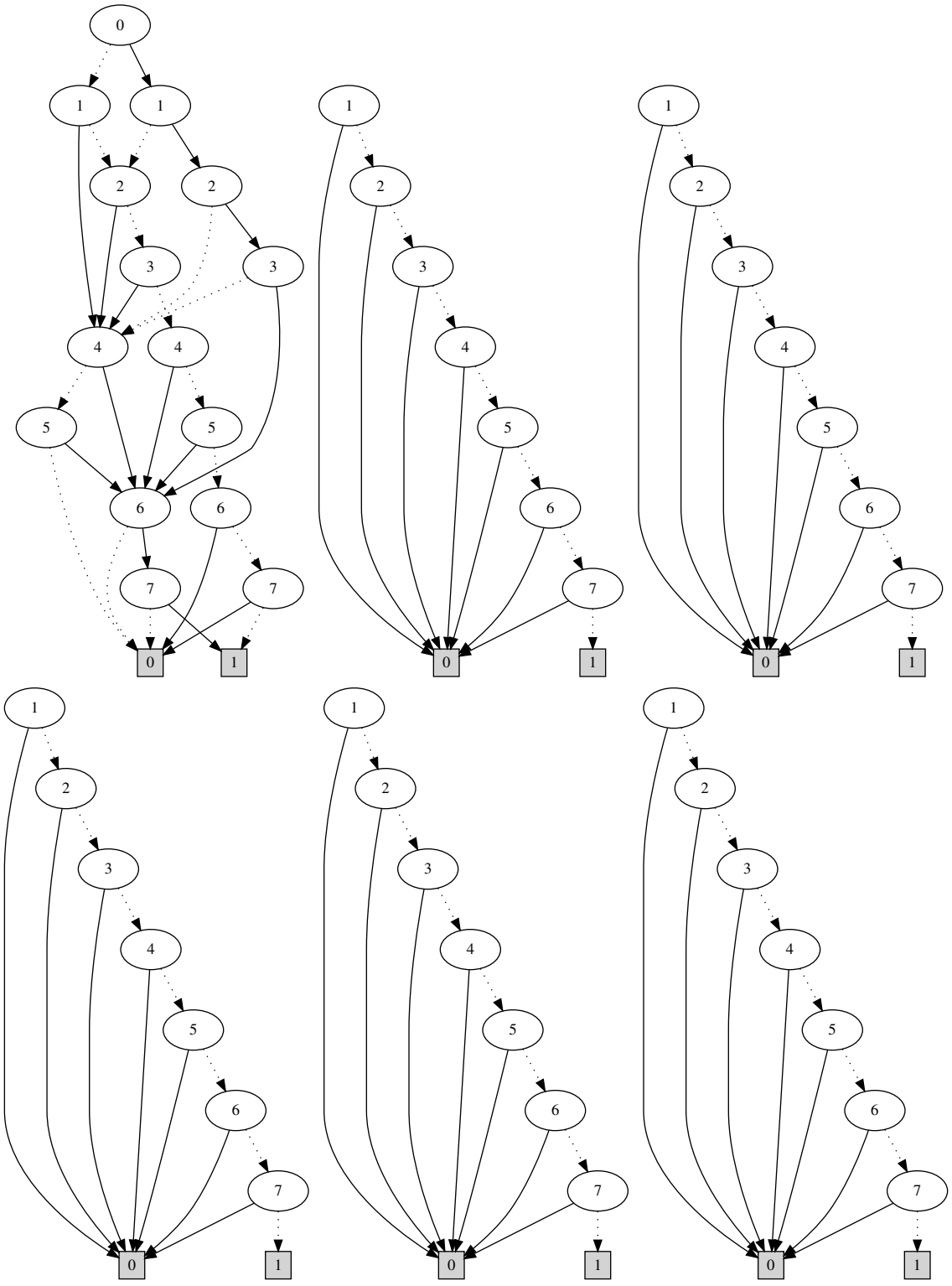


Figure 3.2: BDDs of 16-bit output (bit0 to bit15)

On careful observation of the BDDs, we can see that the complexity of the graph increases towards the middle bits of the output. For our approach, we only consider the biggest BDD, bit 8 for approximation and check its effects on accuracy and space consumed.

3.4 Map the BDD to a Crossbar

The output from the BuDDy package, which we are utilizing, is the node table of each bit of the output in text format. For the successful mapping from BDD to crossbar, we converted the BDD to graphical representation, while the crossbar will be in matrix representation.

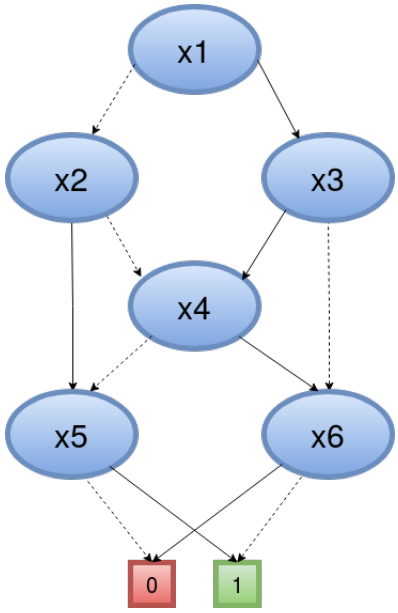


Figure 3.3: BDD of bit 3 of output

The networkx and numpy packages in Python make it a suitable programming language to be used. Hence, we can convert the BDD to a digraph, and then subsequently map it to a memristor crossbar matrix (See Appendix D).

x1	0	1	0	0	0	0
-x1	1	0	0	0	0	0
0	x2	0	0	1	0	0
0	-x2	0	1	0	0	0
0	0	x3	1	0	0	0
0	0	-x3	0	0	1	0
0	0	0	x4	0	1	0
0	0	0	-x4	1	0	0
0	0	0	0	x5	0	1
0	0	0	0	0	-x6	1
0	0	0	0	0	0	1

Figure 3.4: Corresponding matrix of bit 3 of output

The resultant matrix transformation of the digraph, when mapped on to a meristor crossbar network on a circuit, would yield a network in the form as given in Fig 3.5.

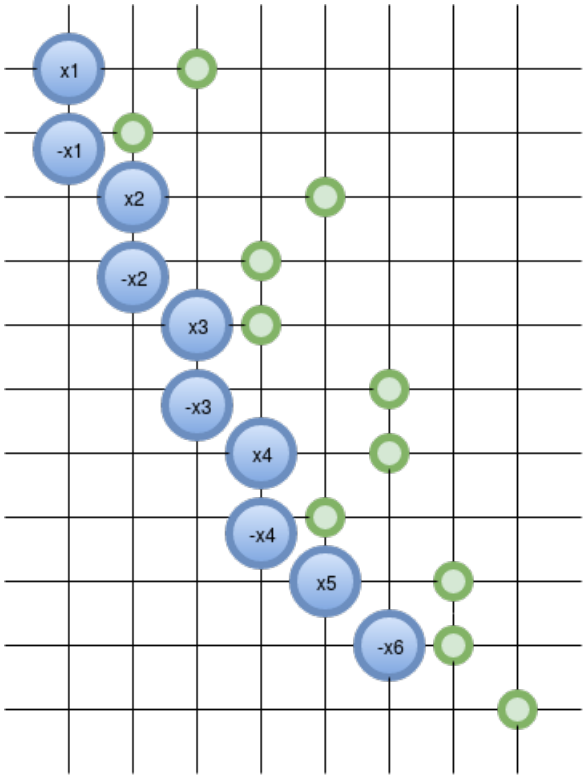


Figure 3.5: Corresponding memristor crossbar network of bit 3 of output

3.5 BDD Approximation

The third step is to approximate this resultant crossbar into a smaller matrix by randomly perturbing the original BDD using an approximate optimization algorithm. Here, we use simulated annealing to implement this.

In simulated annealing, we compare a randomly chosen initial solution to neighbor solutions, that are variations of the original initial solution. If the cost of the neighbor solution is less than our original solution, then we adopt this neighbor solution as our original solution; otherwise, our original solution stays the same. Subsequently, more neighbour solutions are generated and checked

with the original initial solution, until an optimum accuracy level and appropriately small crossbar size is obtained. This approximate BDD may be slightly less accurate compared to the original BDD, but computes less space and time (See Appendix D).

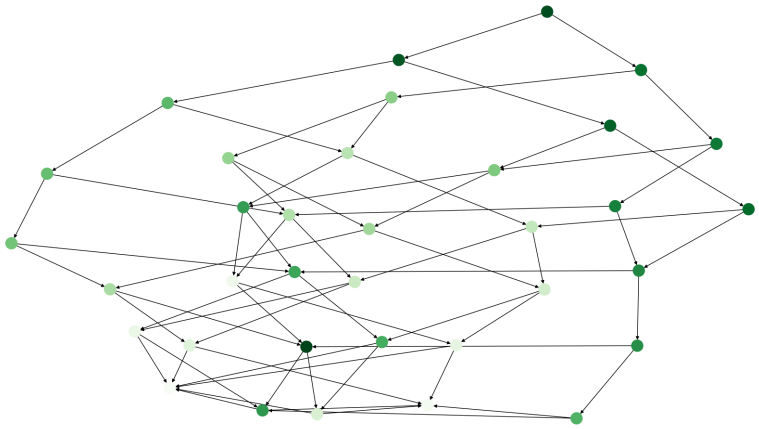


Figure 3.6: Unaltered 8th bit BDD

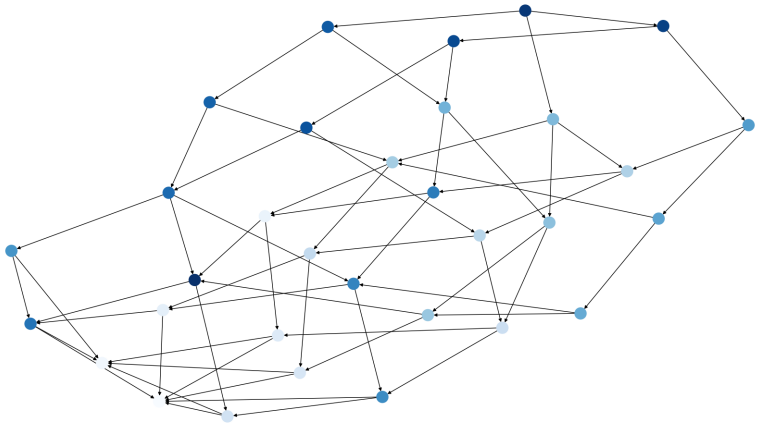


Figure 3.7: Approximated 8th bit BDD

The correctness of the approximate crossbar design is tested by code, by passing the crossbar design generated against a theoretical output value. The cumulative errors is calculated for a large number of inputs and checked.

CHAPTER 4: RESULTS AND CONCLUSIONS

On conversion of the polynomial computation program to a boolean vector using the BuDDy package in C++, we obtain BDDs of the corresponding 16-bit output. interestingly, the BDDS increase in size towards the middle of the 16 bits.

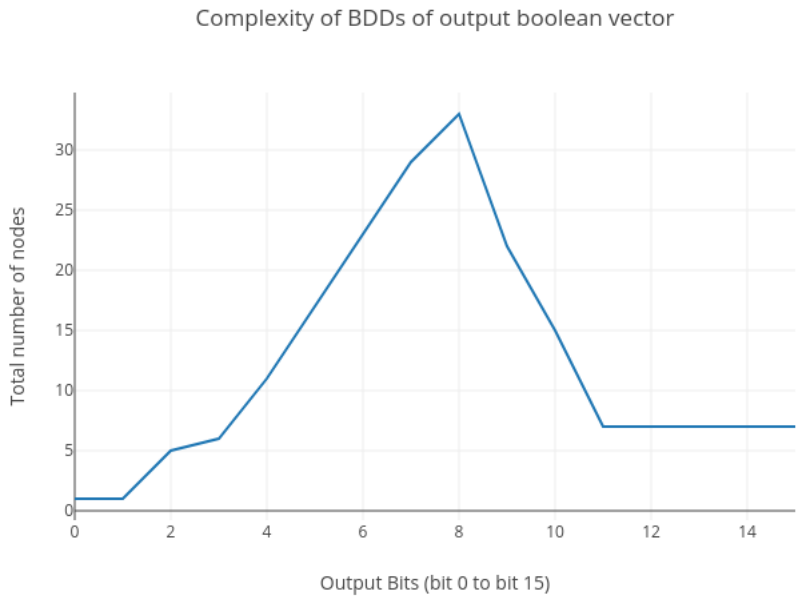


Table 4.1: Tabular representation of complexity of BDDs of output vector

Bit no:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Total no: of nodes:	1	1	5	6	11	17	23	29	33	22	15	7	7	7	7	7

The most complex BDD is chosen out of the 16 BDDs, which is the middlemost eighth BDD of bit 8 of the output, and approximated. This is subsequently tested for correctness and feasibility.

For approximation, we eliminated three nodes at a time; while subsequent neighbor solutions differed from each other by one node, for local optimization.

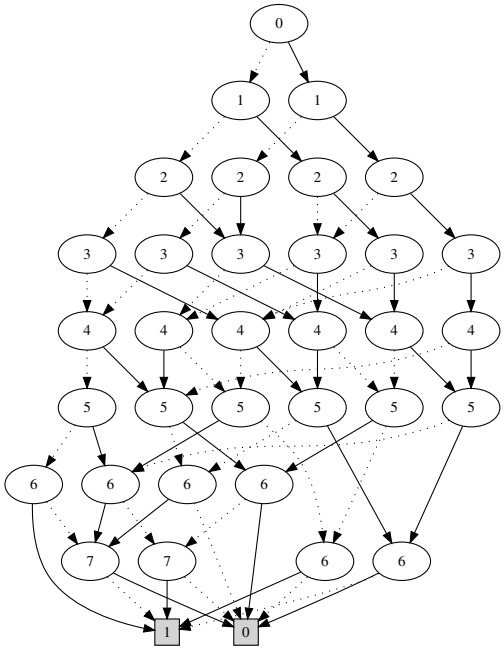


Figure 4.1: Unaltered BDD of bit 8 of output

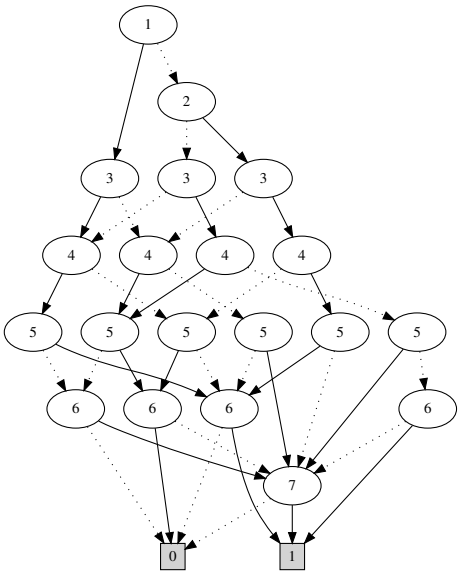


Figure 4.2: The approximated BDD of bit 8 of output

Table 4.2: Performance evaluation of Approximate BDD vs. unaltered BDD

	Unaltered BDD	Approximated BDD
Crossbar matrix size	61 x 34	57 x 31
Total number of nodes	33	32
Number of paths to true terminal	79	25

After BDD approximation, the corresponding crossbar design was generated and tested. Observations to be noted were:

1. The reachability of nodes can be assumed to be affected. The number of paths that return to the true terminal of both unaltered and approximated BDDs were computed. The approximate BDD was seen to have a third of the paths that lead to the true terminal compared ot the unaltered one.

2. By the elimination of even one node from the original BDD, the size of the memristor crossbars were reduced substantially.

3. There is a need to introduce an accuracy threshold value, to keep performance in check.

The experimental observations show that by implementing an approximate BDD, and thus, in turn, a smaller memristor crossbar network, we can implement high level programs with substantially smaller number of electrical components.

Programs that have high error resilience can be found in fields like image processing, polynomial computations, and ordered differential equations. Utilization of approximation for optimization in such applications are far-reaching.

In my thesis, after BDD approximation, the corresponding crossbar design was generated and tested. The approximate crossbar generated by our code, was tested against the theoretical value obtained from polynomial computation. This was performed by randomly generating values for

x in the polynomial computation, f(x) = 5 * x + 11, particularly the eighth bit of the BDD. This value is tested against the approximated values of the eighth bit of x. The approximation resulted in reduction of BDD sizes.

Another method of verification, that can be performed on the crossbar and its approximation is test by simulation . For this, the memristor crossbar networks can be simulated by a simulation software, such as Xyces or SPICE simulation. This is an aspect of testing, that can be looked at in future research.

As a future scope of research, the domain of compute code kernels used can be extended to ODEs (Ordered Differential Equations). The applications of ODEs are widespread in the fields of Computer Science, particularly in Robotics and Probabilistic computing, as well as in other fields like Physics.

The use of BDDs have largely helped in the process of efficient mapping and careful extraction of the appropriate data. Other variations of BDDs, such as Free Binary Decision Diagrams (FDDs)[14], whose variable ordering is more unregulated, and Sentential Deciison Diagrams (SDDs) whose variable ordering is more restricted. The two variations have their own advantages and disadvantages, which, after careful observation and research, can yield exponentially smaller decsion diagrams. Hence, the variations of the decision diagrams adopted in the research approach is another interesting topic to be researched upon.

The scope of memristors and stochastic computing is vast and exciting. The applications of variations in the Decision Diagram used, evolution of usage of sneakpaths, from a curse to a blessing, difference in the package used to map IR to BDD, may yield more developments and advancements in the future of computer architecture world.

APPENDIX A: C COMPUTE KERNEL CODE

```
// Polynomial computation
// Written by Anagha Sivakumar
// Input: linear polynomial of the form ax+b, like 5x+11

#include<stdio.h>
#include<string.h>

int main()
{
        float a,b,x;
        printf("\n Solver for polynomials of the form ax+b");
        printf("\n Enter the values of a and b\n");
        scanf("%f",&a);
        scanf("%f",&b);
        x=-b/a;
        printf("%f",x);
        return x;
}
```

APPENDIX B: LLVM IR

```
; ModuleID = 'poly.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [42 x i8] c"\0A Solver for
polynomials of the form ax+b\00", align 1
@.str.2 = private unnamed_addr constant [3 x i8] c"%f\00", align 1
@str = private unnamed_addr constant [30 x i8] c"\0A Enter the
values of a and b\00"

; Function Attrs: nounwind optsize uwtable
define i32 @main() #0 {
  %a = alloca float, align 4
  %b = alloca float, align 4
  %1 = bitcast float* %a to i8*
  call void @llvm.lifetime.start(i64 4, i8* %1) #3
  %2 = bitcast float* %b to i8*
  call void @llvm.lifetime.start(i64 4, i8* %2) #3
  %3 = tail call i32 (i8*, ...) @printf(i8* getelementptr
  inbounds ([42 x i8], [42 x i8]* @.str, i64 0, i64 0)) #4
  %puts = tail call i32 @puts(i8* getelementptr inbounds
  ([30 x i8], [30 x i8]* @str, i64 0, i64 0))
  %4 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr
  inbounds ([3 x i8], [3 x i8]* @.str.2, i64 0, i64 0), float*
  nonnull %a) #4
```

```
  %5 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr
  inbounds ([3 x i8], [3 x i8]* @.str.2, i64 0, i64 0), float*
  nonnull %b) #4
  %6 = load float, float* %b, align 4, !tbaa !1
  %7 = fsub float -0.000000e+00, %6
  %8 = load float, float* %a, align 4, !tbaa !1
  %9 = fdiv float %7, %8
  %10 = fpext float %9 to double
  %11 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
  ([3 x i8], [3 x i8]* @.str.2, i64 0, i64 0), double %10) #4
  %12 = fptosi float %9 to i32
  call void @llvm.lifetime.end(i64 4, i8* %2) #3
  call void @llvm.lifetime.end(i64 4, i8* %1) #3
  ret i32 %12
}

; Function Attrs: argmemonly nounwind
declare void @llvm.lifetime.start(i64, i8* nocapture) #1

; Function Attrs: nounwind optsize
declare i32 @printf(i8* nocapture readonly, ...) #2

; Function Attrs: nounwind optsize
declare i32 @__isoc99_scanf(i8* nocapture readonly, ...) #2
```

```
; Function Attrs: argmemonly nounwind
declare void @llvm.lifetime.end(i64, i8* nocapture) #1

; Function Attrs: nounwind
declare i32 @puts(i8* nocapture) #3

attributes #0 = { nounwind optsize uwtable "disable-tail-calls"
="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="false"
"no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-
buffer-size"="8" "target-cpu"="x86-64" "target-features"="+fxsr,
+mmx,+sse,+sse2" "unsafe-fp-math"="false"
"use-soft-float"="false" }
attributes #1 = { argmemonly nounwind }
attributes #2 = { nounwind optsize "disable-tail-calls"="false"
"less-precise-fpmad"="false" "no-frame-pointer-elim"="false"
"no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-
protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="
+fxsr,+mmx,+sse,+sse2" "unsafe-fp-math"="false"
"use-soft-float"="false" }
attributes #3 = { nounwind }
attributes #4 = { optsize }

!llvm.ident = !{!0}

!0 = !{!"clang version 3.8.0-2ubuntu4 (tags/RELEASE_380/final)"}
```

```
!1 = !{!2, !2, i64 0}
!2 = !{!"float", !3, i64 0}
!3 = !{!"omnipotent char", !4, i64 0}
!4 = !{!"Simple C/C++ TBAA"}
```

APPENDIX C: C++ BDD CODE

```
// BDD equivalent of linear polynomial solver
// Anagha Sivakumar
// 5 * x + 11 => run it n times
#include <iostream>
#include <bdd.h>
#include "bvec.h"
#include<fstream>

using namespace std;
int main()
{
        FILE * fp;
        fp = fopen ("bdd6.txt","w");

        bdd_init(100,1000);
        bdd_setvarnum(64);
        bdd_reorder(BDD_REORDER_SIFT);

        bvec x = bvec_var(8,0,1);
        bvec a = bvec_con(8,5);
        bvec b = bvec_con(16,10);

        // 5 * x
        bvec ax = bvec_mul(a,x);
```

```
        // 5x + -10 = 0          =>       x = 2 ( FTFFFFFF )
        bvec axplusb = bvec_add(ax,b);

        bdd_fnprintdot("bdd0.dot",axplusb[0]);
        bdd_fnprintdot("bdd1.dot",axplusb[1]);
        bdd_fnprintdot("bdd2.dot",axplusb[2]);
        bdd_fnprintdot("bdd3.dot",axplusb[3]);
        bdd_fnprintdot("bdd4.dot",axplusb[4]);
        bdd_fnprintdot("bdd5.dot",axplusb[5]);
        bdd_fnprintdot("bdd6.dot",axplusb[6]);
        bdd_fnprintdot("bdd7.dot",axplusb[7]);
        bdd_fnprintdot("bdd8.dot",axplusb[8]);
        bdd_fnsave("bdd8bit.txt",axplusb[8]);
        bdd_fnprintdot("bdd9.dot",axplusb[9]);
        bdd_fnprintdot("bdd10.dot",axplusb[10]);
        bdd_fnprintdot("bdd11.dot",axplusb[11]);
        bdd_fnprintdot("bdd12.dot",axplusb[12]);
        bdd_fnprintdot("bdd13.dot",axplusb[13]);
        bdd_fnprintdot("bdd14.dot",axplusb[14]);
        bdd_fnprintdot("bdd15.dot",axplusb[15]);
        bdd_done();
}
```

APPENDIX D: PYTHON CODE TO MAP BDD TO CROSSBAR AND APPROXIMATE

```
# BDD Approximation
# Anagha Sivakumar

# Function to map BDD (as a networkx graph) to a crossbar
def bddnx2xbar(bdd_in):
    node_vals = []
    child_nodes = []
    nodes = list(bdd_in.nodes())
    nodes.sort()
    for tmp_node in nodes:
        current_edges = list(bdd_in.out_edges([tmp_node],
        data=True))
        for tmp_edge in current_edges:
            to_node = tmp_edge[1]
            if to_node != 0:
                edge_weight = tmp_edge[2]['weight']
                if edge_weight < 1:
                    node_vals.append(-tmp_edge[0])
                else:
                    node_vals.append(tmp_edge[0])
                child_nodes.append(to_node)
    node_vals = node_vals[::-1]
    child_nodes = child_nodes[::-1]

    row_idx = 0
```

```
col_idx = 0
n_nodes_to_map = len(node_vals)
xbar_mat = np.zeros((100,100), dtype=int)
memr_idx = {}

for node_idx in range(n_nodes_to_map):
    curr_node = node_vals[node_idx]
    if node_idx > 0:
        if abs(curr_node) != abs(node_vals[node_idx-1]):
            col_idx += 1
    xbar_mat[row_idx][col_idx] = curr_node
    memr_idx[curr_node] = (row_idx, col_idx)
    row_idx += 1
zero_rows = np.where(~xbar_mat.any(axis=1))[0]
zero_cols = np.where(~xbar_mat.any(axis=0))[0]
xbar_mat = xbar_mat[:zero_rows[0]+1, :zero_cols[0]+1]
one_idx = (xbar_mat.shape[0]-1, xbar_mat.shape[1]-1)
xbar_mat[one_idx[0]][one_idx[1]] = 1
memr_idx[1] = one_idx

for node_idx in range(n_nodes_to_map):
    curr_node = node_vals[node_idx]
    curr_child = child_nodes[node_idx]
    curr_node_idx = memr_idx[curr_node]
    curr_child_idx = memr_idx.get(curr_child, [0])
```

```
        if len(curr_child_idx) < 2:
            curr_child_idx = memr_idx[-curr_child]
        one_row = curr_node_idx[0]
        one_col = curr_child_idx[1]
        xbar_mat[one_row][one_col] = 1
    return xbar_mat
#################### end of function ######################

# Function to modify the BDD (network graph) by eliminating
a subset of nodes

def modifybdd(bdd_in,elim_list):
    bdd_out =  bdd_in.copy(as_view=False)
    child_nodes = []
    nodes = list(bdd_in.nodes())
    nodes.sort()
    for tmp_node in elim_list:
        outgoing_edges = list(bdd_out.out_edges([tmp_node],
        data=True))
        incoming_edges = list(bdd_out.in_edges([tmp_node],
        data=True))
        for tmp_edge in incoming_edges:
            gp_node = tmp_edge[0]
            gp_weight = tmp_edge[2]['weight']
            for tmp_edge1 in outgoing_edges:
```

```
                child_node = tmp_edge1[1]
                child_weight = tmp_edge1[2]['weight']
                if gp_weight < 1:
                    bdd_out.add_edge(gp_node, child_node,
                    weight=0.0 )
                else:
                    bdd_out.add_edge(gp_node, child_node,
                    weight=1.0 )
        bdd_out.remove_node(tmp_node)
    return bdd_out

######################## end of function ##########################

# Function to compute the cost of the memristor crossbar matrix
def computecost(xbar_mat):
    dim = xbar_mat.shape
    cost = dim[0]+dim[1]
    return cost

#################### end of function ############################

# move Function
def move(old_bdd,elim_nodes):
    item = elim_nodes[0]
    old_elim_nodes.append(item)
```

```
    elim_nodes.remove(item)
    while True:
        from random import sample
        new_item = sample(key_list, 1)
        if new_item[0] not in elim_nodes and new_item[0] not in
        old_elim_nodes:
            break
    elim_nodes.append(new_item[0])
    return modifybdd(bdd_nx, elim_nodes)

########################## end of function ######################

# Acceptance probability function
def acceptance_probability(old_c, new_c,T):
    numerator = old_c - new_c
    from math import exp
    ap = numerator/T
    return ap
########################## end of function ######################

# Simulated annealing function
def anneal(current_solution):
    current_xbar = bddnx2xbar(current_solution)
    old_cost = computecost(current_xbar)
    # print "OLD cost = ",old_cost
```

```
    T = 1.0
    T_min = 0.001
    alpha = 0.9
    i=1
    while T > T_min:
        while i <= len(key_list)-number_of_items:
            # print "i",i
            new_solution = move(current_solution,elim_nodes)
            new_xbar = bddnx2xbar(new_solution)
            new_cost = computecost(new_xbar)
            # print "NEW cost = ",new_cost
            # print "OLD cost = ",old_cost
            ap = acceptance_probability(old_cost, new_cost, T)
            # print "Acceptance probability: ",ap
            rp = random.uniform(0.0,1.0)
            if ap > rp:
                # print "Random value between 0 and 1: ",rp
                current_solution = new_solution
                old_cost = new_cost
            i += 1
        T = T*alpha
    return current_solution
############################## end of function ########################

############################## Main function ##########################
```

```
# importing the required module
import matplotlib.pyplot as pyplot
import networkx as nx
import numpy as np
import random
import sys
inFile = sys.argv[1]

with open(inFile,'r') as f:
    content = f.readlines()

# To remove leading and trailing whitespaces
content = [x.strip() for x in content]

#To split string separated by space into words
content = [x.split(',') for x in content]
# Ignore the first two lines
content = content[2:]

# Bdd node table to dictionary
bdd_dict = {}
# Include terminal nodes 0 and 1
bdd_dict[0] = [0]
bdd_dict[1] = [1]
```

```
key_list = []
for line in content:
    line = line[0].split(' ')
    key = int(line[0])
    tmp_node = line[1:]
    bdd_dict[key] = tmp_node
    key_list.append(key)

# BDD dictionary to digraph
bdd_nx =  nx.DiGraph()
bdd_nx.add_nodes_from(key_list)
for tmp_key in key_list:
    curr_node = bdd_dict[tmp_key]
    zero_child = int(curr_node[1])
    one_child = int(curr_node[2])
    bdd_nx.add_edge(tmp_key, zero_child, weight = 0.0)
    bdd_nx.add_edge(tmp_key, one_child, weight = 1.0)

# BDD digraph to crossbar

input_xbar = bddnx2xbar(bdd_nx)
old_dim =input_xbar.shape
print "Original xbar dimensions =", old_dim[0],old_dim[1]

# 1 - Randomly perturb the BDD to get a current solution
```

```
old_elim_nodes = []
from random import sample
number_of_items = len(key_list)/2
elim_nodes = sample(key_list, number_of_items)
current_solution = modifybdd(bdd_nx,elim_nodes)

# Simulated annealing - BDD approximation
sol = anneal(current_solution)

# Output
print "Solution bdd and matrix:"
print sol
output_xbar = bddnx2xbar(sol)
print output_xbar
new_dim =output_xbar.shape
print "New xbar dimensions =", new_dim[0],new_dim[1]
# output_cost = computecost(output_xbar)
# print "Output cost =",output_cost
import pylab as plt
from networkx.drawing.nx_agraph import graphviz_layout
nx.draw(bdd_nx, pos=graphviz_layout(bdd_nx), node_size=300,
cmap=plt.cm.Greens,
        node_color=range(len(bdd_nx)),
        prog='neato')
plt.show()
```

```
nx.draw(sol, pos=graphviz_layout(sol), node_size=300,
cmap=plt.cm.Blues,
        node_color=range(len(sol)),
        prog='dot')
plt.show()

########################## end of function ######################
```

LIST OF REFERENCES

[1] Leon Chua, *Memristor-the missing circuit element*, IEEE Transactions on circuit theory, 1971

[2] R. E. Bryant, *Binary decision diagrams and beyond: enabling technologies for formal verification*, International Conference on Computer Aided Design (ICCAD), 1995

[3] 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* , IEEE International Conference on Rebooting Computing 2017, Washington D.C., 2017

[4] D. Chakraborty and S. K. Jha, *Automated synthesis of compact crossbars for sneak-path based in-memory computing*, Design Automation and Test in Europe Conference and Exhibition, 2017

[5] Said Hamdioui, Lei Xie, Hoang Anh Du Nguyen, Mottaqiallah Taouil, Koen Bertels, Henk Corporaal, Hailong Jiao, Francky Catthoor, Dirk Wouters, Linn Eike, Jan van Lunteren, *Memristor based computation-in-memory architecture for data-intensive applications*, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015

[6] Armin Alaghi, Cheng Li, John P. Hayes , *Stochastic circuits for real-time image-processing applications*, DAC '13 Proceedings of the 50th Annual Design Automation Conference, 2013

[7] Mathias Soeken, Daniel Groe, Arun Chandrasekharan, Rolf Drechs, *BDD Minimization for Approximate Computing*, Asia and South Pacific - Design Automation Conference (ASP-DAC), 2016

[8] Pinaki Mazumder, Sung Mo Kang, Rainer Waser, *Memristors: Devices, Models, and Applications [Scanning the Issue]*, Proceedings of the IEEE (Volume: 100, Issue: 6, Page(s): 1911 - 1919 i), 2012

[9] Alvaro Velasquez, Sumit Kumar Jha, *Parallel Computing using Memristive Crossbar Networks: Nullifying the Processor-Memory Bottleneck*, Design & Test Symposium (IDT), 2014

[10] Shahar Kvatinsky, Avinoam Kolodny, Uri C. Weiser, Eby G. Friedman, *Memristor-based IMPLY logic design procedure*, IEEE 29th International Conference on Computer Design (ICCD), 2011

[11] Dwaipayan Chakraborty, Sumit Kumar Jha, *Automated synthesis of compact crossbars for sneak-path based in-memory computing*,Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017

[12] Yuval Cassuto; Shahar Kvatinsky; Eitan Yaakobi, *Sneak-Path Constraints in Memristor Crossbar Arrays*, IEEE International Symposium on Information Theory, 2013

[13] Zahiruddin Alamgir, Karsten Beckmann, Nathaniel Cady, Alvaro Velasquez, Sumit Kumar Jha, *Flow-based computing on nanoscale crossbars: Design and implementation of full adders*, IEEE International Symposium on Circuits and Systems (ISCAS), 2016

[14] A. U. Hassen, D. Chakraborty, and S. K. Jha, *Free Binary Decision Diagram Based Synthesis of Compact Crossbars for in-Memory Computing of Boolean Functions* , Transactions on Circuits and Systems (TCAS) II, 2018

[15] A. Velasquez and S. K. Jha, *Parallel computing using memristive crossbar networks: Nullifying the processor-memory bottleneck*, Design & Test Symposium (IDT), 2014 9th International, 2014

[16] A. U. Hassen, B. Chandrasekar, and S. K. Jha, *Automated synthesis of stochastic computational elements using decision procedures*, IEEE International Symposium on Circuits and Systems (ISCAS), 2016

[17] Zdenek Vasicek, Lukas Sekanina, *Search-Based Synthesis of Approximate Circuits Implemented into FPGAs*,26th International Conference on Field Programmable Logic and Applications (FPL), 2016

[18] James King, Sheir Yarkoni, Mayssam M. Nevisi, Jeremy P. Hilton, Catherine C. McGeoch, *Benchmarking Adiabatic Quantum Optimization for Complex Network Analysis*, 2015

[19] Adewole A.P.,Otubamowo K.,Egunjobi T.O., *A Comparative Study of Simulated Annealing and Genetic Algorithm for Solving the Travelling Salesman Problem*, International Journal of Applied Information Systems (IJAIS), 2012