Saeideh Shirinzadeh · Rolf Drechsler

In-Memory Computing

Synthesis and Optimization

Springer

In-Memory Computing

Saeideh Shirinzadeh • Rolf Drechsler

In-Memory Computing

Synthesis and Optimization

Springer

Saeideh Shirinzadeh
University of Bremen and DFKI GmbH
Bremen, Germany

Rolf Drechsler
University of Bremen and DFKI GmbH
Bremen, Germany

ISBN 978-3-030-18025-6 ISBN 978-3-030-18026-3 (eBook)
https://doi.org/10.1007/978-3-030-18026-3

© Springer Nature Switzerland AG 2020
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Acknowledgements

We would like to particularly appreciate all those who have contributed to the results included in this book. Our special thanks go to Mathias Soeken for his invaluable investments of time and brilliant ideas, which were of significant importance for this book. We also would like to acknowledge Pierre-Emmanuel Gaillardon for his contributions, which helped us to improve this book. Many thanks to Giovanni De Micheli and Daniel Große for their helpful feedback and inspiring discussions.

We would like to express our appreciation to all of the colleagues in the group of computer architecture at the University of Bremen for their support.

Bremen, Germany
January 2019

Saeideh Shirinzadeh
Rolf Drechsler

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Today's computer architectures are mainly structured based on the model proposed by John von Neumann, the Hungarian-American mathematician in his *First Draft of a Report on the EDVAC* [59]. The von Neumann architecture in an abstract way, consists of a *Central Processing Unit* (CPU) which is made of control and arithmetic/logic units, a main memory unit also called *primary storage* to store the data and instructions, an external storage unit, i.e., *secondary storage*, which in contrary to the main memory does not have direct access to the CPU, input/output devices, and buses for transmitting data, addresses, and commands between CPU and main memory.

Among the different storage components provided by the memory hierarchy used in today's computer architectures, we refer to commonly used hard disk as the secondary storage and the *Random-Access Memory* (RAM) as the primary storage, the so-called memory. The amount of disk in any typical computer is considerably larger than RAM due to its high price. This makes it costly to allocate large amounts of RAM enough to be used as the only storage space, and using the disk only for archiving permanently. Accordingly, use of hard disk can not be avoided when processing data larger than the storage volume provided by RAM. However, the access time for information stored on the hard disk is at least 20 times the amount of time needed by RAM. This latency caused by the communication between memory and processor limits the overall performance of the CPU as it can not work at a higher speed than the required waiting time for the data to be retrieved which is referred to as memory bottleneck.

The memory bottleneck is nowadays the main obstacle in the advancement of computer systems. The emergence of applications such as *Internet of Things* (IoT) and *big data* has drawn attentions to alleviate the memory issue due to the need to process and retrieve huge amount of data in real time. In this condition, further optimization to the existing memory hierarchy can not be a solution anymore, rather a revolutionary approach is required. In-memory computing allowed by advanced

© Springer Nature Switzerland AG 2020

S. Shirinzadeh, R. Drechsler, *In-Memory Computing*,
https://doi.org/10.1007/978-3-030-18026-3_1

non-volatile memory technologies is a promising candidate which abolishes the communication between memory and processor by merging them.

Resistive RAM (RRAM) is a promising non-volatile technology which stores data in form of electrical resistance. The resistive state of the device can be switched between two low and high states representing the binary values. This allows to perform logic primitives, and hence makes it possible to process information within the memory. Besides the resistive switching capability, RRAM possesses interesting properties such as zero standby power, high scalability, low read/write latencies, and compatibility with *Complementary Metal Oxide Semiconductor* (CMOS) technology [36, 93, 95, 96, 99]. These features enable high density hybrid CMOS-RRAM structures realizing digital logic-in-memory circuits and systems.

RRAM crossbar array architectures can be programmed to calculate arbitrary logical functions performed within current computer processors. As computation is performed inside RRAM devices, memory is automatically updated with the result, without additional latency to transfer and write the information which is inevitable in classical von Neumann architectures, where memory and processor are separated (see Fig. 1.1).

The logic computation capability within RRAM devices can also be generalized to stand-alone programmable hardwares. Such in-memory circuits and systems are advantageous to the current silicon based hardwares fabricated with the CMOS technology which suffer from deteriorated performance resulted by continuously scaling down the minimum physical features approaching the atomistic and quantum boundaries. As the steadily shrinking technology sizes is bringing the CMOS era to its end [38], the necessity to move towards promising post CMOS technologies has resulted in high research attention to such fields.

According to the discussions above, there is a need to fill the gap between the traditional synthesis flow and the specific requirements of synthesis with RRAM devices. In this context, this book aims at providing a comprehensive and automated design flow for in-memory computing. The presented contributions enable to synthesize highly efficient programmable logic-in-memory hardwares and architectures.

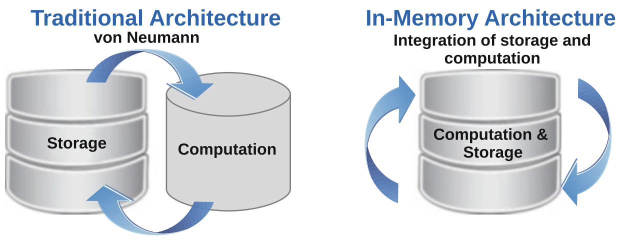


Fig. 1.1 von Neumann vs. in-memory computer architecture

1.1 Overview

This book explores synthesis for in-memory computing from two different perspectives, i.e., (1) based on a customized approach at gate level which employs logic representations, and (2) an instruction oriented approach for efficient manipulation, compilation, and execution of programs on a logic-in-memory computer architecture [72, 78]. The customized approach proposes design methodologies and optimization algorithms for each representation with respect to area and latency upon the realizations of their logic primitives. The instruction based approach proposes an automated compiler to execute instructions on a logic-in-memory computer architecture. The proposed approach optimizes the programs with respect to latency and the number of required devices. It also addresses the write endurance issue by applying wear leveling techniques.

The proposed customized approach aims at filling the gap between the traditional synthesis flow and the technology specific requirements for synthesis of in-memory computing circuits using resistive devices. For this purpose, the customized synthesis approach employs the well known logic representations such as *Binary Decision Diagram* (BDD) (e.g., in [23]), *And-Inverter Graph* (AIG) [48], and the recently introduced *Majority-Inverter Graph* (MIG) [1]. The approach includes three stages, (1) finding efficient realizations with RRAM devices for logic primitive of each representation, (2) then, defining the design methodology to map the representations into RRAM array, (3) and finally optimizing the representations with respect to the number of RRAM devices and operations determined by the design methodology (see Fig. 1.2). The contributions towards the customized synthesis approach include many effective heuristic algorithms for optimization of the aforementioned logic representations with respect to different cost metrics. These algorithms can also be beneficial for classical logic synthesis and other applications using BDDs, AIGs, and MIGs.

By means of the instruction based approach, we fully automatize and optimize a logic-in-memory computing architecture consisting of banks of standard memristive arrays. The aim here is to start from a gate level description and present it efficiently

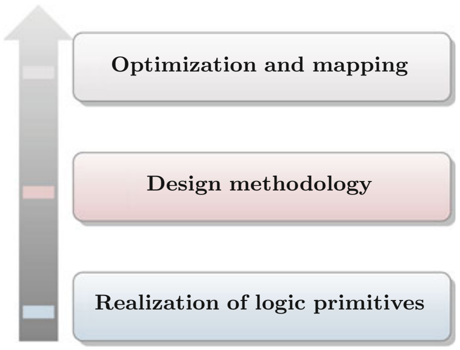


Fig. 1.2 Design flow for the proposed customized synthesis approach

by MIGs. Then, the target MIG is optimized and translated into understandable commands for the logic-in-memory computer. The compiled programs include the scheduled list of instructions, i.e., reads and writes or in other words computations, which can be executed by applying appropriate voltage levels to certain wordlines and bitlines of the RRAM array. Furthermore, we address the issue of lower write endurance of RRAM devices and propose wear-leveling techniques to increase the lifetime of the architecture.

1.2 Outline

This book consists of six chapters including the current introductory chapter. Chapter 2 presents the required background. This includes brief introductions on the RRAM technology and its features, the employed logic representations, and basic concept of evolutionary computation which are used for optimization of BDDs. Chapters 3–5 present the main technical contents of the book which are briefly described in the following.

- The optimization algorithms proposed for BDDs as a representation used by the customized synthesis approach are also developed for general purpose BDD optimization by setting different cost metrics. These algorithms are presented in Chap. 3 and allow multi-objective BDD optimization which has been performed for the first time. The proposed algorithms allow to apply preferences and can be applied to a wide range of applications utilizing BDDs. The experimental result reveal that considerable efficiency has been achieved in comparison with the state-of-the-art BDD optimization techniques. The chapter also allows BDD approximation which is highly timely. Approximation has been incorporated with optimization allowing much higher gain at low costs of inaccuracy which is suitable for error resilient applications.
- Chapter 4 presents the customized synthesis approach. This includes three design approaches based on the features of BDDs, AIGs, and MIGs representing the target Boolean functions. The chapter also includes several MIG optimization algorithms with respect to different criteria, while the optimization of AIGs has been directly performed through rewriting algorithms embedded within other ABC [5] logic synthesis tool. The chapter shows the design preferences varying with respect to the latency and area of the resulting implementations which are provided by use of the three different logic representations and two basic operations enabled within RRAM devices. Also, implementation issues on memristive crossbar array are extensively discussed within the chapter with example.
- Chapter 5 discusses the instruction based synthesis method for a programmable logic-in-memory computer architecture. The chapter includes three main contributions, i.e., automated compilation of the logic-in-memory programs given a gate level description which has been performed for the first time, optimization

of the resulting programs with respect to the number of required RRAM devices and the length of instruction sequences, and wear leveling of the memory arrays by applying techniques to balance writes over entire devices which is addressed at both times of compilation and execution. The experimental results presented in the chapter show considerable improvements in comparison to the naïve implementations. The contributions presented in Chap. 5 resulted in the following publications.

This book is concluded in Chap. 6.

Chapter 2
Background

This chapter aims at keeping this book self-contained by providing brief introductions on the basic concepts required for the following chapters. The chapter includes four introductory sections. The first section introduces the RRAM technology and its fundamental properties. This is followed by a section discussing the current basic logic operations which are executable within RRAM devices. This book conducts customized synthesis based on logic representations which are introduced in Sect. 2.3. Since optimization for one of the logic representations is performed through a multi-objective approach using evolutionary heuristics, Sect. 2.4 explains the basics and general flow of such optimization algorithms.

2.1 RRAM Technology

The abrupt switching capability of an oxide insulator sandwiched by two metal electrodes was known from 1960s, but it did not come into interest for several decades until feasible device structures were proposed. Nowadays, a variety of two-terminal devices based on resistance switching property exist which use different materials. These devices possess resistive switching characteristics between two high and low resistance values and are known by various acronyms such as OxRAM, ReRAM, and RRAM [94].

An RRAM cell is a two-terminal device which is made of a metal oxide sandwiched by two metal electrodes. RRAM basically performs like an electrical resistor which internal resistance allocates two stable low and high values designating the binary values 1 and 0, respectively. The switching between the alternate states is possible by applying appropriate voltages to the device which can be exploited to execute logic operations within RRAM devices. RRAM device is non-volatile as it maintains its resistance state until being changed under voltage bias with required polarity and adequate amplitude. Figure 2.1 shows the structure of an RRAM device

© Springer Nature Switzerland AG 2020

S. Shirinzadeh, R. Drechsler, *In-Memory Computing*,
https://doi.org/10.1007/978-3-030-18026-3_2

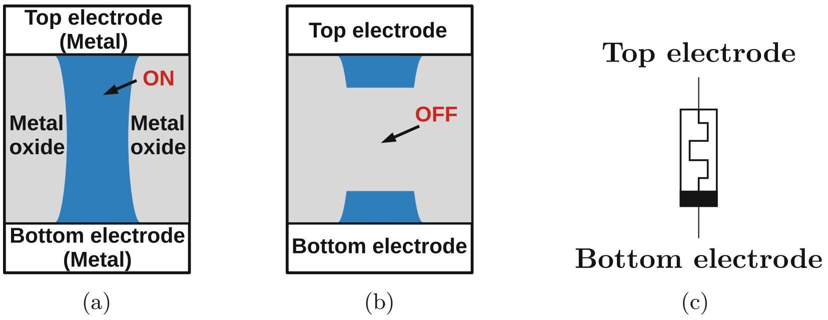


Fig. 2.1 Structure of an RRAM device: (**a**) low-resistance state (on), (**b**) high-resistance state (off). (**c**) Symbol of an RRAM device

in its two different resistance states which correspond to on and off switches. As the figure shows, the device is on when it is in low-resistance state or in other words the electrical conductance is high which is due to the dopant concentration between the electrodes.

High scalability of RRAM devices [94] makes it possible to implement ultra dense resistive memory arrays [44]. Such architectures using memristive devices are of high interest for their possible applications in non-volatile memory design [40, 56], digital and analog programmable systems [18, 34, 61], and neuromorphic computing structures [86]. Furthermore, the resistive switching property in RRAM devices also allows advanced computer architectures different from classical von Neumann architectures by providing memories capable of computing [35, 51].

RRAM devices have also attracted renewed attention to the theory of memristors proposed by Chua in 1971 [19]. Chua reasoned a forth fundamental passive circuit element from the symmetric equations between the three known elements, resistor, capacitor, and inductor (see Fig. 2.2). The new element was called memristor as a short for memory and resistor. In [20], RRAM devices were suggested as a physical implementation for memristors. Although some researchers argued differences between a memristor and an RRAM device [92] due to the lack of magnetic flow, the resistive switching property is shared by both devices [20, 87, 94]. Since different RRAM devices have already been fabricated and their functionality is proven, here we prefer to use the term RRAM device. Also, RRAM device is referred with the terms resistive and memristive device or switch throughout this book interchangeably.

The memristor represents a non-linear relationship between electrical charge and magnetic flux [19], as shown in Fig. 2.2. This definition is later extended to time invariant memristive devices controlled by current even without magnetic charge such that

$$dx/dt = f(x, i)v(t) = R(x, i).i(t).$$

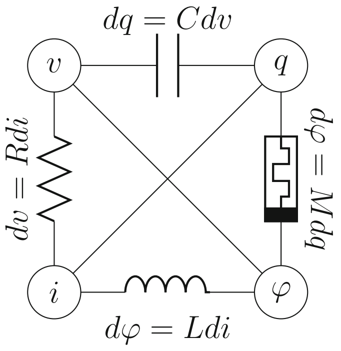


Fig. 2.2 Relationship between fundamental passive circuit elements derived based on duality [19]

In 2008, HP labs fabricated a device which was claimed to be the missing memristor [87]. The HP device was made of TiO_2 doped with oxygen vacancies on one side (TiO_{2-x}) sandwiched between two platinum electrodes . The doped region has a lower resistivity than the undoped region. By applying a voltage of suitable polarity and magnitude across the device, the doped region can be expanded or contracted thereby resulting in a change in resistance. When the voltage is withdrawn, the states of the oxygen vacancy carriers remain unchanged, and thus the device can remember or memorize its last resistance value. In addition to TiO_2, several other materials have also been used for fabricating memristive devices [80].

For analyzing RRAM based circuits, various circuit models for the memristive devices have been proposed. This allows a designer to simulate the circuit designs using standard circuit simulation tools and analyze their performance. Some of the simulation models that have been proposed so far include linear ion drift model [87], threshold adaptive memristor model (TEAM) [49], and voltage threshold adaptive memristor model (VTEAM) [52].

One of the first RRAM models to be proposed is the linear ion drift model, which is based on the simplified view of the HP memristive device as shown in Fig. 2.3. In this model, the memristive device is viewed as a combination of two variable resistors in series, one corresponding to the doped region and the other to the undoped region. The width of the doped region w is referred to as the *state variable*, and determines the conductivity of the memristor. The following equations describe the drift-diffusion velocity and the time varying voltage in this model:

$$\frac{dw}{dt} = \frac{\mu_v R_{on} i(t)}{D}$$

$$v(t) = \left(\frac{w(t)}{D} R_{on} + \left(1 - \frac{w(t)}{D} R_{off}\right)\right) i(t)$$

where D is the width of the memristive device, μ_v is the average ion mobility of the TiO_2 region, and $w(t)$ is the thickness of the doped region as a function of time t, also called the state variable. R_{on} is the resistance when the width of the doped

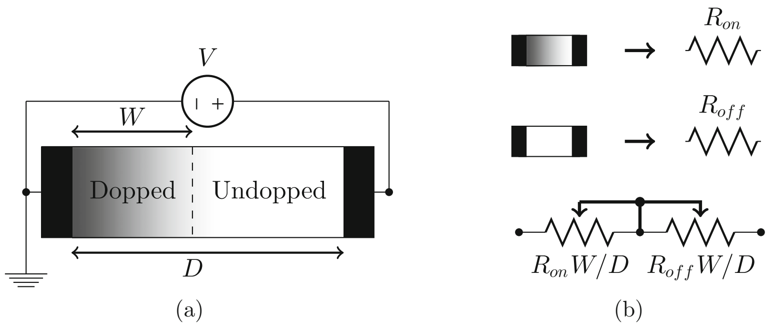


Fig. 2.3 (**a**) Schematic diagram of TiO_2 memrisive device proposed in [87]. (**b**) RRAM as resistor in on and off states [87]

region $w(t)$ is D, and R_{off} is the resistance when $w(t)$ is 0. The total memristance of the device is given by:

$$M(q) = R_{off}\left(1 - \frac{\mu_v R_{on}}{D^2} q(t)\right)$$

where $v(t)$, $i(t)$ and $q(t)$ respectively denote the voltage, current and total charge flowing through the device at time t.

To overcome the limitations of the model when w approaches the boundaries of the device, and also to introduce non-linearities in ion drift, various window functions have been proposed such as in [45] and [8]. There are also other memristor models such as TEAM [49] and VTEAM [52] that directly incorporate non-linear behavior in the ion drift phenomena in a more accurate manner. However, these models are computationally more intensive.

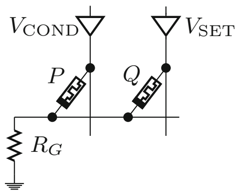
2.2 Logic-in-Memory with RRAM

This section explains how three fundamental operations can be executed within an RRAM device to be used as the basis for design of RRAM-based logical circuits.

2.2.1 Material Implication

Material Implication (IMP), i.e. $q' \leftarrow p$ IMP $q = \overline{p} + q$, and FALSE operation, i.e. assigning the output to logic 0, are sufficient to express any Boolean function [12]. Figure 2.4 shows the implementation of an IMP gate which was proposed in [12]. P and Q designate two resistive devices connected to a load resistor R_G.

Fig. 2.4 IMP operation. (**a**) Implementation of IMP using RRAM devices. (**b**) Truth table for IMP ($q' \leftarrow p$ IMP $q = \overline{p} + q$) [12]



(a)

p	q	q'
0	0	1
0	1	1
1	0	0
1	1	1

(b)

Three voltage levels V_{SET}, V_{COND}, and V_{CLEAR} are applied to the devices to execute IMP and FALSE operations by switching between low-resistance (logic 1) or high-resistance (logic 0) states.

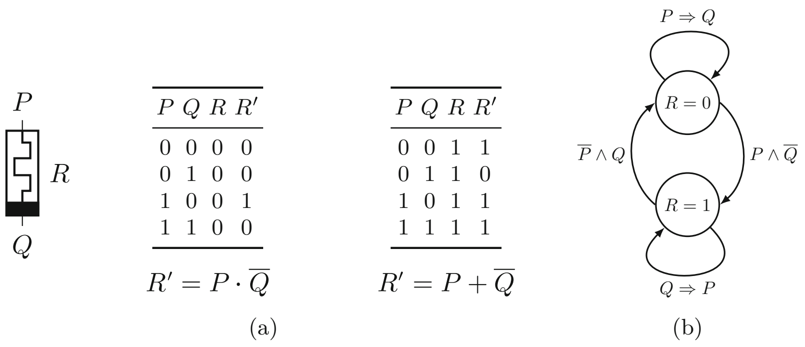
The FALSE operation can be performed by applying V_{CLEAR} to an RRAM device. An RRAM device can also be switched to logic 1 by applying a voltage larger than a threshold V_{SET} to its voltage driver. To execute IMP, two voltage levels V_{SET} and V_{COND} are applied to the switches P and Q simultaneously. The magnitude of V_{COND} is smaller than the required threshold to change the state of the switch. However, the interaction of V_{SET} and V_{COND} can execute IMP according to the current states of the switches, such that switch Q is set to 1 if $p = 0$ and it retains its current state if $p = 1$ [12].

2.2.2 *Resistive Majority Operation*

RRAM is a two-terminal device which internal resistance R can be switched between two logic states 0 and 1 designating high and low resistance values, respectively. Denoting the top and bottom terminals by P and Q, the memory can be switched with a negative or positive voltage V_{PQ} based on the device polarity. The truth tables in Fig. 2.5 show how the next state of the switch (R') is resulted from the interaction of values of P, Q, and the current state (R). The built-in majority operation described in Fig. 2.5 can be formally expressed as the following [35]:

$$\begin{aligned}
R' = \mathrm{RM}_3(P, Q, R) &= (P \cdot \overline{Q}) \cdot \overline{R} + (P + \overline{Q}) \cdot R \\
&= P \cdot R + \overline{Q} \cdot R + P \cdot \overline{Q} \cdot \overline{R} \\
&= P \cdot R + \overline{Q} \cdot R + P \cdot \overline{Q} \cdot R + P \cdot \overline{Q} \cdot \overline{R} \\
&= P \cdot R + \overline{Q} \cdot R + P \cdot \overline{Q} \\
&= M(P, \overline{Q}, R)
\end{aligned}$$

The operation above is referred to three input resistive majority denoted by RM_3, with $\mathrm{RM}_3(x, y, z) = M(x, \bar{y}, z)$ [35]. According to RM_3, the next state of a resistive switch is equal to a result of a built-in majority gate when one of the three



P	Q	R	R'
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	0

$$R' = P \cdot \overline{Q}$$

P	Q	R	R'
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	1

$$R' = P + \overline{Q}$$

Fig. 2.5 The intrinsic majority operation within an RRAM device presented by (**a**) truth table, and (**b**) finite state machine [35]

variables x, y, and z is already preloaded and the variable corresponding to the logic state of the bottom electrode is inverted.

Throughout this book, we use MAJ as another acronym for the RM$_3$ operation when compared with IMP, while RM$_3$ is used where it is the only operation, i.e. in Chap. 5.

2.2.3 Memristor Aided Logic

MAGIC is the acronym for memristor aided logic [50], which is a stateful logic design style where the resistance values represent the logic states. In contrary to IMP, in MAGIC the input and output values are stored in different RRAM devices. A MAGIC gate operation requires two sequential steps. In the first step, the output device is initialized to a known logical state (either 0 or 1). For non-inverting gates like AND and OR, the output device is initialized to 0, while for inverting gates such as NOT, NOR, and NAND the initialization sets the device to 1. In the second step, a suitable voltage V_0 is applied to the input RRAM devices. The voltage across the output device depends upon the logical state of the input and output switches [50].

All basic gates (NOT, AND, OR, NOR, NAND) can be executed within RRAM devices using the MAGIC operation. Though all gates can be implemented using the MAGIC only; however, only NOR and NOT gates can be mapped to RRAM arrays due to the connection pattern among input and output devices. Other gates can only be used as stand-alone logic. Hence, for synthesizing larger Boolean functions on RRAM crossbar arrays, i.e. arrays consisting of horizontal and vertical nanowires, where the RRAM devices are fabricated at the junctions, the functions are first represented in terms of NOR and NOT gates. Then, various mapping techniques [89, 90] are used to map the gates to the crossbar.

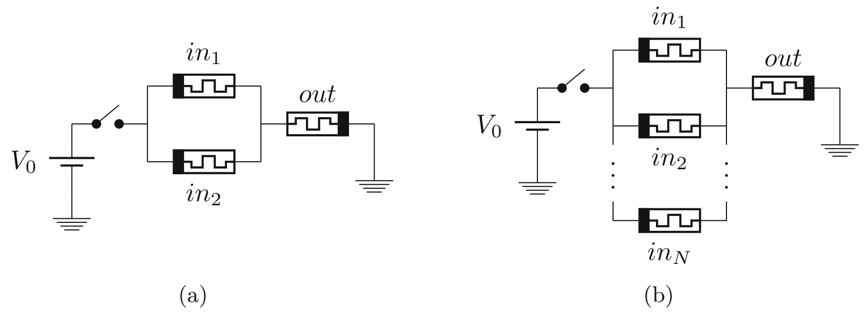


Fig. 2.6 (**a**) 2-input MAGIC NOR gate. (**b**) N-input MAGIC NOR gate [50]

Figure 2.6 shows MAGIC NOR gates with two and N input variables. The implementation for both cases is performed in the same manner and takes two computational steps. The procedure is similar to that explained above for general case MAGIC-based computation of logic gates. In the first step, the output device is set to 1. In this case, if all of the input variables are equal to 0 (high resistance state), the voltage dropping on the output device is less than the required threshold to change the resistance state and therefore the output remains 1. In other input combinations, i.e. when at least of the input variables is equal to 1 (low resistance state), the voltage share on the output device is greater than the required threshold and therefore can switch the output to logic 0.

2.3 Logic Representations

This section introduces the logic representations which are used by this book. These representations include *Binary Decision Diagram* (BDD), *And-Inverter Graph* (AIG), and *Majority-Inverter Graph* (MIG).

Figure 2.7 shows all of the three aforementioned representations for a Boolean function with three input variables. All of the graphs have three nodes and three levels, i.e. partitions with equal distance or computational latency from the root function f. Number of nodes and levels, also known as size and depth, are considered important features of a representation because of determining the area and latency of the resulting circuits, respectively. In particular, the latency of the resulting circuits highly matters for in-memory computing. Therefore, optimization of the logic representations with respect to the number of levels of the graphs is very important. This section discusses the features of three mentioned representations, while their optimization is discussed in detail in Chaps. 3 and 4.

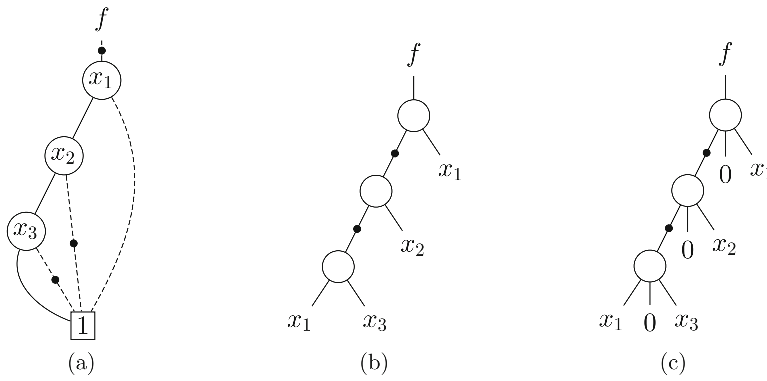
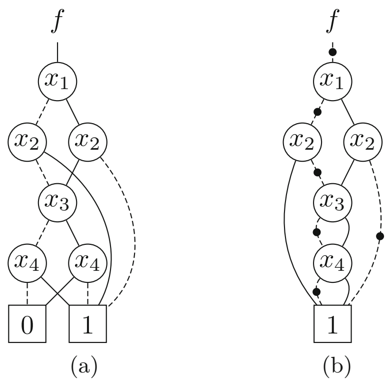


Fig. 2.7 Logic representations for an example function with three input variables. (**a**) BDD, (**b**) AIG, and (**c**) MIG

Fig. 2.8 Initial BDD representations with and without complemented edges for the function $f = (x_1 \oplus x_2) \vee (x_3 \oplus x_4)$, using the ascending variable ordering. (**a**) BDD only with regular edges. (**b**) BDD with regular and complemented edges



2.3.1 Binary Decision Diagrams

Binary Decision Diagrams (BDD) can provide a compact graph representation for most of the practical relevant Boolean functions. For an arbitrary function $f(x)$, the BDD representation is derived from the Shannon decomposition $f = x_i f_{x_i} \oplus \bar{x}_i f_{\bar{x}_i}$ [70]. Applying this decomposition recursively allows dividing the function into many smaller sub-functions. The main reason of the compactness of BDDs is that such functions often have common *subfunctions* where for a given function $f(x_1, x_2, \ldots, x_n)$ its subfunctions are $f(0, x_2, \ldots, x_n)$ and $f(1, x_2, \ldots, x_n)$ as well as their subfunctions and so on. A function $f(x_1, \ldots, x_n)$ is called a *bead* if it depends on its first variable, or in other words, if its truth table representation is not of the form $\alpha\alpha$ for any bitstring α of length 2^{n-1} [47]. The nodes of a BDD for a Boolean function f are all subfunctions of f which are beads. The size of a BDD can further be reduced with complemented edges such that a subfunction and its complement can be represented by the same node [13] (see Fig. 2.8).

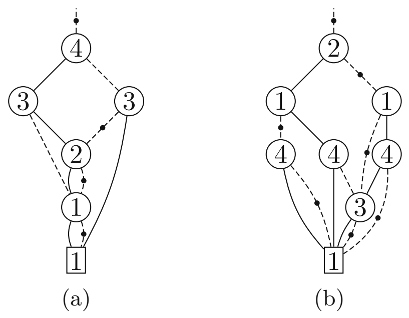


Fig. 2.9 Two BDD representations for function $\bar{x}_1\bar{x}_2\bar{x}_3 \vee \bar{x}_1x_2x_4 \vee x_1x_2\bar{x}_3\bar{x}_4 \vee x_1\bar{x}_2x_3x_4$ with different variable orderings; (**a**) optimized BDD with respect to number of nodes, (**b**) optimized BDD with respect to number of one-paths

It can easily be seen that the set of subfunctions changes when the order of input variables is changed. In fact the *variable ordering* of a BDD has a significant impact on the number of nodes and paths in a BDD. As an example, the function

$$\bar{x}_1\bar{x}_2\bar{x}_3 \vee \bar{x}_1x_2x_4 \vee x_1x_2\bar{x}_3\bar{x}_4 \vee x_1\bar{x}_2x_3x_4$$

from [32] and depicted in Fig. 2.9 has its minimal number of nodes, $N = 5$, for the variable ordering $x_4 < x_3 < x_3 < x_1$. The minimal number of one-paths evaluating the function to one, i.e. all paths from the start vertex to the 1-terminal that have an even number of complemented edges (including the complement of the start vertex), $P = 4$, is obtained for the variable ordering $x_2 < x_1 < x_4 < x_3$.

Throughout this book, we consider initial BDD representations before optimization with a fixed ascending variable ordering $x_1 < x_2 < \cdots < x_n$, where n is the number of input variables, e.g., in Fig. 2.8, $n = 4$ and therefore the ordering is $x_1 < x_2 < x_3 < x_4$.

Improving the variable ordering to find a BDD with at most N nodes is known to be NP-complete [10] and several heuristics have been proposed that are described in the next chapter.

2.3.2 Homogeneous Logic Representations

In this work we use *And-Inverter Graphs* (AIGs) [48] and *Majority-Inverter Graphs* (MIGs) [1] as homogeneous logic representation. Each node in the graphs represents one logic operation, $x \cdot y$ (conjunction) in case of AIGs, and $M(x, y, z) = x \cdot y + x \cdot z + y \cdot z$ (majority of three) in case of MIGs. Inverters are represented in terms of complemented edges; regular edges represent non-complemented inputs. Homogeneous logic representations allow for efficient and simpler algorithms due to their regular structure—no case distinction is required for the logic operations. Consequently, such logic representations are the major data structure in state-of-the-art logic synthesis tools.

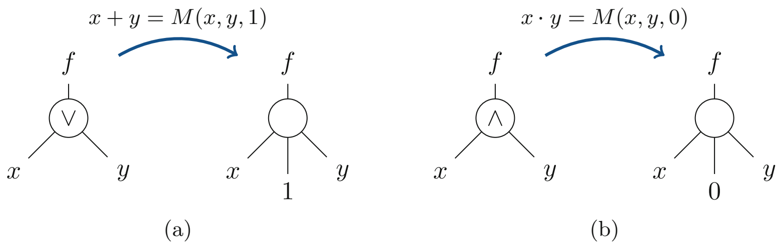


Fig. 2.10 Converting AIG and OIG representations to MIG by adding third constant inputs. (**a**) An AND gate transposed to majority gate. (**b**) Conversion of an OR gate into a majority gate

MIG [1] has a high flexibility in depth optimization, i.e. lowering the number of levels of the graph, and therefore enables design of high speed logic circuits and FPGA implementations [2]. In comparison with the well-known data structures BDDs and AIGs, MIGs have experimentally shown better results in logic optimization, especially in propagation delay [1]. In [35], it was shown that MIGs are highly qualified for logic synthesis of RRAM-based circuits since they can efficiently execute the intrinsic *resistive majority* operation in RRAM devices indicated by RM3 (see Sect. 2.2.2). This enables more efficient in-memory computing exploiting a majority oriented paradigm.

MIGs can efficiently represent Boolean functions enabled by the expressive power of the *majority operator* $M(a, b, c) = a{\cdot}b{+}a{\cdot}c{+}b{\cdot}c = (a{+}b){\cdot}(a{+}c){\cdot}(b{+}c)$. Indeed, a majority operator can be configured to behave as a traditional conjunction (AND) or disjunction (OR) operator. As shown in Fig. 2.10, in the case of 3-input majority operator, fixing one input to 0 realizes an AND while fixing one input to 1 realizes an OR. As a consequence of the AND/OR inclusion by majority operator, traditional *And-Or-Inverter Graphs* (AOIGs) are a special case of MIGs and MIGs can be easily derived from AOIGs [1, 83]. Accordingly, MIGs are at least as compact as AOIGs. However, even smaller MIG representation can be obtained when fully exploiting the majority functionality, i.e. with nonconstant inputs [3].

2.4 Evolutionary Multi-objective Optimization

Evolutionary Algorithms (EAs) have been used by this book to optimize synthesized circuits based on BDDs with respect to several design criteria, mainly area and latency. Therefore, this section explains the evolutionary multi-objective optimization by first formally defining a multi-objective optimization problem and then describing the general framework of evolutionary algorithms.

2.4.1 Multi-objective Optimization

Multi-objective optimization is the problem of multiple criteria decision making, that is concerned with mathematical optimization problems involving more than one objective function to be optimized simultaneously. Such problems can arise in every field of science, like engineering, economics and logistics, which has led to an increasing need for efficient and reliable multi-objective optimization methods. The task is challenging due to the fact that, instead of a single optimal solution, multi-objective optimization results in a set of solutions representing different trade-offs among conflicting criteria. Knowledge about such solution set helps the decision maker to choose the best compromise solution. This way, in case of high dimensional design problems, the large design space can be dramatically reduced to a set of optimal trade- offs.

An arbitrary optimization problem with m objectives can be defined as

$$\min \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x}))^{\mathrm{T}},$$

where solution $\mathbf{x} = (x_1, x_2, \ldots, x_n)^{\mathrm{T}}$ is described as a decision vector from the decision space $\Omega \subseteq \mathbb{R}^n$, and $\mathbf{f} : \Omega \rightarrow \Lambda$ is a set of m objective functions which evaluates a specific solution by mapping it to objective space $\Lambda \subseteq \mathbb{R}^m$.

Let $\mathbf{x}, \mathbf{y} \in \Omega$, then the *Pareto-dominance*, denoted by $\prec$, can be defined as

$$\begin{aligned} \mathbf{x} \prec \mathbf{y} :\Leftrightarrow \exists j \in \{1,2, \ldots, m\} :& \\ & f_j(\mathbf{x}) < f_j(\mathbf{y}) \wedge \forall i \neq j : \; f_i(\mathbf{x}) \leq f_i(\mathbf{y}). \end{aligned}$$

According to the definition above, an optimal solution, also called *non-dominated* is a solution that is not dominated by any other solution. The set of optimal solutions is the so-called *Pareto set*, denoted by $\omega \subseteq \Omega$. It must fulfill the property

$$\forall p \in \omega : \nexists q \in \Omega : q \prec p.$$

The *Pareto front* λ is defined as the image of the Pareto set in the objective space, i.e, $\lambda = f(\omega) \subseteq \Lambda$. Finding a well converged set of solutions to the Pareto set is the goal of a *Multi-Objective Evolutionary Algorithm* (MOEA). The final output of a MOEA is a set of non-dominated solutions, so-called *Pareto set approximation*.

2.4.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a class of probabilistic optimization methods which simulate the process of natural evolution [101]. It should be noted that the term EA throughout this book does not refer to other classes of evolutionary computation such as evolutionary strategy, or evolutionary programming. The concept of EA is closely related and to a large extent equal to *Genetic Algorithm* (GA), and hence both terms are interchangeably used in this book.

For a given multi-objective problem, EAs work on a set of possible solutions which is called population with each solution in the population being referred to as an individual, resembling the terms used for describing natural evolutionary phenomena. Similarly, every iteration of an EA is called a generation.

The size of the population and the number of generations are two important parameters of an EA. The bigger the population and the more generations means higher chance to reach converged and diverse solutions, i.e. better estimation of Pareto front considering both aspects of the quality of the solutions and the variety of the trade-offs they provide. Having few individuals in the population and small number of generations limit the directions of the evolution which leads to immature convergence. Hence, the population size and the number of generations are determined based on the amount of available memory and time as well as complexity of the problem.

In summary, an EA evolves a population of solutions iteratively by making offspring, i.e. children, from promising solutions used as parents. Each cycle completes with selecting the better solutions to be used as the next generation through a survival strategy. The algorithm ends after a certain number of generations or when the required merits are met by the found solutions.

The general flow of an evolutionary cycle is shown in Fig. 2.11. In the first generation, a population is generated randomly or according to a set of previously known solutions if applicable. Then, each individual in the population is evaluated

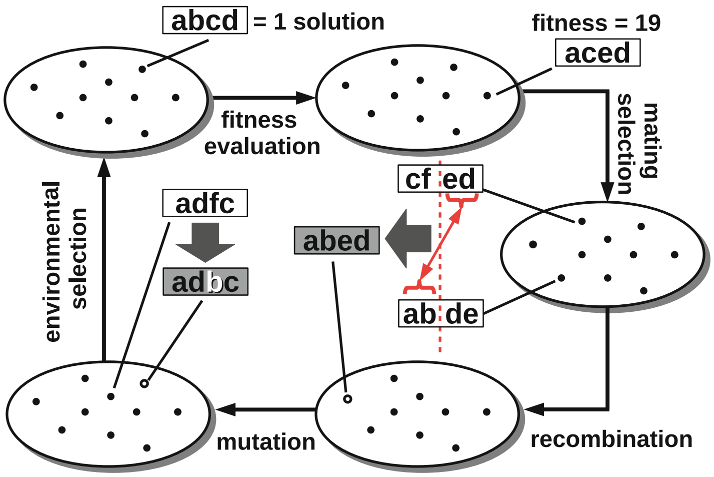


Fig. 2.11 General flow for an evolutionary algorithm [101] with alphabetically encoded solutions with length of four

based on its qualities in the objective space, i.e. assigning objective functions. Having the metrics describing the quality of the solutions, they can be marked with fitness values determining their comparative quality within the population. This stage is shown by *fitness evaluation* in Fig. 2.11. *Mating selection* then chooses parents to make the offspring. This is performed in a probabilistic manner. *Binary tournament* is an approach that is commonly used for mating selection. Binary tournament chooses two individuals randomly, and keeps the one with the better fitness value as a selected parent. This procedure continues until the number of required parents are selected.

After mating selection, the parents are recombined to make an offspring. The most commonly recombination operator among EAs is *binary crossover* which selects two parents randomly and breaks them from one or more points. Then children are made by exchanging different segments from both parents. For example, Fig. 2.11 shows recombination for parents *cfed* and *abde* which are broken from the middle. Then, one child *abed* is made from putting the head of the second parent together with the tail of the first. More complicated crossover operators can be obtained by using three or more parents, i.e. non-binary crossover, or several breaking points.

The probability of crossover is an important parameter for an EA. This probability shows the percentage of the population to which the crossover operators are applied. Without crossover, i.e. 0% crossover probability, the offspring is a full copy of the parents, while a 100% crossover probability means that the whole offspring has been made by crossover. Usually, the probability of crossover is set to a high value hoping that more promising solutions can be obtained by the parents of the current generation.

The offspring created by crossover is still further variated by *mutation*. The mutation operator is considered to randomly change individuals slightly. The purpose of mutation is to ensure higher diversity and allow to escape local minima. As Fig. 2.11 shows an individual encoded alphabetically as *adfc* is mutated by changing the third letter f to b. For a binary encoded solution, i.e. bit string, mutation could be flipping one pit at a random position. Similarly, different mutation operators can be defined for differently encoded population of solutions.

Mutation probability shows the number of times an individual is mutated. This probability is usually set to $\frac{1}{n}$, where n is the length of the vector describing a solution. This means that for example in a binary encoded solution only a single bit is subjected to the mutation operator at a random position.

After applying the mutation operators, the offspring is ready and can be used by *environmental selection* to make the population of the next generation at last stage of the cycle. To make sure that the promising solutions in the current generation are not eliminated, the concept of *elitism* has been suggested. A non-elitist EA selects the next generation only from the offspring, while an elitist EA applies the environmental selection deterministically to the union of parents and offspring sets and keeps the best ones as population of the next generation.

Chapter 3
BDD Optimization and Approximation: A Multi-criteria Approach

3.1 Related Work

Binary Decision Diagrams (BDDs) are extensively used in VLSI CAD for synthesis and verification [29]. Also, *Artificial Intelligence* (AI) benefits from BDDs, e.g., in software model checking [30], sparse-memory applications [43], and enhancing search methods [31].

BDD optimization techniques aim to find an optimal variable ordering which leads to a BDD with minimum cost which is mainly the number of nodes, the so-called BDD size, for most of the applications using BDDs. Many VLSI CAD applications map BDDs directly to target circuits, and therefore, a smaller BDD results in smaller chip area [26, 29]. Also the number of one-paths, i.e. paths evaluating the function to true, is influential in several applications. As an example, in formal verification using SAT-solving, the number of required steps to solve a SAT problem can be measured by the number of paths in BDDs [64]. In logical synthesis it has been shown that a reduction in the number of paths enhances the minimization of *Disjoint-Sum-of-Product* representations which can be directly extracted from BDDs [58, 98]. Besides optimization, approximation as an emerging computing paradigm allows to further minimize BDDs at a cost of inaccuracy for applications in which small variations can be tolerated.

There are several approaches for exact size minimization guaranteeing to find the minimal BDD [28, 33, 42]. Nevertheless, high complexity of the exact methods is a reason for heuristic methods to be of higher interest. Sifting [65] is a well known node minimization algorithm based on a hill-climbing framework. The technique is based on a rule that swaps two adjacent variables in a BDD without changing the function (see Fig. 3.1). Sifting uses this feature by moving each variable downwards and upwards in the variable order. The BDD size resulting from reordering every variable is recorded during this procedure and finally the variable is moved back and fixed to the position where the BDD with the minimum number of nodes was found. Results obtained by Sifting can be far from the optimum when the number of

© Springer Nature Switzerland AG 2020
S. Shirinzadeh, R. Drechsler, *In-Memory Computing*,
https://doi.org/10.1007/978-3-030-18026-3_3

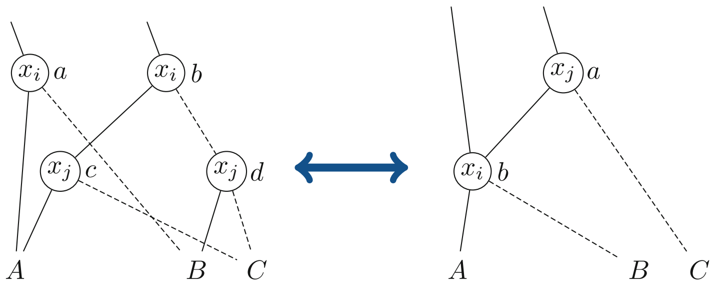


Fig. 3.1 Sifting [65] technique for variable reordering of BDDs [29]

inputs increases. Employing other heuristic approaches such as *Simulated Annealing* (SA) [11] and *Evolutionary Algorithms* (EAs) [24, 73] for BDD node minimization, the number of nodes can even decrease to half of that achieved by Sifting for large circuits.

As this chapter explains the use of EAs for BDD optimization in detail, the framework of SA is explained here briefly. SA starts with generating a random variable ordering and then evaluates the resulting BDD by counting its number of nodes. Before a termination criterion is satisfied, an adjacent ordering is accepted instead of the current variable permutation if it leads to a BDD with a lower number of nodes. Otherwise, the new variable ordering might be accepted in a probabilistic manner. In fact, SA allows to accept a variable ordering with a certain probability in order to make the algorithm capable of escaping a local minimum.

The majority of BDD optimization approaches are based on node minimization, however it has been proven that the number of BDD paths are also important in some applications such as SAT-solving or synthesis [29]. *Modified Sifting* (MS) [32] is a BDD optimization method aiming at minimizing the number of one-paths. The framework of MS is the same as Sifting with the objective being set to the number of one-paths. Similar to Sifting, MS examines just a few possibilities of the whole search space that makes it faster compared to simulation based methods such as SA and EA. Consequently, the BDDs found by MS might have a considerable higher number of one-paths than the optimum when the number of variables increases. An evolutionary algorithm was proposed in [39] for BDD path minimization which was denoted by EA. EA has a better performance than MS for the same reasons mentioned above. EA applies roulette wheel mating selection for creating an offspring with a cardinality of half of the current population. In each iteration, after employing variation operators the offspring is produced and then the new population is created by combining the offspring with the best half of the individuals in the current population.

All of the BDD optimization techniques explained above optimize BDDs with respect to one of the cost metrics. An approach introduced in [57] employs Sifting and MS independently to find a candidate BDD with a trade-off between the number of nodes and one-paths without using a multi-objective framework. The algorithm simply stores the number of nodes and one-paths returned by both methods. Thereafter, a comparison operator checks the node counts found by two approaches. If they are equal, the optimized BDD is characterized by the BDD found by MS, as it may have lower number of one-paths. Otherwise, the same scenario is repeated for equal one-path counts but the BDD found by Sifting is selected this time since the number of nodes are expected to be lower. In the case that both objectives in the BDDs found by two methods are different, the user is supposed to manually choose between the resulting BDDs. The only advantage that this approach provides in comparison with Sifting and MS is the possibility of preferring a BDD with one lower objective when the other objective is identical in both options.

Besides heuristic optimization, approximation of BDDs can be also considered for further gain in error-resilient applications. Approximate computing is an emerging approach that provides higher efficiency with a loss of quality by relaxing the strict requirements. A wide variety of modern applications such as media processing, recognition, and data mining tolerate acceptable error rates that can be exploited by approximate computing circuits and systems to lower the costs in time or energy [37]. In [71], an approach for approximate synthesis of multi-level logic circuits was proposed, which minimizes circuit area for a given error threshold. Another approach has been proposed in [91] that introduces a systematic methodology to formulate the problem of approximate computing and mapping it into an equivalent traditional synthesis problem.

As approximation is becoming an emerging computing paradigm for exploiting the intrinsic error resilience in many of modern applications, the need for approximate computing design methods increases. In [82], this requirement was addressed for applications using BDDs by introducing a minimization approach for approximate computing. The paper proposes approximation operators for eliminating a number of nodes in BDDs as well as several algorithms to compute different error metrics. In this work we have used the same algorithms proposed in [82] for computing the error rate and the worst case error.

3.2 Multi-objective BDD Optimization

3.2.1 Algorithm Features

The proposed multi-objective BDD minimization approach [73] is a non-dominated sorting based algorithm structurally similar to NSGA-II [22] with variation operators specifically designed for escaping invalid variable permutations. NSGA-II has been shown excellent performance on problems with two or three objectives. However, as a Pareto-dominance algorithm it fails to rank solutions in the presence of many objectives. Since BDD optimization is characterized as a bi-objective problem in this chapter, NSGA-II would be sufficient and more reasonable than paying high penalties when using many objective optimization approaches, e.g., [88].

The relation employed by NSGA-II to rank a population is briefly discussed in the following. According to non-dominated sorting, all individuals should be assigned to a level of dominance that is equal to their overall fitness values used for ranking the population. To find all members of the first non-dominated front, each individual should be compared with every other individual in the population to find if it is non-dominated. This procedure can be completed by exploiting a fast non-dominated sorting scheme to find all levels of dominance. Fast non-dominated sorting decreases computational complexity by counting the number of individuals which dominate or are dominated by each individual in the population. To preserve a well-distributed optimal set, density of individuals is also considered besides non-domination rank in the selection process. The density estimation metric so-called crowding distance is calculated to discriminate between individuals belonging to the same non-dominated front. Thus, between two individuals with different non-dominated ranks, the individual with lower rank is preferred. Otherwise, the individual with greater value of crowding distance is preferred in the case of equal ranks.

Algorithm 1: Framework of the general MOB

```
1  $P_0 \leftarrow$ InitializePopulation;
2  CountNodesOne-paths($P_0$);
3  $\{F_1, F_2, \ldots\} \leftarrow$;
4  Non-dominated-sort($P_0$);
5  $t \leftarrow 0$;
6  while the stopping criterion is not met do
7      $Q_t \leftarrow$ MakeOffspringPopulation(P$_{t+1}$);
8      CountNodesOne-paths($Q_t$);
9      $R_t \leftarrow P_t \cup Q_t$;
10     $\{F_1, F_2, \ldots\} \leftarrow$;
11     Non-dominated-sort($R_t$);
12     $P_{t+1} \leftarrow \emptyset$;
13     $i \leftarrow 1$;
14     while $|P_{t+1}| + |F_i| \leq N$ do
15         $P_{t+1} \leftarrow P_{t+1} \cup F_i$;
16         $i \leftarrow i + 1$;
17     end
18     CrowdingDistanceSort($F_i$);
19     $P_{t+1} \leftarrow P_{t+1} \cup F_i[1 : (N - |P_{t+1}|)]$;
20     $t \leftarrow t + 1$;
21 end
```

The general framework of the MOB algorithm is described in Algorithm 1. First, a random parent population P_0 of size N is initialized. Each individual of the population is characterized by a permutation of the variable indices of the BDD. Objective values are assigned to the population by counting the number of nodes, denoted by N, and one-paths, denoted by P. Then, the population is sorted based on non-domination. Steps 6–18 are iterated until the stopping criterion is satisfied. After applying binary tournament, recombination, and mutation operators an offspring Q_t of size equal to the parent population is created in step 6. To ensure elitism, Q_t is combined with the current population P_t resulting in a new population R_t. In step 9, non-dominated sorting is employed to classify R_t into different non-domination levels (F_1, F_2, and so on). Thereafter, individuals are added to P_{t+1} starting from the first non-dominated front F_1. This procedure is continued to fill population P_{t+1} with subsequent non-dominated fronts until their sizes are smaller than the free available slots in P_{t+1} (step 17), formally expressed as

$$P_{t+1} \leftarrow P_{t+1} \cup F_i[1 : (N - |P_{t+1}|)].$$

Otherwise, the lesser crowded individuals in the current front F_i are chosen after sorting the front according to the crowding distance.

3.2.1.1 Variation Operators

In this section, the employed variation operators are introduced. The offspring population is produced by recombination and mutation operators. Recombination includes specific crossover operators which avoid omissions and duplication in variable indices. Indeed, the employed crossover operators ensure that the permutations remain valid. Three crossover operators are used in the proposed BDD optimization algorithm including partially matched crossover, inversion and reproduction. Partially matched crossover (PMX) [60] is performed by randomly choosing two crossover points which break two parents in three sections. Two children are produced by combination of sections from both parents such that each previously used variable index is substituted for a new one to guarantee valid permutations. Inversion introduced in [39] breaks each parent into three sections by choosing two random positions. Then, one child is produced from each parent by inverting the order of indices in a randomly chosen part. The last employed crossover operator only copies each parent with no genetic change. This means the created children are identical to their parents.

After recombination, three mutation operators all described in [39] are employed such that one of them might affect a child according to given probabilities. The first operator exchanges two random variable indices of the individual. The second operator applies the first mutation technique twice on the same permutation. In the third mutation operator a position is chosen randomly and its content is exchanged with an adjacent index.

3.2.1.2 Priority

In many real world applications, one or more objectives are of higher level of significance. In such optimization problems, although it is desirable to minimize all objectives, it is preferred to escape penalties for objectives with higher priorities as a cost of improving lesser important criteria. Hence, the proposed MOB optimization approach is equipped with the ability to handle priority in order to meet requirements of specific applications. For instance, the number of nodes is more important than the number of one-paths in formal verification. On the other hand, the number of paths are regarded as an influential objective in SAT-solving.

Several techniques have been developed to model priorities in a multi-criteria optimization problem. Weighted sum is a widely used classical method that is able to handle objective priorities. It scalarizes the set of objectives into a fitness value by multiplying each objective with a user supplied weight. For a solution $x \in \Omega$, the fitness function f($\mathbf{x}$) is given by

$$f(\mathbf{x}) = \sum_{i=1}^{m} w_i . x_i .$$

It is obvious that larger coefficients, denoted by ω_i, represent higher objective priorities in the fitness value. Another approach so-called *priority preferred* is proposed in [68]. It incorporates a lexicographic ordering of objectives with relation *preferred* that is a refinement of Pareto-dominance [25]. In this work, the model explained in [68] is adapted to non-domination instead of *preferred*. Let $p = \{p_1, p_2, \ldots, p_m\}$ be a priority vector determining priorities assigned to the objectives for an m-objective problem. Each component p_i, $i \in \{1, 2, \ldots, m\}$ can adopt values from the set $\{1, 2, \ldots, n\}$, $n \le m$ (n is equal to m in case that all objectives have different priorities). Considering a minimization problem, we assume that a lower value of p_i means objective i is of higher priority. Given two solutions $\mathbf{x}$, $y \in \Omega$, $\mathbf{x}|_j$ and $\mathbf{y}|_j$ represent subvectors of $\mathbf{x}$ and $\mathbf{y}$ only including objective functions with priority of j, priority-dominance is defined as

$$\begin{aligned} \mathbf{x} \prec_p \mathbf{y} :\Leftrightarrow \exists j \in \{1, 2, \ldots, n\} : \mathbf{x}|_j \prec \mathbf{y}|_j \wedge \\ \forall k < j : \mathbf{y}|_k \not\prec \mathbf{x}|_k . \end{aligned}$$

More informally, the relation defined above employs Pareto-dominance to compare objective functions with equal priorities. In other words, $\mathbf{x}$ priority-dominates $\mathbf{y}$ if there is a subvector of objective functions with identical priority in $\mathbf{x}$ that dominates the corresponding subvector in $\mathbf{y}$, and at the same time $\mathbf{x}|_j$ is not dominated by any subvector of priority value higher than j in $\mathbf{y}$.

Priority-dominance can be defined more simply in the bi-objective BDD optimization problem. In this case, the priority vector consists of two values, i.e. 1 for objective with higher significance and 2 for the other objective. The priority subvector for each individual is equal to the corresponding objective function representing the number of nodes or one-paths. The procedure to compare individuals according to a given priority vector is described in Algorithm 2. In steps 6–13, an individual $\mathbf{x}$ is compared with $\mathbf{y}$ based on priority-dominance. For this purpose, relation dominance is applied to the corresponding objective functions with high priority in $\mathbf{x}$ and $\mathbf{y}$. Obviously, if these values are different $\mathbf{x}$

priority-dominates or is priority-dominated by **y**. Otherwise, the other objective is taken into account to discriminate between individuals. In order to determine the non-domination front for each individual, two entities are also calculated in parallel with the comparisons. S_x represents the set of solutions which are dominated by individual **x**, and n_x is the number of individuals dominating **x**. Thereafter, these entities can be used for fast non-domination sorting as described in [22].

Algorithm 2: Priority-dominance

```
1  GetPriorityVector($p$) ;   // To determine priority subvectors
2  forall the $x \in \Omega$ do
3      $S_x \leftarrow \emptyset$ ;        // Set of individuals dominated by x
4      $n_x \leftarrow 0$ ;   // The number of individuals dominating x
5      forall the $y \in \Omega$ do
6          $i \leftarrow 1$ while $i \leq 2$ do
7              if $x|_i \prec y|_i$ then
8                  $S_x \leftarrow S_x \cup \{\mathbf{y}\}$
9              end
10             else if $y|_i \prec x|_i$ then
11                 $n_x \leftarrow n_x + 1$
12             end
13             else
14                 $i \leftarrow i + 1$
15             end
16         end
17     end
18 end
```

3.2.2 Experimental Results

In order to evaluate the performance of MOB and compare it with other discussed existing approaches, several experiments are carried out on a benchmark set including 25 Boolean functions. The functions are taken from ISCAS89 [14] and LGSynth91 [97] benchmark sets with a range of input variables from 7 to 54. MOB has been run 30 times independently for each function. In all experiments, the population size is set to 100 and the algorithm terminates after 1000 iterations. Similarly to EA, the probabilities of crossover operators are set to 0.98, 0.01, and 0.01 for PMX, inversion and reproduction, respectively. Three mutation operators have an overall probability of $1/n$, where n is the number of variables of the BDD. The CUDD package [85] is used for BDD representation and comparison of results with node minimization approaches.

To evaluate the performance of the general MOB algorithm, i.e. without priority, statistical studies have been performed on the final populations for all 30 runs of each benchmark function. Complete statistics representing the average values, the smallest and the greatest observations of results are shown in Table 3.1. It is worth mentioning that for a function with four or more number of variables, a BDD minimal in both metrics of nodes and paths does not exist necessarily [32]. Thus, the values of nodes and one-paths achieved by MOB might represent the real optimum for one or both objectives or even none of them. Indeed, with no priority to nodes or one-paths, results shown in Table 3.1 are supposed to express a good trade-off between objectives.

In order to analyze the descriptive statistics of results obtained by the general bi-objective MOB, we assume the minimum found for each objective as a metric roughly estimating the optimal value. Statistics of the number of nodes (N) and one-paths (P) shown in Table 3.1 demonstrate that the average values represented by mean and median are closer to the minimum of the number of nodes and paths rather than the maximum values found in the whole runs. For example, for function s1196 mean values for the number of nodes and one-paths are only 10.86% and 13.22% greater than their minimum values, respectively. The worst observations of the number of nodes and one-paths are 70.37% and 67.88% greater than minimums found for the corresponding objectives. This property in the first three statistics is especially apparent for the functions resulting in larger BDDs in size and number of paths. Considering the effect of maximum number of nodes and one-paths on worsening their average values, the discussion above reflects the fact that the final populations are of high quality in comparison with the best ever found values. In general, the number of nodes and paths of the BDDs found by MOB are mostly spread over values expressing an optimal BDD which indicates the quality of results.

As discussed before, the number of nodes or one-paths of BDDs resulting by the general MOB algorithm does not represent the optimum value for each objective necessarily. Instead, the minimum BDD, either in the number of nodes or paths, is accessible for a single-objective method. Therefore, in order to make a fair comparison the results of MOB with priority to the number of nodes or one-paths are compared with the introduced node or path minimization methods, respectively. For this purpose, the BDDs possessing the smallest values for the preferred objective in all 30 runs are chosen as the results of the prioritized MOBs shown in Tables 3.2 and 3.3.

In Table 3.2, the BDDs found by node minimization techniques discussed in Sect. 3.1, Sifting and SA, are compared to the optimum BDDs obtained by MOB with priority to the number of nodes. It can be easily concluded from the table that the number of BDD nodes found by Sifting for the 25 benchmark circuits are far away from the values returned by the two other approaches except for a few cases. However, Sifting obtains smaller one-path counts than the compared methods for a couple of functions that is an unintended result of lower capability of finding the minimized BDDs. On the other hand, SA shows high performance by achieving BDDs with the minimum number of nodes for most of the functions. Considering all results shown in Table 3.2, MOB with priority to the number of nodes shows obviously higher quality than the other two methods with respect

Table 3.1 Descriptive statistics for MOB

		Min		Mean		Median		Max	
Benchmark	#I/O	*N*	*P*	*N*	*P*	*N*	*P*	*N*	*P*
s1196	32/31	611	2581	677.37	2922.21	675	2857	1014	4333
s1238	32/31	627	2518	707.34	2827.9	679	2806	1057	4479
s1488	14/25	373	352	396.17	365.45	402	357	464	517
s208	18/9	50	53	59.52	62.67	60	61	63	202
s27	7/4	10	16	10	16	10	16	10	16
s298	17/20	74	70	74.51	73.4	74	74	77	74
s344	24/26	104	330	104.73	331.45	104	330	110	346
s382	24/27	121	230	138.73	255.82	136	247.5	208	332
s386	13/3	109	57	114.16	64.89	113	65	133	82
s400	24/27	121	230	142.147	252.05	135	247	188	332
s444	24/27	119	230	134.95	261.88	132	258	188	340
s510	25/13	146	153	149.49	166.78	150	159	156	183
s526	24/27	116	157	128.74	164.15	129	161	147	197

(continued)

Table 3.1 (continued)

		Min		Mean		Median		Max	
Benchmark	#I/O	*N*	*P*	*N*	*P*	*N*	*P*	*N*	*P*
s641	54/42	461	1512	533.51	1594.76	527	1593	604	1793
s713	54/42	435	1523	527.23	1647.11	517	1627	631	1936
s820	23/24	221	146	224.94	157.13	224	151	240	180
s832	23/24	220	146	224.75	158.37	225	152	235	179
alu4	14/8	564	1372	573.19	1428.55	572	1372	644	1599
clip	9/5	75	214	75	214	75	214	75	214
misex1	8/7	35	34	35.4	35.24	35	36	37	36
sao2	10/4	81	97	84.43	102.15	85	102	96	111
t481	16/1	21	841	21.94	903.31	21	841	32	1009
cordic	23/2	43	9440	49.41	20,260.62	47	25,797	82	34,393
misex3	14/14	480	1973	545.74	2120.83	536	2056	751	3046
seq	41/35	1208	1766	1275.76	1901.32	1270	1883	1461	2301
AVG		257	1041.64	280.36	1531.52	277.32	1738.5	348.12	2329.2

Table 3.2 Comparison of results with priority to the number of nodes with existing node minimization approaches

	Sifting [65]		SA [11]		MOB	
Benchmark	*N*	*P*	*N*	*P*	*N*	*P*
s1196	642	3511	600	4079	599	3907
s1238	642	3511	600	4079	599	3909
s1488	391	551	369	582	373	407
s208	61	79	41	1867	41	1867
s27	10	17	10	17	10	16
s298	78	73	74	74	74	74
s344	104	330	104	330	104	330
s382	121	315	119	332	119	332
s386	123	70	110	65	110	58
s400	121	315	119	332	121	292
s444	161	447	119	332	119	332
s510	165	206	148	192	146	181
s526	141	368	113	207	115	189
s641	629	2164	408	2883	385	2671
s713	629	2164	408	2883	389	2617
s820	258	184	221	180	221	177
s832	258	184	221	180	221	151
alu4	804	2876	564	1430	564	1430
clip	87	326	75	275	75	214
misex1	35	39	35	39	35	36
sao2	86	97	81	130	81	111
t481	21	1009	21	1009	21	841
cordic	43	38,482	43	38,482	42	27,329
misex3	602	2654	478	3485	478	3408
seq	2163	62,587	1193	2265	1193	2259
AVG	333	4902.36	250.96	2629.16	249.4	2129.12

to both metrics. The average of results for the whole benchmark set obtained by MOB reveals a decrease of 2090 and 39 in the number of nodes in comparison with Sifting and SA. Moreover, MOB with priority to the number of nodes finds BDDs with noticeably lower one-path counts for the majority of the benchmark functions. To make a more precise comparison, the average of the number of one-paths in MOB is 97.36% and 19.02% smaller than the corresponding values in Sifting and SA, respectively. Indeed, MOB with priority to the number of nodes finds better minimized BDDs than the well known node minimization techniques besides achieving a large reduction in the number of one-paths.

Since the number of nodes are assumed to be of higher importance in this experiment, finding minimal BDDs in size is the principal goal of the MOB with priority to the number of nodes. Nonetheless, there are three functions for which MOB fails to find the BDD representations with minimum number of nodes

Table 3.3 Comparison of results with priority to the number of one-paths with existing one-path minimization approaches

	MS [32]		EA [39]		MOB	
Benchmark	*N*	*P*	*N*	*P*	*N*	*P*
s1196	1523	2874	1142	2508	1087	2509
s1238	1523	1874	1105	2508	1064	2509
s1488	500	369	410	352	408	352
s208	62	53	62	53	62	53
s27	13	16	11	16	10	16
s298	91	70	88	70	76	70
s344	104	330	104	330	104	330
s382	152	238	188	230	188	230
s386	158	61	121	57	113	57
s400	152	238	190	230	188	230
s444	154	243	188	230	188	230
s510	184	170	159	153	155	153
s526	153	162	138	156	138	157
s641	768	1700	911	1444	611	1447
s713	768	1700	1458	1447	616	1452
s820	310	155	250	146	230	146
s832	310	155	249	146	230	146
alu4	621	1545	597	1372	572	1372
clip	127	262	75	214	75	214
misex1	42	34	37	34	36	34
sao2	102	99	100	97	86	97
t481	35	1009	50	841	21	841
cordic	49	30,093	171	8332	73	9440
misex3	946	2303	727	1976	577	1973
seq	1398	1841	1499	1744	1391	1744
AVG	409.8	1903.76	401.2	987.36	331.96	1032.08

obtained by SA. For functions s1488, s400, and s526, BDDs found by MOB have in total 8 more nodes than the three BDDs resulted by SA. On the other hand, the average of one-path counts for the mentioned functions shows a decrease of 233 compared to the total number of one-paths in the corresponding BDDs found by SA. Considering the fact that MOB with priority to the number of nodes is a multi-objective algorithm, finding BDDs with a small increase in the node counts as a fair cost of reducing the number of paths is acceptable.

Table 3.3 shows the comparison of results obtained by MOB with priority to the number of one-paths with BDDs found by MS and EA. It can be clearly seen that results obtained by MS are dominated by EA and MOB. Therefore, the quality of BDDs optimized by EA and MOB are compared in the following. According to the table, MOB has found the minimum number of one-paths that were also found by EA for the majority of benchmark circuits. However, for a few functions BDDs obtained by MOB have higher one-path counts than the corresponding BDDs by EA. For functions s1196, s1238, s641, and s713 the BDDs found by MOB show few increases in the numbers of one-paths while the node counts are considerably lowered. Only the function cordic shows a high difference between the numbers of one-paths resulted by EA and MOB. The BDD representing cordic found by MOB is 13% larger than the BDD resulted by EA with respect to the number of one-paths. It is also worth mentioning that the number of nodes in the BDD found by MOB for the function cordic is decreased to less than half of the nodes in the BDD found by EA. Although MOB has almost failed to find the minimal BDD with respect to the significant objective for cordic, it has compensated this failure with a fair reduction in the other objective. For function misex3, MOB has even found a BDD noticeably smaller in size with a lower number of one-paths compared to the BDD obtained by EA.

The total number of one-paths obtained by MOB is almost equal to half of the one-paths achieved by MS with a reduction of 18.99% in the number of nodes. In comparison with EA, MOB shows an increase of 4.52% in the one-path counts that is mainly caused by result of *cordic*. Also, MOB has 17.26% less BDD nodes compared to EA for the whole benchmark set. Although, the results obtained by MOB are slightly worsened in the number of one-paths compared to EA, MOB has noticeably lowered the number of BDD nodes. In general, it is fair to say that MOB with priority to the number of one-paths shows a high performance with respect to both objectives in comparison with existing node minimization techniques.

In order to have a thorough assessment on the experimental results, multiplication of the number of nodes and one-paths is considered as a metric representing the quality of BDDs. In fact, a lower *Node-One-Path-product* (NOP) means a better trade-off between both optimization criteria. The values of NOPs obtained by all types of MOB and the other methods are shown in Table 3.4. According to the table, the general MOB with no priority has achieved the minimum average of all NOP values among all other approaches that reveals its high performance as a multi-objective technique. The average of NOP values of the general MOB is decreased to 92.78%, 27.02%, 43.84%, and 32.28% of the corresponding values in Sifting, SA, MS, and EA, respectively. Moreover, the two other versions of MOB have also considerable lower NOP average values than that of node or path minimization methods. In other words, priority based MOB types find a good trade-off that is usually the minimal BDD with respect to the objective of higher importance besides reducing the value of the other objective.

To evaluate the effect of priority on MOB, average of results with priority to nodes and one-paths are respectively compared to the average of median values of nodes and one-paths obtained by the general MOB. The integration of results shown in Tables 3.1, 3.2, and 3.3 describes that after applying priority to the number of nodes and one-paths, MOB has achieved 10.06% and 40.63% less BDD nodes and one-paths than the general MOB. Since one-path counts are spread over a larger range than nodes, results of MOB with priority to the number of one-paths show higher level of BDD abstraction in respect to one-paths. In general, incorporating

Table 3.4 Comparison of Node-One-Path-products (NOPs)

Benchmark	Sifting [65]	SA [11]	MS [32]	EA [39]	MOB		
					With priority to one-paths	With priority to nodes	General
s1196	2,254,062	2,447,400	4,377,102	2,864,136	2,727,283	2,340,293	1,906,860
s1238	2,254,062	2,447,400	4,377,102	2,771,340	2,669,576	2,341,491	1,889,888
s1488	215,441	217,341	184,500	144,320	143,616	151,811	142,065
s208	4819	76,547	3286	3286	3286	76,547	3286
s27	170	170	208	176	160	160	160
s298	5694	5476	6370	6160	5320	5476	5476
s344	34,320	34,320	34,320	34,320	34,320	34,320	34,320
s382	38,115	39,508	36,176	43,240	43,240	39,508	33,082
s386	8610	7150	9638	6897	6441	6380	6380
s400	38,115	39,508	36,176	43,700	43,240	35,332	32,562
s444	71,967	39,508	37,422	43,240	43,240	39,508	33,082
s510	33,990	28,416	31,280	24,327	23,715	26,426	23,562

s526	51,888	23,391	24,786	21,528	21,666	21,735	20,640
s641	1,361,156	1,176,264	1,305,600	1,315,484	884,117	1,028,335	806,837
s713	1,361,156	1,176,264	1,305,600	2,109,726	894,432	1,018,013	802,382
s820	48,246	39,780	48,050	36,500	33,580	39,117	33,222
s832	48,246	39,780	48,050	36,354	33,580	33,371	32,928
alu4	2,312,304	806,520	959,445	819,084	784,784	806,520	784,784
clip	28,362	20,625	33,274	16,050	16,050	16,050	16,050
misex1	1365	1365	1428	1258	1224	1260	1260
sao2	8342	10,530	10,098	9700	8342	8991	8342
t481	21,189	21,189	35,315	42,050	17,661	17,661	17,661
cordic	1,654,726	1,654,726	1,474,557	1,424,772	689,120	1,147,818	686,774
misex3	1,597,708	1,665,830	2,178,638	1,436,552	1,138,421	1,629,024	1,116,951
seq	135,375,681	2,702,145	2,573,718	2,614,256	2,425,904	2,694,987	2,305,520
AVG	5,953,189.36	588,846.12	765,285.56	634,692.24	505,923.96	503,018.24	429,762.96

priority with MOB has improved the performance of the algorithm with respect to the user preferred objective in addition to preserving favorable multi-objective features.

3.3 Approximate BDD Optimization

3.3.1 Basic Concepts

Approximate BDD optimization aims at minimizing a BDD representing a given Boolean function f by approximating it with a Boolean function $\hat{f}$ and respecting a given threshold based on an error metric. The associated exact decision problem is called *Approximate BDD Minimization* (ABM, [82]) and asks for a given function f, an error metric e, a threshold t, and a size bound b, whether there exists a function $\hat{f}$ such that $e(f, \hat{f}) \leq t$ and $B(\hat{f}) \leq b$.

In this chapter, we consider the frequently used worst-case error and error rate as error metrics e. The *worst-case error*

$$\mathrm{wc}(f, \hat{f}) = \max\{|\,\mathrm{int}(f(x)) - \mathrm{int}(\hat{f}(x))|\ \forall x \in \mathbb{B}^n\}, \tag{3.1}$$

where 'int' returns the integer representation of a bit vector, is the maximum difference between the approximate output value and a correct version for all possible inputs. This metric is sometimes also referred to as *error significance* in the literature. The *error rate*

$$\mathrm{er}(f, \hat{f}) = \frac{\sum\limits_{x \in \mathbb{B}^n} [f(x) \neq \hat{f}(x)]}{2^n} \tag{3.2}$$

is the ratio of the errors observed in the output value as a result of approximation to the total number of input combinations. The product of error rate and the total number of input combinations corresponds to the Hamming distance when applied to single-output functions. The first two metrics consider the integer representation of the function values instead of the binary representation. This is useful since in most applications a change in more significant bits has a much higher effect than changes in less significant bits.

In order to approximate a BDD, we use the operators which can efficiently be implemented using the APPLY algorithm [65]. These operators should (1) change the function and (2) reduce the size of the BDD. The functional change should at best not be too drastically, i.e. the effect on the error metric is kept small. In this chapter, we use co-factoring on some variable x_i, i.e.

$$\hat{f} \leftarrow f_{x_i} \quad \text{or} \quad \hat{f} \leftarrow f_{\bar{x}_i}, \tag{3.3}$$

as the simplest operator, however, with quite a significant effect on the error metric. A more complex algorithm is called *heavy branch subsetting* [63] which we apply partially to the level of x_i and all lower levels (towards the terminals), denoted $\lfloor f \rfloor_{x_i}$ and $\lceil f \rceil_{x_i}$ and called *rounding up* and *rounding down*. For each node that appears at level x_i or lower (in other words for each node labeled x_j with $j \geq i$), the lighter child, i.e. the child with the smaller ON-set, is replaced by a terminal node. The terminal node is $\bot$ (false) when rounding down, and $\top$ (true) when rounding up.

3.3.2 Algorithm Requirements

Approximate BDD optimization has some specific requirements in comparison with a typical multi-objective optimization problem that should be fulfilled. The simplest model for the multi-objective BDD approximate optimization considering the two error metrics described in Sect. 3.3.1 is a three-objective problem with the first objective set to the BDD size. This model might need to avoid occurrence of non-comparable solutions by adding density information due to the presence of three objectives. The model is also not a true projection of the problem for ignoring the main purpose of approximate BDD computing which is finding the BDD with minimum size. Moreover, it does not guarantee to maintain the validity of the approximated BDDs. In this section, we briefly review the possible solutions to make a specific problem based model for the multi-objective optimization for BDD approximation.

By approximate computing we accept to pay some costs as errors to find a smaller BDD which might not exist using only variable ordering techniques. Thus, the error metrics can not be considered as important as the number of BDD nodes. This means that the optimization algorithm should be capable of handling higher priority of the first criterion. A more comprehensive priority scheme which allows different priority scenarios for all objectives might be also of interest for some Boolean functions when one or two of the error metrics highly affect the precision and validity of approximation. For example, for a BDD representing an adder a high value of error-rate might be tolerated if the integer values of the real and the approximated functions are slightly different even for a large number of assignments. While a high value of worst-case error can mean a failed approximation since it shows that the output of the approximated function is not acceptable for at least one input assignment.

Besides the discussion above on the priority of objectives, the algorithm should also guarantee that the final solutions represent valid BDDs according to the user defined standards. Hence, for each error metric a threshold value should be set. It is worth noting that it is sufficient to satisfy the error thresholds. In fact, the algorithm should ensure not to lose the minimum BDD size at a cost of finding error values smaller than required. This has been already considered by giving higher priority to the size of BDDs, but the threshold setting still needs further considerations.

The values of errors caused by approximation highly depend on the characteristics of the functions represented by BDDs. It is almost impossible for a user to anticipate the behavior of a function under approximation [82]. For some functions, a small operation on the BDD under approximation can cause dramatic inaccuracies without a noticeable improvement in the size, whereas high gain can be achievable at low error costs for other functions. In this case, approximate BDD optimization may not lead to the best possible error values. If the error constraints are set too loose, the process will terminate without finding the best possible error values. On the other hand, strict constraints may allow only few individuals in the population which cover a small percentage of the whole search space. This can limit the directions of evolution and prevent to obtain minimal BDDs. Therefore, a proper adaptive error constraint scheme depending on the behavior of the function under approximation can be highly effective on the quality of results.

According to the discussion above on the requirements of the multi-objective BDD approximation, we need to consider a proper relation to discriminate the solutions to this problem. This discriminative relation will replace the strict Pareto dominance relation used for algorithm in Sect. 3.2.1 in order to meet the aforementioned problem specific requirements and speed up the selection procedure because of benefiting from the rough comparisons enabled by approximation [76, 79].

3.3.3 *Algorithm Features*

This section first explains the prioritized ε-preferred relation which is employed for comparison of solutions, and then presents the proposed multi-objective algorithm for the problem of approximate BDD optimization.

3.3.3.1 Prioritized ε-Preferred

As discussed before, the strict relation of Pareto-dominance is not required for the approximate BDD optimization. In this case, we are only interested in finding BDDs with smaller sizes, and the error metrics does not matter until they satisfy the threshold values. To find the ranking method appropriate for this purpose, the so-called relation *preferred* proposed in [25] is first introduced here. Preferred is a refinement of Pareto-dominance which respects the number of objective functions for discriminating solutions. For i and $j \in \{1, 2, \ldots, m\}$, relation preferred is defined as

$$\mathbf{x} \prec_{\text{preferred}} \mathbf{y} :\Leftrightarrow |\{i : f_i(\mathbf{x}) < f_i(\mathbf{y})\}| > |\{j : f_j(\mathbf{y}) < f_j(\mathbf{x})\}|. \tag{3.4}$$

More informally, the equation above means that solution x is preferred to y if the number of objective functions of x which are smaller than their corresponding values of y is larger than the number of objective functions of y smaller than the same components of x.

Relation preferred is not transitive and as a result it can not be a partial order. Consequently, concepts such as non-dominating levels represented for Pareto-dominance [22] where each solution in a level dominates all solutions belonging to the levels with higher fitness values can not be defined for preferred. For a population compared based on preferred, the solutions can be pairwise comparable [25]. To make this more clear we give an example. Let $f(x) = (8, 7, 1)$, $f(y) = (1, 9, 6)$, and $f(z) = (7, 0, 9)$ be solutions mapped into the objective space. According to relation preferred, x is preferred to y and y is preferred to z. For a transitive relation it could be concluded that x has the same relation with z, while here solution z is preferred to x. Indeed, the solutions x, y, and z describe a cycle in comparison with each other.

Solutions inside a cycle are denoted as incomparable and are ranked equally. To sort solutions based on preferred, the population is divided into the so-called *Satisfiability Classes* (SCs, [25]). Each SC represents the fitness value for a set of solutions that are not comparable according to the concept explained above. Solutions of SCs indicated by smaller values are preferred to solutions of SCs with larger indices during selection. However, due to the non-transitivity of relation preferred all solutions belonging to an SC are not necessarily preferred to the entire solutions with larger SC indices. For example, two solutions x and y in two successive SCs might be incomparable but x is classified in a better SC since it has been also incomparable with another solution z which is preferred to y.

In [88], relation *ε-preferred* was proposed to make preferred more robust for many objective optimization problems. ε-preferred adds the parameter ε to enhance the quality of final solutions by specifying limits for each dimension. Comparing two objective vectors $f(x) = (1, 1, 100)$ and $f(y) = (5, 5, 5)$ based on preferred, solution x is ranked better while it might not be recognized efficient by the user for the high value of 100 in one dimension. This problem is addressed by setting maximum values ε_i, $i \in \{1, \dots, m\}$ for each optimization objective.

Using ε-preferred, we first need to compare two given solutions $x, y \in \Omega$ based on the number of times that each solution violates ε. Assuming $i, j \in \{1, \dots, m\}$, a relation *ε-exceed* is defined as

$$\begin{aligned}\mathbf{x} \prec_{\varepsilon\text{-exceed}} \mathbf{y} :\Leftrightarrow\ & |\{i : f_i(\mathbf{x}) < f_i(\mathbf{y}) \wedge |f_i(\mathbf{x}) - f_i(\mathbf{y})| > \varepsilon_i\}| > \\ & |\{j : f_j(\mathbf{x}) > f_j(\mathbf{y}) \wedge |f_j(\mathbf{x}) - f_j(\mathbf{x})| > \varepsilon_j\}|. \qquad (3.5)\end{aligned}$$

According to the equation above solution x ε-exceeds y if the number of times that x exceeds ε in any dimension is smaller than the same count for y. ε-exceed means the same as ε-preferred if the solutions have different ε-exceeding counts, otherwise preferred is required to discriminate between solutions. Using ε-exceed the complete definition of ε-preferred is formally expressed by

$$\begin{aligned}&\mathbf{x} \prec_{\varepsilon\text{-preferred}} \mathbf{y} :\Leftrightarrow \\ &\qquad \mathbf{x} \prec_{\varepsilon\text{-exceed}} \mathbf{y} \vee (\mathbf{y} \not\prec_{\varepsilon\text{-exceed}} \mathbf{x} \wedge \mathbf{x} \prec_{\text{preferred}} \mathbf{y}). \qquad (3.6)\end{aligned}$$

As discussed in Sect. 3.3.2, the main goal of BDD approximation is size minimization which makes priority an undeniable requirement. Considering objective priorities with relation preferred was introduced in [67]. The model incorporates a lexicographical ordering of objectives with respect to a user defined priority vector. To fulfill the requirements of the approximate BDD optimization problem we use the prioritized ε-preferred relation proposed in [27]. Prioritized ε-preferred denoted by $\prec_{\text{prio-}\varepsilon\text{-pref}}$ exploits the same model used in [67] and is able to handle different levels of priorities for all of the optimization objectives.

Let $p = \{p_1, p_2, \ldots, p_m\}$ be a priority vector determining priorities assigned to an m-objective problem. Each component p_i, $i \in \{1, 2, \ldots, m\}$ can adopt values from the set $\{1, 2, \ldots, n\}$, $n \leq m$, where n is equal to m in case that all objectives have different priorities. Considering a minimization problem, we assume that a lower value of p_i means objective i is of higher priority. Given two solutions $x, y \in \Omega$, $x|_j$ and $y|_j$ represent subvectors of x and y only including objective functions with priority of j, prioritized ε-preferred is defined as

$$\begin{aligned}&\mathbf{x} \prec_{\text{prio-}\varepsilon\text{-pref}} \mathbf{y} :\Leftrightarrow \exists j \in \{1, 2, \ldots, n\} : \\ &\qquad \mathbf{x}|_j \prec_{\varepsilon\text{-preferred}} \mathbf{y}|_j \wedge \forall k < j : \mathbf{y}|_k \not\prec_{\varepsilon\text{-preferred}} \mathbf{x}|_k. \qquad (3.7)\end{aligned}$$

The relation defined above employs ε-preferred to compare subvectors of objective functions with equal priorities. x is prioritized ε-preferred to y if there is a subvector of objective functions in x with priority value j that is ε-preferred to the corresponding subvector in y, and at the same time $x|_j$ is not ε-preferred by any subvector of priority higher than j in y.

The proposed multi-objective algorithm adopts the relation prioritized ε-preferred, since it satisfies the requirements of approximate BDD optimization problem, i.e. the higher significance of size and meeting user defined thresholds for the error metrics.

3.3.3.2 Selection of ε

To ensure removal of invalid solutions, they should be ranked at the end of population and are not survived for the next generations. However, this approach fails to sort the population properly in the presence of a large number of invalid solutions which is caused by low error thresholds. ε can be highly effective for the hard task of threshold setting in the problem of BDD approximation. Indeed, error estimation requires high expertise. For many Boolean functions, there might not be any solution satisfying a small value of threshold, and on the other hand for a high threshold the search might end up missing solutions with low error values. Using ε makes it possible to set large threshold values to ensure that the algorithm will not be stuck in finding incomparable invalid solutions, and at the same time, the search is directed toward desired error values considering limits for each error metric.

The value of ε does not matter for the prioritized optimization objective, i.e. the BDD size. As explained before, prioritized ε-preferred performs comparisons based on relation ε-preferred in subvectors of objectives function with equal priorities starting from the most significant criterion. It is clear that comparisons of solutions for a single objective function $\prec_{\varepsilon\text{-preferred}}$ works the same as Pareto-dominance. In fact, $x|_1 \prec_{\varepsilon\text{-preferred}} y|_1$ here can only mean that the BDD size of solution x is smaller than the same metric in y which fulfills the problem requirement of size minimization. Hence, we can simply set ε_1 to any value such as zero.

ε values for each of the two error metrics can be manually set to a fraction of their thresholds. Nonetheless, an automatic adaptive ε selection scheme can be influential on the performance of the algorithm. In [27], it was proposed to assign the averages of objective functions as the ε values for each optimization criterion. An adaptive method is proposed in here which is based on the standard normal distribution of those error values of the parent population which satisfy the thresholds. The ε_i, $i \in \{2, 3\}$ at each generation is equal to $N(\mu_i, \sigma i^2)$, where μ_i and σ_i are respectively the mean value and standard deviation of the set of the valid i^{th} error values. If there is no valid error value for dimension i, which is probable in the early generations, the ε_i is set to the default manual value.

Using the proposed adaptive ε selection method, in the primary generations ε values are larger with a high probability since the objective functions are spread widely over the objective space. As the number of generations increases the ε values decrease due to the fact that the population gets more evolved which leads to the concentration of objective functions on smaller values. This way a higher diversity is provided in the early generations, while convergence tends to be more influential in the final generations.

3.3.3.3 Evolutionary Cycle and Operators

The proposed approach minimizes BDDs by performing both variable reordering and approximation. For this purpose, an elitist genetic algorithm has been employed which general framework is based on the MOB algorithm introduced in Sect. 3.2.1 but exploits relation prioritized ε-preferred instead of Pareto-dominance to find the smallest approximated BDDs with valid error values.

The approximated BDDs resulted by the proposed algorithm are represented by a variable ordering and an approximation vector. Hence, each individual inside the population has two parts which are initialized independently. One part designates the exact BDD by a permutation of input variables of the Boolean function and the other is a vector consisting of pairs designating the approximation operators and the BDD level indices where each operator should be applied. The length of the approximation vector is determined by the maximum allowed number of approximation operators which is defined as an option for initializing the population. The approximation vector in each initial individual might contain a randomly generated number of approximation operators from one to this maximum value.

After initialization, the population is evaluated. First the exact BDDs are created according to the variable orderings, and then the approximation vectors are applied to the corresponding BDDs to create the approximated ones. Thereafter, the objective functions, i.e. the size and error values of approximated BDDs, are set to each individual of the population. Then, the relation prioritized ε-preferred is used to sort the population into satisfiability classes. Binary tournament is used for mating selection and then the variation operators are applied to make an offspring of the same size as the parent population. The union of both parent and offspring populations are then sorted and the best individuals are survived as the next generation according to their fitness values. This procedure is continued until a certain number of iterations.

As mentioned before, the fitness value for each solution is equal to the index of the satisfiability class the solution belongs to. In fact, the density information of population is not considered in selection. By giving higher priority to one of the objectives the search is focused on the best ever found value of the prioritized criterion instead of finding a good distribution of solutions not preferred to each other.

It is obvious that there should be two types of variation operators for the distinct parts of each individual, i.e. both cross over and mutation operators should be specifically defined for the permutations of variable orderings and the approximation vectors to prevent invalid solutions. For crossover of the variable orderings, the *Partially Matched Crossover* [60] is used as explained in Sect. 3.2.1 to guarantee valid permutations. The crossover operator for the approximation vectors selects two positions randomly and breaks the parents into three sections. Then two children are produced by combination of sections from both parents.

The same mutation operators explained in Sect. 3.2.1 are utilized for the mutation of variable orderings. Three mutation operators are defined independently for the approximation vectors. The first operator selects one position in the vector randomly and increments or decrements its content by equal probabilities. The validity of the solutions is also maintained by only allowing changes which keep the indices denoting either approximation operators or BDD levels within their acceptable ranges. The second mutation operator applies the first operator on the same approximation vector twice. The third operator chooses one pair of approximation operator and level index and increments or decrements both with equal probabilities.

3.3.4 Experimental Results

The performance of the proposed algorithm is assessed on a set of multiple-output benchmark set including 20 Boolean functions. The benchmarks are from ISCAS89

[14] with a range of input variables from 13 to 54, and primary outputs from 9 to 52. For each benchmark, the population size is set to three times the number of input variables but not larger than 120. The algorithm terminates after 500 generations and the results shown in Table 3.5 represent the best out of ten independent runs for each benchmark function. The sum of the probabilities of crossover operators for both variable ordering and approximation vectors are set to 1. A probability of $1/n$, where n is the number of input variables of the Boolean function, i.e. the length of the ordering vectors, is equally distributed over the three mutation operators.

The maximum number of times that approximation operators are applied to any solution in the population is set to 3 during all experiments. This gives a fixed length of 6 to all approximate vectors since they are formed of pairs of indices denoting approximation operators and their corresponding BDD levels. Thus, the overall probability for the three mutation operators designed for approximation vectors is set to 1/6. For BDD representation of the benchmark functions and for the implementation of the approximation operators we have used the CUDD package [85].

Table 3.5 shows the results obtained by the proposed adaptive ε-preferred evolutionary algorithm. The results are also compared with non-optimized initially ordered BDDs indicated by #N-initial, BDDs optimized by Sifting reordering technique [65], and the results obtained by the proposed algorithm when ε is set manually, to see the effect of the adaptive scheme.

The threshold values for the error rate is set to 10%, i.e. each output of the approximated BDD can be different from the corresponding output in the exact BDD for a maximum of 10% of the whole input assignments. We have used the same value of 10% as the threshold for the worst case error denoted by WC. According to the definition of the worst case error, such a threshold means that the maximum difference between the equivalent integer values of the output vectors of the approximated BDD and the exact one can not exceed 2^{m-2} for a Boolean function with m outputs. This is equal to the 10% of the largest possible value that the exact function can adopt. The default manual values of ε for each error metric are set to 5%, and as discussed before, the adaptive ε at each generation is selected from the normal distribution of the valid solutions with error values under thresholds.

It should be noted that the results in Table 3.5 with absolute 0 error values, show that no approximation have been performed. Actually, the approximated BDD representations satisfying the defined thresholds have not been reachable for one or both of the error metrics. Therefore, the algorithm converges to optimized BDDs represented by the best found variable orderings and empty approximation vectors.

The experimental evaluations show a noticeable size reduction at a small cost of error for both manual and adaptive versions of the proposed algorithm. More precisely, the adaptive ε approach obtains a size improvement of 68.02% on average in comparison with the initial BDDs. This improvement has been achieved at a low cost of total inaccuracy, i.e. the average of both error metrics, equal to 2.12%, which is insignificant compared to the amount of size reduction.

Comparison of the results with the Sifting reordering technique [65] also shows that the proposed algorithm using both manual and adaptive ε lowers the number

Table 3.5 Experimental evaluation and comparison of approximated BDDs with initially ordered and reordered BDDs by Sifting [65]

			#N-Sifting	Manual ε				Proposed adaptive ε			
Benchmark	#I/O	#N-initial	[65]	#N	ER (%)	WC (%)	Impr. (%)	#N	ER (%)	WC (%)	Impr. (%)
s208	18/9	1033	61	30	9.32	0.19	97.09	29	7.76	0.19	97.19
s298	17/20	125	78	72	0.78	3.12	42.40	73	0.78	3.12	41.60
s344	24/26	206	104	106	6.25	3.12	48.54	103	6.25	0.39	50.00
s349	24/26	206	104	105	0.00	0.00	49.02	104	0.00	0.00	49.51
s382	24/27	168	121	121	0.00	0.00	27.97	119	0.00	0.00	29.16
s386	13/13	281	123	114	0.39	0.19	59.43	110	0.39	0.19	60.85
s400	24/27	168	121	122	2.25	0.19	27.38	126	0.00	0.00	25.00
s420	34/17	262,227	185	63	6.24	<0.001	99.97	63	6.24	<0.001	99.97
s444	24/27	226	161	126	0.00	0.00	44.24	122	0.00	0.00	46.01
s510	25/13	19,076	165	147	0.00	0.00	99.22	146	0.00	0.00	99.23
s526	24/27	232	141	116	0.00	0.00	50.00	119	0.00	0.00	48.70
s641	54/42	1352	629	408	8.98	3.12	69.82	405	2.72	6.83	70.04
s713	54/42	1352	629	403	2.72	6.83	70.19	387	2.72	6.83	71.37
s820	23/24	2651	258	221	0.09	9.76	91.66	219	0.09	9.76	91.73
s832	23/24	2651	258	218	0.09	9.37	91.77	218	0.09	9.37	91.77
s953	45/52	1723	402	102	0.38	5.42	94.08	106	0.19	5.42	93.84
s967	45/52	1755	417	264	9.84	3.61	84.95	259	9.78	3.61	85.24
s1196	32/32	2295	642	627	2.59	<0.001	72.67	617	<0.001	<0.001	73.11
s1238	32/32	2295	642	627	2.59	<0.001	72.67	600	2.59	<0.001	73.85
s1488	14/25	1016	391	375	0.00	0.00	63.09	383	0.00	0.00	62.30
AVG	15,051.90		281.60	218.35	2.62	2.24	67.80	215.40	1.96	2.28	68.02

#I number of inputs, #O number of outputs, #N-*initial* non-approximated BDD size, #N-*Sifting* BDD size obtained by Sifting reordering technique [65], #N BDD size after approximation, ER error rate, WC worst case error, improvement is calculated compared to the initially ordered non-approximated BDD of BDD nodes considerably. The average number of BDD nodes over the entire benchmark set by the proposed adaptive ε-preferred algorithm is reduced by 23.51% compared to the same value obtained by Sifting [65]. The gain achieved in the BDD sizes is more than 10 times the total inaccuracy of 2.12%, which proves that incorporating approximation with BDD optimization, i.e. variable reordering, has been highly effective.

According to Table 3.5, the algorithm using adaptive ε outperforms the manual approach in both BDD sizes and inaccuracy. The difference between the average size improvement achieved by the manual and adaptive ε methods is very small. However, the average error rate by the adaptive approach is reduced by 25.19% compared to the algorithm using the manual ε. This confirms that the adaptive ε approach has been successful in guiding the search towards minimum BDDs with smaller error values.

With error thresholds of 10% it is fair to say both methods could find approximated BDDs with smaller sizes as well as quite low error cases mostly far from the threshold values. This proves that incorporating ε has been highly effective on the quality of results.

It worth noting that an approximation of a BDD meeting a certain threshold does not necessarily exist. This case occurs for $s1488$ so that the search fails to find BDDs with worst case errors below 25% for both manual and adaptive methods. Therefore, the output for $s1488$ is only an optimized BDD defined with an efficient variable ordering vector and a zero approximation vector.

3.3.5 Assessment of ε-Setting Methods

To see the effect of different ε schemes on the quality of results, we have performed experiments on a set of six selected benchmark functions by setting the threshold values for both error rate and the worst case error at 0%, 5%, 10%, 25%, 50%, 75% and 95%. The default manual ε values are set to half of the thresholds for both error metrics and each benchmark has been run five times for each given threshold value for the same number of 500 generations. We have also compared the performance of our algorithm with the adaptive ε scheme proposed in [27], which is the average value of each objective function over the entire parent population at each generation.

Figure 3.2 shows the BDD sizes obtained by the three different ε-selection methods at the given threshold values. We have removed duplication of data points representing BDDs with equal sizes which is the reason for having less than seven measurements in a few plots. It is obvious that the BDDs representing different functions slope down to smaller sizes with different paces. However, all of the plots show that the BDD sizes decrease as the threshold values increase as expected, which is the main purpose of approximation.

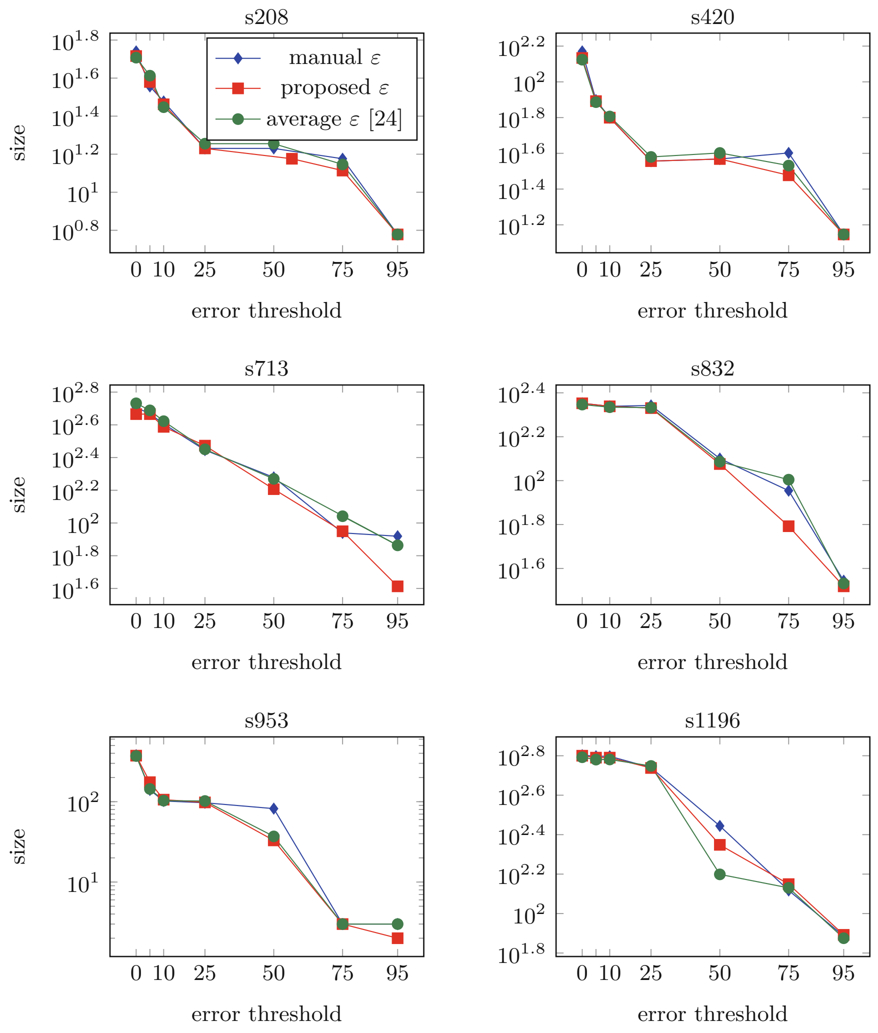


Fig. 3.2 Comparison of different ε selection approaches

A quick look at Fig. 3.2 reveals that smaller BDD sizes have been obtained by the proposed adaptive ε in comparison with the manual ε and average ε [27] approaches. The results by the proposed ε scheme show smaller BDD sizes compared to the two other ε selection methods for all of the functions, except s1196 at threshold 50%. These results confirm that the proposed adaptive ε approach is also influential for the convergence of the BDD sizes as well as the error values.

3.4 Summary

Since many applications in electronic design automation and formal verification successfully use BDDs, their optimization is of high interest. Also, approximation can be exploited to increase the gain in a variety of applications using BDDs which can tolerate some levels of inaccuracy. Both optimization and approximation of BDDs are addressed in this chapter.

The proposed BDD optimization is a bi-objective non-dominated sorting evolutionary algorithm which aims at finding minimal BDDs with respect the number of nodes and one-paths. The proposed algorithm also benefits from an objective priority facility allowing the user to give higher importance to one of the optimization criteria according to the application's requirements. Experimental results show that the multi-objective BDD optimization approach is able to find more compact BDDs considering both objectives in comparison with the existing BDD optimization methods.

The proposed BDD approximation approach is an adaptive ε-preferred evolutionary algorithm for a three-objective optimization problem. The first objective, i.e. the number of nodes of BDDs, was set to a higher priority which projects the main purpose of the BDD approximation. The two other objectives are error metrics measuring the inaccuracy caused by approximation. For given error thresholds, the algorithm is designed to find the BDD with the minimum possible number of nodes and the adaptive ε scheme allows to prefer lower error values for equal BDD sizes. The experimental results show that the proposed approach has accomplished a considerable size reduction of BDDs for small error values.

Chapter 4
Synthesis for Logic-in-Memory Computing Using RRAM

4.1 Related Work

Most of the initial related work for synthesis of logic-in-memory circuits with RRAM devices exploit IMP, which was discussed in detail in Sect. 2.2.1, as the basic operation to compute within RRAM devices. In [54], it was shown that any Boolean function can be implemented within limited number of computational steps using only $N+3$ RRAM devices, where N is the number of input variables. Here, the unfavorable nature of sequential operations for RRAM-based in-memory computing has been mostly exploited to reduce the number of required RRAM devices. In [55], this approach has been improved by reducing the number of RRAM devices by one, such that the target function can be executed using $N + 2$ devices. Both of the approaches presented in [54, 55] require long sequences of computational steps. RRAM devices are of small sizes and also allow higher packaging density for having one terminal less than the MOS transistors. This makes a low area overhead approach at a cost of high number of computational steps disadvantageous.

A realization for a 2-input NAND gate was proposed in [12] based on IMP which requires three resistive switches and executes the NAND function within three computational steps. NAND is a universal logic operation and therefore it allows to implement any Boolean function within a network of NAND gates. In [51], the use of IMP-based logic gates was extended to k-input NOR gates. In this extension, k-input RRAM devices are placed in series on a horizontal nanowire besides an output device in a similar structure to IMP-based NAND gate. For k inputs, the execution of NOR operation requires two computational steps. The first step initializes the output device to zero and the NOR function is executed within the next step by applying voltage levels V_{SET} and V_{COND} similarly to what explained in Sect. 2.2.1 for regular IMP operations.

© Springer Nature Switzerland AG 2020
S. Shirinzadeh, R. Drechsler, *In-Memory Computing*,
https://doi.org/10.1007/978-3-030-18026-3_4

It is obvious that NAND-based and the NOR-based synthesis approaches both can efficiently make use of logic representations, i.e., AIG and *Or-Inverter Graph* (OIG). Also, OIG can also be used for design with NOR gates based on MAGIC [50] and MIG can easily be exploited for MIG-based synthesis using MAJ operation [35]. The efficiency of the logic representations are used by several state-of-the-art which are discussed in the following since the focus of this chapter is on such approaches.

In [17], IMP was used to synthesize combinational logic circuits with resistive memories using OIGs. The approach applies an extension of the delay minimization algorithm proposed in [4] to the OIGs and also uses an area minimization to lower the costs of the equivalent circuits constructed with resistive memories.

The delay minimization algorithm visits the elements of the graph in a topological order starting from the primary inputs and then arranges the children nodes of each multi-input node in a priority queue such that the two children with minimum delay are combined to form a tree. This procedure is repeated until a binary tree is formed. For applying the area minimization the nodes of OIG are sorted topologically and then the nodes of the same topological level are compared to detect the common children and eliminate the replicated nodes. This process is continued until no further node sharing is possible in the graph.

The OIGs are mapped to circuits of resistive memories on a node-by-node basis. For an OIG node with two regular edged children, one operation is first required to invert one of the children to have the negated input of the IMP operation, i.e., $q\prime = \neg p \vee q$. Then, the OR operation is executed by the second IMP. This process is performed similarly for an OIG node with both children complemented. A node with a single complemented edge child is favorable during mapping since there is no need to an additional negation step.

Another approach using AIGs was proposed in [15] for synthesis of in-memory computing logic circuits. The approach uses the state-of-the-art synthesis tool ABC [5] to map an arbitrary Boolean function to an AIG and optimize it. An AIG representing a given function is then mapped to an equivalent network of IMP gates according to the IMP-based realization of NAND gate proposed in [12]. The approach executes a given Boolean function using $N + 2$ RRAM devices such as proposed in [55], where N is the number of input variables of the target Boolean function. Nevertheless, some extra RRAM devices are also considered to maintain values of the IMP gates which have more than one fanout. Using only two RRAM devices in addition to the input devices, the IMP gates which have more than one fanout may lose their information before being used at the fanouts' targets. In this case the data loss due to such fanouts can be resolved by either replicating the logic producing the fanouts' outputs or copying the gate's result in a temporary RRAM device. Authors in [15] have mentioned a scheduling scheme that they use in the presence of lossy fanouts to make a trade-off between the additional required number of computational steps and RRAM devices.

BDD-based synthesis of Boolean functions using resistive memories has been proposed in [16]. Two IMP-based realizations are proposed for a 2-to-1 multiplexer

(MUX) one for a minimum number of resistive switches and the other for a minimum number of steps when lower latency is of higher importance than area. It has not been referred to any BDD optimization method in [16] to lower either the number of RRAM devices or steps. For a given Boolean function, the approach maps the corresponding BDD representation to a netlist of RRAM devices using any of the MUX realizations. This is carried out using two mapping approaches, one fully sequential which is slow but needs a small number of RRAM devices, and the other partially parallel which performs much faster but needs some considerations for the complemented edges and fanouts.

Two methods for serial and parallel implementation of BDDs are used in [16] which lead to different cost metrics. In the serial method, the BDD is traversed in depth-first-manner and each node corresponding to a MUX realization is computed in order. This method decreases the number of required RRAM devices but as a consequence the length of computational steps increases significantly with respect to the size of Boolean function which might not be of interest for many applications. Moreover, a high number of RRAM devices should be considered to prevent missing intermediate results. The parallel method proposed in [16] computes all of BDD nodes in a level at the same time. This way the number of RRAM devices for the mapped BDD is as large as the maximum level size as well as some extra devices required for complemented edges and fanouts. Although, the number of RRAM devices is increased in comparison with the serial implementation, the number of computational steps can be remarkably lowered to the number of BDD levels, i.e., the number of input variables of the given function.

4.2 In-Memory Computing Design with RRAM

The following section presents the proposed synthesis approach for RRAM-based logic-in-memory computing using the three representations explained in Sect. 2.3, i.e., BDD, AIG, and MIG. For each representation, two realizations for a graph node are presented using IMP and MAJ as well as the methodology to map the graph to its equivalent circuit constructed by RRAM devices. Furthermore, optimization algorithms are proposed for each logic representation to lower the cost metrics of the resulting in-memory computing circuits in contrary to the conventional optimization algorithms that are mainly designed to reduce the size, i.e., the number of nodes of the graph also called area, or the depth, i.e., the number of levels of the graph.

The proposed synthesis approach includes (1) realizations of logic primitives with RRAM devices, (2) design methodology, and (3) optimization algorithms. These three design stages besides the experimental results are presented in the rest of this section for each graph based representation in the same order as introduced in Sect. 2.3.

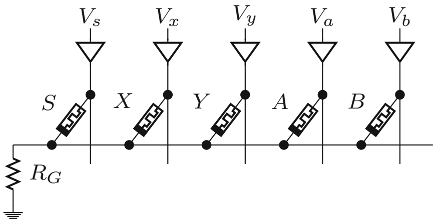


Fig. 4.1 The realization of an IMP-based MUX using RRAM devices [16]

4.2.1 *BDD-Based Synthesis*

4.2.1.1 Memristive Realization for MUX

Figure 4.1 shows the IMP-based realization for the 2-to-1 multiplexer (MUX) proposed in [16]. The implementation requires six computational steps and five RRAM devices of which three, named S, X, and Y, store the inputs, and the two others, A and B, are required for operations and thus their initial values changes during operations. The corresponding implication steps of the MUX realization shown in Fig. 4.1 are as follows:

1: $S = s, X = x, Y = y, A = 0, B = 0$

2: $a \leftarrow s \text{ IMP } a = \bar{s}$

3: $a \leftarrow y \text{ IMP } a = \bar{y} + \bar{s}$

4: $b \leftarrow a \text{ IMP } b = y \cdot s$

5: $s \leftarrow x \text{ IMP } s = \bar{x} + s$

6: $b \leftarrow s \text{ IMP } b = x \cdot \bar{s} + y \cdot s$

In the first step, devices keeping the input variables and the two extra work switches are initialized. The remaining steps are performed by sequential IMP operations that are executed by applying simultaneous voltage pulses V_{COND} and V_{SET}.

To find the MAJ-based realization of MUX, we first express the Boolean function of a multiplexer with majority gates and then simply convert it to MAJ by adding a complement attribute to each gate. For this purpose, the AND and OR operations are represented by majority gates using a constant as the third input variable, i.e., 0 for AND and 1 for OR [1]. Accordingly, a MUX with input variables x, y and a select input s can be expressed as

$$\begin{aligned} x \cdot \bar{s} + y \cdot s &= M(M(x, \bar{s}, 0), M(y, s, 0), 1) \\ &= M(M(x, \bar{s}, 0), M(y, 0, s), 1) \\ &= \text{MAJ}(\text{MAJ}(x, s, 0), \overline{\text{MAJ}}(y, 1, s), 1). \end{aligned}$$

The equations above can be executed by three MAJ operations as well as a negation. Therefore, the MAJ-based realization of the MUX can be obtained by the following operations after a data loading step.

1: $S = s, X = x, Y = y, A = 0, B = 0, C = 1$

2: $P_A = x, Q_A = s, R_A = 0 \Rightarrow R'_A = x \cdot \bar{s}$

3: $P_S = y, Q_S = 1, R_S = s \Rightarrow R'_S = y \cdot s$

4: $P_B = 1, Q_B = s, R_B = 0 \Rightarrow R'_B = \overline{y \cdot s}$

5: $P_C = a, Q_C = b, R_C = 1 \Rightarrow R'_C = x \cdot \bar{s} + y \cdot s$

The proposed MAJ-based MUX can be realized quite similarly to the IMP-based circuit shown in Fig. 4.1 without the need to the load resistance such that the bottom electrodes of the memristive switches are electrically connected via a horizontal nanowire and the switching can be done by applying voltage levels to the top and bottom electrodes. As can be seen, the MAJ-based realization of MUX needs one more RRAM device and one less step. Considering area and delay two equally important cost metrics, using IMP or MAJ does not make a difference in the circuits synthesized by the proposed BDD-based approach. Indeed, the MAJ-based realization of BDD nodes allows faster circuits, while the IMP-based realization leads to circuits with smaller area consumption. Such property in both realizations can be exploited when higher efficiency in delay or area is intended.

4.2.1.2 Design Methodology

In order to escape heavy delay penalties, we assume parallelization per level for BDD-based synthesis [16, 74]. As explained before, in the parallel implementation, each time one BDD level is evaluated entirely starting from the level designating the last ordered variable to the first ordered variable the so-called root node. This is performed through transferring the computation results between successive levels, i.e., using the outputs of each computed level as the inputs of the next level. Using IMP, the results of previous levels are read and copied wherever required within the first loading step of the next level, while for executing MAJ the results are read and then applied as voltages to the rows and columns.

Regardless of the possible fanouts and complemented edges in the BDD, the number of RRAM devices required for computing by this approach is equal to five or six times the maximum number of nodes in any BDD level. In a similar way, the number of computational steps is six or five times the number of BDD levels, for the IMP-based and MAJ-based realizations, respectively. A multiple row crossbar architecture entirely based on resistive switches can be used to realize the presented parallel evaluation.

Table 4.1 The cost metrics of logic representations for RRAM-based in-memory computing

Metric	Definition\value
N$_i$	No. of nodes in the ith level
CE$_i$	No. of ingoing complemented edges in the ith level
RE$_i$	No. of ingoing regular edges in the ith level
FO	Maximum no. of nonconsecutive fanouts in any BDD level
D	The depth of the graph
L$_{CE}$	No. of levels with ingoing complemented edges
L$_{RE}$	No. of levels with ingoing regular edges
R	No. of RRAM devices
S	No. of computational steps
#R	*BDD:* $\max\limits_{0\le i\le D}(K\cdot N_i + CE_i) + FO \quad \text{IMP}: K=5,\ \text{MAJ}: K=6$
	AIG: $\text{IMP}: \max\limits_{0\le i\le D}(3\cdot N_i + RE_i) \quad \text{MAJ}: \max\limits_{0\le i\le D}(3\cdot N_i + CE_i)$
	MIG: $\max\limits_{0\le i\le D}(K\cdot N_i + CE_i) \quad \text{IMP}: K=6,\ \text{MAJ}: K=4$
#S	*BDD:* $K\cdot D + L_{CE} \quad \text{IMP}: K=6,\ \text{MAJ}: K=5$
	AIG: $\text{IMP}: 3\cdot D + L_{RE} \quad \text{MAJ}: 3\cdot D + L_{CE}$
	MIG: $K\cdot D + L_{CE} \quad \text{IMP}: K=10,\ \text{MAJ}: K=3$

The cost metrics of the proposed BDD-based synthesis approach are given in Table 4.1. However, the larger part of the costs representing area and delay of the resulting circuits are explained above, some additional RRAM devices addressing complemented edges and fanouts are still required.

Every complemented edge in the BDD requires a NOT gate to invert its logic value. As shown in the computational steps for both IMP-based and MAJ-based realizations, inverting a variable can be executed after an operation with a zero loaded RRAM device (see step 2 in the IMP-based and step 4 in the MAJ-based realization descriptions for MUX). Accordingly, for each MUX with a complemented input an extra RRAM device should be considered and set to FALSE ($Z = 0$) that can be performed in parallel with the first loading step without any increase in the number of steps. Then, an IMP or MAJ operation should be executed to complete the logic NOT operation. It is obvious that the required computational steps for all complemented edges in a level can be carried out simultaneously that means for any level with ingoing complemented edges only one extra step is required. This implies that the number of additional steps required for inverting all of the complemented edges can not exceed the number of BDD levels. Therefore, the number of steps to evaluate a BDD possessing complemented edges is equal to the number of BDD levels with ingoing complemented edges besides the basic value required for the level counts [74].

It is obvious that the RRAM devices keeping the outputs of each BDD level can be reused and assigned to the inputs of the next successive level. Nevertheless, the results of nodes targeting levels which are not right after their origin level might be lost during computations if their corresponding RRAM devices are rewritten by the next operations. Thus, we consider extra RRAM devices for such nonconsecutive fanouts to retain the result of their origin nodes to be used as an input signal of their target nodes. The required number of RRAM devices for this is equal to the maximum number of such fanouts over all BDD levels. This will not increase the number of steps because copying the results of nodes with nonconsecutive fanouts in additional RRAM devices and using the stored value in the fanouts' targets can be performed simultaneously in the first data loading step of nodes on the both sides of the fanouts.

4.2.1.3 Optimization

Optimization of BDDs in this work is carried out as a bi-objective problem aiming at minimizing the number of RRAM devices and computational steps simultaneously, i.e., finding a trade-off between the number of RRAM devices and computational steps of the resulting circuits. For this purpose, we have exploited MOB, i.e., a multi-objective genetic algorithm which framework was explained in detail in Sect. 3.2.1. The general framework of the employed optimization algorithm is based on a non-dominated sorting genetic algorithm [22] which has been experimentally proven useful for solving NP-complete problems such as BDD optimization [73, 74]. MOB is also capable of handling higher priority to any of the cost metrics, which allows to design smaller in-memory computing circuits at a fair cost of latency or vice versa. The reader is referred to Sect. 3.2.1 for the details of MOB.

Figure 4.2 shows an example with two BDDs both representing a 4-variable 2-output Boolean function. The left BDD has the initial ordering, whereas the second

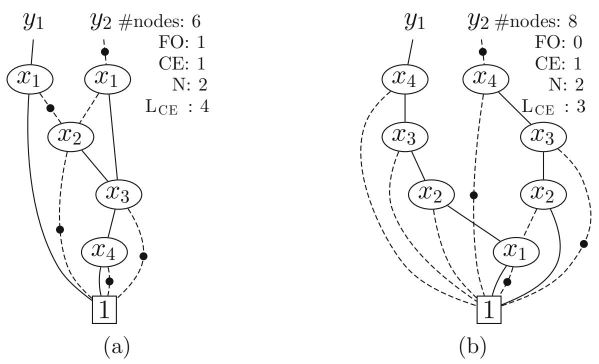


Fig. 4.2 Cost metrics of RRAM-based in-memory computing for an arbitrary BDD, (**a**) before optimization (initial), $\#R = 12$, $\#S = 28$, and (**b**) after optimization (optimized), $\#R = 11$, $\#S = 27$

BDD has the ordering obtained by MOB. The number of required RRAM devices for computing BDD levels ($N + CE$) (see Table 4.1) is equal before and after optimization since both BDDs have a maximum number of two nodes and one ingoing complemented edge. However, there is a nonconsecutive fanout of node x_3 targeting x_1 before optimization requiring an extra RRAM device to maintain the intermediate result. In the optimized BDD the inputs of all of the nodes come from the consecutive levels or the constant 1 which has reduced the number of required RRAM devices by 1. The number of computational steps has been also reduced after optimization since one level has been released from complemented edges.

As can be seen, the numbers of RRAM devices and computational steps decrease although the number of BDD nodes increases. The effect of BDD optimization sounds to be too small for the example function by reducing each one of the cost metrics only by one. Nevertheless, this reduction can be much more visible for larger functions due to the higher possibility of finding BDDs with smaller number of nonconsecutive fanouts, complemented edges, and level sizes caused by larger search space.

4.2.1.4 Evaluation and Comparison

The proposed BDD-based synthesis approach is evaluated using a set of 25 benchmark functions selected from LGsynth91 [97]. The results have been also compared with the similar existing approach [16] introduced in Sect. 3.1, which use the same data structures for RRAM-based in-memory computing. For each benchmark function, MOB has been run 10 times with a termination criterion of 500 generations. The population is three times as large as the number of input variables of each function with a maximum allowed size of 120. The rest of the experimental setup including the genetic operators and their probabilities are the same as used in Sect. 3.2.2.

Table 4.2 presents the results of the three versions of MOB and compares them with results of the BDD-based synthesis approach proposed in [16]. For MOB with priority to the number of RRAM devices and computational steps, we chose results with the smallest number of RRAM devices and steps among all runs and populations. The results shown in the table for the general MOB have been also selected such that they represent a good trade-off between the minimum and maximum values found by the prioritized algorithms. It is worth mentioning that the runtime varies between 0.56 to 187.22 s for the benchmark functions 5xp1_90 in the shortest case and seq as the longest, respectively.

According to Table 4.2, the number of RRAM devices obtained by MOB with priority to R for the IMP-based realization is reduced by 5.74% on average compared to the corresponding value by the general MOB. MOB with priority to the number of steps also achieves smaller latency by reducing the average step count up to 0.74% in comparison to the general MOB. It should be noted that optimization can not noticeably lower the number of computational steps. As shown in Table 4.1, the main contribution to the step count is the number of BDD levels which is most

Table 4.2 Comparison of results of the BDD-based synthesis approach by general and prioritized MOB with Chakraborti et al. [16]

				MOB				MOB with priority to R				MOB with priority to S			
		Chakraborti et al. [16]		IMP		MAJ		IMP		MAJ		IMP		MAJ	
Benchmark	PI/PO	R	S	R	S	R	S	R	S	R	S	R	S	R	S
5xp1_90	7/10	84	73	57	49	62	42	57	49	62	42	65	42	62	42
alu4_98	14/8	642	334	697	93	833	78	590	96	847	77	1014	77	1069	77
apex1	45/45	1626	705	1139	282	1334	236	1082	283	1312	237	3040	277	3185	232
apex2	39/3	122	237	165	235	186	196	172	237	186	196	188	235	173	196
apex4	9/19	2073	447	2028	63	2558	54	2028	63	2558	54	2224	62	2588	53
apex5	117/88	806	888	537	772	582	654	509	775	582	654	591	771	704	654
apex6	135/99	770	1169	179	814	189	679	160	815	180	679	220	813	235	678
apex7	49/37	290	437	118	328	143	279	102	332	134	280	190	328	151	279
b9	41/21	125	298	64	267	73	226	59	269	79	226	77	267	92	226
clip	9/5	120	89	93	63	107	54	93	63	107	54	93	63	107	54
cm150a	21/1	56	127	28	127	30	106	28	127	30	106	28	127	30	106
cm162a	14/5	46	102	42	91	42	78	32	93	36	79	38	89	43	75
cm163a	16/5	42	116	29	112	31	92	29	112	30	92	31	108	32	92
cordic	23/2	32	149	25	140	24	117	21	140	24	117	26	140	29	117
misex1	8/7	83	69	79	50	91	42	59	52	69	44	79	50	91	42
misex3	14/14	444	185	436	91	503	77	436	91	503	77	681	86	781	72
parity	16/1	23	113	6	112	7	96	6	112	7	96	6	112	7	96
seq	41/35	1566	692	1199	250	1320	210	1129	251	1320	210	1207	248	1398	207
t481	16/1	26	107	12	100	14	84	12	100	14	84	16	100	30	84
table5	17/15	580	168	667	109	768	92	637	113	763	95	1346	107	1511	90
too_large	38/3	282	232	214	229	174	191	164	232	232	191	182	229	212	191
x1	51/35	230	398	185	333	219	282	172	339	203	289	186	333	217	282
x2	10/7	60	80	45	65	52	55	42	67	49	57	45	65	52	55
x3	135/99	770	1169	221	813	252	678	161	815	191	679	215	813	252	678
x4	94/71	401	642	177	573	324	479	177	573	231	481	209	573	333	479
AVG		451.96	361.04	337.67	246.44	396.72	207.08	318.28	247.96	389.96	207.84	479.88	244.6	535.36	206.28

PI/PO number of primary inputs/number of primary outputs, *R* number of RRAM devices, *S* number of computational steps probably equal to the number of input variables of the target function and therefore hard to reduce, unless one or more variables are entirely removed from the BDD representation.

In comparison with results of [16] our BDD-based synthesis approach has achieved better performance in both cost metrics. The average values of results over the whole benchmark set by the general MOB for the IMP-based realization, which is also used by [16], shows reduction of 21.11% and 31.74% in the number of RRAM devices and computational steps, respectively. The reduction in the number of steps reaches 42.64% for the MAJ-based realization which also has 12.22% smaller number of RRAM devices.

4.2.2 AIG-Based Synthesis

4.2.2.1 Memristive Realization for AND/NAND Gate

Realization of NAND gate using resistive switches based on material implication has been proposed in [12]. The proposed NAND gate in [12] corresponds to a node with complemented fanout in an AIG and therefore can be utilized as the IMP-based implementation realizing AIGs with RRAM devices. In this case, a FALSE operation is required for any regular edge in the graph. The implementation proposed in [12] requires three resistive memories connected by a common horizontal nanowire to a load resistor, i.e., structurally similar to the circuit shown in Fig. 4.1 with a different number of switches. The interaction of the tri-state voltage drivers on the RRAM devices execute the NAND operation within three computational steps listed below.

$$\textbf{1:}\ X = x, Y = y, A = 0$$

$$\textbf{2:}\ a \leftarrow x \text{ IMP } a = \bar{x}$$

$$\textbf{3:}\ a \leftarrow y \text{ IMP } a = \bar{x} + \bar{y}$$

Using MAJ, AIG can be also implemented with equal number of RRAM devices and computational steps. A majority operation of two variables x and y together with a constant logic value of 0 ($M(x, 0, y)$) [1] executes the AND operation. This corresponds to MAJ$(x, 1, y)$ which only needs one extra operation to preload operand y in a resistive switch. The required steps are as follows

$$\textbf{1:}\ X = x, Y = y, A = 0$$

$$\textbf{2:}\ P_A = y, Q_A = 1, R_A = 0 \Rightarrow R'_A = y$$

$$\textbf{3:}\ P_A = x, Q_A = 1, R_A = y \Rightarrow R'_A = x \cdot y.$$

4.2.2.2 Design Methodology

Although both of the realizations using IMP and MAJ for the AIG-based synthesis approach impose sequential circuit implementations, they allow a reduction in area by reusing RRAM devices released from previous computations. According to the parallel evaluation method, we only consider one AIG level each time, such that the employed RRAM devices to evaluate the level can be reused for the next levels. Starting from the inputs of the graph, the RRAM devices in a level are released when all the required computational steps are executed. Then, the RRAM devices are reused for the upper level and this procedure is continued until the target function is evaluated. Depending on the use of IMP or MAJ in the realization, such an implementation requires as many NAND or AND gates as the maximum number of nodes in any level of the AIG. Hence, the corresponding number of RRAM devices and computational steps for synthesizing the AIG is three times the number of required majority gates and three times the number of levels, respectively.

However, still some additional RRAM devices should be allocated for the required NOT operations, i.e., the regular edges in the IMP-based realization, where the outputs of AIG nodes are already negated due to being implemented by NAND gates, and the complemented edges in the MAJ-based realization. Table 4.1 shows the number of RRAM devices and computational steps of the resulting RRAM-based circuits. Since the implementation starts from the input of AIG, the ingoing regular edges for the IMP-based realization, and the ingoing complemented edges for the MAJ-based realization of any level should be first inverted similarly to the procedure explained for BDDs. Therefore, the total number of RRAM devices required for the synthesis of the whole graph for both IMP-based and MAJ-realizations is equal to the maximum of three times the number of nodes in the level plus the number of ingoing edges to be inverted over all AIG levels.

4.2.2.3 Optimization

For AIG optimization we have used ABC [5] commands. To address the area of the resulting circuits using RRAM devices we use the command *dc2* which minimizes the number of nodes in the graph. Latency of the circuits has been also reduced before mapping them to their corresponding netlist of RRAM devices by another ABC command $if - x - g$. The command minimizes the depth of AIG which is actually the most significant term in the required number of computational steps given in Table 4.1 due to the factor of three for both IMP-based and MAJ-based realizations. Both of the area and depth AIG rewriting commands by ABC do not target the extra number of RRAM devices and computational steps caused by the NOT operations for synthesis. Nevertheless, applying any of the aforementioned commands iteratively can noticeably reduce the cost metrics of RRAM-based in-memory computing.

It should be noted that we can not optimize AIG for both cost metrics since area minimization leads to worsening the latency and on the other hand depth minimization increases the number of nodes in the graph. Thus, according to the application one can choose the optimization command regarding the area or delay of the resulting circuits.

4.2.2.4 Evaluation and Comparison

Results of the proposed AIG-based synthesis approach for in-memory computing are presented in Table 4.3 for both area and depth rewriting methods by ABC [5]. A quick look at Table 4.3 reveals that the number of RRAM devices is smaller for the

Table 4.3 Results of AIG-based synthesis using size and depth rewriting by ABC [5]

	Area minimization				Depth minimization			
	IMP		MAJ		IMP		MAJ	
Benchmark	*R*	*S*	*R*	*S*	*R*	*S*	*R*	*S*
5xp1_90	60	38	45	39	131	35	133	35
alu4_98	692	78	696	79	1046	71	1010	71
apex1	1513	67	1346	67	2192	55	1816	55
apex2	171	83	189	87	302	70	325	71
apex4	2153	67	2040	67	2710	55	2282	55
apex5	698	51	630	51	1033	55	831	55
apex6	639	54	685	55	791	47	681	47
apex7	147	59	147	58	228	47	196	47
b9	123	31	123	31	128	27	123	27
clip	78	37	82	37	176	39	176	39
cm150a	63	36	72	39	95	29	81	28
cm162a	42	39	42	39	42	31	42	31
cm163a	48	35	48	35	48	27	48	27
cordic	71	37	73	39	100	37	108	37
misex1	49	31	55	31	66	25	70	27
misex3	900	79	820	79	1239	67	1105	67
parity	64	38	64	43	64	59	64	67
seq	1040	86	1000	87	1723	71	1453	71
t481	48	25	48	27	100	37	92	37
table5	1155	83	869	83	1878	71	1466	71
too_large	153	80	159	83	279	82	321	83
x1	267	39	271	39	382	39	410	39
x2	45	27	35	27	53	27	55	27
x3	573	54	699	55	801	37	711	37
x4	298	37	282	37	371	31	301	31
AVG	443.6	51.6	420.8	52.56	639.12	46.84	556	47.28

MAJ-based realization, while the operation counts are almost equal. According to Table 4.3, area and depth rewriting reduce the total number of RRAM devices and operations respectively by 24.31% and 10.04% on average compared to each other.

Table 4.4 makes a comparison with the AIG-based approach proposed in [15] for a different benchmark set with single output functions, i.e., $PO = 1$. Since the number of required RRAM devices for the benchmark set are not given in [15], we can only compare with respect to the number of operations. The number of operations obtained by our proposed method using the IMP-based realization, which is also used by [15], is 7 times smaller than that of [15] for both rewritings. Furthermore, the method proposed in [15] fails to keep the number of computational steps at a reasonable value when the number of inputs increases. For example, the number of operations by [15] for functions sym10 and t481 is equal to 1172 and

Table 4.4 Comparison of results by the proposed AIG-based synthesis with Bürger et al. [15]

		Bürger et al. [15]	Area minimization				Depth minimization			
			IMP		MAJ		IMP		MAJ	
Benchmark	PI	S	R	S	R	S	R	S	R	S
9sym_d	9	1418	231	71	225	71	370	62	342	62
con1f1	7	18	18	18	18	19	18	21	18	21
con2f2	7	19	15	22	15	23	23	19	25	18
exam1_d	3	12	9	17	9	19	9	17	9	19
exam3_d	4	12	12	21	12	23	19	18	21	18
max46_d	9	427	138	61	134	63	195	49	177	51
newill_d	8	50	24	33	24	34	59	41	61	42
newtag_d	8	21	24	23	24	22	24	27	24	27
rd53f1	5	27	16	21	16	23	26	26	22	27
rd53f2	5	57	23	31	25	31	44	26	44	27
rd53f3	5	32	16	24	16	27	16	24	16	27
rd73f1	7	238	79	47	81	47	127	42	153	43
rd73f2	7	46	24	24	24	27	24	31	24	35
rd73f3	7	104	29	34	27	35	66	38	62	38
rd84f1	8	351	128	54	128	55	167	51	153	51
rd84f2	8	47	32	24	32	27	32	31	32	35
rd84f3	8	23	24	15	24	12	24	19	24	15
rd84f4	8	345	127	51	137	50	186	51	182	50
sao2f1	10	102	31	48	30	50	68	42	68	43
sao2f2	10	112	63	34	65	35	103	42	97	43
sao2f3	10	380	63	54	62	54	108	54	100	55
sao2f4	10	252	66	53	62	54	126	51	122	51
sym10	10	1172	375	79	401	79	575	66	569	67
t481	16	1564	277	93	275	95	500	78	452	79
xor5	5	32	16	24	16	27	16	24	16	27
AVG		274.44	74.2	39.04	75.28	40.08	117	38	112.52	38.84

1564, respectively. While using our method, both functions can be synthesized with less than 80 operations.

It is worth mentioning that the runtime for each benchmark function in both Tables 4.3 and 4.4 is in the range of milliseconds.

4.2.3 *MIG-Based Synthesis*

This section presents the approach for MIG-based synthesis of logic-in-memory computing circuits. The presented approach is based on a realization proposed for the majority gate with RRAM devices and its corresponding MIG mapping methodology. Then, several MIG optimization algorithms are presented with respect to area and delay of the resulting implementations, i.e., the number of RRAM devices and computational steps, respectively [75].

4.2.3.1 Memristive Realization for Majority Gate

Two realizations are proposed for majority gate based on IMP and MAJ [75, 81]. The proposed IMP-based realization of majority gate is similar to the circuit shown in Fig. 4.1 with six RRAM devices. The realization requires ten sequential steps to execute the majority function. The corresponding steps for executing the majority function are as follows

01: $X = x, Y = y, Z = z$
$A = 0, B = 0, C = 0$
02: $a \leftarrow x \text{ IMP } a = \bar{x}$
03: $b \leftarrow y \text{ IMP } b = \bar{y}$
04: $y \leftarrow a \text{ IMP } y = x + y$
05: $b \leftarrow x \text{ IMP } b = \bar{x} + \bar{y}$
06: $c \leftarrow y \text{ IMP } c = \overline{x + y}$
07: $c \leftarrow z \text{ IMP } c = \overline{x \cdot z + y \cdot z}$
08: $a = 0$
09: $a \leftarrow b \text{ IMP } a = x \cdot y$
10: $a \leftarrow c \text{ IMP } a = x \cdot y + y \cdot z + x \cdot z.$

Three RRAM devices denoted by X, Y, and Z keep the input variables and the remaining three other RRAM devices A, B, and C are required for retaining the intermediate results and the final output. In the first step, the input variables are loaded and the other RRAM devices are assigned to FALSE to be used later for the next operations. Another FALSE operation is also performed in step 8, to clear an RRAM device which is not required anymore for inverting an intermediate result which is not required anymore. Finally, the Boolean function representing a majority gate is executed by implying results from the seventh and ninth step.

It is obvious that the MAJ-based majority gate can be realized with smaller number of RRAM devices and computational steps due to benefiting from the discussed built-in majority property. Using MAJ, the majority gate will require only four RRAM devices that can be placed in the same structure shown in Fig. 4.1. Furthermore, the majority function can be executed within only three steps carrying out simple operations. The MAJ-based computational steps for the proposed RRAM-based realization are

1: $X = x, Y = y, Z = z, A = 0$

2: $P_A = 1, Q_A = y, R_A = 0 \Rightarrow R'_A = \bar{y}$

3: $P_Z = x, Q_Z = \bar{y}, R_Z = z \Rightarrow R'_Z = M(x, y, z)$.

In the first step, the initial values of input variables as well as an additional RRAM device are loaded by applying voltage levels to their voltage divers. Step 2 executes the required NOT operation in RRAM device A. This can be done with applying appropriate voltage levels to the terminals of the switch A, for cases $y = 0$ and $y = 1$, respectively. In the last step, the majority function is executed by use of MAJ within RRAM device Z.

4.2.3.2 Design Methodology

Although the realization imposes sequential circuit implementation, it allows a reduction in area by reusing RRAM devices released from previous computations. The presented synthesis approach considers one MIG level at each time, such that the employed RRAM devices to evaluate the level can be used for other levels. Starting from the input of the graph, the RRAM devices in a level are released when all the required instructions are executed. Then, the RRAM devices are reused for the upper level and this procedure is continued until the target function is evaluated. As explained before for the BDD and AIG-based approaches, such an implementation requires as many majority gates as the maximum number of nodes in any level of the MIG. Hence, depending on the use of IMP or MAJ in the realization, the corresponding number of RRAM devices for synthesizing the MIG is six times or four times the number of required majority gates, and the number of computational steps is ten times or three times the number of levels, respectively. As explained before, still some additional RRAM devices are needed in the presence of complemented edges. Table 4.1 shows the number of RRAM devices and instructions of the resulting in-memory computing implementations.

4.2.3.3 Optimization

We are interested in compact MIG representations because they translate into smaller and faster physical implementations. In order to manipulate MIGs and reach advantageous MIG representations, a dedicated Boolean algebra was introduced in [1]. The axiomatic system for the MIG Boolean algebra, referred to as Ω, is defined by the five primitive axioms.

The axioms are inspired from median algebra [41] and the properties of the median operator in a distributive lattice [9]. A strong property of MIGs and their algebraic framework concerns reachability. It has been proved that by using a sequence of transformations drawn from Ω it is possible to traverse the entire MIG representation space [3]. In other words, given two equivalent MIG representations, it is possible to transform one into the other by just using axioms in Ω. This is of high interest to logic synthesis because it guarantees that the best MIG can always be reached. Unfortunately, deriving a sequence of Ω transformations is an intractable problem. As for traditional logic optimization, heuristic techniques provide here fast solutions with reasonable quality [21].

$$\Omega \begin{cases} \textbf{Commutativity} - \Omega.C \\ M(x, y, z) = M(y, x, z) = M(z, y, x) \\ \textbf{Majority} - \Omega.M \\ M(x, x, z) = x \\ M(x, \bar{x}, z) = z \\ \textbf{Associativity} - \Omega.A \\ M(x, u, , M(y, u, z)) = M(z, u, M(y, u, x)) \\ \textbf{Distributivity} - \Omega.D \\ M(x, y, M(u, v, z)) = M(M(x, y, u), M(x, y, v), z) \\ \textbf{Inverter Propagation} - \Omega.I \\ \overline{M}(x, y, z) = M(\bar{x}, \bar{y}, \bar{z}) \end{cases}$$

Although Ω axioms are sufficient to transform a given MIG to any equivalent one, the length of the transformation sequence might be impractical to execute. To solve this problem, a more advanced set of transformations derived from the basic rules in Ω was proposed in [1] which is denoted by Ψ. The following set includes those axioms of Ψ that are used in this chapter.

$$\Psi \begin{cases} \textbf{Relevance - } \boldsymbol{\Psi.R} \\ M(x, y, z) = M(x, z, z_{x/\bar{y}}) \\ \textbf{Complementary Associativity - } \boldsymbol{\Psi.C} \\ M(x, u, M(y, \bar{u}, z)) = M(x, u, M(y, x, z)) \end{cases}$$

where $z_{x/\bar{y}}$ means replacing x with $\bar{y}$. We refer the reader to paper [3] for an in-depth discussion on MIG optimization recipes.

Having the cost metrics shown in Table 4.1, an MIG representing a given Boolean function can be optimized by applying a set of valid transformations to find an equivalent but more efficient MIG. MIG optimization in terms of delay and area aims at finding the smallest depth, i.e., the number of levels, or the size of the graph, i.e., the number of nodes. Using RRAM devices for implementation, the metrics determining area and delay depend on a combination of MIG features that some of them are not intended in conventional area and depth optimization. However, a reduction in area and especially depth might lower costs of an RRAM-based implementation. Thus, specific optimization techniques are required to find an optimum MIG with respect to the number of RRAM devices and computational steps.

Three MIG rewriting algorithms for logic synthesis of in-memory computing circuits are presented in this section. The first algorithm optimizes MIGs with respect to both objectives simultaneously, while the two others aim at reducing the number of instructions or RRAM devices. For a better understanding of the MIG optimization algorithms tailored for in-memory computing, conventional area and depth optimization algorithms for standard implementation of MIGs proposed in [1] are introduced first.

Algorithm 1: Conventional MIG area optimization (based on [1])

```
1 for (cycles = 0; cycles < effort; cycles++) do
2     $\Omega.M$; $\Omega.D_{R\to L}$;  ⎫
3     $\Omega.A$; $\Psi.C$;             ⎬ eliminate
4     $\Omega.M$; $\Omega.D_{R\to L}$;  ⎭
5 end
```

The framework for area optimization given in Algorithm 1 is based on conventional MIG area optimization algorithm proposed in [1]. Using *eliminate* ($\Omega.M$; $\Omega.D_{R\to L}$) some of the MIG nodes can be removed by repeatedly applying majority rule ($\Omega.M$) and distributivity from right to left ($\Omega.D_{R\to L}$) to the MIG. Assuming x, y, z, u, and v as input variables $\Omega.D_{R\to L}$ transforms $M(M(x, y, u), M(x, y, v), z)$ to $M(x, y, M(u, v, z))$ which means the total number of nodes is decreased from three to two. In order to enable further reduction in the number of nodes, the MIG is reshaped by use of associativity axioms $\Omega.A$, $\Psi.C$, which allow to move the variables between adjacent levels. Then, *eliminate* is applied again to optimize the size of the newly arranged MIG. The area optimization algorithm can be iterated for a maximum number of cycles called *effort*. From the point of area in an RRAM-based circuit, although Algorithm 1 can reduce the number of RRAM devices by removing unnecessary nodes, it does not address the issue of complemented edges that are important in both aforementioned cost metrics.

Algorithm 2: Conventional MIG depth optimization (based on [1])

```
1 for (cycles = 0; cycles < effort; cycles++) do
2     $\Omega.M$; $\Omega.D_{L\to R}$; $\Omega.A$; $\Psi.C$;  ⎫
3     $\Psi.R$;                                            ⎬ push-up
4     $\Omega.M$; $\Omega.D_{L\to R}$; $\Omega.A$; $\Psi.C$;  ⎭
5 end
```

Algorithm 2 is structurally similar to the MIG depth optimization procedure proposed in [1] with slightly shorter iterations. In general, the depth of the graph is of high importance in MIG optimization to lower the latency of the resulting circuits. The depth of the MIG can be reduced by pushing the critical variable with the longest arrival time to upper levels. This can be possible by the process *push-up* shown in Algorithm 2. *Push-up* includes majority, distributivity, and associativity axioms. It is obvious that the majority rule may reduce depth by removing unnecessary nodes. Applying distributivity from left to right ($\Omega.D_{L\to R}$) such that $M(x, y, M(u, v, z))$ is transformed to $M(M(x, y, u), M(x, y, v), z)$ may also result in an MIG with smaller depth. If either x or y is the critical variable with the latest arrival, distributivity can not reduce the depth of $M(x, y, M(u, v, z))$. However, if z is the critical variable, applying $\Omega.D_{L\to R}$ will reduce the depth of MIG by pushing z one level up. In the cases that the associativity rules ($\Omega.A$, $\Psi.C$) are applicable, the depth can be reduced by one if the axioms move the critical variable to the upper level. After performing *push-up*, the relevance axiom ($\Psi.R$) is applied to replace the reconvergent variables that might provide further possibility of depth reduction for another *push-up*.

Although Algorithm 2 decreases the number of instructions in an in-memory computing circuit, it does not consider the issue of complemented edges. Moreover, the depth reduction by Algorithm 2 is only possible at a cost in area, since $\Omega.D_{L\to R}$ adds one extra node to the graph. This may increase the area of the resulting circuit if the size of the critical level, i.e., the level with the maximum number of required RRAMs, is increased. $\Omega.A$ and $\Psi.C$ can also have a similar effect on the maximum level size by moving one node to the critical level. A simple example for this is applying $\Omega.A$ to $M(x, u, M(y, u, M(p, q, r)))$ that has a depth of three and one node in each level. The transformation results in $M(M(p, q, r), u, M(y, u, x))$ of depth two and two nodes in the lower level. Although the late arrival variable ($M(p, q, r)$) is pushed up, the number of nodes in one level, that might be the critical level, has increased from one to two. This effect is not of interest for implementation of MIGs with resistive arrays, however using $\Psi.C$ might be with a positive spin in this case because of the possibility of reducing the number of complemented edges.

Algorithm 3: MIG rewriting for optimization of RRAM costs

```
1 for (cycles = 0; cycles < effort; cycles++) do
2     $\Omega.M_{L\to R}$; $\Omega.D_{L\to R}$; $\Omega.A$; $\Psi.C$;  \
3     $\Omega.I_{R\to L(1-3)}$;                                     } push-up
4     $\Omega.M_{L\to R}$; $\Omega.D_{L\to R}$; $\Omega.A$; $\Psi.C$;  /
5     $\Omega.A$; $\Omega.D_{R\to L}$;
6 end
```

None of the algorithms explained above suggest a solution for the issue of complemented edges that contain an important part of both cost metrics of in-memory computing circuits. Moreover, a single-objective MIG optimization algorithm considers either area or delay that leads to circuits worsened with respect to the other objective. Hence, an MIG optimization algorithm is presented to obtain efficient RRAM-based logic circuits with a good trade-off between both cost metrics. The algorithm includes a combination of conventional area and depth rewritings besides techniques tackling complemented edges from both aspects of area and delay.

As shown in Algorithm 3, the presented MIG rewriting for in-memory computing costs starts with applying *push-up* to obtain a smaller depth. Then, the complemented edges are aimed by applying an extension of inverter propagation axiom from right to left ($\Omega.I_{R\rightarrow L}$) for the condition that the considered node has at least two outgoing complemented edges. The three cases satisfying this condition and their equivalent majority gates are shown below and discussed in the following considering their effect on both cost metrics.

$$M(\bar{x}, \bar{y}, \bar{z}) = \overline{M}(x, y, z) \tag{4.1}$$

$$\overline{M}(\bar{x}, \bar{y}, z) = M(x, y, \bar{z}) \tag{4.2}$$

$$M(\bar{x}, \bar{y}, z) = \overline{M}(x, y, \bar{z}) \tag{4.3}$$

In the first case, the ingoing complemented edges of the gate are decreased from three to zero, while one complement attribute is moved to the upper level, i.e., the level including the output of the gate. Assuming that the current level, i.e., the level including the ingoing edges, is the critical level with the maximum number of required RRAMs, this case is favorable for area optimization. However, if the upper level is the critical level, the number of required RRAMs will increase by only one. Similar scenarios exist for the two other cases, although the last case might be less interesting because the number of complemented edges in both levels is changed equally by one. That means a penalty of one is possible as the cost for a reduction of one, while transformations (1) and (2) may result in RRAM reductions of three and two, respectively.

To reduce the number of instructions, the number of levels possessing complemented edges should be reduced. Depending on the presence of complemented edges by other gates in both levels, the two first transformations given above might reduce or increase the number of instructions or even leave it unchanged. Case (1) is beneficial if the upper level already has complement edges and also the transformation removes all the complemented edges from the current level. It might be also neutral if none of the levels are going to be improved to a complement-free level. The worst case occurs when moving the complement attribute to the upper level which increments the number of levels with complement edges. Similar arguments can be made for the remaining cases. However, case (2) is more favorable because it never adds a level with complemented edges and case (3) can not be advantageous because it can never release a level from complemented edges.

Figure 4.3 shows a simple MIG that is applicable to transformation (2) ($\Omega.I_{R\rightarrow L(2)}$). The transformation has released one level of the MIG from the complement attribute (indicated by a black dot on the edge), which results in a smaller number of instructions. Furthermore, as a result of removing one complemented edge from the critical level, the required number of RRAM devices is decreased by one.

After applying inverter propagation axioms ($\Omega.I_{R\rightarrow L(1-3)}$) in line 3 of Algorithm 3, the MIG is also reshaped and more chances for reducing the depth might be created. Thus, *push-up* is applied to the entire MIG again to reduce the number of instructions as much as possible. In the last step, the number of RRAMs

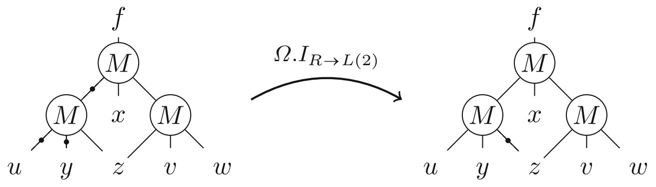


Fig. 4.3 Applying an extension of $\Omega.I_{R\rightarrow L}$ to reduce the extra RRAMs and steps caused by complemented edges

are reduced to make a trade-off between both cost metrics. Applying $\Omega.A$, some of changes by *push-up* that have increased the maximum level size can be undone. Finally, distributivity from right to left ($\Omega.D_{R\rightarrow L}$) is applied to the graph to reduce the number of nodes in levels.

Algorithm 4: MIG rewriting for step minimization

```
1 for (cycles = 0; cycles < effort; cycles++) do
2     $\Omega.M_{L\rightarrow R}$; $\Omega.D_{L\rightarrow R}$; $\Omega.A$; $\Psi.C$;  }
3     $\Omega.I_{R\rightarrow L}$                                                    }
4     $\Omega.I_{R\rightarrow L(1-3)}$                                               } push-up
5     $\Omega.M_{L\rightarrow R}$; $\Omega.D_{L\rightarrow R}$; $\Omega.A$; $\Psi.C$;  }
6 end
```

Due to the importance of latency in logic synthesis, and the issue of sequential implementation, Algorithm 4 is presented to reduce the number of computational steps. In the presented step count optimization algorithm, two axioms of inverter propagation are applied to the MIG after *push-up*. First, only the axiom presented by case (1), i.e., the base rule of inverter propagation from right to left ($\Omega.I_{R\rightarrow L}$), is applied to the entire MIG to lower the number of levels with complemented edges. Since the transformation moves one complement attribute to the upper level, it might create new inverter propagation candidates for the all three discussed cases if the upper level already has one or two ingoing complemented edges. Hence, $\Omega.I_{R\rightarrow L(1-3)}$ is applied again to ensure maximum coverage of complemented edges. Although case (3) can not reduce the number of steps, it is not excluded from $\Omega.I_{R\rightarrow L(1-3)}$ due to its effect on balancing the levels' sizes. Finally, *push-up* is applied to the MIG to reduce the depth more if new opportunities are generated. It should be noted that the number of instructions is mainly determined by the MIG depth. In fact, in the worst case caused by complemented edges, the total number of computational steps would be equal to four times the number of levels, i.e., the MIG depth.

Algorithm 5 is proposed to reduce the number of required RRAM devices [83]. The algorithm starts with *eliminate* to reduce the number of nodes. Then, it applies $\Omega.A$, $\Psi.C$ to reshape the MIG to enable further reduction of area and applies *eliminate* for the second time as suggested in [1]. After eliminating the unnecessary nodes, we use $\Omega.I_{R\to L(1-3)}$ to reduce the number of additional RRAM devices required for complemented edges. At the end, since the MIG might have been changed after the three inverter propagation transformations, $\Omega.I_{R\to L}$ is applied again to ensure the most costly case with respect to the complemented edges is removed. In general, applying the last two lines of Algorithm 5 over the entire MIG repetitively can lead to much fewer RRAM cost.

Algorithm 5: MIG rewriting for RRAM device minimization

```
1 for (cycles = 0; cycles < effort; cycles++) do
2     $\Omega.M$; $\Omega.D_{R\to L}$;   ⎫
3     $\Omega.A$; $\Omega.C$;            ⎬ eliminate
4     $\Omega.M$; $\Omega.D_{R\to L}$;   ⎭
5     $\Omega.I_{R\to L(1-3)}$;
6     $\Omega.I_{R\to L}$;
7 end
```

4.2.3.4 Evaluation and Comparison

To assess the performance of the presented customized logic-in-memory computing approach, experiments are carried with the same benchmark functions used to assess the performance of the proposed BDD-based and AIG-based approaches. The number of cycles (effort) is set to 40 in all experiments. The results of the three proposed MIG optimization algorithms are given in Table 4.5.

According to Table 4.5, MIGs using the MAJ-based realization have much smaller costs in comparison with IMP-based MIGs obtained from the same optimization algorithms that is an expected result of the different number of RRAMs and computational steps required by both realizations. The results show that the proposed step and RRAM minimization algorithms have achieved the smallest number of computational steps and RRAM devices, respectively. This extreme reduction on the other hand has led to a high penalty in the other cost metric for both algorithms such that they have resulted in MIGs with the highest numbers of RRAM devices and computational steps which are target as the optimization objective.

The proposed algorithm tailored for both cost metrics of RRAM-based in-memory computing has achieved circuits for which the number of RRAM devices and computational steps are between the minimum and maximum boundaries found by step and RRAM minimization algorithms. This confirms the capability of the proposed MIG rewriting technique to find a good trade-off between both objectives. To support this statement we make more precise comparisons with the results of step and RRAM count minimization algorithms using the MAJ-based realization. The number of RRAMs by the bi-objective algorithm for RRAM costs optimization is reduced by 19.78% on average compared to the step minimization algorithm at a cost of 21.09% increase in the number of computational steps. The

Table 4.5 Results of MIG-based synthesis approach by the three proposed MIG optimization algorithms

	RRAM costs optimization				Step minimization				RRAM minimization			
	IMP		MAJ		IMP		MAJ		IMP		MAJ	
Benchmark	*R*	*S*	*R*	*S*	*R*	*S*	*R*	*S*	*R*	*S*	*R*	*S*
5xp1	199	99	149	36	264	77	182	28	141	175	97	63
alu4	2160	176	1370	72	2461	165	1717	56	1252	318	878	115
apex1	3676	165	2343	56	4335	121	2972	44	2410	329	1682	119
apex2	531	143	358	56	653	132	435	47	288	341	202	124
apex4	4728	143	2820	64	5340	132	3602	48	3454	286	2414	104
apex5	1482	141	1053	47	1975	98	1286	35	1176	284	816	102
apex6	1652	121	1018	44	1742	99	1191	36	1124	253	786	92
apex7	408	132	277	48	526	121	348	44	300	220	200	80
b9	252	87	168	32	252	66	168	28	252	121	168	44
clip	312	110	217	40	380	99	275	36	218	187	152	68
cm150a	147	77	95	32	132	88	90	32	132	143	88	52
cm162a	90	86	60	30	90	66	65	24	90	117	60	40

cm163a	102	76	68	27	102	66	68	24	102	97	68	34
cordic	189	121	134	48	229	99	162	39	144	248	96	87
misex1	111	66	76	24	130	55	94	20	92	99	64	36
misex3	2207	165	1444	67	2621	143	1762	52	1303	307	913	111
parity	216	132	152	53	216	154	152	48	111	275	77	100
seq	3189	153	1970	64	3551	132	2498	60	1747	341	1217	124
t481	148	142	90	52	188	110	123	40	102	241	68	87
table5	2630	154	1723	64	3393	142	2252	52	1579	363	1107	132
too_large	510	164	322	64	587	121	392	48	234	341	162	124
x1	569	99	435	36	711	77	509	28	322	176	226	64
x2	66	76	46	26	94	66	68	24	66	98	44	35
x3	1729	99	1008	44	1787	99	1201	36	1110	231	772	84
x4	599	77	391	28	694	66	563	24	570	121	380	44
AVG	1116.08	120.16	711.48	46.16	1298.12	103.76	887	38.12	732.76	228.48	509.48	82.6

close percentages demonstrate that the bi-objective MIG rewriting has been almost equally affected both cost metrics, i.e., the number of computational steps is not worsened meaningfully more than the improvement achieved in the RRAM count. A similar comparison of MIG optimization for RRAM costs with results obtained by RRAM device minimization rewriting shows an average decrease of 44.11% in the number of steps at a fair cost of 39.64% in the average value of required RRAM devices.

Table 4.6 compares the proposed MIG rewriting algorithm for RRAM costs with conventional area and depth minimization algorithms. According to the results demonstrated in Table 4.6, MIGs obtained by area and depth minimization algorithms have smaller average values of RRAM devices and computational steps respectively compared to the proposed bi-objective RRAM cost optimization algorithm. It is obvious that as the number of nodes of MIG decreases its depth increases and vice versa, due to the unfavorable effects of *eliminate* and *push-up* in the area and depth rewriting techniques. However, this effect has been lowered by the proposed optimization algorithm for RRAM costs by considering both cost metrics as well as the number of complemented edges in the MIG. The average step count of the proposed bi-objective algorithm for the MAJ-based realization is reduced by 32.55% in comparison with the corresponding value obtained by the area minimization algorithm at a cost of 24.65% increase in the number of RRAM devices. A similar comparison with the depth minimization algorithm shows reduction in both cost metrics. More precisely, the number of computational steps and RRAM devices by the bi-objective MIG optimization approach are decreased by 2.15% and 14.89% on average compared to the depth minimization algorithm.

Comparison of results for the MAJ-based realization by the proposed step and RRAM minimization approaches in Table 4.5 with results by conventional depth and area minimization algorithms demonstrated in Table 4.6 reveals that considering complemented edges has been really influential in getting further reduction in the target cost metric. The number of RRAM devices by the proposed RRAM minimization algorithm is reduced by 10.74% on average compared to the corresponding value by MIG area minimization algorithm. Similarly, comparison of the average step count by the proposed step minimization approach with the same metric resulted by the depth minimization shows a reduction of 29.71%.

4.2.4 Discussion on Design Preferences

Figure 4.4 compares the average values of synthesis results over the whole benchmark set for the three discussed logic representations. For a fair comparison and due to the high importance of latency, the values shown in Fig. 4.4 are chosen

Table 4.6 Comparison of results of the MIG-based synthesis approach with MIG area and depth minimization algorithms based on [1]

	Area minimization				Depth minimization				RRAM costs optimization			
	IMP		MAJ		IMP		MAJ		IMP		MAJ	
Benchmark	*R*	*S*	*R*	*S*	*R*	*S*	*R*	*S*	*R*	*S*	*R*	*S*
5xp1	170	110	121	40	213	110	153	40	199	99	149	36
alu4	1542	286	1111	104	1858	242	1324	88	2160	176	1370	72
apex1	2647	241	1854	92	3399	187	2373	68	3676	165	2343	56
apex2	355	275	249	104	583	231	423	84	531	143	358	56
apex4	3854	198	2703	76	4122	176	2894	64	4728	143	2820	64
apex5	1240	275	861	100	1757	143	1227	52	1482	141	1053	47
apex6	1097	198	763	72	1277	143	891	52	1652	121	1018	44
apex7	300	176	200	64	389	143	275	52	408	132	277	48
b9	252	99	168	36	252	88	168	32	252	87	168	32
clip	256	132	184	52	276	121	198	44	312	110	217	40
cm150a	132	99	88	36	132	99	88	36	147	77	95	32
cm162a	90	99	60	36	90	77	60	28	90	86	60	30

(continued)

Table 4.6 (continued)

	Area minimization				Depth minimization				RRAM costs optimization			
	IMP		MAJ		IMP		MAJ		IMP		MAJ	
Benchmark	*R*	*S*	*R*	*S*	*R*	*S*	*R*	*S*	*R*	*S*	*R*	*S*
cm163a	102	77	68	28	102	77	68	28	102	76	68	27
cordic	199	164	143	56	242	132	174	48	189	121	134	48
misex1	101	77	73	28	128	66	92	24	111	66	76	24
misex3	1547	253	1112	92	2118	231	1488	84	2207	165	1444	67
parity	224	176	160	64	224	176	160	64	216	132	152	53
seq	2032	308	1457	112	2566	242	1798	88	3189	153	1970	64
t481	102	209	71	76	168	132	120	48	148	142	90	52
table5	1598	286	1126	104	2719	231	1881	84	2630	154	1723	64
too_large	315	341	213	124	512	264	370	96	510	164	322	64
x1	442	164	309	64	736	110	528	40	569	99	435	36
x2	66	88	45	35	92	77	66	28	66	76	46	26
x3	1075	198	750	72	1363	143	951	52	1729	99	1008	44
x4	570	121	380	44	591	88	409	32	599	77	391	28
AVG	812.32	186	570.76	68.44	1036.36	149.16	727.16	54.24	1116.08	120.16	711.48	46.16

from optimization results with respect to the number of computational steps, i.e., MOB with priority to the number of steps for BDDs, MIG rewriting for step count minimization, and AIG depth rewriting.

According to Fig. 4.4, BDDs clearly achieve smaller number of RRAM devices and therefore can be a better choice when area is considered a more critical cost metric. On the other hand, the step counts obtained by the BDD-based synthesis are much higher than the same values resulted by AIG-based and MIG-based methods. Comparison of synthesis results by the AIGs and MIGs also shows that the average number of computational steps for the MIG-based method using the MAJ-based realization is reduced by 19.37% compared to the AIG-based synthesis using depth minimization. This confirms the advantage of MIGs in providing higher speed in-memory computing circuits in comparison with the two other representations.

4.3 Crossbar Implementation for the Proposed Synthesis Approach

The number of required RRAM devices and computational steps for the presented in-memory computing approach using any of the three mentioned logic representations have been discussed before. This section shows how an entire graph can be computed on a crossbar and what determines its required dimensions. We present step by step implementation of an example MIG shown in Fig. 4.5 which represent a three input XOR gate.

Both MAJ-based and IMP-based implementations for in-memory computing logic circuits can be executed on a standard crossbar architecture as shown in Fig. 4.6a. In such an architecture, an entire row should be allocated for computing

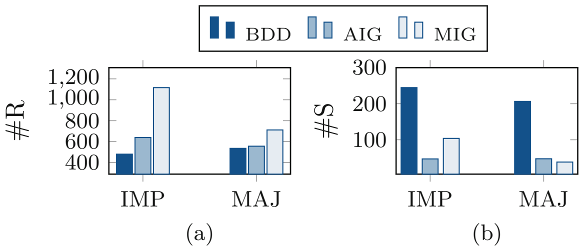


Fig. 4.4 Comparison of synthesis results by logic representations for RRAM-based in-memory computing, (**a**) the average number of RRAM devices, (**b**) the average number of computational steps

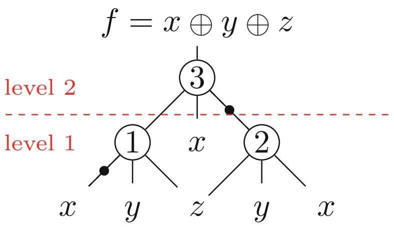


Fig. 4.5 MIG representing a three bit XOR gate

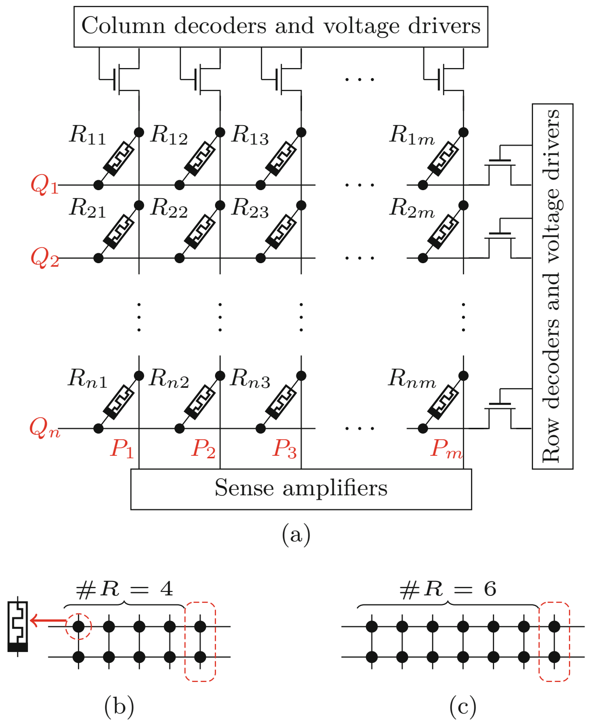

Column decoders and voltage drivers
R_{11} R_{12} R_{13} R_{1m}
R_{21} R_{22} R_{23} R_{2m}
R_{n1} R_{n2} R_{n3} R_{nm}
Q_1 Q_2 Q_n
P_1 P_2 P_3 P_m
Sense amplifiers
Row decoders and voltage drivers
(a)
$\#R = 4$
(b)
$\#R = 6$
(c)

Fig. 4.6 Crossbar implementation for the presented synthesis approach for logic-in-memory computing, (**a**) Standard crossbar architecture, (**b**) upper-bound crossbar for MAJ-based, and (**c**) IMP-based implementations for MIG shown in Fig. 4.5

a single graph node which represents a data structure primitive, e.g., a majority gate if MIG is used for synthesis. To compute an entire level of a data structure simultaneously, all nodes of the level should be computed in parallel in separate rows. This means that a number of rows equal to the maximum level size in the entire graph is required. For example, for a logic representation whose largest level has four nodes, a crossbar architecture with at least four rows is needed, independent of the type of the utilized representation or the basic operation of MAJ or IMP. Nevertheless, the number of RRAM devices at each row, i.e, the number of columns, is determined by the RRAM-based realization of the exploited data structure primitive and therefore absolutely depends on both of the aforementioned conditions.

In the following, it is assumed that all of the primary inputs are already written in the memory.

4.3.1 MAJ-*Based Implementation*

The values given for MAJ-based implementation in Table 4.1 indeed present the upper bounds for the crossbar realization. According to Table 4.1, the MAJ-based synthesis of the MIG shown in Fig. 4.5 can be realized using a maximum of nine RRAM devices, since the critical level needs 2×4 for its nodes and one more RRAM for the ingoing complemented edge. Also, each level can require three computational steps (2×3), which results in a total of eight steps for the whole MIG considering the presence of complemented edges at both levels.

Here, it is showed that the resulting MAJ-based implementation can be executed much more efficiently with respect to both time and area.

The crossbar with the upper bound dimensions for the MAJ-based implementation of 3-bit XOR gate is shown in Fig. 4.6b. The MIG has a maximum level size of two, and accordingly the required crossbar needs two rows. Each row consists of four RRAM devices to compute a node and one additional device to be used in case of having a complemented edge. We assume that a maximum of two ingoing edges for an MIG node can be complemented after rewriting, from which one can directly be used as the second inverted operand of MAJ and thus, only one needs to be negated. The RRAM devices allocated for the complemented edges are displayed in red dashed surrounds at the end of the rows.

The implementation steps for the MAJ-based computation of the MIG shown in Fig. 4.5 are listed below:

Initialization	$R_{ij} = 0 : Q_{ij} = 1,\ P_{ij} = 0;$
1: Loading the third operands	$Q_1 = Q_2 = 0,\ P_1 = P_2 = z;$
	$R_{11} : \text{MAJ}(z, 0, 0) = M(z, 1, 0) = z;$
	$R_{21} : \text{MAJ}(z, 0, 0) = M(z, 1, 0) = z;$
2: Negation for node 2	$Q_1 = Q_2 = x,\ P_1 = x,\ P_2 = 1;$
	$R_{25} : \text{MAJ}(1, x, 0) = M(1, \bar{x}, 0) = \bar{x};$
3: Computing level 1	*Node 1:* $P_1 = y,\ Q_1 = x,\ R_{11} = z$
	$R_{11} : \text{MAJ}(y, x, z) = M(y, \bar{x}, z);$
	Node 2: $P_1 = y,\ Q_2 = \bar{x}\ (@R_{25}),\ R_{21} = z;$
	$R_{21} : \text{MAJ}(y, \bar{x}, z) = M(y, x, z);$
4: Computing level 2 (root node)	$P_1 = x,\ Q_1 = @R_{21},\ R_{11} = M(\bar{x}, y, z);$
	$R_{11} : \text{MAJ}(x, @R_{21}, @R_{11}) = M(x, \overline{@R_{21}}, @R_{11}) :$
	$M(M(\bar{x}, y, z), x, \overline{M}(x, y, z));$

It is assumed that all of RRAM devices are first loaded with zero. For initialization, voltage levels 1 and 0 should be applied to the bottom electrodes (Q_{ij}) and the top electrodes (P_{ij}), respectively. This step is not considered in the step count. Step 1 starts to compute the nodes 1 and 2 in level 1 (see Fig. 4.5) by loading the variable

z as the third operands of MAJ. As said before, every node of the level should be computed in a separate crossbar row. Accordingly, nodes 1 and 2 are respectively computed in row 1 and 2 by selecting R_{11} and R_{21} as the corresponding third operands, i.e., the destinations of the operations. Then, the primary inputs are read from memory and applied to the corresponding row and columns to execute the operations.

It is worth noting that the MAJ-based realization of majority gate in Sect. 4.2.3 also allocates RRAM devices for the first and the second operands. This is not actually required by the MAJ operation, dissimilar to IMP which needs all variables to be accessible on the same row. Nevertheless, all of the input RRAM devices are already considered in the crossbar with the upper bound dimensions shown in Fig. 4.6b. Furthermore, in the MAJ-based realization of majority gate, we simply assumed that the second operand needs to be inverted and considered an RRAM device for it, while this is not always required. An MIG node with a single ingoing complemented edge can actually be implemented faster by skipping the negation and using the complemented edge directly as the second operand. In the MIG shown in Fig. 4.5, nodes 1 and 3 (the root node) are ideal for MAJ due to possessing a single complemented edge but node 2 requires one negation which needs to be performed first.

The negation required for node 2 is performed in step 2 at R_{25} by setting its bottom electrode (Q_2), to the value of x and its top electrode (P_2) to 1. It should be noted that R_{25} is not independently accessible and the entire second row and column are exposed to these voltage levels. Therefore, we need to ensure that the previously stored values are retained. By setting Q_2 to x, the bottom electrode of R_{21} is also changed to x. To maintain the value stored in R_{21}, its top electrode (P_1) should be also set to the same voltage level x as shown in step 2. By doing this, the top electrode of R_{11} changes to x, and thus its bottom electrode (Q_1) also needs to be set to x for keeping the current state of the devise.

The entire level 1 is computed simultaneously in step 3. One read is required to apply the value of $\bar{x}$ to Q_2. As shown in the step 3, both nodes can be computed in the same column since their first operands are equal, which is not necessarily true in all the cases. Step 4 computes the root node of the MIG. This can be done at one of the RRAM devices storing the intermediate results from the previous step and does not require any data loading. Since the value of node 2 is complemented, it is more efficient to use the value stored in R_{21} as the second operand to skip negation. This requires one read from R_{21} and R_{11} can be set to the third operand which is also the final destination of the computation.

4.3.2 IMP-*Based Implementation*

The dimensions of the required crossbar for the IMP-based implementation is shown in Fig. 4.6c which has one extra RRAM at the end of each row for the complemented edges. According to Table 4.1, the example MIG shown in Fig. 4.5 with a maximum level size of 2 needs an upper bound of 12 (2×6) RRAM devices placed in two rows in addition to one more for the ingoing complemented edge. As Table 4.1 suggests, the computation needs 22 steps, 2×10 for the two levels plus two more steps for the complemented edges, including the IMP operations and the loads.

The required steps for the IMP-based implementation of the MIG shown in Fig. 4.5 are listed in the following:

Initialization	$R_{ij} = 0;$
1: Loading variables for level 1	$R_{11} = x,\ R_{12} = y,\ R_{13} = z;$ $R_{21} = x,\ R_{22} = y,\ R_{23} = z;$
2: Negation for node 1	$R_{17} \leftarrow x \text{ IMP } R_{17} : R_{17} = \bar{x};$
3–11: Computing level 1	*Node 1:* $R_{14} = M(\bar{x}, y, z);$ *Node 2:* $R_{24} : M(x, y, z);$
12: Loading variables for level 2	$R_{11} = x,\ R_{12} = M(x, y, z),$ $R_{13} = M(\bar{x}, y, z),$ $R_{14} = R_{15} = R_{16} = R_{17} = 0;$
13: Negation for node 3	$R_{17} \leftarrow R_{12} \text{ IMP } R_{17} :$ $R_{17} = \overline{R_{12}} = \overline{M}(x, y, z);$
14–22: Computing level 2 (root node)	$R_{14} = M(M(\bar{x}, y, z), x, \overline{M(}x, y, z));$

To explain the implementation step-by-step, we use names R_1 to R_6 for the RRAM devices at each row to denote X, Y, Z, A, B, and C, respectively as used in Sect. 4.2.3. For initialization, all of the RRAM devices in the entire crossbar are cleared. In contrary to MAJ, IMP needs all variables used for computation to be stored in the same horizontal line. This means that there may be a need to have several copies of primary inputs or intermediate results at different rows simultaneously, as shown in step 1, where the variables of nodes 1 and 2 are loaded into RRAM devices in both rows. Step 2 computes the complemented edge of node 1 in the seventh RRAM device considered for this case at the end of first row, R_{17}. Steps 3-11 compute both nodes at level one and store the results in the forth RRAM device at the corresponding crossbar row, similarly to the RRAM device A used in Sect. 4.2.3.

The same procedure continues to compute the second level, which only consists of the MIG root node. Two out of the three inputs of node 3 are intermediate results, which have to be first read and then copied into the corresponding RRAM devices at row 1 besides other input and work devices as shown in step 12. In step 13, the complemented edge originating at node 2 is negated, and then root node is computed in step 22.

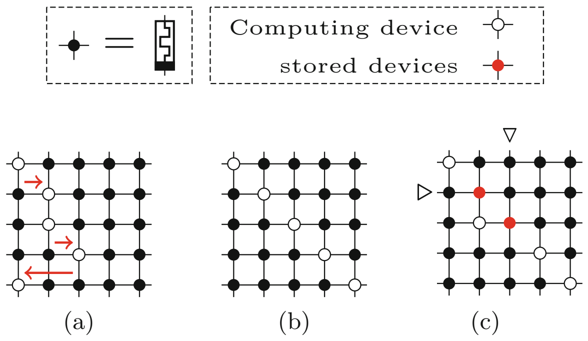


Fig. 4.7 Selecting crossbar computing RRAM devices to avoid computation interferences for MAJ-based implementation, (**a**) performing conflicting computations at different columns, (**b**) diagonal computation, (**c**) retaining previously stored devices

4.3.3 Discussion

The MAJ-based implementation for the example MIG in Fig. 4.5 was carried out using a small number of RRAM devices and within only four steps far less than the upper bounds, while computational steps or RRAM devices could be saved during the IMP-based implementation. Length of the computational sequence required for data loading and negation of MAJ-based implementation can be even shortened much more for larger Boolean functions. Number of RRAM devices can also be much lower than the MAJ-based upper bounds given in Table 4.1 by performing operations successively in the devices carrying the results of previous levels.

It is obvious that using MAJ provides higher efficiency especially for MIG-based synthesis. Moreover, IMP requires additional voltages, which is not the case for MAJ, and as a result needs more complex control scheme and peripheral circuitry. However, MAJ-based implementation needs active read operations for each cycle, while IMP-based implementation can reduce this requirement and propagate data natively within the memory array. MAJ-based implementation also does not allow to independently set the values of the top electrodes of the computing RRAM devices placed in the same columns. Such computation correlations do not occur during IMP-based operations since all IMP operations are executed with the same voltage levels V_{SET} and V_{COND}.

Nevertheless, dependency of the voltage levels of crossbar's rows and columns can be managed in many cases due to the commutativity property of the majority operation. This allows to perform computations simultaneously at RRAM devices in the same column if they share a single operand to be applied to the entire column. Using an automated procedure, the RRAM devices allocated for parallel computations can be placed in different non-conflicting columns as shown in Fig. 4.7a. When none of the MAJ operations share any operand, the computations should be performed diagonally (Fig. 4.7b).

The rows or columns with computing devices may also possess previously stored RRAM devices, which values have to be maintained during computations by applying equal voltage levels to their top and bottom electrodes, i.e., their row and column drivers. For example in Fig. 4.7c, the second column has a stored device in the second row from top and a computing one in the third row. To keep the value of the stored device unchanged, the same voltage level, which is applied to its column for computing, has to be applied to its row. Setting the voltage level of the third column from left also needs a similar consideration as the column has a stored device located in a computing row.

It is obvious that a computation can coexist with the stored device in the same row or column if one of its operands is equal to what applied to the coordinates of the stored device. However, presence of several stored devices in a row or column may make it complex to arrange safe computations which can be handled by freeing such rows or columns from computation. As Fig. 4.7 shows, considerations regarding the conflicting computing or previously stored devices increase the area of the crossbar architecture since a larger number of columns or rows may be required, although the number of required RRAM devices does not change. Nevertheless, the number of steps can increase if the crossbar array does not meet the required number of rows and columns, which needs to move some computations into the successive steps.

4.4 Summary

This chapter presents a comprehensive approach for logic synthesis of in-memory computing circuits using the logic representations BDDs, AIGs, and MIGs. The synthesis for in-memory computing is carried out within three stages for each representation, i.e., realization of the logic primitives, design methodology for computing on a memristive crossbar, and optimization to lower the implementation costs. The chapter also shows that the presented approach provides valid and efficient crossbar implementations.

Experimental results reveal higher efficiency in comparison with the state-of-the-art. The results for each logic representation differ from values addressing area and latency which fulfills different design preferences. Using BDDs for synthesis leads to smaller number of RRAM devices at a high cost in the number of computational steps in comparison with the two other logic representations. Thus, the BDD-based approach is useful when lower area overhead is of high interest. The proposed AIG-based synthesis approach provides a fair trade-off between both cost metrics among the experimented representations, while MIGs in particular show a high capability in reduction of length of the operation sequences. This makes MIG a suitable representation for faster logic-in-memory implementations.

Chapter 5
Compilation and Wear Leveling for Programmable Logic-in-Memory Architecture

5.1 Related Work

The *Programmable Logic-in-Memory* (PLiM) computer architecture is a regular resistive memory array architecture which is also capable of performing logic operations. The memory is based on traditional multi-bank architecture and is connected to the PLiM controller block (see Fig. 5.1a). This controller is a light-weight synchronous block that controls the memory array's access bus to allow running a computation mode. The computation mode runs a sequential execution of a given set of instructions that represent a program. The program is stored on the memory array, which is automatically updated when the program is executed. The transition between the computation mode and the memory mode is controlled by the *Logic-in-Memory* (LiM) input.

Dissimilar to the processors in modern computer architectures where computations are performed on embedded circuits and systems, the PLiM computes a sequence of instructions on banks of RRAM crossbar arrays (see Fig. 5.1b) which makes it programmable for any purpose without any change in hardware or wiring. Although the PLiM implementations are not structured as connected hardware elements, the procedure to design for PLiM can algorithmically borrow from classical hardware synthesis.

The PLiM instructions consist of the majority oriented logic operation, i.e. RM_3 (see Sect. 2.2.2), natively constructed within RRAM devices [35]. PLiM revolves around a single instruction, i.e. an $\text{RM}_3(P, Q, R)$ operation. The instruction takes three bits stored in the memory as operands, P, Q, and R. P is applied to the top electrode and Q to the bottom electrode of an RRAM device with a resistance state of R which is updated accordingly when the RM_3 instruction is executed.

The single instruction scheme simplifies the architecture as it is directly associated with the memory's intrinsic logic operation. The source, destination, and the processing unit of the architecture is the memory block itself. Performing the instruction simply means loading the bit-level values of P and Q from memory

© Springer Nature Switzerland AG 2020

S. Shirinzadeh, R. Drechsler, *In-Memory Computing*,
https://doi.org/10.1007/978-3-030-18026-3_5

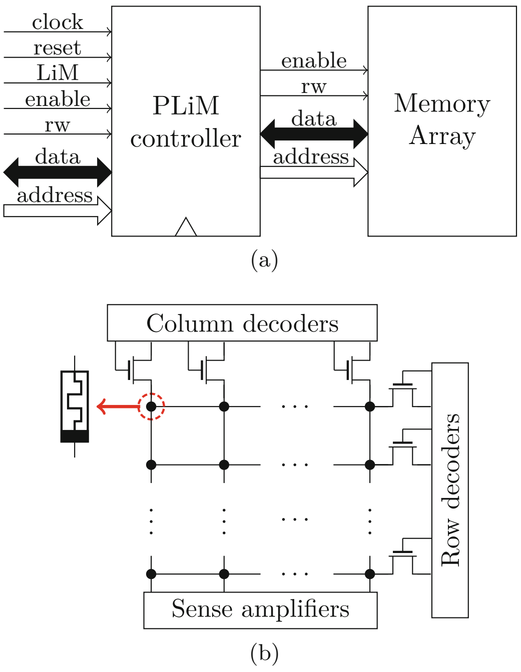


Fig. 5.1 (**a**) The PLiM architecture [35]. (**b**) A single memory bank of resistive crossbar array on which a PLiM implementation is executed

and applying them to the RRAM device *R*. The instruction itself is stored on the same memory bank which should be loaded first. Then, the operands are loaded from the memory, and finally, the operands are applied to the electrodes of the destination RRAM, i.e. addressed with the row (wordline) and column (bitline) in which the device is located in the crossbar array. When the operation is performed, the resistance state of the destination RRAM device is updated by the result of the instruction.

In [6], an in-memory computing architecture based on crossbar RRAM (called ReVAMP) was proposed which uses RM3 as the basic resistive operation similarly to PLiM and therefore makes use of MIGs efficiently. The number of executable RM3 instructions per cycle was increased in [6] enabled by heuristic bit-level parallelization algorithms. Since only one word can be read during a cycle, ReVAMP stores as many as possible instruction operands on a word until its capacity allows according to the crossbar dimensions. This way several instruction operands can be read and applied to the wordlines/bitlines after one read cycle. The parallelization is increased by finding and trying to create shared operands within MIG nodes which can be computed concurrently. Also, technology mapping for the architecture was addressed by investigating the impact of the word length [6], and generally the crossbar dimensions on the overall delay [7].

As explained above, the ReVAMP architecture in [6] allows partial parallelization. While the issue of multiple reads for parallel instruction executions is addressed, the approach does not consider the probable computational conflicts which may overwrite and distort the instructions' outputs (see Sect. 4.3.3). Accordingly, this chapter considers only a single instruction to be performed in each cycle as suggested for PLiM and proposes optimization algorithms to minimize the delay and number of required devices for this case.

5.2 The PLiM Compiler

The concept of the PLiM computer was proposed in [35] as explained before. However, no automated scheme was proposed for the control and optimization during the synthesis process. This section proposes an automated compiler for the PLiM. For an arbitrary Boolean function, the compiler produces a chain of RM3 instructions which compute the function on PLiM's memory arrays. The compilation also determines the RRAM devices which are used for each operand of an instruction. To improve readability of the PLiM programs, we present them as a set of primitive logical functions sufficient to compute any function which we refer to as PLiM commands. This section aims at programming PLiM to compute an arbitrary function within a finite sequence of these commands.

$$\begin{array}{ll}
\texttt{ZERO}(z) & z \leftarrow \mathrm{RM3}(0, 1, z) = \langle 00z \rangle = 0 \\
\texttt{ONE}(z) & z \leftarrow \mathrm{RM3}(1, 0, z) = \langle 11z \rangle = 1 \\
\texttt{BUF}(a, z) & \texttt{ZERO}(z);\ z \leftarrow \mathrm{RM3}(a, 0, z) = \langle a10 \rangle = a \\
\texttt{NOT}(a, z) & \texttt{ZERO}(z);\ z \leftarrow \mathrm{RM3}(1, a, z) = \langle 1\bar{a}0 \rangle = \bar{a} \\
\texttt{RM}(a, b, z) & z \leftarrow \mathrm{RM3}(a, b, z)
\end{array}$$

Each command consists of one, e.g., $\texttt{ZERO}(z)$, or two, e.g., $\texttt{BUF}(a, z)$, RM3 instructions, where z is the destination RRAM device storing the result of the command [83, 84].

5.2.1 *Motivation*

The target automated compiler aims at leveraging MIGs in order to derive RM3 instruction sequences, which can run as programs on the PLiM architecture. Several issues should be properly addressed during translation of an MIG into PLiM understandable commands or in other words compilation for PLiM. The order of the MIG nodes selected for execution highly matters as it affects the required number of RRAM devices and the number of instructions. Even for computing a certain node, different selections of operands can result in different numbers of RRAM devices and instructions [83, 84].

In its current form, the PLiM architecture can only handle serial operations [35]. Therefore, only one MIG node might be computed each time and the total number of instructions is equal to the sum of instructions required to compute each MIG node. Accordingly, reducing the size of the MIG is considered to have a significant impact on the PLiM program with respect to the number of instructions. In addition, further MIG optimization may be possible to lower the costs caused by complemented edges. While the presence of a single complemented edge in an MIG node is of interest for benefiting from the intrinsic majority operation inside an RRAM device, a second or third complemented edge imposes extra costs in both number of instructions and required RRAM devices. Hence, MIG area rewriting, besides reducing number of nodes with multiple complemented edges, can be highly effective for optimizing the number of instructions, while the latter can also lower the number of required RRAM devices.

As an example, consider the two equivalent MIGs in Fig. 5.2a, before optimization on the left and after optimization on the right. Translating them into PLiM commands yields:

$$\begin{array}{r|r}
\texttt{NOT}(i_3, z_1) & \texttt{BUF}(i_3, z_1) \\
1 \rhd \texttt{RM}(i_1, i_2, z_1) & 1 \rhd \texttt{RM}(i_2, i_1, z_1) \\
\texttt{NOT}(z_1, z_2) & 2 \rhd \texttt{RM}(i_2, i_4, z_1) \\
2 \rhd \texttt{RM}(i_2, i_4, z_2) &
\end{array}$$

Here z_i refers to the address of RRAM devices storing the result or an operand of an instruction. The commands given on the left side represent the PLiM program to compute the initial MIG, while the commands on the right represent the MIG after optimization. Before optimization, the MIG shown in Fig. 5.2a requires more commands and two RRAM devices, z_1 and z_2, to be implemented on PLiM. Although the optimization does not reduces the number of nodes and just affects the complemented edges, the required number of RRAM devices and commands both decrease by one. The latter indeed means a reduction from 6 to 4 in the number of instructions, i.e. clock cycles, as some commands include two instructions. The effect of multiple complement edge elimination is much larger when translating a large MIG.

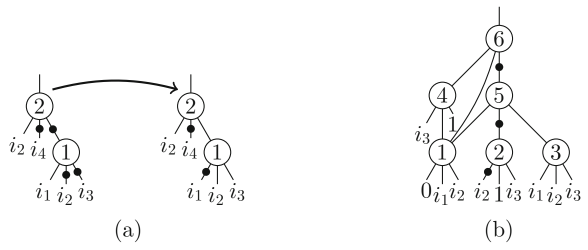


Fig. 5.2 Reducing the number of instructions and RRAM devices, after (**a**) MIG rewriting, (**b**) after node selection and translation

As mentioned before, not only the MIG structure has an effect on the PLiM program, but also the order in which nodes are translated as well as the selection of the node's inputs as operands P, Q, and destination R in the RM3 instruction. For example, consider the MIG in Fig. 5.2b. Translating it in a naïve way, i.e. in order of their node indices and selecting the RM3 operands and destination in order of their inputs (from left to right), will result in PLiM program shown in the following on the left side which needs 13 commands and 7 RRAM devices. By changing the order in which the nodes are translated and also the order in which operands are selected for the RM3 instructions, a shorter program requiring 11 commands and only 4 RRAM devices can be found for the same MIG representation on the right side:

```
    NOT($i_1, z_1$)      3 ▷ RM($i_1, z_4, z_5$)   |      BUF($i_2, z_1$)      3 ▷ RM($i_1, z_3, z_4$)
    BUF($i_2, z_2$)          NOT($i_3, z_6$)       |  1 ▷ RM($i_1, 1, z_1$)    5 ▷ RM($z_1, z_2, z_4$)
1 ▷ RM($0, z_1, z_2$)        ONE($z_7$)            |      ONE($z_2$)               BUF($i_3, z_2$)
    BUF($i_3, z_3$)      4 ▷ RM($z_2, z_6, z_7$)   |  2 ▷ RM($i_3, i_2, z_2$)  4 ▷ RM($z_1, 0, z_2$)
2 ▷ RM($1, i_2, z_3$)    5 ▷ RM($z_2, z_3, z_5$)   |      NOT($i_2, z_3$)      6 ▷ RM($z_1, z_4, z_2$)
    NOT($i_2, z_4$)      6 ▷ RM($z_7, z_5, z_2$)   |      BUF($i_3, z_4$)
    BUF($i_3, z_5$)                                |
```

Based on this observations, the next section describes algorithms for automatically finding a good MIG representation and for translating an MIG representation in an effective way to get a small PLiM program.

5.2.2 MIG Rewriting

As discussed in Sect. 5.2.1, both the size of an MIG and the distribution of complemented edges have an effect on the PLiM program in number of instructions and number of RRAM devices. Hence, we are interested in an MIG rewriting algorithm that (1) reduces the size of the MIG, and (2) reduces the number of MIG nodes with multiple complemented edges.

Algorithm 1: MIG rewriting for PLiM architecture

```
1 for (cycles = 0; cycles < effort; cycles++) do
2     $\Omega.M$; $\Omega.D_{R\to L}$;
3     $\Omega.A$; $\Omega.C$;
4     $\Omega.M$; $\Omega.D_{R\to L}$;
5     $\Omega.I_{R\to L(1-3)}$;
6     $\Omega.I_{R\to L}$;
7 end
```

The proposed MIG rewriting approach is given in Algorithm 1 and follows the rewriting idea of [1]. This algorithm is identical to Algorithm 5 in Chap. 4 for RRAM device minimization which aims at reducing the number of MIG nodes and nodes with multiple complemented edges. However, to keep this chapter self-contained, the algorithm has been repeated in here and explained briefly. Algorithm 1 can be iterated for a certain number of times, controlled by *effort*. The first three lines of the algorithm are based on the conventional MIG area rewriting approach proposed in [1]. It is clear that $\Omega.M$ reduces the size of MIG by eliminating the unnecessary nodes (see Sect. 4.2.3.3). Distributivity from right to left ($\Omega.D_{R\rightarrow L}$) also reduces the number of nodes by one. These node elimination techniques are repeated after reshaping the MIG by applying $\Omega.A$; $\Omega.C$, which may provide further size reduction opportunities.

To reduce the number of nodes with multiple inverted edges, we first apply an extended inverter propagation axiom from right to left denoted by $\Omega.I_{R\rightarrow L(1-3)}$. $\Omega.I_{R\rightarrow L(1-3)}$ includes the three transformations (1) $M(\bar{x}, \bar{y}, \bar{z}) = \overline{M(x, y, z)}$, (2) $\overline{M(\bar{x}, \bar{y}, z)} = M(x, y, \bar{z})$, and (3) $M(\bar{x}, \bar{y}, z) = \overline{M(x, y, \bar{z})}$ (for more information see Sect. 4.2.3.3). Finally, at the end of the algorithm, $\Omega.I_{R\rightarrow L}$ is applied again to ensure the most costly case is eliminated.

Algorithm 2: Outline of compilation algorithm

```
   Input  : MIG $M$
   Output: PLiM program $P = \{I_1, I_2, \ldots, I_k\}$
1  foreach leaf in $M$ do
2      set COMP[$v$] ← $\top$;
3  end
4  foreach MIG node in $M$ do
5      if all children of $v$ are computed then
6          $Q$.enqueue($v$);
7      end
8  end
9  while $Q$ is not empty do
10     set $c$ ← $Q$.pop();
11     set $P$ ← $P \cup$ translate($c$);
12     set COMP[$c$] ← $\top$;
13     foreach parent of $c$ do
14         if all children of $v$ are computed then
15             $Q$.enqueue($v$);
16         end
17     end
18 end
```

5.2.3 *Compilation*

This section describes how an optimized MIG is compiled into a PLiM program. Algorithm 2 gives an overview of the compilation algorithm, details are explained in the remainder of this section. The algorithm keeps track of a map `COMP[`v`]` that stores for each MIG node v whether it has been computed or not. Initially, all leafs, i.e. primary inputs and the constant, are set to be computed. A priority queue Q keeps track of all vertices that can possibly be translated, called candidates. A vertex is a *candidate* if all its children are computed. The sorting criteria for Q is described in Sect. 5.2.3.1.

The main loop of the algorithm starts by popping the best candidate c from the priority queue and translating it into a sequence of PLiM instructions. Section 5.2.3.2 describes the translation process in detail. Afterwards, for each parent, it is checked whether it is computable, and if this is the case, it is inserted into Q.

5.2.3.1 Candidate Selection

As shown before, the order of computing MIG nodes also changes the number of required RRAM devices. In fact, a different order of node computation can lead to a greater or less number of RRAM devices by different allocating and releasing times. Here, it is shown that how adopting a candidate selection strategy can effectively lower the size of resistive memory array required for a given task.

The proposed candidate selection strategy is based on two principles: (1) releasing the RRAM devices in-use as early as possible, and (2) allocating RRAMs at the right time such that they are blocked as short as possible. We show two example MIGs to clarify the principles. Figure 5.3a shows an MIG with two candidate nodes u and v, for which all of their children nodes are already computed. Candidate u has two *releasing children*, i.e. children who have single fan-out, while v has only one releasing child. In the case that u is selected for computation first, the RRAM devices keeping its releasing children can be freed and reused for the next candidate.

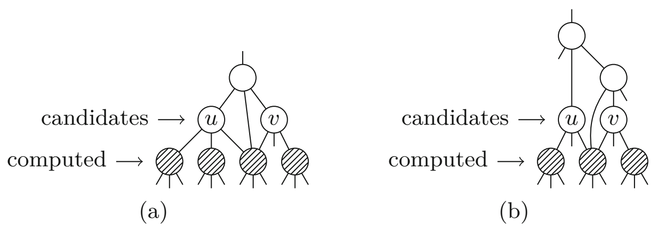


Fig. 5.3 Reducing the number of RRAM devices for PLiM implementations by selecting the candidate with (**a**) more releasing children, and (**b**) smaller fanout level index

Figure 5.3b shows a small MIG with two candidates u and v to illustrate the second principle. The output of u is only required when v is already computed. In other words, the number of RRAM devices in use can increase if u is computed before v, since the device keeping u can not be released before computing the root node of the MIG. This way, v is computed when an RRAM device has been already allocated to retain the value of u. The number of additional RRAM devices required in such condition can be considerable for large number of nodes.

In order to sort nodes in the priority queue in Algorithm 2, two nodes u and v are compared. Node u is preferred over v if (1) its number of releasing children is greater, or (2) if u's parent with the largest level (ordered from primary inputs to primary outputs) is on a lower level than v's parent with the smallest level. If no criteria is fulfilled, u and v are compared according to their node index.

5.2.3.2 Node Translation

This section explains how an MIG node is translated into one RM$_3(A, B, Z)$ instruction with operands A and B, and destination Z. The operands A and B can be RRAM devices or constants and the destination Z is an RRAM device. Recall that the instruction computes $Z \leftarrow \langle A\bar{B}Z\rangle$. In the ideal case each MIG node can be translated into exactly one RM$_3$ instructions and can reuse one of its children's RRAM devices as destination. In other cases, additional RM$_3$ instructions and/or additional devices are required.

5.2.3.3 Select Operand *B*

We first select which of the node's children should serve as operand B, i.e. the second operand of the RM$_3$ instruction. In total, four cases with subcases are checked in the given order which are illustrated in Fig. 5.4. Only the last two subcases require two additional instructions and one additional RRAM. It should be noted that $@X_i$ in the figure refers to the address of RRAM device X_i.

(a) *There is exactly one complemented child:* B is the RRAM device storing this complemented child.
(b) *There is more than one complemented child, but also a constant child:* The nonconstant complemented child is selected for B, since constants allow for more flexibility when selecting the remaining operands.
(c) *There is no complemented child, but there is a constant child:* B is assigned the inverse of the constant. Since we consider MIGs that only have the constant 0 child, B is assigned 1.
(d) *There is more than one complemented child, but at least one with multiple fanouts:* We select the RRAM device of the child with multiple fanouts, as this excludes its use as destination.

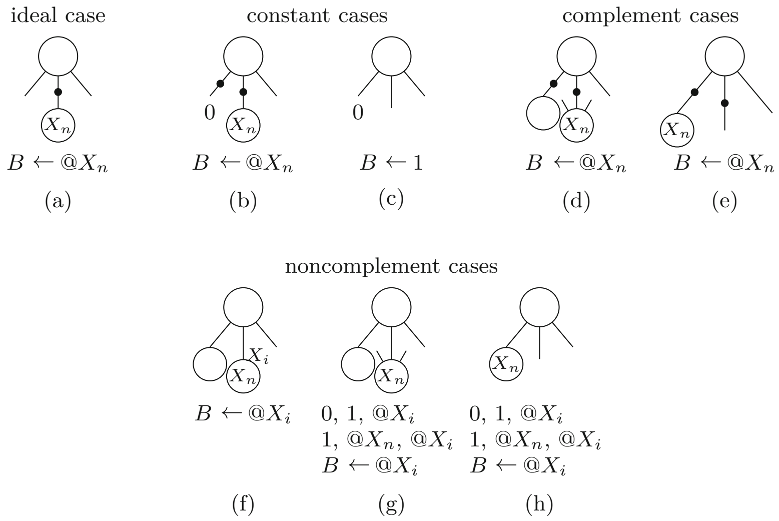


Fig. 5.4 Selecting the second operand of an instruction denoted by B

(e) *There is more than one complemented child, none with multiple fanouts:* The RRAM device of the first child is selected.

(f) *There is no complemented child, but for one child there exists an RRAM device with its complemented value:* Each node is associated with an RRAM device which holds or has held its computed result. In addition, if its inverted value is computed once and stored in an additional RRAM device X_i, it is remembered for future use. In this case B can be assigned X_i.

(g) *There is no complemented child, but one child has multiple fanouts:* The child with multiple fanouts is selected with the same argumentation as above. Since it is not inverted, an additional RRAM device X_i needs to be allocated, loaded with 0, and then set to the complement of X_n. As described above, X_i is associated to the child for future use.

(h) *There is no inverted child and none has multiple fanouts:* The fist child is selected and an additional RRAM device X_i is allocated to store the complement of X_n.

5.2.3.4 Select Destination *Z*

After the inverter selection, the destination RRAM device, i.e. the third argument of the RM3 instruction is selected. The aim is to reuse one of the children's RRAM devices as work device instead of creating a new one. In total, four cases (with subcases) are checked which are illustrated in Fig. 5.5. Only the first case allows to reuse an RRAM device of the children for the destination, all the other cases require

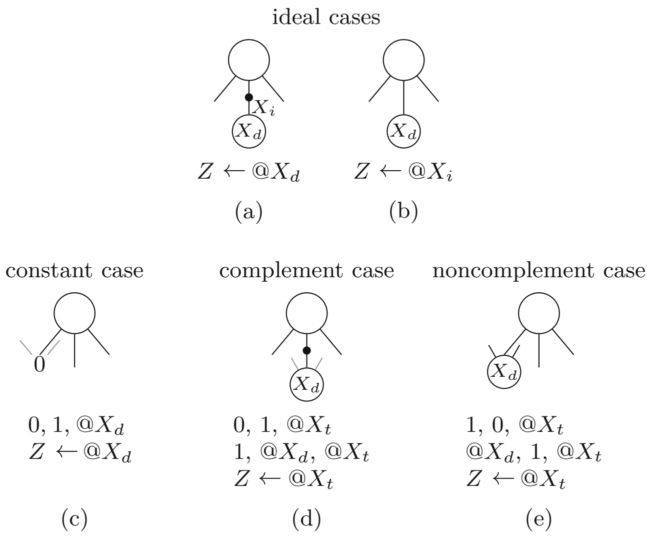


Fig. 5.5 Selecting the third operand or destination of an instruction denoted by Z

one or two additional instructions and one additional RRAM device. Note that one of the children has already been selected as operand B and that this is implicitly considered in the following descriptions.

(a) *There is a complemented child with one fanout, and there exists an RRAM device with its complemented value:* The existing RRAM device X_i for the complemented value can be used and it is safe to override it, since the child does not fan out to other parents.
(b) *There is a noncomplemented child with one fanout:* The RRAM device of this child can be used as destination and it is safe to override it. Note that case (a) is preferable compared to this one to avoid complemented children for operand A, i.e. the first operand of the instruction.
(c) *There is a constant child:* If there is a constant child (with or without multiple fanouts) a new RRAM device is allocated and initialized to the constant value (considering complemented edges into account).
(d) *There is a complemented child:* If there is an inverted child X_d (with or without multiple fanouts), a new RRAM device X_t is allocated and initialized to the complement of X_d using two RM3 instructions.
(e) *There is a noncomplemented child with multiple fanouts:* The first child X_d is selected and it's value is copied into a new allocated RRAM device X_t using two RM3 instructions.

5.2.3.5 Select Operand *A*

The child that is selected as operand A, i.e. the first operand of the instruction, is uniquely determined at this point since operand B and destination Z have been selected. Consequently, there is no case distinction with respect to preference. However, there are still different actions to be taken depending on the child node.

5.2.4 Experimental Results

The results of evaluating our approach for the EPFL benchmarks[1] is given in Table 5.1. The second column includes results for a naïve translation, where only the candidate selection scheme is disabled, based on the initial nonoptimized MIGs. The third and the forth columns represent results after MIG rewriting and both rewriting and compilation, respectively. The number of iterations of the MIG rewriting algorithm is set to 4 during all experiments. The instructions and required RRAM devices are shown by I and R, and the number of MIG nodes N is provided to give a better understanding of the MIG before and after rewriting. It is clear that N also shows the number of MIG nodes for the compiled PLiM, since the same MIG after rewriting has been used.

As expected, the number of MIG nodes have been reduced or remained unchanged for a few cases after MIG rewriting. Although, the number of nodes after MIG rewriting does not show a significant reduction, the sum of the number of instructions is reduced up to 20.09% compared to the naïve translation. This besides the 14.83% reduction achieved in the total number of RRAM devices imply the effectiveness of the employed techniques for removing multiple inverted edges.

Performing both MIG rewriting and our optimized compilation approach, the number of required instructions and RRAM devices reduces notably. The sum of the number of instructions and RRAM devices for the compiled PLiM are reduced by up to 19.95% and 61.4%, respectively in comparison with the corresponding values obtained for the naïve PLiM. This represents a significant reduction in both the latency and especially storage space metrics.

5.3 Logic-in-Memory Computing Write Traffic

As a non-volatile memory technology, RRAM has many advantages when compared to SRAM and eDRAM. However, RRAM has a limited write endurance. In the best existing RRAM architectures, a cell can endure about 10^{10} [53] to 10^{11} write counts [46]. There are many approaches that try to improve memory lifetime. Some

[1] http://lsi.epfl.ch/benchmarks.

Table 5.1 Experimental evaluation for the PLiM compiler

		Naïve			MIG rewriting					Rewriting and compilation			
Benchmark	PI/PO	#*N*	#*I*	#*R*	#*N*	#*I*	Impr. (%)	#*R*	Impr. (%)	#*I*	Impr. (%)	#*R*	Impr. (%)
adder	256/129	1020	2844	512	1020	2037	28.38	386	24.61	1911	32.81	259	49.41
bar	135/128	3336	8136	523	3240	5895	27.54	371	29.06	6011	26.12	332	36.52
div	128/128	57,247	146,617	687	50,841	147,026	−0.03	771	−12.22	147,608	−0.68	590	14.12
log2	32/32	32,060	78,885	1597	31,419	60,402	23.43	1487	6.89	60,184	23.71	1256	21.35
max	512/130	2865	6731	1021	2845	5092	24.35	867	15.08	4996	25.78	579	43.29
multiplier	128/128	27,062	76,156	2798	26,951	56,428	25.91	1672	40.24	56,009	26.45	419	85.03
sin	24/25	5416	12,479	438	5344	10,300	17.09	426	2.73	10,223	18.08	402	8.22
sqrt	128/64	24,618	60,691	375	22,351	47,454	21.81	433	−15.46	49782	17.97	323	13.87
square	64/128	18,484	54,704	3272	18,085	33,625	38.53	3247	0.76	33,369	39.00	452	86.19
cavlc	10/11	693	1919	262	691	1146	40.28	236	9.92	1124	41.43	102	61.07
ctrl	7/26	174	499	66	156	258	48.29	55	16.66	263	47.29	39	40.91
dec	8/256	304	822	257	304	783	4.74	257	0.00	777	5.47	258	−0.39
i2c	147/142	1342	3314	545	1311	2119	36.05	487	10.64	2028	38.81	234	57.06
int2float	11/7	260	648	99	257	432	33.33	83	16.16	428	33.95	41	58.59
mem_ctrl	1204/1231	46,836	113,244	8127	46,519	85,785	24.25	6708	17.46	84,963	24.97	2223	72.65
priority	128/8	978	2461	315	977	2126	13.61	241	23.49	2147	12.76	149	52.70
router	60/30	257	503	117	257	407	19.09	112	4.27	401	20.28	64	45.30
voter	1001/1	13,758	38,002	1749	12,992	25,009	34.19	1544	11.72	24,990	34.24	1063	39.22
$\sum$		236,710	608,655	22,760	225,560	486,324	20.09	19,383	14.83	487,214	19.95	8785	61.40

#*N* number of MIG nodes, #*I* number of RM$_3$ instructions, #*R* number of RRAM devices, improvement is calculated compared to naïve approaches are based on a write balancing scheme [62, 69, 100] mainly proposed for another non-volatile memory technology known as *Phase-Change Memory* (PCM), which has a lower write endurance compared to RRAM [93]. There are also other approaches which use error correcting techniques to fix the hard failures [66].

When using RRAM devices for in-memory computing, some memory cells face much higher write counts than others. In this section, we briefly survey the sources of unbalanced write traffic caused by different synthesis approaches for in-memory computing as well as basic operations allowing to use the resistive switching property.

So far a variety of approaches has been proposed for synthesis of logic-in-memory computing circuits, mostly based on material implication (IMP) [51, 54] and some using other basic operations, i.e. MAGIC [50] and RM_3 (also called MAJ throughout this book) [35]. In all these approaches, the target function can only be executed using a certain number of RRAM devices and after a number of operations, which increases rapidly as the size of the function increases. Hence, lowering the number of operations and the number of resistive memories is considered the main target of current in-memory computing synthesis approaches.

In [12], an IMP-based NAND gate was proposed. The gate is implemented with two resistive switches and the NAND function is executed within three computational steps. Regardless of the required operations to load the primary inputs of the gate into the two allocated devices, only one of them, the so-called work device, is rewritten after each operation while the other keeps its initial value. Using IMP for synthesis, this unbalanced distribution of writes happens due to the lack of commutativity property, which results in higher write traffic in the memory cell storing the output of the operation. Nevertheless, the write traffic can be spread more evenly over the entire memory by considering extra RRAM devices to replace those with high write counts. This will also require additional operations to copy the contents of RRAM devices into fresh ones or ones with lower write counts. However, being unwilling to spend more time or area, this intrinsic unbalanced write can become even worse. For example, [55] proposes a synthesis approach that considers only two work resistive devices besides N input devices, where N is the number of input variables of the Boolean function. It is obvious that the work devices used in implementations based on such an approach suffer from short lifetime.

RM_3 [35] was shown more efficient compared to material implication with respect to both area and latency when using *Majority-Inverter Graphs* (MIG, [1]) [75]. Regarding write endurance, RM_3 has more flexibility in comparison with IMP by sharing the writes between three operands instead of one. Nevertheless, RM_3 does not benefit from the commutativity property of the majority function due to the inversion of the second operand. Moreover, similarly to IMP, storing the result of the RM_3 in one of the involved memory devices can cause an unbalanced write if the same device is updated each time in a chain of operations.

In the following, it is shown how compilation for PLiM is performed in an endurance-aware manner by allocating wear leveling techniques.

5.4 Endurance-Aware Compilation of PLiM

In this section, we propose four techniques which jointly balance the write traffic for the PLiM computer [77]. Two techniques work directly based on the write counts of the memory cells, and the others based on MIG rewriting and compilation to address the intrinsic sources of unbalanced write existing in MIGs.

5.4.1 Direct Wear Leveling for PLiM

During PLiM compilation, write endurance can easily be considered whenever an RRAM device is requested such that if there are any freed RRAM devices at the time of request, the one with the smallest write count is returned. We call this *"minimum write count strategy"* which is a simple but also an essential technique to lower the deviation of writes. However, it is not still sufficient to obtain a good write balance. In this work, we also consider the conflicts of area and latency with the write distribution of the resulting PLiM programs. Indeed, we try to address those parts of the compilation procedure that worsen the write traffic but keep the number of instructions and RRAM devices as low as possible.

For example, computing the MIG shown in Fig. 5.6 by PLiM, the writes can not be balanced completely over all allocated RRAM devices unless additional instructions and devices are consumed. We assume that node A is already computed and node B is the best candidate for PLiM to be computed next. In this case, the compiler chooses the RRAM device storing the value of node A, let us call it X, as the destination of RM3. This is because the RRAM devices storing the two other children nodes of B have more than one fanout. Selecting a node with multiple fanouts as the destination of RM3 will require two extra instructions and one extra RRAM device to copy the values of the nodes to use them later at their other fanout targets. Therefore, node A is written again regardless of its current number of writes, and its content is replaced by the result of computation of node B. A similar situation occurs when computing node C. To compute node C, the compiler first sets the only complemented child shown with a dotted edge, i.e. node D, as the second operand of RM3 for the sake of efficiency. Between the two other remaining children, only node B has a single fanout whose value is stored in RRAM device X. Accordingly, again X will be selected as the destination of operation in order to avoid extra costs, while the RRAM device keeping the value of node D might have much smaller write count.

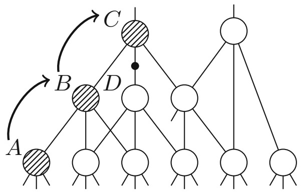


Fig. 5.6 Example MIG vulnerable to unbalanced write caused by PLiM's area latency considerations

As shown in the example above, rewriting the same RRAM device repeatedly can continue for a large number of node computations depending on the situation of single fanouts and complemented edges in the MIG. This condition causes an unbalanced write distribution which can not be controlled without extra costs in terms of execution time and area. A solution for this problem is to consider a maximum allowed number of writes per RRAM device. This way, the RRAM devices which reach this maximum write count are kept out of the free set and can not be requested anymore. As a result, a fresh RRAM device should be allocated which increases the number of devices. Moreover, the number of instructions can also increase since the ideal case of nodes with a single fanout child can only be exploited if the write threshold is satisfied. Otherwise, two extra instructions must be performed for computing the candidate node to load one of its children nodes into a fresh RRAM device or one with a valid write count to be used as the destination. We refer to this technique as the *"maximum write count strategy."*

The two strategies mentioned above can be utilized for any in-memory computing method with endurance considerations in general. To achieve a more comprehensive wear leveling scheme for the PLiM computer, in the following we refer to some key characteristics specifically applicable to PLiM. We first study how a slightly different MIG rewriting algorithm can enhance the distribution of writes over the memory. Then, we propose an endurance-aware node selection during compilation to get a better distribution of write traffic.

5.4.2 Endurance-Aware MIG Rewriting

Algorithm 3 describes the proposed endurance-aware MIG rewriting. The algorithm improves the write balance as well as maintaining the main role of MIG rewriting, i.e. minimizing the number of instructions, by applying $\Omega.M$ and $\Omega.D_{R\to L}$ to reduce the number of nodes, and inverter propagation axioms, $\Omega.I_{R\to L(1-3)}$ and $\Omega.I_{R\to L}$ to control the extra instructions required for complemented edges (see Sect. 4.2.3.3).

Algorithm 3: Endurance-aware MIG rewriting for PLiM architecture

```
1  for (cycles = 0; cycles < effort; cycles++) do
2      $\Omega.M$; $\Omega.D_{R\to L}$;
3      $\Omega.I_{R\to L(1-3)}$;
4      $\Omega.I_{R\to L}$;
5      $\Omega.A$;
6      $\Omega.I_{R\to L(1-3)}$;
7      $\Omega.I_{R\to L}$;
8      $\Omega.M$; $\Omega.D_{R\to L}$;
9      $\Omega.I_{R\to L}$;
10 end
```

In comparison to Algorithm 1, we have removed $\Omega.C$ which was disadvantageous due to removing a single complemented edge of an MIG node, which is actually considered an ideally low cost case for the RM_3 operation. Instead, $\Omega.A$ is sandwiched by two sets of inverter propagation axioms (Steps 3–7) to maximize the gain by presence of ideal nodes with a single inverted child. This also reshapes the MIG and creates more opportunities for further reduction in the number of nodes (Step 8). At the end, $\Omega.I_{R\rightarrow L}$ (Step 9) is applied to eliminate the costly nodes with three inverted children.

Algorithm 3 does not explicitly target cases such as repeatedly writing a memory cell due to the condition of fanouts or complemented edges, but it can decrease the number of nodes further and results in a more useful distribution of complemented edges. This effects lead to a lower total number of writes and increase the possibility of regular change of the switching device.

5.4.3 Endurance-Aware Node Selection

As explained before, the write traffic highly depends on the MIG features, including the condition of inverters and fanouts in the graph. Here, we discuss another MIG feature which can have a considerable effect on the write distribution.

The structure of the MIG can help to distribute the writes more evenly if RRAM devices in use are released and reused with a similar frequency. Indeed, the issue with unbalanced write traffic happens when some RRAM devices are blocked for a long time and some others are rewritten very often. In such a situation, the number of writes for the devices used to execute the PLiM program can vary widely. Thus, some RRAM devices reach to the end of their lifetime much faster than others which shortens the normal duration that the PLiM computer can work reliably.

Figure 5.7 shows an MIG with the problem explained above. Let us assume that node A is already computed and X_A designates the RRAM device keeping its value. It is obvious that X_A is still required and can not be released until node G is computed. Node A targets several nodes in higher levels which means longer waiting time for X_A, while all other nodes only target the nodes in the very next levels. In comparison, the RRAM devices storing the values of nodes B and C,

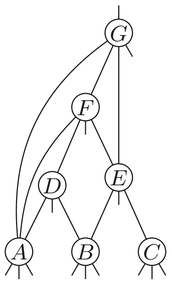


Fig. 5.7 Example MIG vulnerable to unbalanced write due to possessing nodes with long storage durations

indicated by X_B and X_C, respectively, can be released when nodes D and E are computed and one of them can be rewritten again to compute node F. X_B and X_C can also be rewritten before by the values of their target nodes D and E. Considering that the initial number of writes has been equal for all the mentioned RRAM devices, X_B and X_C might have been rewritten at lease one or two times more than X_A, which we call a blocked device, when the root node G is computed.

The impact of blocked RRAM devices on write traffic can be much more noticeable in large MIGs with high level differences. Nevertheless, write balance can be enhanced if we compute the nodes with long waiting time as late as possible. In the example above, the write traffic can be managed if the compiler computes nodes B and C before A. For this purpose, we reverse the priority of nodes in the compilation procedure. Thus, the candidate nodes with the smallest fanout level index, which means the shortest storage duration, are computed first. The second node selection principle, i.e. number of releasing RRAM devices, is considered if there are several candidate nodes with equal fanout level indices. In this case, the node with the largest number of releasing RRAM devices is selected for computation. Algorithm 4 describes the procedure to select between two candidate nodes.

Algorithm 4: Comparison of MIG nodes for the endurance-aware node selection

```
1  forall the MIG nodes A and B ∈ Candidate Set do
2      if (A/B has the smaller fanout level index) then
3          Select A/B;
4      end
5      else if |releasing_RRAMs(A)| > |releasing_RRAMs(B)| then
6          Select A;
7      end
8      else
9          Select B;
10     end
11 end
```

Although using this endurance-aware node selection strategy for PLiM compiler improves the memory write balance, it slightly lowers efficiency in comparison with the initial node selection scheme. Nevertheless, the small increase in the number of instructions and RRAM devices might not matter regarding the write traffic improvement.

It is worth noting that the unbalanced write distribution caused by blocked RRAM devices waiting for quite long time can not be eliminated but only decreased. In fact, an evenly distributed write traffic for an MIG with large differences between fanout origins and targets is not possible by only postponing computation of nodes with longer waiting times. The sequential nature of PLiM architecture anyway imposes a waiting list of blocked RRAM devices which can not be released until the root node or nodes close to it are computed. We consider this factor as the main difficulty of balancing writes for a generic MIG-based in-memory computing architecture constructed upon a resistive crossbar array.

The issue of blocked RRAM devices could be considered as an objective during MIG rewriting to keep the level differences between connected nodes low. However, MIGs optimized this way might not be favorable with respect to the length of instructions. The proposed MIG rewriting focuses on reducing the number of nodes and complemented edges which have a direct impact on latency. Reducing the number of nodes already makes the connections between nodes more complicated which means higher number of nodes targeting farther levels.

5.4.4 Experimental Results

We have carried out experiments on the same set of benchmarks used for evaluation of the PLiM compiler.[2] The benchmark set consists of 18 functions including large arithmetic and random control functions possessing up to 1204 primary inputs and 1231 primary outputs. The number of cycles for MIG rewriting, i.e. effort, is set to 5 in all experiments. The standard deviation, which is known as a robust statistical metric, is used to describe the distribution of writes over entire memory cells required for computing an MIG representing a Boolean function. Minimum and maximum number of writes are also given for a more precise statistical description.

Table 5.2 shows the standard deviation, and minimum and maximum number of writes performed for the RRAM devices required by the PLiM computer for each benchmark function. The results are given in an incremental manner in different columns to clearly show the effect of each proposed endurance-aware technique, excluding the maximum write strategy. The maximum write strategy is not integrated into the techniques used in Table 5.2. Full endurance considerations including all of the four presented schemes is provided in another table for different values of write threshold. In Table 5.2, the improvement of standard deviation obtained by each technique is compared to the corresponding naïve implementations, which only benefit from node translation but not MIG rewriting and node selection.

To see the effect of the general purpose compilation, we have also provided the statistical values representing the write distribution obtained by the PLiM compiler (see Sect. 5.2) in the second group columns. The PLiM compiler [83, 84] tackles the number of instructions and RRAM devices as the cost metrics, however, the techniques employed also improve write traffic due to the shorter release time of RRAM devices. Comparison of the first three group columns shows that the write balance improvement of the PLiM compiler [83] is increased from the overall average value of 30.95–57.07% after only adding the presented minimum write strategy. This improvement increases further up to 64.42% by use of the proposed endurance-aware MIG rewriting instead of the previous version in Algorithm 1. Finally, adding the endurance-aware compilation to the proposed MIG rewriting

[2]http://lsi.epfl.ch/benchmarks.

Table 5.2 Experimental evaluation of the proposed endurance-aware techniques

	Naïve		PLiM compiler [83]			Min. write strategy			Min. write strategy + endu.-aware MIG rewr.			Min. write strategy + endu.-aware MIG rewr. and compil.		
Benchmark	Min/max	STDEV	Min/max	STDEV	Impr. (%)	Min/max	STDEV	Impr. (%)	Min/max	STDEV	Impr. (%)	Min/max	STDEV	Impr. (%)
adder	0/84	12.60	0/34	6.09	51.66	2/9	0.90	92.86	2/9	2.49	80.24	2/13	1.55	87.70
bar	0/87	11.97	0/122	7.97	33.41	0/106	5.64	52.88	0/26	3.07	74.35	0/18	1.69	85.88
div	2/459	121.98	1/757	227.73	−86.69	2/596	233.38	−91.33	1/381	140.49	−15.17	1/340	119.53	2.01
log2	1/286	49.92	2/89	15.73	68.48	14/57	3.62	92.75	7/55	3.24	93.51	10/55	3.27	93.45
max	1/77	11.80	1/105	11.35	3.81	1/88	7.72	34.58	2/88	7.41	37.20	2/17	2.65	77.54
multiplier	0/1196	179.90	2/289	32.77	81.78	5/206	18.74	89.58	4/24	1.69	99.06	5/24	1.53	99.15
sin	0/166	27.17	0/50	8.47	68.82	6/40	2.79	89.73	3/42	3.04	88.81	3/48	4.42	83.73
sqrt	2/440	145.16	1/360	94.11	35.16	4/228	73.90	49.09	4/267	82.34	43.28	2/230	68.36	52.91
square	0/577	95.43	0/48	4.35	95.44	1/43	1.87	98.04	1/28	1.95	97.96	1/22	1.96	97.95
cavlc	0/73	15.36	0/44	10.30	32.94	0/18	3.16	79.43	0/18	3.17	79.36	0/13	2.36	84.64
ctrl	0/28	7.44	1/38	9.10	−22.31	1/15	3.12	58.06	1/17	3.93	47.18	1/11	1.79	75.94
dec	1/6	0.46	2/5	0.45	2.17	2/6	0.57	−23.91	2/6	0.57	−23.91	2/6	0.57	−23.91
i2c	0/114	14.07	0/88	10.87	22.74	0/24	4.23	69.94	0/31	4.44	68.44	0/16	3.04	78.39
int2float	0/49	13.28	0/52	11.87	10.61	0/22	4.70	64.61	0/19	3.93	70.41	0/12	2.69	79.74
mem_ctrl	0/553	80.47	0/175	21.07	73.81	0/89	11.78	85.36	0/56	9.12	88.67	0/97	11.03	86.29
priority	0/496	45.57	0/101	14.37	68.46	2/35	5.98	86.88	2/45	7.52	83.50	2/49	7.43	83.70
router	0/52	8.71	0/33	6.30	27.66	0/21	3.57	59.01	1/23	3.91	55.11	1/19	3.57	59.01
voter	2/156	31.66	0/188	35.09	−10.83	13/793	19.11	39.64	13/25	1.53	95.17	10/23	1.59	94.98
AVG	0.5/272.16	48.49	0.55/163.72	29.33	30.95	2.94/133.11	22.48	57.07	2.38/67.70	15.07	64.42	2.33/56.27	13.27	72.17

Min/max minimum/maximum number of writes, *STDEV* standard deviation of write counts, *impr.* improvement of standard deviation is calculated compared to naïve and minimum write count strategy reduces the standard deviation of writes further by 72.17% in comparison to the naïve approach.

Table 5.3 compares the number of instructions and RRAM devices required for the PLiM programs using only endurance-aware rewriting and both endurance-aware rewriting and compilation with the naïve approach. As shown in the table, the average number of instructions and RRAM devices slightly increases by adding the endurance-aware compilation. Nevertheless, considering the 72.17% improvement achieved in the write traffic, the PLiM programs obtained by the endurance-aware MIG rewriting and compilation also have average reductions of 36.48% and 18.18% in the number of instructions and RRAM devices, respectively. It should be noted that the minimum write count strategy does not influence the number of required instructions and RRAM devices. The minimum write strategy selects an RRAM device with the smallest write count from the set of free RRAM devices when requested. This only contributes to a better distribution of the write counts, and can not increase or decrease the number of instructions or RRAM devices.

Affording a number of instructions and RRAM devices higher than that shown in Table 5.3, almost any desired write traffic is accessible using the suggested maximum write count strategy. Table 5.4 shows the results of full endurance considerations exploiting the minimum and maximum write strategies as well as

Table 5.3 Number of instructions and RRAM devices required for endurance-aware compilation of PLiM

	Naïve		Endurance-aware MIG rewriting		Endurance-aware rewriting and compilation	
Benchmark	#*I*	#*R*	#*I*	#*R*	#*I*	#*R*
adder	2844	512	1656	258	1657	355
bar	8136	523	5103	264	5103	391
div	146,617	687	100,071	620	101,318	496
log2	78,885	1597	48,692	1344	48,665	1347
max	6731	1021	3902	660	4042	749
multiplier	76,156	2798	45,966	4297	45,847	4301
sin	12,479	438	9156	377	9211	373
sqrt	60,691	375	39,434	333	39,464	371
square	54,704	3272	29,757	2387	30,044	2714
cavlc	1919	262	1045	148	1058	186
ctrl	499	66	268	38	273	48
dec	822	257	777	258	777	258
i2c	3314	545	1946	305	1971	365
int2float	648	99	368	50	378	63
mem_ctrl	113,244	8127	74,577	4357	74,588	4724
priority	2461	315	1417	138	1464	140
router	503	117	371	66	364	73
voter	38,002	1749	20,208	1337	20,406	1667
AVG	33,814.16	1264.44	21,373	957.61	21,479.44	1034.50

#*I* number of RM_3 instructions, #*R* number of RRAMs

Table 5.4 Results of full endurance considerations with maximum write strategy for write values of 10, 20, 50, and 100

	10			20			50			100		
Benchmark	#I	#R	STDEV	#I	#R	STDEV	#I	#R	STDEV	#I	#R	STDEV
adder	1659	362	1.12	1657	355	1.55	–	–	–	–	–	–
bar	5351	603	1.37	5103	391	1.69	–	–	–	–	–	–
div	103,057	11,691	0.91	102,187	5551	2.04	101,615	2327	11.76	101,431	1295	31.59
log2	51,063	5559	0.85	49,603	2643	1.36	48,666	1347	3.25	48,665	1347	3.27
max	4076	776	2.06	4042	749	2.65	–	–	–	–	–	–
multiplier	47,819	5225	0.97	45,854	4301	1.51	45,847	4301	1.53	–	–	–
sin	9562	1069	1.18	9336	534	2.28	9211	373	4.42	–	–	–
sqrt	40,991	4548	1.04	39,956	2248	3.29	39,642	994	14.55	39,514	598	33.81
square	31,411	3478	1.07	30,051	2714	1.95	30,044	2714	1.96	–	–	–
cavlc	1067	186	2.05	1058	186	2.36	–	–	–	–	–	–
ctrl	274	48	1.74	273	48	1.79	–	–	–	–	–	–
dec	777	258	0.57	–	–	–	–	–	–	–	–	–
i2c	1985	377	2.54	1971	365	3.04	–	–	–	–	–	–
int2float	381	63	2.47	378	63	2.69	–	–	–	–	–	–
mem_ctrl	78,216	9194	2.07	75,403	5940	5.32	74,619	4777	10.57	74,588	4724	11.03
priority	1545	206	2.27	1494	140	6.27	1464	140	7.43	–	–	–
router	367	74	2.91	364	73	3.57	–	–	–	–	–	–
voter	21,538	2350	0.86	20,408	1667	1.58	20,406	1667	1.59	–	–	–
AVG	22,285.50	2559.27	**1.55**	21,661.94	1568.11	2.66	21,507.61	1173.77	4.27	21,488.50	1091.50	6.47

#I number of RM$_3$ instructions, *#R* number of RRAMs, *STDEV* standard deviation of write counts, – shows that the value has not been changed

the endurance-aware MIG rewriting and compilation. The number of instructions, RRAM devices and the standard deviation of the writes performed to execute the PLiM programs under maximum write constraints of 10, 20, 50, and 100 are presented to show the relation between the improvement in write distribution and additional penalties with respect to delay and area. Indeed, Table 5.4 provides different trade-offs between write endurance, latency, and area for the resulting implementations.

According to Table 5.2, results for some benchmarks remain unchanged for constraint values which are higher than the benchmarks' natural maximum number of writes. This unchanged values are indicated by dashes in the table and their value can be found on the corresponding cells with a lower write constraint. As expected, the number of instructions and RRAMs decrease as the write constraint becomes looser, while the write deviation worsens. An average improvement of 96.8% in standard deviation is obtained when a maximum allowed write value of 10 is set in the compiler. This improvement in the write distribution is obtained at a cost of 50.59% increase in the number of RRAM devices compared to the naïve approach. Nevertheless, the number of instructions has slightly increased and still shows a considerable reduction in comparison to the naïve solution. Results for a maximum write count of 100 can be considered as a good trade-off, due to an overall improvement of 86.85% in the write balance as well as reducing the average number of instructions and RRAM devices by 36.45% and 13.67%, respectively.

5.5 Summary

This chapter presents algorithms to optimize and automatically translate large Boolean functions into programs for the in-memory computing PLiM architecture. The PLiM uses a majority based operation for computing on a resistive crossbar array and therefore, the target functions for implementation are first presented by MIGs. We observed that both the MIG representation and the way in which an MIG is compiled has a large impact on the resulting PLiM programs—in terms of required instructions as well as number of RRAM devices. Experiments show that compared to a naïve translation approach the number of instructions and especially the number of RRAM devices can be reduced considerably. Our algorithm unlocks for the first time the potential of the PLiM architecture to process large scale computer programs using in-memory computing. This makes this promising emerging technology available for nontrivial applications.

The chapter also addresses endurance constraints for in-memory computing synthesis approaches. We proposed techniques for endurance-aware optimization and compilation for the PLiM architecture. The proposed approach maintains a trade off between write endurance and other cost metrics of the resulting implementations including area and latency. The experimental results reveal considerable reduction in the standard deviation of writes as well as noticeable improvements in the lengths of instructions and number of RRAM devices compared to a naïve approach.

Chapter 6
Conclusions

In today's computer architectures, i.e., von Neumann architecture, processors are responsible for computing while the memory stores the raw or processed data. However, the advancements in the processors of modern computers have far exceeded that of memory. The considerably higher latency of memory compared to processor on one hand, and the requirement of communication between these two in the von Neumann architecture on the other hand, has limited the overall performance of current computing systems which is known as *memory wall*. The growing need to deal with higher amount of data demanded by emerging applications such as *Internet of Things* (IoT) and *big data* has prompted much research to alleviate this problem [84]. Among these solutions, *in-memory computing* sounds to be very promising as it allows to go beyond the memory wall by integrating the storage and computing paradigms.

This book studies in-memory computing enabled by an emerging memory technology called *Resistive Random Access Memory* (RRAM) from two perspectives, i.e., customized and instruction based. The first perspective customizes the classical approaches for logic synthesis such that they can be exploited to compute within RRAM devices. The presented approach employs traditional and recent logic representations which are already proven to be efficient in classical logic synthesis both theoretically and experimentally. The design procedure starts with finding efficient realizations with RRAM devices for the logic primitive of each representation, then continues based on a design methodology for implementation on RRAM array. This customized approach also provides several heuristic optimization algorithms for the logic representations with respect to the area and latency which are shown to be very efficient in comparison with the state-of-art. These contributions regarding the customized synthesis approach for in-memory computing and optimization algorithms for the employed logic representations, are presented in Chaps. 3 and 4.

The instruction-based approach presented in Chap. 5 considers in-memory computing for a computer architecture which is programmable and thus can compute any arbitrary logic function. This approach provides a step-by-step automated procedure

© Springer Nature Switzerland AG 2020

S. Shirinzadeh, R. Drechsler, *In-Memory Computing*,
https://doi.org/10.1007/978-3-030-18026-3_6

which starts from a gate level description and obtains an efficient logic-in-memory program addressing factors such as latency, area, and reliability. To be more precise, the proposed instruction oriented approach introduces an automated compiler for logic-in-memory computer architectures for the first time, optimization algorithms to obtain faster programs requiring smaller number of RRAM devices, and several wear leveling techniques to increase the lifetime of architecture. The experimental results prove the functionality of the proposed techniques when compared to the naïve implementations.

In summary, the contributions of this book propose a comprehensive framework for the promising field of logic-in-memory computing at gate level. This provides a firm foundation for this emerging computing paradigm allowing various applications and creates opportunities for further research and enhancements in different aspects of programmable memristive hardwares and architectures.

References

1. Amarù, L.G., Gaillardon, P.E., De Micheli, G.: Majority-inverter graph: a novel data-structure and algorithms for efficient logic optimization. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 194:1–194:6. IEEE, Piscataway (2014)
2. Amarù, L., Petkovska, A., Gaillardon, P.E., Bruna, D.N., Ienne, P., De Micheli, G.: Majority-inverter graph for FPGA synthesis. In: Proceedings of the 19th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI) (2015)
3. Amarù, L.G., Gaillardon, P.E., De Micheli, G.: Majority-inverter graph: a new paradigm for logic optimization. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **35**(5), 806–819 (2016)
4. Beatty, J.C.: An axiomatic approach to code optimization for expressions. J. ACM **19**(4), 613–640 (1972)
5. Berkeley Logic Synthesis and Verification Group: ABC—a system for sequential synthesis and verification (2005). http://www.eecs.berkeley.edu/~alanmi/abc/
6. Bhattacharjee, D., Devadoss, R., Chattopadhyay, A.: ReVAMP: ReRAM based VLIW architecture for in-memory computing. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 782–787. IEEE, Piscataway (2017)
7. Bhattacharjee, D., Amarù, L., Chattopadhyay, A.: Technology-aware logic synthesis for ReRAM based in-memory computing. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1435–1440. IEEE, Piscataway (2018)
8. Biolek, D., Biolkova, V., Biolek, Z.: SPICE model of memristor with nonlinear dopant drift. Radioengineering **18**(2), 210–214 (2009)
9. Birkhoff, G., Kiss, S.A.: A ternary operation in distributive lattices. Bull. Am. Math. Soc. **53**(8), 749–752 (1947)
10. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Comput. **45**(9), 993–1002 (1996)
11. Bollig, B., Löbbing, M., Wegener, I.: Simulated annealing to improve variable orderings for OBDDs. In: International Workshop on Logic Synth (1995)
12. Borghetti, J., Snider, G.S., Kuekes, P., Yang, J.J., Stewart, D.R., Williams, R.S.: 'Memristive' switches enable 'stateful' logic operations via material implication. Nature **464**, 873–876 (2010)
13. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: Proceedings of the 27th Annual Design Automation Conference (DAC), pp. 40–45. IEEE, Piscataway (1990)

© Springer Nature Switzerland AG 2020
S. Shirinzadeh, R. Drechsler, *In-Memory Computing*,
https://doi.org/10.1007/978-3-030-18026-3

14. Brglez, F., Bryan, D., Kozminski, K.: Combinational profiles of sequential benchmark circuits. In: International Symposium on Circuits and Systems, pp. 1929–1934. IEEE, Piscataway (1989)
15. Bürger, J., Teuscher, C., Perkowski, M.: Digital logic synthesis for memristors. Reed-Muller **2013**, 31–40 (2013)
16. Chakraborti, S., Chowdhary, P., Datta, K., Sengupta, I.: BDD based synthesis of Boolean functions using memristors. In: 9th International Design and Test Symposium (IDT), pp. 136–141. IEEE, Piscataway (2014)
17. Chattopadhyay, A., Rakosi, Z.: Combinational logic synthesis for material implication. In: 2011 IEEE/IFIP 19th International Conference on VLSI and System-on-Chip, pp. 200–203. IEEE, Piscataway (2011)
18. Cheng, K.T.T., Strukov, D.B.: 3D CMOS-memristor hybrid circuits: devices, integration, architecture, and applications. In: Proceedings of the 2012 ACM International Symposium on Physical Design, pp. 33–40. ACM, New York (2012)
19. Chua, L.: Memristor-The missing circuit element. IEEE Trans. Circuit Theory **18**(5), 507–519 (1971)
20. Chua, L.: Resistance switching memories are memristors. Appl. Phys. A **102**(4), 765–783 (2011)
21. De Micheli, G.: Synthesis and Optimization of Digital Circuits. McGraw-Hill Higher Education, New York (1994)
22. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
23. Drechsler, R., Sieling, D.: Binary decision diagrams in theory and practice. Softw. Tools Technol. Transfer **3**(2), 112–136 (2001)
24. Drechsler, R., Becker, B., Göckel, N.: A genetic algorithm for variable ordering of OBDDs. IEE Proc. Comput. Digit. Tech. **143**(6), 364–368 (1996)
25. Drechsler, N., Drechsler, R., Becker, B.: Multi-objective optimisation based on relation favour. In: International Conference on Evolutionary Multi-Criterion Optimization, vol. 1993, pp. 154–166. Springer, Berlin (2001)
26. Drechsler, R., Shi, J., Fey, G.: Synthesis of fully testable circuits from BDDs. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **23**(3), 440–443 (2004)
27. Drechsler, N., Sülflow, A., Drechsler, R.: Incorporating user preferences in many-objective optimization using relation ε-preferred. Nat. Comput. **14**(3), 469–483 (2015)
28. Ebendt, R., Günther, W., Drechsler, R.: An improved branch and bound algorithm for exact BDD minimization. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **22**(12), 1657–1663 (2003)
29. Ebendt, R., Fey, G., Drechsler, R.: Advanced BDD Optimization. Springer, Berlin (2005)
30. Edelkamp, S., Mehler, T.: Byte code distance heuristics and trail direction for model checking Java programs. In: Workshop on Model Checking and Artificial Intelligence (MoChArt), pp. 69–76 (2003)
31. Edelkamp, S., Reffel, F.: OBDDs in heuristic search. In: Annual Conference on Artificial Intelligence. Lecture Notes in Computer Science, vol. 1504, pp. 81–92. Springer, Berlin (1998)
32. Fey, G., Drechsler, R.: Minimizing the number of paths in BDDs: theory and algorithm. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **25**(1), 4–11 (2006)
33. Friedman, S.J., Supowit, K.J.: Finding the optimal variable ordering for binary decision diagrams. In: Design Automation Conference, pp. 348–356. IEEE, Piscataway (1987)
34. Gaillardon, P.E., Tang, X., Sandrini, J., Thammasack, M., Omam, S.R., Sacchetto, D., Leblebici, Y., De Micheli, G.: A ultra-low-power FPGA based on monolithically integrated RRAMs. In: 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1203–1208. IEEE, Piscataway (2015)
35. Gaillardon, P.E., Amarú, L., Siemon, A., Linn, E., Waser, R., Chattopadhyay, A., De Micheli, G.: The programmable logic-in-memory (PLiM) computer. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 427–432. IEEE, Piscataway (2016)

36. Hamdioui, S., Taouil, M., Haron, N.Z.: Testing open defects in memristor-based memories. IEEE Trans. Comput. **64**(1), 247–259 (2015)
37. Han, J., Orshansky, M.: Approximate computing: an emerging paradigm for energy-efficient design. In: IEEE European Test Symposium, pp. 1–6. IEEE, Piscataway (2013)
38. Haron, N.Z., Hamdioui, S.: Why is CMOS scaling coming to an END? In: International Design and Test Workshop, pp. 98–103. IEEE, Piscataway (2008)
39. Hilgemeier, M., Drechsler, N., Drechsler, R.: Minimizing the number of one-paths in BDDs by an evolutionary algorithm. In: IEEE Congress on Evolutionary Computation, pp. 1724–1731. IEEE, Piscataway (2003)
40. Ho, Y., Huang, G.M., Li, P.: Dynamical properties and design analysis for nonvolatile memristor memories. IEEE Trans. Circuits Syst. **58**(4), 724–736 (2011)
41. Isbell, J.R.: Median algebra. Trans. Am. Math. Soc. **260**(2), 319–362 (1980)
42. Ishiura, N., Sawada, H., Yajima, S.: Minimization of binary decision diagrams based on exchanges of variables. In: International Conference on Computer-Aided Design, pp. 472–475. IEEE, Piscataway (1991)
43. Jensen, R.M., Hansen, E.A., Richards, S., Zhou, R.: Memory-efficient symbolic heuristic search. In: Proceedings of International Conference on Automated Planning and Scheduling, pp. 304–313. AAAI Press, Palo Alto (2006)
44. Jo, S.H., Kim, K.H., Lu, W.: High-density crossbar arrays based on a Si memristive system. Nano Lett. **9**(2), 870–874 (2009)
45. Joglekar, Y.N., Wolf, S.J.: The elusive memristor: properties of basic electrical circuits. Eur. J. Phys. **30**(4), 661 (2009)
46. Kim, Y.B., Lee, S.R., Lee, D., Lee, C.B., Chang, M., Hur, J.H., Lee, M.J., Park, G.S., Kim, C.J., Chung, U.I., Yoo, I.K., Kim, K.: Bi-layered rram with unlimited endurance and extremely uniform switching. In: Symposium on VLSI Technology, pp. 52–53. IEEE, Piscataway (2011)
47. Knuth, D.E.: The Art of Computer Programming, vol. 4A. Addison-Wesley, Reading (2011)
48. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust Boolean reasoning for equivalence checking and functional property verification. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **21**(12), 1377–1394 (2002)
49. Kvatinsky, S., Friedman, E.G., Kolodny, A., Weiser, U.C.: TEAM: threshold adaptive memristor model. IEEE Trans. Circuits Syst. I **60**(1), 211–221 (2013)
50. Kvatinsky, S., Belousov, D., Liman, S., Satat, G., Wald, N., Friedman, E., Kolodny, A., Weiser, U.: MAGIC – memristor-aided logic. IEEE Trans. Circuits Syst. II **61**(11), 895–899 (2014)
51. Kvatinsky, S., Satat, G., Wald, N., Friedman, E.G., Kolodny, A., Weiser, U.C.: Memristor-based material implication (IMPLY) logic: design principles and methodologies. IEEE Trans. Very Large Scale Integr. VLSI Syst. **22**(10), 2054–2066 (2014)
52. Kvatinsky, S., Ramadan, M., Friedman, E.G., Kolodny, A.: VTEAM: a general model for voltage-controlled memristors. IEEE Trans. Circuits Syst. II **62**(8), 786–790 (2015)
53. Lee, H.Y., Chen, Y.S., Chen, P.S., Gu, P.Y., Hsu, Y.Y., Wang, S.M., Liu, W.H., Tsai, C.H., Sheu, S.S., Chiang, P.C., Lin, W.P., Lin, C.H., Chen, W.S., Chen, F.T., Lien, C.H., Tsai, M.J.: Evidence and solution of over-reset problem for HfOX based resistive memory with sub-ns switching speed and high endurance. In: IEEE International Meeting on Electron Devices, pp. 19.7.1–19.7.4. IEEE, Piscataway (2010)
54. Lehtonen, E., Laiho, M.: Stateful implication logic with memristors. In: IEEE/ACM International Symposium on Nanoscale Architectures, pp. 33–36. IEEE, Piscataway (2009)
55. Lehtonen, E., Poikonen, J.H., Laiho, M.: Two memristors suffice to compute all boolean functions. Electron. Lett. **46**(3), 239–240 (2010)
56. Liu, K.C., Tzeng, W.H., Chang, K.M., Chan, Y.C., Kuo, C.C., Cheng, C.W.: The resistive switching characteristics of a Ti/Gd_2O_3/Pt RRAM device. Microelectron. Reliab. **50**(5), 670–673 (2010)

57. Lu, K., Ramin, Y., Edwin, Y., Constantinos, K.: Structural optimization of reduced ordered binary decision diagrams for SLA negotiation in IaaS of cloud computing. In: Proceedings of the 10th International Conference on Service-Oriented Computing, pp. 268–282. Springer, Berlin (2012)
58. Mishchenko, A., Perkowski, M.: Fast heuristic minimization of exclusive-sums-of-products. In: International Workshop on Applications of the Real-Muller Expansion in Circuit Design (2001)
59. Neumann, J.V.: First draft of a report on the EDVAC. Tech. rep. (1945)
60. Oliver, I.M., Smith, D.J., Holland, J.R.C.: Study of permutation crossover operators on the traveling salesman problem. In: International Conference on Genetic Algorithms, pp. 224–230. Erlbaum Associates, Hillsdale (1987)
61. Pershin, Y., Di Ventra, M.: Practical approach to programmable analog circuits with memristors. IEEE Trans. Circuits Syst. Regul. Pap. **57**(8), 1857–1864 (2010)
62. Qureshi, M.K., Karidis, J., Franceschini, M., Srinivasan, V., Lastras, L., Abali, B.: Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In: IEEE/ACM International Symposium on Microarchitecture, pp. 14–23. IEEE, Piscataway (2009)
63. Ravi, K., Somenzi, F.: High-density reachability analysis. In: Proceedings of IEEE International Conference on Computer Aided Design (ICCAD), pp. 154–158. IEEE, Piscataway (1995)
64. Reda, S., Drechsler, R., Orailoglu, A.: On the relation between SAT and BDDs for equivalence checking. In: Proceedings of the International Symposium on Quality Electronic Design, pp. 394–399. IEEE, Piscataway (2002)
65. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of IEEE/ACM International Conference on Computer-Aided Design, pp. 42–47. IEEE, Piscataway (1993)
66. Schechter, S., Loh, G.H., Strauss, K., Burger, D.: Use ECP, not ECC, for hard failures in resistive memories. In: International Symposium on Computer Architecture, pp. 141–152, ACM, New York (2010)
67. Schmiedle, F., Drechsler, N., Drechsler, R.: Priorities in multi-objective optimization for genetic programming. In: Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation, pp. 129–136. Morgan Kaufmann Publishers, San Francisco (2001)
68. Schmiedle, F., Drechsler, N., Große, D., Drechsler, R.: Heuristic learning based on genetic programming. Genet. Program Evolvable Mach. **3**(4), 363–388 (2002)
69. Seong, N.H., Woo, D.H., Lee, H.S.: Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In: International Symposium on Computer Architecture, pp. 383–394. ACM, New York (2010)
70. Shannon, C.E.: A symbolic analysis of relay and switching circuits (1940). Thesis (M.S.)–Massachusetts Institute of Technology, Dept. of Electrical Engineering
71. Shin, D., Gupta, S.K.: Approximate logic synthesis for error tolerant applications. In: 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010), pp. 957–960. IEEE, Piscataway (2010)
72. Shirinzadeh, S., Drechsler, R.: Logic synthesis for in-memory computing using resistive memories. In: IEEE Computer Society Annual Symposium on VLSI, pp. 375–380. IEEE, Piscataway (2018)
73. Shirinzadeh, S., Soeken, M., Drechsler, R.: Multi-objective BDD optimization with evolutionary algorithms. In: Genetic and Evolutionary Computation Conference, pp. 751–758. ACM, New York (2015)
74. Shirinzadeh, S., Soeken, M., Drechsler, R.: Multi-objective BDD optimization for RRAM based circuit design. In: IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, pp. 46–51. IEEE, Piscataway (2016)
75. Shirinzadeh, S., Soeken, M., Gaillardon, P.E., Drechsler, R.: Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 948–953. IEEE, Piscataway (2016)

76. Shirinzadeh, S., Soeken, M., Große, D., Drechsler, R.: Approximate BDD optimization with prioritized ε-preferred evolutionary algorithm. In: Genetic and Evolutionary Computation Conference, pp. 79–80. Association for Computing Machinery, New York (2016)
77. Shirinzadeh, S., Soeken, M., Gaillardon, P.E., De Micheli, G., Drechsler, R.: Endurance management for resistive logic-in-memory computing architectures. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1092–1097. IEEE, Piscataway (2017)
78. Shirinzadeh, S., Soeken, M., Gaillardon, P.E., Drechsler, R.: Logic synthesis for majority based in-memory computing. In: Vaidyanathan, S., Volos, C. (eds.) Advances in Memristors, Memristive Devices and Systems, pp. 425–448. Springer, Berlin (2017)
79. Shirinzadeh, S., Soeken, M., Große, D., Drechsler, R.: An adaptive prioritized ϵ-preferred evolutionary algorithm for approximate BDD optimization. In: Genetic and Evolutionary Computation Conference, pp. 1232–1239. ACM, New York (2017)
80. Shirinzadeh, S., Datta, K., Drechsler, R.: Logic design using memristors: an emerging technology. In: IEEE International Symposium on Multiple-Valued Logic, pp. 121–126. IEEE, Piscataway (2018)
81. Shirinzadeh, S., Soeken, M., Gaillardon, P.E., Drechsler, R.: Logic synthesis for RRAM-based in-memory computing. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(7), 1422–1435 (2018)
82. Soeken, M., Große, D., Chandrasekharan, A., Drechsler, R.: BDD minimization for approximate computing. In: Asia and South Pacific Design Automation Conference, pp. 474–479. IEEE, Piscataway (2016)
83. Soeken, M., Shirinzadeh, S., Gaillardon, P.E., Amarú, L.G., Drechsler, R., De Micheli, G.: An MIG-based compiler for programmable logic-in-memory architectures. In: 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 117:1–117:6. IEEE, Piscataway (2016)
84. Soeken, M., Gaillardon, P.E., Shirinzadeh, S., Drechsler, R., De Micheli, G.: A PLiM computer for the internet of things. IEEE Comput. **50**(6), 35–40 (2017)
85. Somenzi, F.: CUDD: CU Decision Diagram package release 2.5.0 (2012)
86. Strukov, D.B.: Smart connections. Nature **476**, 403–405 (2011)
87. Strukov, D.B., Snider, G.S., Stewart, D.R., Williams, R.S.: The missing memristor found. Nature **453**(7191), 80–83 (2008)
88. Sülflow, A., Drechsler, N., Drechsler, R.: Robust multi-objective optimization in high dimensional spaces. In: International Conference on Evolutionary Multi-Criterion Optimization, pp. 715–726. Springer, Berlin (2007)
89. Talati, N., Gupta, S., Mane, P., Kvatinsky, S.: Logic design within memristive memories using memristor-aided loGIC (MAGIC). IEEE Trans. Nanotechnol. **15**(4), 635–650 (2016)
90. Thangkhiew, P.L., Gharpinde, R., Chowdhary, P.V., Datta, K., Sengupta, I.: Area efficient implementation of ripple carry adder using memristor crossbar arrays. In: IEEE International Design and Test Symposium, pp. 142–147. IEEE, Piscataway (2016)
91. Venkataramani, S., Sabne, A., Kozhikkottu, V.J., Roy, K., Raghunathan, A.: SALSA: systematic logic synthesis of approximate circuits. In: Design Automation Conference, pp. 796–801. IEEE, Piscataway (2012)
92. Vongehr, S., Meng, X.: The missing memristor has not been found. Sci. Rep. **5**, 11657 (2015)
93. Wang, J., Dong, X., Xie, Y., Jouppi, N.P.: Endurance-aware cache line management for non-volatile caches. ACM Trans. Archit. Code Optim. **11**(1), 4:1–4:25 (2014)
94. Wong, H.S.P., Lee, H., Yu, S., Chen, Y., Wu, Y., Chen, P., Lee, B., Chen, F.T., Tsai, M.: Metal-oxide RRAM. Proc. IEEE **100**(6), 1951–1970 (2012)
95. Wong, H.S.P., Lee, H.Y., Yu, S., Chen, Y.S., Wu, Y., Chen, P.S., Lee, B., Chen, F.T., Tsai, M.J.: Metal-oxide RRAM. Proc. IEEE **100**(6), 1951–1970 (2012)
96. Xu, C., Niu, D., Muralimanohar, N., Balasubramonian, R., Zhang, T., Yu, S., Xie, Y.: Overcoming the challenges of crossbar resistive memory architectures. In: IEEE International Symposium on High Performance Computer Architecture, pp. 476–488. IEEE, Piscataway (2015)
97. Yang, S.: Logic synthesis and optimization benchmarks user guide: Version 3.0 (1991)

98. Ye, Y., Roy, K.: A graph-based synthesis algorithm for AND/XOR networks. In: Proceedings of the 34th Design Automation Conference, pp. 107–112. IEEE, Piscataway (1997)
99. Zhang, H., Chen, G., Ooi, B.C., Tan, K.L., Zhang, M.: In-memory big data management and processing: a survey. IEEE Trans. Knowl. Data Eng. **27**(7), 1920–1948 (2015)
100. Zhou, P., Zhao, B., Yang, J., Zhang, Y.: A durable and energy efficient main memory using phase change memory technology. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, pp. 14–23. ACM, New York (2009)
101. Zitzler, E., Laumanns, M., Bleuler, S.: A tutorial on evolutionary multiobjective optimization. In: Gandibleux, X., Sevaux, M., Sörensen, K., T'kindt, V. (eds.) Metaheuristics for Multiobjective Optimisation. Springer, Berlin (2004)

Index

© Springer Nature Switzerland AG 2020
S. Shirinzadeh, R. Drechsler, *In-Memory Computing*,
https://doi.org/10.1007/978-3-030-18026-3