Our sponsors

Mission: The International Workshop on Logic and Synthesis is the premier forum for research in synthesis, optimization, and verification of integrated circuits and systems. Research on logic synthesis for emerging technologies and for novel computing platforms, such as nanoscale systems and biological systems, is also strongly encouraged. The workshop encourages early dissemination of ideas and results. The workshop accepts complete papers highlighting important new problems in the early stages of development, without providing complete solutions. The emphasis is on novelty and intellectual rigor.

Topics of interest include, but are not limited to: hardware synthesis and optimization; software synthesis; hardware/software co-synthesis; power and timing analysis; testing, validation and verification; synthesis for reconfigurable architectures; hardware compilation for domain-specific languages; design experiences. Submissions on modeling, analysis and synthesis for emerging technologies and platforms are particularly encouraged.

The workshop format includes paper presentations, posters, invited talks, social lunch and dinner gatherings. Accepted papers are distributed exclusively to IWLS participants.

IWLS 2016

IWLS 2016

25th International Workshop on
Logic and Synthesis

June 10 – June 11, 2016

Thompson Conference Center

Austin, Texas, USA

Foreword

Welcome to the 25th International Workshop on Logic and Synthesis. IWLS provides an informal forum for exchanging ideas in all areas of synthesis, optimization and verification of integrated circuits.

The final workshop program consists of 22 papers, of which 16 have been accepted for oral presentations and 6 have been accepted as posters. This achievement would not have been possible without the work and dedication of the Program Committee Members, who developed thorough reviews for each submission, and contributed to an engaged discussion during the selection process.

We are also excited to have two distinguished keynote speakers in this year's IWLS: Keshav Pingali will discuss the Amorphous Data-parallelism, and Rajit Manohar will deliver a presentation on Design Automation Challenges in Neuromorphic Systems. Also, we have an outstanding special session on emerging technologies, in which Krishnendu Chakrabarty will present Biochemistry Synthesis on Digital Microfluidic Biochips, Priyank Kalla will talk about Photonic Design Automation: Old Wine in New Bottle?, and Rolf Drechsler will discuss Reversible Circuits: Application and Design Challenges. Thanks to our Special Sessions Co-Chairs, Jie-Hong Roland Jiang and Robert Wille, for organizing these sessions.

Lastly, we thank ACM, IEEE, IBM, Synopsys and NanGate for sponsoring this event.

Rolf Drechsler *General Chair*
Andre Reis *Program Chair*

Organizing Committee

General Chair	Rolf Drechsler	University of Bremen/DFKI, Germany
Program Chair	Andre Reis	UFRGS, Brazil
Special Sessions Co-Chair	Jie-Hong Roland Jiang	National Taiwan University, Taiwan
Special Sessions Co-Chair	Robert Wille	Johannes Kepler University Linz, Austria
Publicity Chair	Jody Maick Matos	UFRGS, Brazil
Finance Chair	Dirk Stroobandt	Ghent University, Belgium

Program Committee

L. Amaru, Synopsys, US
V. Bertacco, University of Michigan, US
P. Brisk, University of California Riverside, US
K.-H. Chang, Avery Design Systems, US
R. Drechsler, U. of Bremen/DFKI, Germany
E. Dubrova, KTH, Sweden
S. A. Edwards, Columbia University, US
P. Ienne, EPFL, Switzerland
N. Jayakumar, Juniper Networks, US
H.-R. Jiang, National Chiao Tung University, ROC
J.-H. Jiang, National Taiwan University, ROC
T. Kam, Intel, US
V. Kravets, IBM T. J. Watson, US
S. Krishnaswamy, Columbia University, US
M. Martins, Carnegie Mellon University, US
A. Mishchenko, U. of California Berkeley, US
S. Nowick, Columbia University, US
H. Parandeh Afshar, Qualcomm, US
A. Pellegrini, University of Michigan, US
M. Purnaprajna, Amrita University, India
W. Qian, Shanghai Jiao Tong University, PRC
R. Ribas, UFRGS, Brazil
L. Rosa Jr., UFPel, Brazil
K. Rupnow, NTU and ADSC, Singapore
M. Soeken, EPFL, Switzerland
D. Stroobandt, Ghent University, Belgium
C. Sze, IBM Research, US
T. Villa, Universit di Verona, Italy
I. Wagner, Intel, US
C.-Y. Wang, National Tsing Hua University, ROC
T. Welp, Yale University, US
R. Wille, Johannes Kepler University Linz, Austria

Special Session on Emerging Technologies

Session 1 - SAT-Based Approaches

Session 2 - Word-Level Abstraction and Sequential Optimization

Poster Session

Session 3 - Asynchronous Circuits

Keynote 1

Session 4 - Technology Mapping

Session 5 - Map Reduce / Circuit Testing

Keynote 2

Session 6 - Stochastic and Statistical Methods

Biochemistry Synthesis on Digital Microfluidic Biochips

Krishnendu Chakrabarty[1]

[1]Duke University, USA

Abstract. *Advances in microfluidics have led to the emergence of biochips for automating laboratory procedures in molecular biology. These devices enable the precise control of nanoliter volumes of biochemical samples and reagents. This talk will first introduce electrowetting-based digital microfludic biochips and provide an overview of market drivers such as immunoassays and DNA sequencing. Next, synthesis tools will be described to map bioassay protocols from the lab bench to a droplet-based microfluidic platform. The role of the digital microfluidic biochip as a "programmable and reconfigurable processor" for biochemical applications will be highlighted. Finally, the speaker will describe dynamic adaptation of bioassays through cyberphysical system integration and sensor-driven on-chip error recovery, as well as recent advances in utilizing cyberphysical integration for quantitative gene-expression analysis.*

Krishnendu Chakrabarty received the B. Tech. degree from the Indian Institute of Technology, Kharagpur, in 1990, and the M.S.E. and Ph.D. degrees from the University of Michigan, Ann Arbor, in 1992 and 1995, respectively. He is now the William H. Younger Distinguished Professor of Engineering in the Department of Electrical and Computer Engineering at Duke University. Prof. Chakrabarty is a recipient of the National Science Foundation Early Faculty (CAREER) award, the Office of Naval Research Young Investigator award, the Humboldt Research Award from the Alexander von Humboldt Foundation, Germany, the IEEE Transactions on CAD Donald O. Pederson Best Paper award (2015), 11 best paper awards at major IEEE conferences, and numerous further awards. Prof. Chakrabarty's current research projects include, besides others, digital microfluidics, biochips, and cyberphysical systems. He served for many major journals and conferences such as JETC, ICCAD, DAC, etc.

Photonic Design Automation: Old Wine in New Bottle?

Priyank Kalla[1]

[1]University of Utah, USA

Abstract. *Recent breakthroughs in silicon photonic technology are enabling the integration of optical devices into silicon-based semiconductor processes. This is motivating investigations into applications beyond that of traditional telecom: sensing, filtering, signal processing, quantum technology – and even optical computing. In effect, we are now seeing a convergence of communications and computation, where the traditional roles and boundaries of optics and microelectronics are becoming blurred. Photonic design automation represents an opportunity to take opto-electronic integrated circuit design to a larger scale, facilitating design-space exploration, and laying the foundation for current and future optical applications – thus fully realizing the potential of this technology.*

In this talk, I will describe our work on design automation for integrated optic systems. Using a building-block model for optical devices, we provide an EDA-inspired design flow and synthesis algorithms for optical design automation. I will show how the resulting synthesis problems can be formulated in terms of classical logic and physical synthesis formalisms – e.g., Boolean function decomposition, technology mapping, global and channel routing, thermal-aware (re)synthesis, etc. I will describe our CAD tool development efforts, which actually resulted in the design of an optical logic chip which was fabricated through the OpSIS program.

Priyank Kalla is an Associate Professor in the Electrical & Computer Engineering department at the Univ. of Utah. His areas of interests are in electronic design automation and hardware verification. He received the B.E. degree in Electronics from Sardar Patel University in India (1993) and M.S. and Ph.D. from the Univ. of Massachusetts Amherst in 1998 and 2002, respectively. He has worked with AMD K-7 and the DEC Alpha microprocessor CAD & Test groups. He's a recipient of the US NSF CAREER award and the ACM Trans. on Design Automation best paper award. He was the chair of IEEE technical committee on computer-aided network design and currently also serves as an associate editor for IEEE Trans. on CAD.

Reversible Circuits: Application and Design Challenges

Rolf Drechsler[1]

[1]University of Bremen/DFKI GmbH, Germany

Abstract. *Reversible circuits build the basis for emerging technologies like quantum computation and have promising applications in domains like low power design. Because of that, much progress in the development of design solutions for this kind of circuits has been made in the last decade. This talk provides an overview on reversible circuits and a selection of their most promising applications. Afterwards, it is shown how the design of corresponding reversible circuits differ from conventional EDA problems and recent solutions which address the corresponding challenges are presented.*

Rolf Drechsler received the Diploma and Dr. phil. nat. degrees in computer science from the J. W. Goethe University Frankfurt am Main, Frankfurt am Main, Germany, in 1992 and 1995, respectively. He was with the Institute of Computer Science, Albert-Ludwigs University, Freiburg im Breisgau, Germany and with the Corporate Technology Department, Siemens AG, Munich, Germany. Since October 2001, he has been with the University of Bremen, Bremen, Germany, where he is currently a Full Professor and the Head of the Group for Computer Architecture, Institute of Computer Science. Since 2011, he is also the Director of the Cyber-Physical Systems group at the German Research Center for Artificial Intelligence (DFKI) in Bremen. His research interests include the development and design of data structures and algorithms with a focus on circuit and system design. In these areas, he published more than 250 papers and served in program committees of international conferences such as DAC, DATE, and ICCAD.

Fast Generation of Lexicographic Satisfiable Assignments: Enabling Canonicity in SAT-based Applications

Ana Petkovska[1]
ana.petkovska@epfl.ch

Alan Mishchenko[2]
alanmi@berkeley.edu

Mathias Soeken[1]
mathias.soeken@epfl.ch

Giovanni De Micheli[1]
giovanni.demicheli@epfl.ch

Robert Brayton[2]
brayton@berkeley.edu

Paolo Ienne[1]
paolo.ienne@epfl.ch

[1]Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences, Lausanne, Switzerland
[2]University of California, Berkeley, Department of EECS, Berkeley, USA

ABSTRACT

Lexicographic Boolean satisfiability (LEXSAT) is a variation of the Boolean satisfiability problem (SAT). Given a variable order, LEXSAT finds a satisfying assignment whose integer value under the given variable order is minimum (maximum) among all satisfiable assignments. If the formula has no satisfying assignments, LEXSAT proves it unsatisfiable, as does the traditional SAT. The paper proposes an efficient implementation of the LEXSAT algorithm by combining incremental SAT solving with binary search. Additional methods are proposed that use the lexicographic properties of the assignments to further improve the runtime when generating multiple consecutive satisfying assignments in the lexicographic order. The proposed algorithm outperforms a state-of-the-art LEXSAT algorithm—on average, it is 2.4 times faster when generating a single LEXSAT assignment, and it is 6.3 times faster when generating multiple consecutive assignments.

1. INTRODUCTION

The *lexicographic satisfiability (LEXSAT)* is a decision problem similar to the *satisfiability (SAT) problem*—for a given SAT formula it returns a satisfying assignment, if the problem is *satisfiable (SAT)*, or otherwise it returns *unsatisfiable (UNSAT)*. The only difference is that SAT returns any satisfiable assignment, while LEXSAT returns deterministically the one whose integer value under a given variable order is the minimum (or maximum) among all satisfiable assignments. The assignments with the minimum and maximum integer value are called *lexicographically smallest* and *lexicographically largest* assignment, respectively. For simplicity, we assume that LEXSAT always generates the lexicographically smallest assignment, but the same principles apply for generation of the lexicographically largest assignment.

EXAMPLE 1. *Assuming a 4-input function* $f(x_1, x_2, x_3, x_4)$ *with the SAT assignments for the inputs* $\{0001, 0101, 1010, 1011, 1101\}$*, SAT can return any of the given assignments, while LEXSAT always returns either the lexicographically smallest assignment* 0001 *or the lexicographically largest assignment* 1101*, depending on the user preference.*

Knuth [5] mentions two implementations of an algorithm for generating satisfying assignments in a lexicographic order. The first one [5, Ex. 7.2.2.2-109] calls a SAT solver multiple times. With the first call, it gets a random assignment that it iteratively tries to minimize with the following SAT calls. On the other hand, the second one [5, Ex. 7.2.2.2-275] implements the same concept by modifying the decision heuristics of the SAT solver. The goal is to perform decisions on the input variables in the given order, while for the other variables decisions can be performed in any order. However, Knuth [5] does not provide an evaluation of the performance of these two algorithms.

Independently, Nadel and Ryvchin [7] propose Knuth's LEXSAT algorithm, which they call `OBV-BS`, in the context of *Satisfiability Modulo Theories (SMT)* solving. They also propose another algorithm integrated in a SAT solver. Their results show, first, that the two proposed algorithms are faster than algorithms based on SMT solvers. Second, they show that the `OBV-BS` algorithm, which uses the SAT solver repeatedly, is slower than the one integrated in the SAT solver but it is more robust—it succeeds to find solutions for difficult instances for which the integrated one fails.

With this paper, we propose a scalable and fast LEXSAT algorithm that also uses the SAT solver repeatedly. But, instead of starting from a satisfiable assignment that is iteratively minimized, we start from a potential assignment that is the lexicographically smallest assignment that might be satisfiable. Then, for each variable, we iteratively either confirm that its assignment is identical to the one in the lexicographically smallest SAT assignment, or we increase it, if possible. To achieve a good performance, we also propose a version of the algorithm that is based on the concept of binary search. Moreover, we propose additional methods that use the lexicographic properties of the assignments to further improve the runtime when consecutive SAT assignments are generated in lexicographic order, which is required in such applications as the canonical SAT-based SOP generation [8]. For all algorithms, we propose to use incremental SAT solving to mimic the alternative implementation that modifies the SAT solver, which leads to a good performance while keeping the SAT solver unmodified for general use. With the experimental results, we show that our algorithm is faster than the first algorithm proposed by Knuth [5] when generating single and multiple consecutive LEXSAT assignments.

Following, in this paper, Section 2 motivates using LEXSAT for electronic design automation applications. Section 3 gives the terminology associated with Boolean functions,

and the SAT and LEXSAT problem. Next, in Section 4, we describe two versions of our algorithm and the methods for improving the runtime. We present our experimental setup and discuss the experimental results in Section 5. In Section 6, we argue that our implementation with repetitive SAT calls is expected to be as efficient as an implementation that modifies the SAT solver. Finally, we conclude and present ideas for future work in Section 7.

2. APPLICATIONS OF LEXSAT

Although LEXSAT has emerged only recently, it is potentially very useful for many *Electronic Design Automation (EDA)* applications. For example, Soeken et al. [10] show that LEXSAT enables heuristic NPN classification of large functions with up to 194 variables. In this case, LEXSAT leads to an improved algorithm which was previously limited to functions with up to 16 variables, for which truth tables could be computed [4].

LEXSAT is also proposed and used for fixing cell placement during the physical design stage of an industrial Computer-Aided Design (CAD) flow [7]. By finding the maximal value of a bit-vector, which encodes that a potential violation is solved, a fixer tool generates a placement that has as few violations as possible while giving preference to fixing high-priority violations that are encoded with the most significant bits of the bit-vector.

Another example is a recent work [8] where, since LEXSAT generates assignments in a deterministic lexicographic order, it enables generation of a canonical *Sum Of Products (SOP)* using a SAT solver. For a given function and a variable order, an SOP generated using this method is unique and independent of the input implementation of the function. Moreover, assuming a function $f(x_1, \ldots, x_n)$ if the assignments of the d most left variables x_i, where $1 \leq i < d \leq n$, are fixed to some value, LEXSAT would generate an assignment that is lexicographically closest to the value defined when the d most left variables are assigned to the fixed values and the rest of the variables x_j, where $d+1 \leq j \leq n$ are assigned to 0.

EXAMPLE 2. *For the function $f(x_1, x_2, x_3, x_4)$ from Example 1, if we fix the most left variable x_1 to* 1*, then LEXSAT returns the assignment* 1010 *as lexicographically smallest, because it is the SAT assignment with the smallest integer value after the assignment* 1000*.*

In general, since LEXSAT generates deterministic assignments, it enables canonicity in SAT-based applications with two important consequences: On the one hand, the result of computation depends only on the Boolean function and the user-specified variable order (and is independent on the SAT solver used and on the problem representation, in particular, on the CNF generation algorithm). On the other hand, subproblems encountered during SAT solving can be cached in a way similar to how BDD-based applications cache the results of intermediate computations, resulting in runtime reduction. To this end, BDD-based applications maintain a hash table mapping BDD nodes into the results of computation for these nodes. Similarly, a SAT-based application can use LEXSAT to compute a canonical representation of Boolean functions (such as the canonical SOP mentioned above). This canonical representation can be used as a hash key in the table of computated results, similarly to how BDD nodes are used as hash keys in BDD-based applications.

Applications such as constraint solving [12] and random assignment generation [6] can benefit from LEXSAT because it can be used to derive the closest satisfying assignments for random valuation of inputs. Similarly, LEXSAT enables generating canonical simulation vectors used to generate canonical signatures for Boolean functions using a SAT solver. Additionally, algorithms for approximate computing [9, 11] can use LEXSAT to compute the worst-case error by finding the lexicographically largest solution for the difference between the approximate output and a correct reference version for all possible inputs.

In formal verification, LEXSAT can be used to analyze bugs, which the SAT solver finds when solving verification instances. Suppose, for example, a satisfiable assignment is found that indicates a mismatch between the specification and the implementation of a hardware design. LEXSAT can determine the lexicographically closest correct minterms before and after the buggy minterm. The difference between the two correct minterms outlines the region of the input space where the bug is present. When one bug is characterized in this way, a question can be asked: are there other bugs before and after the given one in the lexicographical order? Repeatedly calling LEXSAT allows us to explore the input space step by step and understand the distribution and the size of buggy regions, which can provide crucial information for debugging.

In summary, an appealing aspect of LEXSAT is that it enables canonicity in SAT-based applications, leading to the same benefits as BDD-based applications reap from the canonicity of BDDs, which are unique for a given function and for a given variable order. Furthermore, there could be practically important applications of LEXSAT in verification, such as "canonical" random simulation based on evenly-distributed input patterns, or bug characterization based on exploration of input space performed by LEXSAT.

3. BACKGROUND INFORMATION

In this section, we give background information related to Boolean functions, as well as to the Boolean SAT and LEXSAT problems.

3.1 Boolean Functions

For a variable v, a *positive literal* represents the variable v, while the *negative literal* represents its negation $\bar{v}$. A *cube*, or product, c, is a Boolean product (AND, $\cdot$) of literals, $c = l_1 \cdot \cdots \cdot l_k$. If a variable is not represented by a negative or a positive literal in a cube, then it is represented by a *don't-care* (-), meaning that it can take both values 0 and 1. A cube with n don't-cares covers 2^n minterms. A *minterm* is the smallest cube in which every variable is represented by either a negative or a positive literal. Let $f(X) : B^n \rightarrow \{0, 1, -\}$, $B \in \{0, 1\}$, be an *incompletely specified Boolean function* of n variables $X = \{x_1, \ldots, x_n\}$. The *support set* of f is the subset of variables that determine the output value of the function f. Any Boolean function can be represented as a two-level *sum of products (SOP)*, which is a Boolean sum (OR, +) of cubes, $S = c_1 + \cdots + c_m$.

A *canonical representation* is a unique representation for a function under certain conditions. For example, given a Boolean function and a fixed input variable order, a canonical SOP is an SOP independent of the original representation of the function.

3.2 Boolean Satisfiability

A disjunction (OR, +) of literals forms a *clause*, $t = l_1 + \cdots + l_k$. A *propositional formula* is a logic expression defined over variables that take values in the set {0, 1}. To solve a SAT problem, a propositional formula is converted into its *Conjunctive Normal Form (CNF)* as a conjunction (AND, ·) of clauses, $F = t_1 \cdot \cdots \cdot t_k$.

A *satisfiability (SAT) problem* is a decision problem that takes a propositional formula in CNF form and returns that the formula is *satisfiable* (*SAT*) if there is an assignment of variables from the formula for which the CNF evaluates to 1. Otherwise, the propositional formula is *unsatisfiable* (*UNSAT*). A program that solves SAT problems is called a *SAT solver*. SAT solvers provide a *satisfying assignment* when the problem is satisfiable.

Modern SAT solvers can receive as input one or more *assumptions*, which are single-literal clauses that hold only for one specific invocation of the SAT solver. The process of determining the satisfiability of a problem under given assumptions is called *incremental SAT solving*. Some SAT solvers support an internal stack of assumptions, which allows for adding and removing assumptions between consecutive SAT call via a *push/pop mechanism*. This enables preserving the state of the SAT solver between the incremental runs, while incremental runs themselves allow for reusing clauses learned from previous calls of the SAT solving procedure. Thus, both incremental SAT solving with assumptions, and incremental adding/removing of assumptions lead to flexibility and efficiency in SAT-based applications.

EXAMPLE 3. *For the function $f(x_1, x_2, x_3, x_4)$ from Example 1, if we give the assumption $x_1 = 1$ as input, then the SAT solver returns one of the assignments* 1010*,* 1011*, or* 1101*, because those assignments are satisfiable considering the given assumption.*

3.3 Lexicographic Boolean Satisfiability

The *lexicographic satisfiability (LEXSAT) problem* is a variation of the SAT problem that takes a propositional formula in CNF form and a given variable order, and returns a satisfying variable assignment whose integer value under the given variable order is minimum (maximum) among all satisfiable assignments. If the formula has no satisfiable assignments, LEXSAT proves it unsatisfiable.

As described in Section 1, Knuth [5] proposes two solutions for generating a LEXSAT assignment. In this paper, we compare to the first implementation that calls the SAT solver multiple times. Assuming a function $f(x_1, \ldots, x_n)$, with the first call, the algorithm generates an initial satisfiable assignment $a_1 \ldots a_n$, or terminates if the problem is UNSAT. Then, if the problem is SAT, it minimizes the assignment iteratively. For this, a pointer d is set to 0 before the first iteration, and later it points the next variable that is assigned to 1 and can be flipped to 0 to lower the assignment. Assignments for the variables x_i for $1 \leq i < d$ are considered to be fixed. Thus, to minimize the assignment, first, d is set to the index of the next variable that is assigned to 1. If $d > n$, then no variable in the assignment can be flipped, and the algorithm returns $a_1 \ldots a_n$. Otherwise, using the assumption mechanism, the SAT solver is called again with the assumptions $x_i = a_i$, for $1 \leq i < d$, and $x_d = 0$. If the problem is SAT, the assignment $a_1 \ldots a_n$ is updated with the newly received assignment; otherwise, the old assignment is kept. Finally, it performs another iteration for minimization to find the next potential 1 to be flipped.

EXAMPLE 4. *For a function $f(x_1, x_2, x_3, x_4, x_5)$, assume that the assignment* 00101 *is received with the first SAT call. Then, in the first iteration for minimization, the pointer d is set to* 3*, since x_3 is the first variable that can be flipped from* 1 *to* 0*. Next, the SAT solver is called with the assumptions $x_1 = 0$, $x_2 = 0$, $x_3 = 0$. If the problem is UNSAT, the value of x_3 remains* 1*, since there is no SAT assignment that satisfies the given assumptions (i.e., that starts with* 000*); thus, the old assignment is kept and in the second iteration for minimization d is set to* 5*. Otherwise, assuming that the SAT solver returns the assignment* 00010*, it is considered as a potential assignment in the second iteration, so $d = 4$.*

4. GENERATING LEXICOGRAPHIC SAT ASSIGNMENTS

In this section, we first describe a simple and a binary search-based version of our algorithm for generation of LEXSAT assignments. Then, we describe several methods that improve their runtime when generating consecutive satisfiable assignments in lexicographic order.

4.1 Simple Version

Instead of using a SAT solver to find the initial assignment, our algorithm receives as input a potential assignment $a_1 \ldots a_n$ that is smaller or equal to the next LEXSAT assignment. When generating consecutive satisfiable assignments in a lexicographic order, this enables the search to start from the last generated LEXSAT assignment. While, for the first LEXSAT assignment or when generating non-consecutive assignments, for a function $f(x_1, \ldots, x_n)$, we assume the smallest possible assignment when $a_i = 0$ for $1 \leq i \leq n$, which means that all variables are assigned 0. Having this initial potential assignment, our algorithm iteratively verifies if the assignment of each variable can be fixed or should be flipped, and converts the potential assignment into the LEXSAT assignment that is returned as output.

Basic idea. A simple version of our algorithm fixes the assignments of the variables one by one. A pointer d, which is initially set to 1, gives the index of the next variable for which the assignment should be fixed, while for the previous variables the assignments $x_i = a_i$, for $1 \leq i < d$, are already fixed. To fix the assignments, a SAT solver is called iteratively with the assumptions $x_i = a_i$, for $1 \leq i \leq d$. If the problem is SAT, then there is a satisfiable assignment which starts with $a_1 \ldots a_d$, so d is incremented. Otherwise, if there is no SAT assignment which starts with $a_1 \ldots a_d$, the problem is UNSAT. In this case, if $a_d = 0$, then we set $a_d = 1$, set $a_i = 0$ for $d < i \leq n$ to keep the assignment the smallest possible for the future iterations, and do another iteration. But, if the problem is UNSAT when $a_d = 1$, then there is no satisfiable assignment both when $a_d = 0$ and $a_d = 1$, and thus the algorithm returns UNSAT. Once $d > n$, the assignments for all variables are fixed and $a_1 \ldots a_n$ is returned as a LEXSAT assignment.

EXAMPLE 5. *To generate the first LEXSAT assignment for a function $f(x_1, x_2, x_3, x_4, x_5)$, the received potential assignment is 00000. Initially, $d = 1$ and the first SAT call assumes $x_1 = 0$. If the problem is SAT, then d is incremented to $d = 2$, and in the next iteration the SAT call assumes*

$x_1 = 0$ *and* $x_2 = 0$. *Otherwise, if the problem is UNSAT, we flip* $a_1 = 1$, *and iterate with the assumption* $x_1 = 1$. *This time, if we receive SAT, we increment* d, *and in the next iteration the SAT call assumes* $x_1 = 1$ *and* $x_2 = 0$. *But, if we receive UNSAT again, it means that there is no assignment both with* $x_1 = 0$ *and* $x_1 = 1$, *and thus we return UNSAT.*

Improving performance by learning from satisfiable assignments. Similarly to the algorithm by Knuth [5] described in Section 3.3, when the SAT solver returns a SAT assignment, we can learn some satisfiable assignments from it. Thus, we always save the last satisfiable assignment, and use it as following. First, same as before, if the first variable assigned to 1 after d is on position $d+t$, where $1 \leq t \leq n-d$, then we can learn and fix to 0 the $t-1$ variables between d and $d+t$. Moreover, in our case, the potential assignment $a_1 \dots a_n$ is the lexicographically smallest assignment that might be satisfiable. Thus, if the potential assignment for a variable x_i is $a_i = 1$, then we cannot flip it to 0 to minimize the assignment as in the algorithm by Knuth. This allows us to learn from the SAT solver all assignments until the first variable for which the potential assignment and the assignment returned by the SAT solver differ. Assume that the last satisfiable assignment returned by the solver is $v_1 \dots v_n$. Instead of incrementing d for one, we can set it to the index i, such that $a_j = v_j$ for $1 \leq j < i$ and $a_i \neq v_i$. Finally, same as the algorithm by Knuth, for a given literal x_d, where $1 < d \leq n$, with $v_d = 1$, if we get UNSAT when flipping its assignment from 1 to 0, we can immediately fix it to 1, as this value is confirmed by the last satisfiable assignment.

Example 6. *For a function* $f(x_1, \dots, x_6)$, *assume that* 101000 *is received as a potential assignment. When the SAT solver is called with the assumption* $x_1 = 1$, *it returns a satisfiable assignment* 101101, *which is saved as a last satisfiable assignment. Besides fixing* $x_1 = 1$, *from this assignment, we can learn and fix* $x_2 = 0$ *and* $x_3 = 1$, *because their potential assignments are confirmed by the last satisfiable assignment. The variable* x_4 *is the most left variable for which the assignments differ and might be flipped to* 0, *so for the next iteration we set* $d = 4$ *and call the SAT solver with the assumptions* $x_1 = 1$, $x_2 = 0$, $x_3 = 1$ *and* $x_4 = 0$. *If the problem is SAT, we fix* x_4 *to* 0 *and update the last satisfiable assignment. But, if the problem is UNSAT, from the last satisfiable assignment* 101101, *we already know that the problem is satisfiable when* $x_4 = 1$, *we can additionally fix* $x_5 = 0$, *and set* $d = 6$ *for the next iteration.*

4.2 Binary Search-Based Version

To further enhance the simple version of our algorithm, instead of fixing the assignments of variables one by one, we propose to set the pointer d using binary search. Two additional pointers l and r show the first and last variable with non-fixed assignment, respectively, which initially are set to $l = 1$ and $r = n$. Then, d is set to the middle variable of the array of variables bounded by x_l and x_r. This assumes the assignments of the left half of the variables x_i, where $1 \leq i \leq d$, in the first iteration. Later, whenever the SAT solver returns SAT, it confirms that a satisfiable assignment that starts with $a_1 \dots a_d$ exist. As shown in Section 4.1, from the returned satisfiable assignment we can confirm and fix t additional assignments from the potential assignment, where $0 < t < n - d$. After this step, the assignments for the variables x_i, where $1 \leq i \leq d + t$ are fixed. For the next iteration, we set $l = d + t + 1$ and $r = n$ to assume the assignments for the non-fixed variables in the right half. Otherwise, if the problem is UNSAT, if $a_d = 0$, then we proceed as in the simple version of the algorithm: we set $a_d = 1$, set $a_i = 0$ for $d < i \leq n$ for the future iterations, and do another iteration; while, if $a_d = 1$, for the next iteration $r = d - 1$ to assume less non-fixed variables.

Example 7. *To generate the first LEXSAT assignment for a function* $f(x_1, x_2, x_3, x_4, x_5, x_6)$, *the initial assignment* 000000 *is received as input. Initially,* $l = 1$, $r = 6$ *and* $d = 3$. *Thus, the first SAT call would assume* $x_1 = 0$, $x_2 = 0$, *and* $x_3 = 0$. *If the problem is SAT and the SAT assignment 000010 is returned, then the assignment* $x_4 = 0$ *is learned since it is the same in the potential assignment, and the values of the pointers are updated to* $l = 5$, $r = 6$ *and* $d = 5$ *for the next iteration. Otherwise, if it is UNSAT, we would first try the assumptions* $x_1 = 0$, $x_2 = 0$, *and* $x_3 = 1$. *This time, if we receive SAT we would proceed same as before; while, if we receive UNSAT again, for the next iteration, we would update the values of the pointers to* $l = 1$, $r = 2$ *and* $d = 1$ *to assume less variables.*

4.3 Runtime Improvement when Generating Consecutive LEXSAT Assignments

Applications such as the SAT-based generation of canonical SOPs [8] generate consecutive satisfiable assignments in lexicographic order. To allow generation of new assignments, each generated assignment is added to the SAT solver as a *blocking clause*, which is an additional clause that blocks known solutions of the SAT problem.

Example 8. *For the function* $f(x_1, x_2, x_3, x_4)$ *from Example 1, the first LEXSAT call returns the assignment* 0001. *If we add this assignment as a blocking clause to the SAT solver, with the next LEXSAT call the assignment* 0101 *is generated because it is the lexicographically smallest satisfiable assignment that is not blocked.*

For these type of algorithms, we present three methods that improve the runtime of the newly proposed algorithm by using the lexicographic properties of the assignments and the fact that the received potential assignment is the last generated LEXSAT assignment.

Fixing leading 1s. When generating consecutive satisfiable assignments in lexicographic order, after some time, assignments that start with one or more consecutive 1s are generated. Generating a lexicographically smallest SAT assignment that starts with one or more consecutive 1s implies that all unblocked satisfiable assignments are greater than the generated, and therefore also start with the same or more consecutive 1s. When generating a LEXSAT assignment, assume that $a_i = 1$, where $1 \leq t$ and $t \leq n$ (i.e., the received potential assignment starts with t consecutive 1s). Then, we can fix these t assignments for the corresponding variables x_i, where $1 \leq t$, and the initial value of l (or of d in the simple version) is set to $t + 1$ to point the first variable that is assigned 0.

Example 9. *For a function* $f(x_1, x_2, x_3, x_4, x_5)$, *assume that the assignment* 11010 *is generated with the previous LEXSAT call and is received as potential assignment. Since the next lexicographical assignment has to be greater than the last generated assignment, we know that it also starts with*

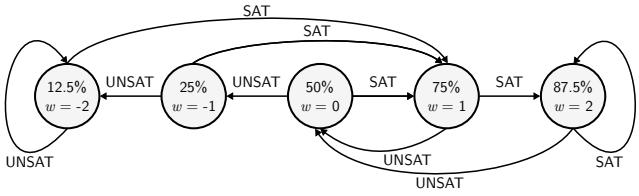


Figure 1: Changing the percentage of assumed variables for the first SAT call of LEXSAT depending on the success of the previous first SAT calls. In this case, at most three iterations of binary search are done at once.

11. *Thus, we can skip assuming assignments for* x_1 *and* x_2*, and directly fix them to* 1*. Initially,* l *is set to* 3*,* r *to* 5*,* d *is computed to be* 4*, and thus, the first SAT call would be with the assumptions* $x_1 = 1$*,* $x_2 = 1$*,* $x_3 = 0$*, and* $x_4 = 1$*.*

Correcting the initial potential assignment. When generating consecutive SAT assignments lexicographically, after generating the first LEXSAT assignment, the initial assignment received as input is the last generated LEXSAT assignment. But, the first possible SAT assignment is actually the assignment whose integer value is one unit greater than the last LEXSAT assignment. Thus, assuming that the last LEXSAT assignment ends with t 1s, i.e., $a_{n-t+1} = 1$, where $0 \leq t \leq n$, we flip the most right 1s by setting $a_{n-t+1} = 0$ and the first 0 starting from the right by setting $a_{n-t} = 1$.

EXAMPLE 10. *For a function* $f(x_1, x_2, x_3, x_4, x_5)$*, assume that the assignment* 11011 *is generated with the previous LEXSAT call and received as a potential assignment. Since the next lexicographical assignment has to be greater than the last generated, the first possible satisfiable assignment is* 11100*. Thus, we flip the* 1*s and the first* 0 *starting from the right to get the potential assignment* 11100*.*

Profiling the success of the first SAT call. For the LEXSAT algorithm, we consider satisfiable SAT calls as successful because they confirm the assumed assignments, while unsatisfiable SAT calls are considered unsuccessful. Further, we propose to profile the success of the first SAT call from the LEXSAT algorithm and use this profile to alter the percentage of assumed assignments in the first SAT calls in the subsequent invocation of the LEXSAT algorithm based on binary search. This method does not apply to the simple version of the algorithm.

The binary search-based LEXSAT algorithm always sets the pointer d to point the middle variable of the array of variables bounded by x_l and x_r, meaning that with the first SAT call we always assume the assignments for the first 50% of the variables between x_l and x_r. In the next iterations, with every satisfiable SAT call, we increase the number of assumed assignments and add 50% more of the right subarray. With every unsatisfiable SAT call, we decrease the number of assumed non-fixed assignments and the next time we use only 50% of the assignments of the left subarray. Thus, for example, assuming 75% of the assignments in the first SAT call is equivalent to having two consecutive iterations with successful SAT calls.

To profile and alter the percentage of assumed assignments in the first SAT call, we keep a variable w which tells us how many iterations to perform at once and in which direction we should perform them. We iterate $|w|$ times to decrease or increase the percentage when $w < 0$ or $w > 0$, respectively. Initially, $w = 0$, which means that we should assume 50% of the assignments. If the first SAT call is satisfiable, we increase w for 1 when $w \geq 0$ or we set $w = 1$ when $w < 0$. If the first SAT call is unsatisfiable, we decrease w for 1 when $w \leq 0$ or we set $w = 0$ when $w > 0$. Figure 1 shows how the percentage of assumed variables for the first SAT call and the value of w changes with the success of the first SAT calls. In this example, maximum three iterations of binary search are performed at once.

EXAMPLE 11. *For a function* $f(x_1, \ldots, x_{10})$*, assume that the assignment* 0000110000 *is received as a potential assignment and* $w = 0$*. Since,* $l = 1$*,* $r = 10$ *and* $w = 0$*, for the first SAT call* d *is computed as* $d = \lfloor (1+10) \cdot 0.5 \rfloor = 5$*. Thus, we assume* $x_1 = 0$*,* $x_2 = 0$*,* $x_3 = 0$*,* $x_4 = 0$ *and* $x_5 = 1$*. If this call is satisfiable, then* w *is set to* 1 *for the next LEXSAT assignment. Assume that with the following SAT calls the LEXSAT assignment* 0000110001 *is generated. Then, when generating the next LEXSAT assignment, for the first SAT call,* $d = \lfloor (1 + 10) \cdot 0.75 \rfloor = 8$ *because* $w = 1$*, so instead of assuming the potential assignments only for the first five inputs as before, we assume the assignments for the first eight inputs. For the remaining SAT calls of the current LEXSAT assignment, we always use the regular binary search algorithm, which always assumes 50% of the assignments.*

5. EXPERIMENTAL RESULTS

In this section, for convenience we call the algorithm from Knuth [5] `KLEX` (Section 3.3), and the simple and binary search-based versions of our algorithm `SIMPLE` and `BINARY`, respectively (Section 4).

We implemented in ABC [2] the three algorithms `KLEX`, `SIMPLE`, and `BINARY`, as well as the methods for improving the runtime from Section 4.3. ABC features an integrated incremental SAT solver derived from an early version of MiniSAT [3]. Also, this SAT solver supports pushing and popping of assumptions.

To evaluate the runtime of the algorithms and the speedup achieved from the additional methods, we use the set of large MCNC benchmarks, as well as a set of logic tables from the instruction decoder unit [1], which we denote with LT-DEC. The names of the LT-DEC benchmarks are given in the form "[N_{PI}].[N_{PO}]", where N_{PI} is the number of primary inputs and N_{PO} is the number of primary outputs. For a given benchmark, each algorithm generates the user specified number of consecutive LEXSAT assignments for each *combinatorial output*, that is each primary output and each latch input. However, to avoid repeatedly calling the procedure for output functions with isomorphic circuit structure, we divide the outputs into equivalence classes. An *equivalence class* contains outputs that implement an identical function expressed over different inputs. Thus, for each benchmark, we actually generate LEXSAT assignments only for the representative of each class.

For a given function and a variable order, the LEXSAT assignments are deterministic and must be generated in the same order when generating consecutive LEXSAT assignments. The correctness of our algorithms is evaluated by generating assignments one by one with each algorithm, and comparing them to ensure that all algorithms generate the same assignments in the same order. For generating a given

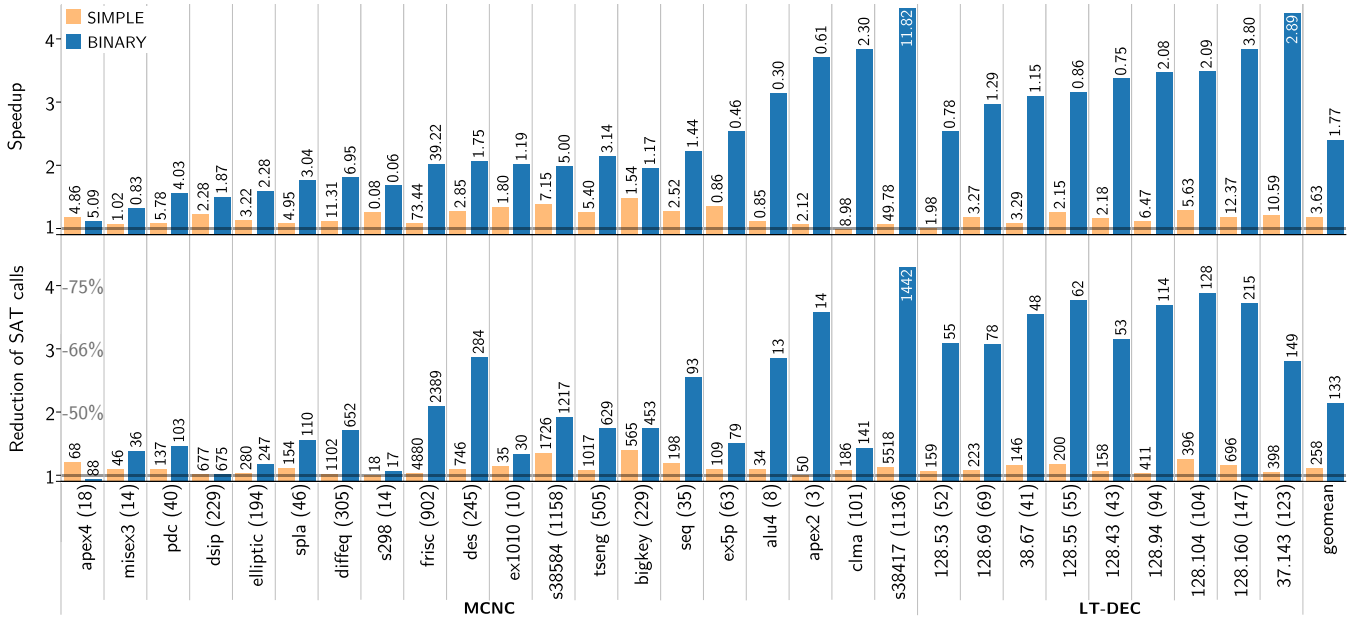


Figure 2: Speedup and reduction of the number of SAT calls achieved by our algorithms `SIMPLE` and `BINARY` compared to the `KLEX` algorithm when generating a single LEXSAT assignment per combinatorial output. Next to each bar is the actual runtime (in milliseconds) and the number of SAT calls, respectively. Next to the name of the benchmark, we give the number of LEXSAT calls in brackets.

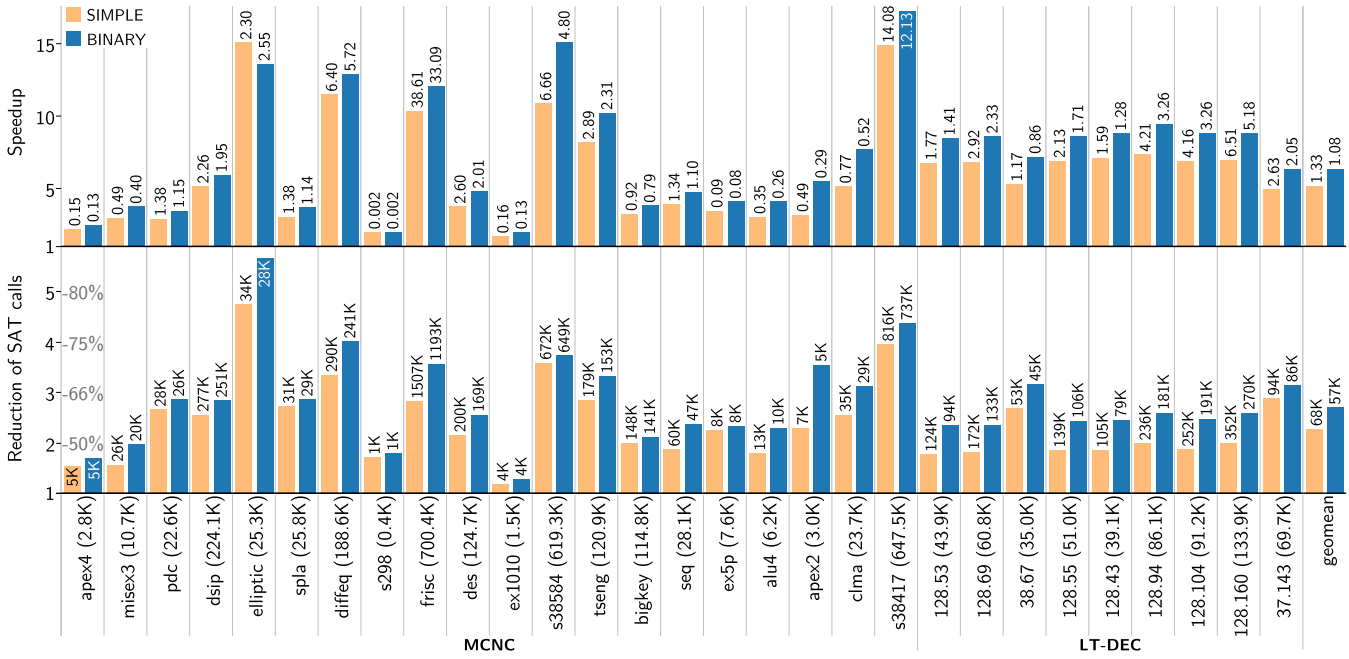


Figure 3: Speedup and reduction of the number of SAT calls achieved by our algorithms `SIMPLE` and `BINARY` compared to the `KLEX` algorithm when generating 1000 consecutive LEXSAT assignments per combinatorial output. Next to each bar is the actual runtime (in seconds) and the number of SAT calls (in thousands), respectively. Next to the name of the benchmark, in brackets, is the number of LEXSAT calls (in thousands).

number of LEXSAT assignments, the number of SAT calls depends on how often the LEXSAT algorithm calls the procedure for SAT solving.

Below we compare the runtime of the algorithm `KLEX`, and the two versions of our algorithm `SIMPLE` and `BINARY` enhanced with the additional methods described in Section 4.3. We evaluate the three algorithms for both generation of a single and multiple consecutive LEXSAT assignments. Afterwards, we evaluate the speedup achieved by each of the additional methods.

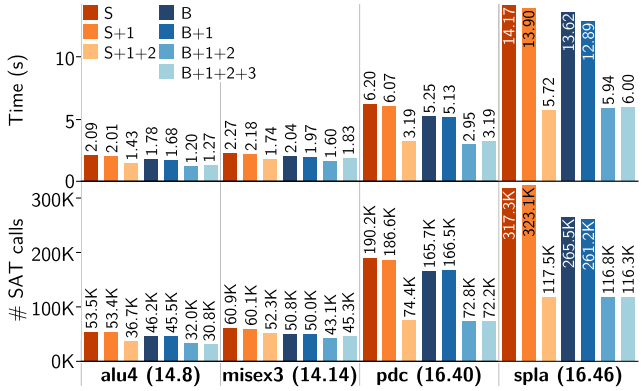


Figure 4: Runtime and number of SAT calls of the SIMPLE (S) and BINARY (B) when different methods for improving the runtime are used for 4 benchmarks from the MCNC set. Next to the name of the benchmark, we give the number of combinatorial inputs and outputs, respectively.

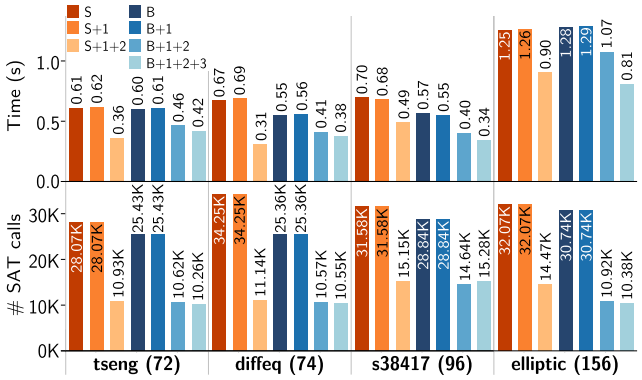


Figure 5: Runtime and number of SAT calls of the SIMPLE (S) and BINARY (B) when different methods for runtime improvement are used for one of the outputs from 4 benchmarks from the MCNC set. Next to the name of the benchmark, we give its number of combinatorial inputs.

5.1 Runtime Comparison

Generation of a single LEXSAT assignment. Some LEXSAT-based application as the NPN classification [10] require multiple LEXSAT assignments, but they are not in a consecutive order or they are for different functions. Thus, first, we evaluate the runtime and number of SAT calls required by each algorithm for generating a single LEXSAT assignment. For each benchmark, a single LEXSAT assignment is generated per combinatorial output. Since the algorithms generate these assignments in few milliseconds, to get a precise comparison, we generate each LEXSAT assignment 1000 times, and then divide the total runtime by 1000. As Figure 2 shows both SIMPLE and BINARY perform better than KLEX for almost all benchmarks. Since the algorithmic steps of SIMPLE are very similar to those of KLEX when generating a single assignment, SIMPLE makes only 9.7% less calls to the SAT solver, and thus is only 14.7% faster than KLEX. On the other hand, assuming more assignments at once with BINARY leads to about 2x less satisfiable assignments and 2x faster runtime than SIMPLE. Finally, BINARY is 2.4x faster than KLEX and makes 2.1x less SAT calls.

Generation of multiple consecutive LEXSAT assignments. On the other hand, applications as the LEXSAT-based generation of canonical SOPs [8] require generation of multiple consecutive LEXSAT assignments. In this case, the methods described in Section 4.3 also contribute to reducing runtime of SIMPLE and BINARY. For this experiment, we generate at most 1000 consecutive LEXSAT assignments for each combinatorial output. For each output we perform the experiment 5 times, and thus the presented results represent the average over 5 runs. As Figure 3 shows, both SIMPLE and BINARY outperform KLEX—for SIMPLE, we have 2.3x less SAT calls on average, which reduces runtime 5.1x, while for BINARY we have 2.7x less SAT calls on average, which reduces runtime 6.3x. Regarding the two proposed versions of our algorithm, on average, BINARY has 16.1% less SAT calls that contribute to 18.9% better runtime than SIMPLE.

Thus, BINARY has the best performance both when generating a single assignment and when generating multiple consecutive LEXSAT assignments.

5.2 Evaluation of the Methods for Runtime Improvement

In Section 4.3, we presented the following three methods for runtime improvement when generating consecutive assignments.

1. Fixing leading 1s.
2. Correcting the initial potential assignment.
3. Profiling the success of the first SAT call.

Since the method for fixing the leading 1s affects the runtime only when generating assignments in which the most significant bits are assigned to 1, we evaluate the methods by generating the complete truth table (i.e., generating all assignments for which the function evaluates to 1) for a subset of the MCNC benchmarks. The selected benchmarks have at most 16 combinatorial inputs, which means that, for each combinatorial output, we can have at most 65536 minterms when the function is 1. Similarly to before, the presented results represent the average over 5 runs. Figure 4 shows the runtime and number of SAT calls for four of the selected benchmarks. First, it shows the results when the algorithms SIMPLE (S) and BINARY (B) are used without the additional methods. We can see that fixing the leading 1s (S+1, B+1) decreases the runtime moderately. Contrarily, if we additionally correct the initial potential assignment (S+1+2, B+1+2) then the runtime decreases for 32%, on average. Finally, for BINARY, although the method for profiling the success of the first SAT call in general decreases the number of SAT calls, for functions with small number of inputs it slightly increases the runtime. However, we have observed reduction of runtime for benchmarks with a large number of combinatorial inputs. Figure 5 shows the runtime and number of SAT calls required to generate 1000 minterms for a single output of 4 large MCNC benchmarks. The considered outputs have more than 70 combinatorial inputs. In this case, the method for fixing leading 1s does not have effect on the number of SAT calls because the most significant bits of all generated assignments are 0s.

Note that in Section 5.1 the results for SIMPLE and BINARY are obtained when all methods are used (i.e., with S+1+2 and B+1+2+3, respectively).

6. ON INTEGRATING THE LEXSAT ALGORITHMS IN A SAT SOLVER

The algorithms presented and evaluated in this paper use repeatedly the SAT solver. Another option is to modify the SAT solver to generate LEXSAT assignments. For convenience, we refer to them with `OUTSAT` and `INSAT`, respectively. Knuth [5, Ex. 7.2.2.2-275] suggests an `INSAT` implementation of `KLEX`. Nadel and Ryvchin [7] show that an `INSAT` algorithm is faster than an `OUTSAT` implementation of the `KLEX` algorithm, but unlike the `OUTSAT` implementation, it is not scalable for difficult instances. In this section, we discuss the difference in these two implementation options.

Generally, in an `INSAT` implementation, the decisions on the input variables are performed in the order and with the values given by the LEXSAT algorithm, while for the other variables decisions can be performed in any order. With this solution, to generate LEXSAT assignments for a function, a single SAT solver instance is created and, for each LEXSAT assignment, the procedure for SAT solving is called only once, with a given order for the input variables. Note that the concepts of the algorithms `SIMPLE` and `BINARY` can also be used to determine the order of issuing decisions and the values for the input variables.

On the other hand, in our `OUTSAT` implementations, for a given function, incremental SAT solving allows generating multiple LEXSAT assignment also by using only a single SAT solver instance. Moreover, for each LEXSAT assignment, the interfaces for pushing and popping assumptions, which we suggest to use, allow to preserve the internal state of the solver between consecutive invocations of the SAT solving procedure. With this, on a higher level, we mimic the solution based on modifying the SAT solver. With such implementation, and by using the algorithm `BINARY`, we expect our `OUTSAT` implementation to be as fast as an `INSAT` implementation, but confirming this experimentally is left for future work.

Moreover, assume a function with n inputs for which the assignments of the first d inputs are already fixed, where $1 \leq d \leq n$. In the `OUTSAT` implementation, the SAT solving procedure can and do change the order of decisions for the least significant $n - d - 1$ variables whose value is not yet fixed, when running the query to fix the value of the variable $d+1$. However, the `INSAT` implementation always makes the same decisions in the same order, and can not change the order even if that would lead to faster UNSAT calls during LEXSAT solving. Thus, for difficult instances, such as functions with large number of variables when different variable orders affect the efficiency of the SAT solving procedure, as well as when all calls are not satisfiable, an `OUTSAT` implementation is more scalable than an `INSAT` implementation.

7. CONCLUSION

This paper presents a novel variation of the Boolean satisfiability problem, called LEXSAT, which in addition to determining the status of a problem (satisfiable or unsatisfiable), also returns satisfying assignments that are minimum (maximum) considering a given variable order. We demonstrate that LEXSAT allows for the development of SAT-based algorithms, which share a number of desirable properties with BDDs but are less likely to suffer from scalability problems besetting BDD-based computations in many EDA applications. In particular, LEXSAT can be used to achieve *canonicity* of the computed results: when for the given Boolean function under the given variable order, the result is deterministic and independent from the SAT solver and the CNF generation algorithm.

The paper also proposes a fast binary search-based algorithm for generation of a single LEXSAT assignment, which is 2.4 times faster than a state-of-the-art LEXSAT algorithm. Furthermore, it proposes several improvements to the LEXSAT algorithms for the typical use-model when it is applied iteratively and the resulting satisfying assignments are monotonically increasing. For such use, the proposed algorithm enhanced with the new features is 6.3 times faster than an existing LEXSAT algorithm. Finally, we propose a way of using incremental SAT solving to improve performance without modifying the SAT solver.

We expect that LEXSAT has many potential uses in EDA. Future work on LEXSAT will focus on exploring several promising applications: SAT-based constraint simulation, SAT-based factoring, SAT-based exclusive sum-of-product minimization, etc.

8. REFERENCES

[1] The EPFL Combinational Benchmark Suite, "Multi-output PLA benchmarks". `http://lsi.epfl.ch/benchmarks`.

[2] Berkeley Logic Synthesis and Verification Group, Berkeley, Calif. *ABC: A System for Sequential Synthesis and Verification*. `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[3] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, volume 2919, pages 502–18. Springer, May 2003.

[4] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko. Fast Boolean matching based on NPN classification. In *Proceedings of the 2013 International Conference on Field Programmable Technology*, pages 310–13, Kyoto, Dec. 2013.

[5] D. E. Knuth. *Fascicle 6: Satisfiability*, volume 19 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., Dec. 2015.

[6] A. Nadel. Generating diverse solutions in SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 287–301, Ann Arbor, Mich., June 2011.

[7] A. Nadel and V. Ryvchin. Bit-vector optimization. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 851–67, Eindhoven, The Netherlands, Apr. 2016.

[8] A. Petkovska, A. Mishchenko, D. Novo, M. Owaida, and P. Ienne. Progressive generation of canonical sum of products using a SAT solver. In *Proceedings of the 25th International Workshop on Logic and Synthesis*, Austin, Tex., June 2016.

[9] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler. BDD minimization for approximate computing. In *Proceedings of the 21st Asia and South Pacific Design Automation Conference*, pages 474–79, Macao, Jan. 2016.

[10] M. Soeken, A. Mishchenko, A. Petkovska, B. Sterin, P. Ienne, R. Brayton, and G. De Micheli. Heuristic NPN classification for large functions using AIGs and LEXSAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, Bordeaux, France, July 2016.

[11] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan. MACACO: Modeling and analysis of circuits for approximate computing. In *Proceedings of the International Conference on Computer Aided Design*, pages 667–73, San Jose, Calif., Nov. 2011.

[12] J. Yuan, A. Aziz, C. Pixley, and K. Albin. Simplifying Boolean constraint solving for random simulation-vector generation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(3):412–20, Mar. 2004.

Progressive Generation of Canonical Sums of Products Using a SAT Solver

Ana Petkovska[1] ana.petkovska@epfl.ch
Alan Mishchenko[2] alanmi@berkeley.edu
David Novo[3] david.novo@lirmm.fr
Muhsen Owaida[4] mohsen.ewaida@inf.ethz.ch
Paolo Ienne[1] paolo.ienne@epfl.ch

[1]Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences, Lausanne, Switzerland
[2]University of California, Berkeley, Department of EECS, Berkeley, USA
[3]Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier, Montpellier, France
[4]Eidgenössische Technische Hochschule Zürich (ETHZ), Zürich, Switzerland

ABSTRACT

We present an algorithm that progressively generates canonical *Sums Of Products (SOPs)* for completely- and incompletely-specified Boolean functions using a satisfiability solver. The progressive generation allows for real time monitoring and early termination, as well as for generation of partial SOPs for specific incremental applications. On the other hand, canonicity brings independence of the original representation and typically yields more regular SOPs that factor better. Also, canonicity is key in applications as constraint solving and random assignment generation, which traditionally rely on BDD-based methods. However, in contrast with BDDs, our algorithm can relax canonicity to improve speed and scalability. On average, our method improves the quality of results up to 10%, in terms of SOP size, compared to a state-of-the-art BDD-based method. Experiments with global circuit restructuring based on SAT-based SOPs show that area-delay product can be improved up to 36%, compared to using BDD-based SOPs.

1. INTRODUCTION

Minimization of the two-level *Sum Of Products (SOP)* representation is well studied due to the wide use of SOPs. In the past, research in SOPs was motivated by mapping into *Programmable Logic Arrays (PLAs)*; now SOPs are supported in many tools for logic optimization and are used for multi-level logic synthesis [3,21], delay optimization [16], test generation [7] and in other areas.

Contrary to the popular belief, research in SOP minimization and its applications is not outdated. As an example, a recent work uses SOPs for delay optimization in technology independent synthesis and technology mapping [16]. In this work, improved quality is achieved by enumerating different SOPs of the local functions of the nodes, factoring them, and finding circuit structures balanced for delay.

Another important application of SOP minimization, targeted in this paper, is *global circuit restructuring*. If a multi-level circuit structure for a Boolean function is not available, or if the circuit structure is redundant, a new circuit structure with desirable properties, such as low area, short delay, good testability or improved implicativity (if the circuit represents constraints in a SAT solver) should be derived.

The best known (and widely used) method for global circuit restructuring is computing SOPs of the output functions in terms of inputs, factoring the multi-output SOPs and deriving a new circuit structure from the shared factored form. The main drawback of this method is the lack of scalability of the *Binary Decision Diagram (BDD)* construction, which is required to compute the SOPs of the output functions. To our knowledge, this method is used in most of the industrial tools, and therefore scalability improvements of this method are highly desirable.

Additional drawback is that BDDs are incompatible with incremental applications since they require building a BDD for the complete circuit before converting it to SOP. Furthermore, for some circuits, the BDD construction suffers from the BDD memory explosion problem—the BDD size is exponential in the number of input variables—and thus, using BDDs is often impractical.

Alternatively, recent progress in the performance of *Boolean satisfiability (SAT) solvers* enabled using SAT in various domains of logic synthesis and verification despite their worst-case exponential runtime. Thus, it has become a trend to replace BDD-based methods with SAT-based ones. For example, this was done for model checking [14], functional dependency [8], functional decomposition [11, 12] and logic don't-care-based optimization [17]. Existing methods for SOP generation using SAT solvers are based on enumeration of satisfying assignments [18]. On the other hand, Sapra et al. [23] proposed SAT-based algorithm for part of ESPRESSO's procedures for SOP minimization. But, since they largely follow the traditional ESPRESSO style of SOP minimization, they operate on existing SOPs and do not consider generating a new SOP from a multi-level representation of the Boolean function. To the best of our knowledge, there is still no complete SAT-based method for SOP generation similar to the *Irredundant Sum-of-Product (ISOP)* algorithm for incompletely specified functions using BDDs [15].

Accordingly, the main contribution of this paper is to propose a new engine for SOP generation and minimization that is completely based on SAT solvers. Our method generates the SOP progressively, building it cube by cube. We guarantee that the generated SOPs are *irredundant*, meaning that no literal and no cube can be deleted without changing the function. As we show in the result section, our algorithm

generates SOPs with the size similar to that of the BDD-based method [15]. Interestingly, for some circuits, we generate smaller SOPs (up to 10%), which is useful in practical applications. For example, when a multi-level description of the circuit is built using an SOP produced by the proposed SAT-based method, the area-delay product of the resulting circuit, assuming unit-area and unit-delay model, decreases up to 36%, compared to using BDDs.

Two main features characterize our SAT-based SOP generation and make it desirable in various domains.

First, we generate an SOP *progressively*, unlike BDD-based methods that attempt to construct a complete SOP at once. The progressive computation allows generation of *a partial SOP* for circuits whose complete SOP cannot be computed given the resource limits. The partial SOPs can be exploited by other applications that do not require the complete circuit functionality, but work with partially defined functions [4,25]. Moreover, for circuits with large SOPs, the progressive generation allows us to predict whether it is feasible to build an SOP for a circuit, and to check if the SOP size is within the limits of the methods that are going to use it. For this, at any moment, we can retrieve the number of outputs for which the SOP is already computed, as well as the finished SOP portion of the currently processed output. We can also easily compute an estimate or a lower-bound of the percentage of covered minterms, considering uniform distribution of minterms in the space or considering the size of the truth table, respectively. In contrast, the termination time and the quality of results of the BDD-based methods are unpredictable since the complete BDD has to be built before converting it to SOP.

Second, counter-intuitive as it may sound, we show that the SAT-based computation can generate *canonical* SOPs. To this end, we combine (1) an algorithm that, under a given variable order, generates consecutive SAT assignments in lexicographic order [20], considering each assignment as integer value, and (2) a deterministic algorithm that expands the received assignments into cubes. For a given function and a variable order, the assignments (i.e., the minterms) are always generated in the same order, and each assignment always results in the same cube. Thus, the resulting SOP is canonical—it is unique and independent of the input implementation of the function. The canonical nature of the resulting SOPs can be useful in those domains where previously only BDDs could be used. For example, constraint solving [26] and random assignment generation [19] can benefit from the canonicity if we iterate repeated generation of random valuation of inputs and get the closest SAT assignment, as it is done in the proposed canonical SOP generation method. Also, the canonicity brings regularity in the SOPs, and thus the results after using algorithms for factoring [21] are expected to better.

In the rest of the paper, we focus on completely-specified functions, but it should be noted that the given SAT-based formulation works for incompletely-specified functions without any changes. Indeed, after extracting the first cube and blocking it in the on-set of the function, the rest of the computation is performed for the incompletely-specified functions, even if the initial function was completely specified. A more interesting extension is to enumerate cubes that are shared across multiple outputs. Implementing this extension and exploring the resulting improvements in quality is left for future work.

The rest of the paper is organized as follows. Section 2 gives background on Boolean functions, the SOP representation and the satisfiability problem. Next, we describe our algorithm for SAT-based progressive generation of irredundant SOPs in Section 3. Section 4 gives our experimental setup and discusses the experimental results. Section 5 gives additional information on the related work. Finally, we conclude and present ideas for future work in Section 6.

2. BACKGROUND INFORMATION

In this section, we define the terminology associated with Boolean functions and the SOP representation, as well as the satisfiability problem.

2.1 Boolean Functions

For a variable v, a *positive literal* represents the variable v, while the *negative literal* represents its negation $\bar{v}$. A *cube*, or a product, c, is a Boolean product (AND, $\cdot$) of literals, $c = l_1 \cdot \cdots \cdot l_k$. If a variable is not represented by a negative or a positive literal in a cube, then it is represented by a *don't-care ($\leq$)*, meaning that it can take both values 0 and 1. A cube with i don't-cares, covers 2^i minterms. A *minterm* is the smallest cube in which every variable is represented by either a negative or a positive literal.

Let $f(X) : B^n \rightarrow \{0, 1, \leq\}$, $B - \{0, 1\}$, be an *incompletely specified Boolean function* of n variables $X = \{x_1, \ldots, x_n\}$. The terms function and circuit are used interchangeably in this paper. The *support set* of f is the subset of variables that determine the output value of the function f. The set of minterms for which f evaluates to 1 defines the *on-set* of f. Similarly, the minterms for which f evaluates to 0 and don't-care define the *ofl-set* and the *don't-care-set*, respectively. In a multi-output function $F = \{f_1, \ldots f_m\}$, each output f_i, $1 \in i \in m$, has its own support set, on-set, offiset and don't-care-set associated with it.

For simplicity, we define the following terms for single-output functions, although our algorithm can handle multi-output functions. Any Boolean function can be represented as a two-level *sum of products (SOP)*, which is a Boolean sum (OR, +) of cubes, $S = c_1 + \cdots + c_k$. Assume that a Boolean function f is represented as an SOP. A cube is *prime*, if no literal can be removed from the cube without changing the value that the cube implies for f. A cube that is not prime, can be *expanded* by substituting at least one literal with a don't-care. The SOP is *irredundant* if each cube is prime and no cube can be deleted without changing the function.

A *canonical representation* is a unique representation for a function under certain conditions. For example, given a Boolean function and a fixed input variable order, a canonical SOP is an SOP independent of the original representation of the function given to the SAT solver. In a similar way, BDDs can be used to generate a canonical SOP that only depends on an input variable order [15].

2.2 Boolean Satisfiability

A disjunction (OR, +) of literals forms a *clause*, $t = l_1 + \cdots + l_k$. A *propositional formula* is a logic expression defined over variables that take values in the set {0, 1}. To solve a SAT problem, a propositional formula is converted into its *Conjunctive Normal Form (CNF)* as a conjunction (AND, $\cdot$) of clauses, $F = t_1 \cdot \cdots \cdot t_k$. Algorithms such as the Tseitin

transformation [24] convert a Boolean function into a set of CNF clauses.

A *satisfiability (SAT) problem* is a decision problem that takes a propositional formula in CNF form and returns that the formula is *satisfiable* (*SAT*) if there is an assignment of the variables from the formula for which the CNF evaluates to 1. Otherwise, the propositional formula is *unsatisfiable* (*UNSAT*). A program that solves SAT problems is called a *SAT solver*. SAT solvers provide a *satisfying assignment* when the problem is satisfiable.

Modern SAT solvers can determine the satisfiability of a problem under given assumptions. *Assumptions* are propositions that are given as input to the SAT solver for a specific single invocation of the SAT solver and have to be satisfied for the problem to be SAT.

EXAMPLE 1. *For the function* $f(x_1, x_2, x_3) = (x_1 + x_2)\bar{x}_3$*, which is satisfiable for the following assignments of the inputs* $\{(0,1,0),(1,0,0),(1,1,0)\}$*, a SAT solver without assumptions can return any of the given assignments. But, if we give as input to the SAT solver the assumption* $x_1 = 1$*, then it returns either* $(1,0,0)$ *or* $(1,1,0)$*, because those two assignments satisfy the given assumption.*

A *lexicographic satisfiability (LEXSAT) problem* is a SAT problem that takes a propositional formula in CNF form and, given a variable order, returns a satisfying variable assignment whose integer value under the given variable order is minimum (maximum) among all satisfiable assignments. If the formula has no satisfiable assignments, LEXSAT proves it unsatisfiable. The LEXSAT problem was first mentioned by Knuth [9]. For our work, we use an efficient algorithm and methods for generating consecutive SAT assignments in lexicographic order [20].

EXAMPLE 2. *For the function* $f(x_1, x_2, x_3)$ *from Example 1, LEXSAT returns either the lexicographically smallest assignment* $(0,1,0)$ *or the lexicographically largest assignment* $(1,1,0)$*, depending on the user preference.*

3. SAT-BASED SOP GENERATION

In this section, we describe our SAT-based algorithm that progressively generates an irredundant SOP for a single-output function. For multi-output circuits, each output is treated separately. In this paper, we focus on completely-specified functions, but the algorithm can be easily used for incompletely-specified functions by providing both the on-set and off-set as input to the algorithm. In the case of a completely specified function one of them is derived by complementing the other.

The presented algorithm iteratively generates minterms, expands them into prime cubes, and adds these cubes to the SOP. The SAT-based heuristics for minterm generation and cube expansion are described in Section 3.1 and Section 3.2, respectively. Finally, to guarantee that the resulting SOP is irredundant, it is post-processed to remove redundant cubes, as described in Section 3.3. Additionally, Section 3.4 describes several techniques that reduce the runtime. The procedures described in the following subsections assume that we are generating the on-set SOP. The same procedures are used to generate the off-set SOP.

The algorithm can be implemented with one SAT solver parameterized to store both on-set and off-set. Alternatively, it can use two solvers, one for on-set and one for off-set. In our implementation of the algorithm, we use four different SAT solvers: for both on-set and off-set, one is used to generate satisfying assignments, the other to expand assignments to cubes. By employing four solvers, we ensure that assignment generation and expansion do not interact with each other during the SOP computation.

3.1 Generation of Minterms

In order to generate minterms for a function f by using a SAT solver, we initialize a SAT solver with the CNF of f. Then, to discard the trivial case when the function has a constant on-set, we solve the SAT problem by asserting that $f = 1$. If the problem is UNSAT, then f is a constant, and we return an SOP with one constant cube.

Generation of non-canonical SOP. Otherwise, if the problem is SAT, an assignment for the inputs is returned for which the function evaluates to 1. From the assignment, we can generate a minterm for the function f in which the variables assigned to 0 and 1 are represented with the negative and positive literal, respectively. For example, for a function $f(x, y, z)$, the assignment $(1,1,0)$ implies the minterm $xy\bar{z}$. Once a minterm is obtained, we expand it into a cube using the heuristic procedure from Section 3.2. Next, we add the cube with its literals complemented to the SAT solver as a *blocking clause*, which is an additional clause that blocks known solutions of the SAT problem. This allows to generate the next minterm that is not covered by any of the previously generated cubes. While the problem is SAT, we iteratively obtain a minterm, expand it to a cube, and add the cube to both the SAT solver and the SOP. The unsatisfiability of the problem indicates that the generated SOP is complete and covers all on-set minterms.

Generation of canonical SOP. However, generating minterms from satisfying assignments received from a SAT solver does not guarantee canonicity, since SAT solvers return minterms in a non-deterministic order that depends on the design of the solver and the CNF generated for the function. Thus, to ensure canonicity, we iteratively use a binary search-based LEXSAT algorithm, called `BINARY` [20], that generates minterms in a lexicographic order that is unique for a given variable order. The algorithm `BINARY` receives as input a potential assignment, which is the lexicographically smallest assignment that might be satisfiable, that is either the last generated minterm or, initially, an assignment with all 0s. Then, `BINARY` tries to verify and fix the assignment of each variable defined with the potential assignment starting from the leftmost variables and moving to right. We also use the proposed methods for runtime improvement [20]: skip verifying the leading 1s, correcting the initial potential assignment, and profiling the success of the first SAT call. Similarly to the non-canonical SOPs, once we obtain a minterm, we expand it into a cube and add it to the SAT solver as a blocking clause.

EXAMPLE 3. *For example, assume that for the function* $f(x_1, \ldots, x_8)$*, the last generated minterm* $(1,1,0,0,0,0,0,1)$ *is received as an initial potential assignment. Since this minterm is covered by the last cube, this assignment is not satisfiable, so we can increase its value for 1 to get the smallest assignment that might be satisfiable* $(1,1,0,0,0,0,\ 1,0)$*. Next, we can skip verifying the assignments* $x_1 = 1$ *and* $x_2 = 1$*, because the next lexicographically smallest assignment has to start with the same leading 1s. Thus, we should*

only check the assignments for x_i, for $3 \in i \in 8$. Due to using binary search, with the first SAT call we assume half of the unverified assignments, and we give to the on-set SAT solver the assumptions $(x_1, \ldots, x_5) = (1, 1, 0, 0, 0)$. Assume that the problem was satisfiable and the SAT solver returned the assignment $(1, 1, 0, 0, 0, 0, 1, 1)$. This assignment proves that an on-set minterm with the assumed values exists, but moreover we can learn that the assignments from the potential minterm $x_6 = 0$ and $x_7 = 1$ are correct. Next, to check if the assignment for the last input x_8 can be set to 0, we call the SAT solver with the assumptions $(x_1, \ldots, x_8) = (1, 1, 0, 0, 0, 0, 1, 0)$. If it returns SAT, we return the potential assignment as a minterm. Otherwise, we flip x_8 to 1 before returning it.

3.2 Expansion of Minterms to Cubes

In this subsection, we describe our SAT-based procedure that receives a minterm and transforms it into a prime cube by iteratively removing literals (i.e., substituting them with don't-cares). For the on-set SOP, a literal can be removed, if after its removal all minterms covered by the cube belong to the on-set.

Greedy deterministic cube expansion. The heuristic removes literals in two rounds. First, we remove literals greedily, in the given order, after ensuring that additional on-set minterms are covered by expanding each literal.

Example 4. *Assume that for the function on Figure 1, the cube c_1 was computed and added to the SAT solver, and as a second minterm $xyzt$ is generated, which can be extended by removing one of the literals x, y or t. If we remove x, we will obtain the cube c_4 that covers only one additional minterm with respect to the existing cube c_1, but if we remove y or t, we will obtain c_2 or c_5, respectively, each of which covers two additional minterms with respect to c_1.*

For Example 4, our expansion procedure skips the opportunity to remove the literal x, and tries to expand other literals if possible. To check if by removing a literal l_i, the expanded cube covers more than one new minterm, we flip l_i and provide it, along with the remaining literals of the cube, as assumptions to an on-set SAT solver that contains the already generated cubes. If the problem is SAT, then we consider this literal for removal since by removing it we cover more than one minterm. Otherwise, we skip removing it temporarily. Once we have found a literal that should be considered for removal, we perform the following *final SAT check* to ensure that it can be removed. First, we assume that the literal is removed from the cube. Next, we run the offset SAT solver with assumptions for all remaining literals of the cube. If the problem is UNSAT, then no minterm covered by the cube belongs to the offset, so we can extend the cube by removing this literal. On the other hand, if the problem is SAT, we cannot extend the cube, since the SAT solver found a minterm that belongs to the offset and is covered by the extended cube.

Expansion to prime cubes. In the first round, we might skip some opportunities for expansion, but in the second round, for each remaining literal, we execute the final SAT check described above. This guarantees that no literal can be further removed, which means that the cube is prime. Since, we always try to remove the literals in the same order, this method generates a canonical SOP.

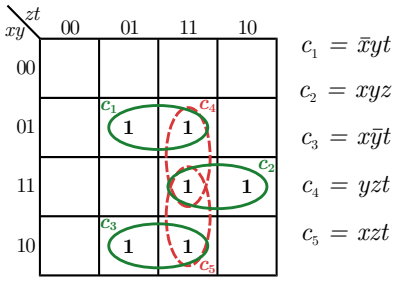


Figure 1: A Karnaugh map for the Boolean function $f(x, y, z, t) = \bar{x}yt + xyz + x\bar{y}t$ with its prime cubes c_i, where $1 \in i \in 5$. The cubes c_1, c_2 and c_3 are essential and they compose the minimum SOP of f.

Fast non-canonical expansion. If generating a canonical SOP is not required, we can substitute the first round of expansion with a faster method to improve runtime: If in an offset SAT solver we assume the values from the received minterm, which is generated from the on-set, the problem is UNSAT and the SAT solver returns the set of literals used to prove unsatisfiability (procedure "analyse_final" in MiniSAT [6]). Thus, we can remove literals that are not returned by the SAT solver. However, the set of remaining literals is not minimal, and thus we run additionally the second round of removal with the final SAT check.

3.3 Removing Redundant Cubes

The cubes expanded with the methods from Section 3.2 are prime by construction. However, by progressively adding cubes to the SAT solver, as described in Section 3.1, we ensure that each cube is irredundant with respect to the preceding cubes, but not with respect to the whole set.

Example 5. *For the function f from Figure 1, assume that the cubes c_1, c_5, c_2 and c_3 are generated in the given order. The cube c_5 is irredundant with respect to c_1, since it additionally covers the minterms $xyzt$ and $x\bar{y}zt$, but it is contained in the union of c_2 and c_3.*

In order to produce an irredundant SOP, after generating all cubes, we iterate through the cubes to detect and remove redundant ones. First, we initialize a new SAT solver with clauses for all generated cubes and we assume that all cubes are required. Then, by using the assumption mechanism, for each cube c_i, we check if there is an assignment for which c_i evaluates to 1 while all the other irredundant cubes evaluate to 0. If the problem is SAT, the cube is irredundant and the SAT solver returns an assignment that corresponds to a minterm which is covered only by c_i. Otherwise, if the problem is UNSAT, then the cube is redundant, and thus it is removed from the SOP and is excluded when checking the redundancy of the following cubes. Since we always try to remove cubes in the order in which they were generated, this method is deterministic and maintains canonicity when canonical SOPs are generated.

Example 6. *Considering the cubes from Example 5, to check whether c_3 is redundant, we set $c_3 = 1$ by assuming the values $x = 1$, $y = 0$ and $t = 1$. For the assumed values, the other cubes evaluate to $c_1 = 0$, $c_2 = 0$ and $c_5 = z$. Setting $z = 0$ leads to $c_5 = 0$. Thus, the problem is SAT and c_3 is irredundant. The returned assignment $(x, y, z, t) = (1, 0, 0, 1)$ defines a minterm $x\bar{y}\bar{z}t$ that is covered only by c_3.*

3.4 Improving the Runtime

In this subsection, we present four techniques that improve the runtime of the algorithm by allowing early termination and by treating some special cases.

Simultaneous on-set and off-set generation. Often, the SOP of the on-set and offset differ in size. For example, a two-input function implementing an AND gate, $f(x, y) = xy$, has on-set SOP, $f = S_{\text{on}} = xy$ with size 1, and offset SOP, $\bar{f} = S_{\text{offi}} = \bar{x} + \bar{y}$ with size 2. Thus, we propose to generate on-set and offset cubes simultaneously, one cube at a time from each set. This way, if one of them is much smaller than the other, we can avoid the situation when the large set of cubes has to be first generated, before a smaller one is discovered.

Prioritizing outputs with large SOPs. Before generating SOPs for each output, we propose to sort outputs by size of their input supports. The outputs with larger supports are processed first since it is more likely that SOP generation for these outputs will exceed resource limits, so we can determine if we should terminate the computation earlier.

Isomorphic circuits. We implemented a method to decrease the runtime by detecting isomorphic outputs. For this, first, we divide the outputs into isomorphic classes. Two outputs are *isomorphic* and belong to the same class, if they implement an identical function using different inputs. Then, for each class, we generate an SOP only for one output, and duplicate it for the others. In Section 4.2, we show that this allows effective generation of an SOP only for 39.7% of the outputs. When running our SAT-based method on industrial benchmarks, which could not be quoted in the paper, we found that typically around 70% of the outputs are isomorphic, which means that this feature leads to at least 3x speedup.

CNF sharing. Generating a CNF for each output is time consuming. Thus, to benefit from the logic sharing among the outputs, we can optionally share one CNF, which corresponds to the complete circuit. For this, we generate the CNF of the circuit, and then, for each output, we initialize the SAT solver only with the part of the CNF for the corresponding output. Besides improving the runtime, as Table 1 shows, this option sometimes leads to better results.

4. EXPERIMENTAL RESULTS

In this section, we describe our experimental setup and compare the proposed SAT-based algorithm with a state-of-the-art BDD-based method.

4.1 Experimental Setup

We implemented the algorithm described in Section 3 as a new command *satclp* in ABC [2]. ABC is an open-source tool, designed for logic synthesis, technology mapping, and formal verification for logic circuits. ABC features an integrated SAT solver based on an early version of MiniSAT [6] that supports incremental SAT solving. Furthermore, ABC provides an implementation of BDD construction for a multi-level circuit (command *collapse*) and the BDD-based ISOP computation [15] (command *sop*). Finally, ABC allows us to analyze the area-delay results when the generated SOPs are used to build a new multi-level circuit representation. A multi-level network is generated using the *fx* command [21]. Then, it is converted into an and-inverter graph, which is

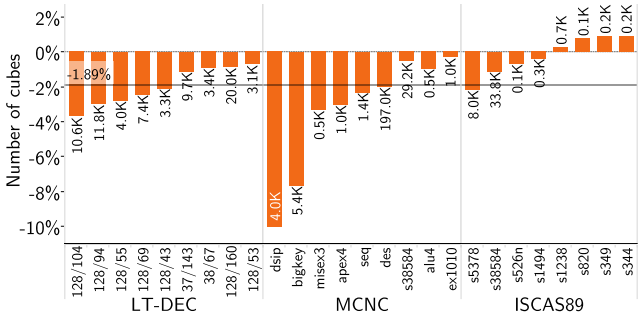


Figure 2: Size of the smallest SOPs generated by the SAT-based algorithm compared to the smallest SOP generated by the BDD-based method. Only the benchmarks for which the SOP size differs are shown. The gray line shows that, on average, the SAT-based method decreases the SOP size by 1.89%. Next to each bar we show the actual number of cubes for the SAT-based SOP.

an internal representation of ABC, and optimized with the *dc2* command. This shows how the difference in the SOPs affects the methods using them.

To evaluate our algorithm, we use the ISCAS'89 benchmarks, a set of large MCNC benchmarks and a set of 9 logic tables from the instruction decoder unit [1], which we denote with LT-DEC. The LT-DEC has been show as well-suited to demonstrate the factoring gains as circuit size increases [10]. The names of the LT-DEC benchmarks are given in the form "$[N_{\text{PI}}]/[N_{\text{PO}}]$", where N_{PI} is the number of primary inputs and N_{PO} is the number of primary outputs. For the main experiments, we discard benchmarks for which the SOP size exceeds the built-in resource limits of the command *sop* or *fx*, and thus, we use 30 (out of 32) and 14 (out of 20) benchmarks from the ISCAS'89 and MCNC set, respectively. With the discarded benchmarks, we demonstrate the generation of partial SOPs.

The reported runtime is the runtime returned by the command *time*. For the BDD-based method we call it before and after calling both *collapse* and *sop*. For our SAT-based algorithm, we call it before and after our command *satclp* that executes the algorithm. Thus, it reports the time for the complete algorithm, including detecting isomorphic outputs, generating CNF and initializing SAT solver instances, as well as the time for all SAT calls for cube generation, expansions and removing redundant cubes.

4.2 SAT- vs. BDD-based SOP Generation

To analyze the performance of the algorithm presented in Section 3, we run both the SAT-based algorithm and the BDD-based method available in ABC. In this section, we present the results of these experiments while analyzing the strengths and shortcomings of our approach.

Although the command *collapse* dynamically finds a good variable order for the BDD, changing the initial order of the primary inputs results in a different BDD structure, which leads to a different SOP. Thus, to obtain a good SOP, we generate five SOPs for the BDD-based method by using five different initial orders of the primary inputs. Similarly, our SAT-based algorithm generates different SOPs for different orders of the primary inputs, which define the order of re-

Table 1: Number of benchmarks (out of the 53 used benchmarks) for which using a given combination of options for the SAT-based algorithm resulted in the smallest SOP in terms of number of cubes (column "#Cubes") or the best area-delay product (column "A·D"). When we select the best result, if we obtain same result with more than one combination of options, we give priority to the default one with least changes. The default options are using the initial order of the benchmark, no canonicity, no CNF sharing, and no reversing of the order.

Order PI	Canonical	Shared CNF	Reverse	#Benchmarks #Cubes	A·D
Initial	No	No	No	15	15
			Yes	1	5
		Yes	No	-	3
			Yes	-	6
	Yes	No	No	23	12
			Yes	8	7
Fanouts	No	No	No	-	3
			Yes	1	1
	Yes	No	No	2	1
			Yes	3	-

Table 2: Comparison of the number of combinational outputs, which are primary outputs and latch inputs, in the used benchmarks and the number of calls of the function for generating SOPs.

	Comb. outputs	Generated SOPs	
LT-DEC	788	682	86.5%
MCNC	2779	1345	48.4%
ISCAS'98	5753	1675	29.1%
Total	9320	3702	39.7%

moving literals from the cubes. We either use the pre-defined order from the benchmark or order the inputs based on their number of fanouts, which currently only works for the combinational benchmarks. We can also, optionally, reverse the selected variable order. Moreover, we can enable or disable generation of canonical SOPs, and we can decide whether to generate one CNF for all outputs as described in Section 3.4. Thus, by trying different orders and changing the options, we generate 12 SOPs using the SAT-based method.

Generating multiple SOPs with each method results in SOPs that differ in size, where the SOP size is equal to the number of cubes that constitute the SOP. Figure 2 shows and compares the benchmarks for which the size of the smallest SOP generated by each method is different. We can see that the SAT-based algorithm decreases the SOP size up to 10% compared to the BDD-based method. Since the results for the SAT-based method are obtained by using several different options, Table 1 shows, in the column "#Cubes", the number of benchmarks for which the smallest SOP, in terms of number of cubes, was generated with each combination of options. We can notice that, for 36 benchmarks we get the smallest SOP when the canonical option is activated, thus for 68% of the benchmarks the canonical SOPs are smaller than the non-canonical ones.

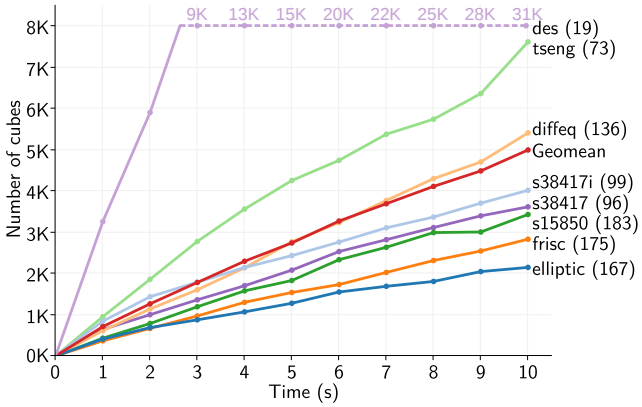


Figure 3: Number of generated cubes for a partial SOP when the time limit is set between 1 and 10 seconds. The number of generated cubes depends on the size of the support set of the function, which is given in brackets. Only for the benchmark des we managed to generate complete SOPs for 3 to 25 outputs, in the given limits. For the other benchmarks, the generated cubes belong to one output.

In terms of scalability, as Table 2 shows, the technique for isomorphic circuits presented in Section 3.4 allows computing an SOP only for 39.7% of the combinational outputs, each of them representing one isomorphic class, while for the other outputs we duplicate the generated SOP of the class representative. This reduces the runtime of our algorithm, and for benchmarks rich in isomorphic outputs, the proposed method is significantly faster than the BDD-based one. For example, the maximum speedup is achieved for the s35932 benchmark from the ISCAS'89 set, for which we generate SOPs only for 10 out of 2048 combinational outputs and thus, on average, the SAT-based method requires 0.06 seconds, while the BDD-based method finished in 1.83 seconds. However, on average, our SAT-based method is 8x slower than the BDD-based method for the public benchmarks. We have observed that the functions for expanding minterms to cubes are the bottleneck. For example, for the LT-DEC benchmarks, on average, 84% of the runtime is spent on this operation, while 8% is spent on minterm generation, 2% on removing redundant cubes, and 6% on other operations, such as dividing the outputs into classes, generating CNF, initializing SAT solver instances, etc.

Additionally, we conducted several experiments on industrial benchmarks, which could not be quoted in the paper. Our conclusion is that the SAT-based method is often as fast the BDD-based one and is definitely more scalable, that is, it can finish on some test-cases where the BDD-based method fails. We believe that the increased scalability is largely due to the fact that most of the industrial test cases have hundreds of inputs and outputs, which makes constructing global BDDs in the same manager problematic for all outputs at once. Our SAT-based method does not suffer from this limitation, because it computes the SOPs one by one, for each output separately. It can be argued that the BDD-based computation can also be performed on a per-output basis. However, in this case, the BDD manager will inevitably find different variable orders for different outputs, which will substantially reduce the quality of factoring. This

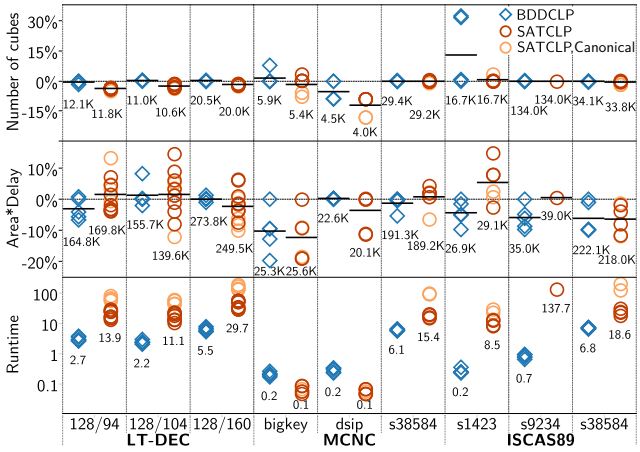


Figure 4: Variability of the results of the BDD-based and SAT-based method for the three largest circuits of each set. The results for the number of cubes and the area-delay product are normalized to one of the seeds of the BDD-based method. The horizontal lines show the averages for each method. Below the symbols are the actual minimal results.

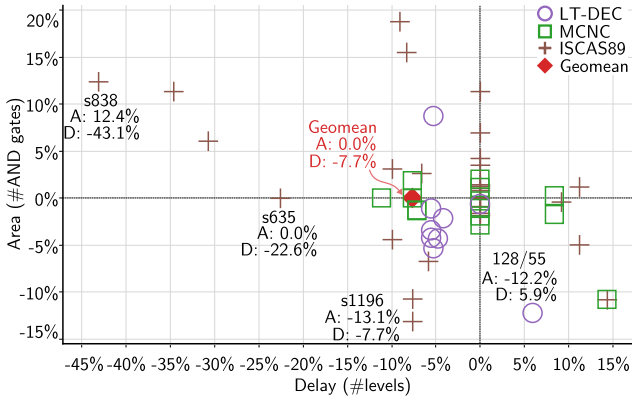


Figure 5: The best results for each benchmark after a multi-level description is built from SOPs generated by our SAT-based algorithm, compared to using a BDD-based SOPs. For most benchmarks, we obtain Pareto optimal solutions.

is because factoring benefits from computing BDD-based SOP using the same variable order, which facilitates creating similar combinations of literals in different cubes, which in turn helps improve the quality of shared divisor extraction and factoring.

Finally, Figure 3 shows the number of generated cubes when we generate non-canonical SOPs and the time limit is set to t seconds, where t is an integer value such that $1 \in t \in 10$. For functions with larger support set, we usually generate less cubes as more time is required for cube expansion. Since we generate cubes progressively, unlike the BDD-based method, we can build partial SOPs even for large circuits, and these can be used for incremental applications. For this experiment, we are still generating both the on-set and the offi-set SOP at the same time. However, for incremental application, we can generate just one of them, which would increase the number of generated cubes for a given time limit.

4.3 Case-Study: SAT-based SOPs for Generation of Multilevel Implementation

As explained in Section 4.2, we generate several SOPs with each method. Consequently, as Figure 4 shows, the different SOPs from each method, which have different size, result in multi-level networks with different area and delay. Figure 5 shows that our algorithm obtains Pareto optimal solutions compared to the BDD-based method for most benchmarks, if for each method we isolate the best circuit structures in terms of area-delay product. Table 1, with the column "A·D", shows the number of benchmarks with the smallest area-delay product generated with each combination of options. For 17 benchmarks we get the best results when the canonical option is activated. Thus, for 38% of the benchmarks generating canonical SOPs improves the results. Similarly, using the initial order of the primary inputs results in the best area-delay product for 91% of the circuits, and for 62% of them it is best to keep the order unreversed.

5. RELATED WORK

After the Quine-McCluskey algorithm [13], many algorithms and heuristics for SOP generation and minimization have been developed. Prior research largely divides into two main classes; BDD-based and ESPRESSO style algorithms.

BDD-based techniques, such as that of Minato-Morreale [15] and SCHERZO [5], first build a BDD or a ZDD to generate an SOP for a given Boolean function, and then apply a heuristic approach to minimize the BDD/ZDD size, which leads to a smaller SOP. If building a BDD is feasible, then an SOP, even a suboptimal one, can be generated. However, for some logic circuits with hundreds of inputs, building BDDs is sometimes not feasible or requires very long runtimes. On the other hand, given a time limit, our method returns a partial SOP when a complete SOP is not possible to obtain. This helps predicting if an SOP can be generated in a short runtime or a long runtime is required. Moreover, it allows to estimate the SOP size and if it is suitable or not for subsequent phases.

A second group of researchers had been inspired by the logic minimizer ESPRESSO [3] to develop fast and eff cient logic minimization tools, such as ESPRESSO-MV [22]. Although ESPRESSO style techniques avoid the memory explosion problem of BDDs, they still incur impractical runtimes for very large Boolean functions. A recent work by Sapra et al. [23] proposed using a SAT solver to implement some of ESPRESSO's operators as a way to speed it up. However, an ESPRESSO based approach requires as input a computed SOP, on which its runtime and end results significantly rely. Contrary, our SAT-based algorithm generates and minimizes cubes one by one, and it returns irredundant SOP in relatively short runtime.

6. CONCLUSION

In this paper, we present a novel algorithm for progressive generation of irredundant canonical SOPs using heuristics based solely on SAT solving. Besides generating SOPs, the canonicity and the progressive generation make our heuristics desirable in many other areas where minterms or cubes are required, and for which the existing methods are either unscalable or impractical to use.

Regarding the quality of results, we show that for computing a complete SOP, on average, the SAT-based computation is as good as the BDD-based method. Moreover, the multi-level circuit structures derived using the SOPs generated by our approach are most often better or Pareto optimal.

Regarding the runtime, the proposed method is somewhat slower than the BDD-based method for most of the public benchmarks, but it is faster for circuits that are rich in isomorphic outputs. Other industrial benchmarks, which we tried but could not use in the paper, show that our method is both faster and more scalable, and therefore a good candidate for global circuit restructuring at least in that particular industrial setting.

The proposed method can also benefit from the ongoing improvement of modern SAT solvers. For example, recently we explored a new push/pop interface for assumptions used in the incremental SAT solving, which led to additional runtime improvements. As we show, for some circuits the results can improve by changing the variable order in which the cubes are expanded, but a careful study of this problem is required to improve further the quality of results.

In addition to runtime improvements, future work will focus on developing a dedicated SAT-based multi-output SOP computation, which computes cubes that are shared between several outputs. A recent publication [10] indicates that a significant improvement in quality (more than 10%) can be achieved by computing and factoring multi-output SOPs. We are not aware of a practical method for BDD-based multi-output SOP computation, so it is likely that SAT will be the only way to work with multiple outputs. Other directions of future work will include exploring the benefits of the progressive generation of canonical minterms and cubes in different areas. One such area is multi-level logic synthesis where incremental SAT-based decomposition methods can be developed based on partial SOPs computed for the output functions.

7. REFERENCES

[1] The EPFL Combinational Benchmark Suite, "Multi-output PLA benchmarks". `http://lsi.epfl.ch/benchmarks`.

[2] Berkeley Logic Synthesis and Verification Group, Berkeley, Calif. *ABC: A System for Sequential Synthesis and Verification*. `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[3] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, Boston, Mass., 1984.

[4] K. Chang, V. Bertacco, I. L. Markov, and A. Mishchenko. Logic synthesis and circuit customization using extensive external don't-cares. *ACM Trans. on Design Automation of Electronic Systems*, 15(3):26:1–24, May 2010.

[5] O. Coudert. Two-level logic minimization: An overview. *Integration, the VLSI journal*, 17(2):97–140, Oct. 1994.

[6] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, volume 2919, pages 502–18. Springer, May 2003.

[7] A. Ghosh, S. Devadas, and A. R. Newton. Test generation and verification for highly sequential circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 10(5):652–67, May 1991.

[8] J.-H. R. Jiang, C.-C. Lee, A. Mishchenko, and C.-Y. R. Huang. To SAT or not to SAT: Scalable exploration of functional dependency. *IEEE Trans. on Computers*, C-59(4):457–67, Apr. 2010.

[9] D. E. Knuth. *Fascicle 6: Satisfiability*, volume 19 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., Dec. 2015.

[10] V. N. Kravets. Application of a key-value paradigm to logic factoring. *Proceedings of the IEEE*, 103(11):2076–92, Nov. 2015.

[11] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung. Bi-decomposing large Boolean functions via interpolation and satisfiability solving. In *Proceedings of the 45th Design Automation Conference*, pages 636–41, Anaheim, Calif., June 2008.

[12] H.-P. Lin, J.-H. R. Jiang, and R.-R. Lee. To SAT or not to SAT: Ashenhurst decomposition in a large scale. In *Proceedings of the International Conference on Computer Aided Design*, pages 32–37, San Jose, Calif., Nov. 2008.

[13] E. J. McCluskey. Minimization of Boolean functions. *Bell System Tech. Journal*, 35(6):1417–44, Nov. 1956.

[14] K. L. McMillan. Interpolation and SAT-based model checking. In *Proceedings of the International Conference on Computer Aided Verification*, volume 2725, pages 1–13. Springer, July 2003.

[15] S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Proceedings of Synthesis And Simulation Meeting and International Interchange*, pages 64–73, Kobe, Japan, Apr. 1992.

[16] A. Mishchenko, R. Brayton, S. Jang, and V. N. Kravets. Delay optimization using SOP balancing. In *Proceedings of the International Conference on Computer Aided Design*, pages 375–82, San Jose, Calif., Nov. 2011.

[17] A. Mishchenko and R. K. Brayton. SAT-based complete don't-care computation for network optimization. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 412–17, Munich, Mar. 2005.

[18] A. Morgado and J. P. M. Silva. Good learning and implicit model enumeration. In *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, pages 131–36, Hong Kong, Nov. 2005.

[19] A. Nadel. Generating diverse solutions in SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 287–301, Ann Arbor, Mich., June 2011.

[20] A. Petkovska, A. Mishchenko, M. Soeken, G. De Micheli, R. Brayton, and P. Ienne. Fast generation of lexicographic satisfiable assignments: Enabling canonicity in SAT-based applications. In *Proceedings of the 25th International Workshop on Logic and Synthesis*, Austin, Tex., June 2016.

[21] J. Rajski and J. Vasudevamurthy. The testability-preserving concurrent decomposition and factorization Boolean expressions. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 11(6):778–93, June 1992.

[22] R. L. Rudell and A. L. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–50, Sept. 1987.

[23] S. Sapra, M. Theobald, and E. M. Clarke. SAT-based algorithms for logic minimization. In *Proceedings of the 21st IEEE International Conference on Computer Design*, page 510, San Jose, Calif., Oct. 2003.

[24] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, Symbolic Computation, pages 466–83. Springer, Berlin, 1983.

[25] A. K. Verma, P. Brisk, and P. Ienne. Iterative Layering: Optimizing arithmetic circuits by structuring the information flow. In *Proceedings of the International Conference on Computer Aided Design*, pages 797–804, San Jose, Calif., Nov. 2009.

[26] J. Yuan, A. Aziz, C. Pixley, and K. Albin. Simplifying Boolean constraint solving for random simulation-vector generation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(3):412–20, Mar. 2004.

SAT-based Functional Dependency Computation

Mathias Soeken[1], Pascal Raiola[2], Baruch Sterin[3], Matthias Sauer[2]
[1] EPFL, Switzerland [2] University of Freiburg, Germany [3] UC Berkeley, CA, USA
mathias.soeken@epfl.ch, {raiolap,sauerm}tf.uni-freiburg.de, sterin@berkeley.edu

***Abstract*—We present an algorithm for computing both functional dependency and unateness of combinational Boolean functions represented as logic networks. The algorithm uses SAT-based techniques from *Combinational Equivalence Checking* (CEC) and *Automatic Test Pattern Generation* (ATPG) to compute the dependency matrix of multi-output Boolean functions. Experiments show the applicability of the methods and the improved robustness compared to existing approaches.**

I. Introduction

In this paper we present an algorithm to compute the *dependency matrix* $D(f)$ for a given combinational multi-output function f. For every input-output pair, the combinational dependency matrix indicates whether the output depends on the input, and whether the output is positive or negative unate in that input [1].

Several algorithms in logic design use the dependency matrix as a *signature* [2] to speed up computation, e.g., Boolean matching [3], functional verification [4], [5], or reverse engineering [6]. Although most of these algorithms make implicit use of the dependency matrix, the name has been used in this paper for the first time. The name is inspired by the output format of functional dependence and unateness properties in the state-of-the-art academic logic synthesis tool ABC [7]. Functional dependency is also related to *transparent logic* [8], [9]. Given a set of inputs X_d and a set of outputs Y_d, the problem is to find a set of inputs X_c that is disjoint from X_d and that distinguishes the output values at Y_d for different input assignments to X_d. In contrast, we consider functional dependence without constraints for all input-output pairs.

Existing algorithms for computing the dependency matrix are based on *Binary Decision Diagrams* (BDDs, [10]) and have been implemented in ABC [7]. It is important to point out that the term *functional dependence* is used to describe a different property in a related context: In [5], [11], [12], the authors refer to functional dependence as the question whether given a set of Boolean functions $\{f_1, \ldots, f_n\}$, there exists an f_i, that can be written as $h(f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n)$. In other words, functional dependence is defined of a Boolean function w.r.t. to a set of Boolean functions. In contrast, we consider the functional dependence of a Boolean function w.r.t. a single variable as functional dependence.

Our algorithm uses techniques from *Combinational Equivalence Checking* (CEC, e.g., [13]) and *Automatic Test Pattern Generation* (ATPG, e.g., [14], [15], [16]). We employ efficient incremental SAT-based solving techniques and extract incidental information from solved instances to reduce runtime consumption on complex circuits.

In an experimental evaluation we demonstrate the applicability of our methods to various benchmark sets. Within reasonable amounts of computing time we are able to accurately compute the dependency matrix. We further show the robustness our proposed algorithms compared to previous state-of-the-art algorithms that time out or suffer from memory explosion on complex circuits.

The rest of the paper is organized as follows. Section II presents the fundamentals of the work. In Section III we present our SAT-based approach to compute the dependecy matrix of combinational circuits. The experimental results are presented in Section IV and Section V concludes the work.

II. Background

A. Functional Dependencies

A Boolean function $f(x_1, \ldots, x_n)$ is *functionally dependent* in x_i if

$$f_{\bar{x}_i} \neq f_{x_i} \tag{1}$$

where the *co-factors* f_{x_i} or $f_{\bar{x}_i}$ are obtained by setting x_i to 1 or 0 in f, respectively. We call f_{x_i} the positive co-factor and $f_{\bar{x}_i}$ the negative co-factor. The function f is is said to be *positive unate* in x_i, if

$$f_{\bar{x}_i} \leq f_{x_i} \tag{2}$$

and *negative unate* in x_i, if

$$f_{\bar{x}_i} \geq f_{x_i}. \tag{3}$$

f is said to be unate in x_i if it is either positive or negative unate in x_i. Clearly, a function f is both positive and negative unate in x_i, if f does not depend on x_i. Hence, we call f *strictly* positive (negative) unate in x_i, if f is positive (negative) unate in x_i *and* depends on x_i. If f is neither positive nor negative unate in x_i, we say that f is *binate* in x_i.

Example 1: The functions $x_1 \wedge x_2$ and $x_1 \vee x_2$ are positive unate in both x_1 and x_2. The function $x_1 \rightarrow x_2$ is negative unate in x_1 and positive unate in x_2. The function $x_1 \oplus x_2$ is binate in both variables.

Let $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ be a multi-output Boolean function where each output is represented by a Boolean function $f_j(x_1, \ldots, x_n)$. The *dependency matrix* $D(f)$ is an $m \times n$ matrix with entries $d_{j,i}$ where

$$d_{j,i} = \begin{cases} \texttt{p} & \text{if } f_j \text{ is strictly positive unate in } x_i, \\ \texttt{n} & \text{if } f_j \text{ is strictly negative unate in } x_i, \\ \texttt{d} & \text{if } f_j \text{ depends on, but is not unate in } x_i, \\ \bullet & \text{otherwise.} \end{cases} \tag{4}$$

Example 2: Let $f : \mathbb{B}^5 \to \mathbb{B}^3$ with $f_1 = x_1 \wedge x_2$, $f_2 = x_3 \to x_5$, and $f_3 = x_1 \oplus x_2 \oplus x_5$. Then

$$D(f) = \begin{bmatrix} \mathtt{p} & \mathtt{p} & \bullet & \bullet & \bullet \\ \bullet & \bullet & \mathtt{n} & \bullet & \mathtt{p} \\ \mathtt{d} & \mathtt{d} & \bullet & \bullet & \mathtt{d} \end{bmatrix}.$$

B. Boolean Satisfiability

In our algorithms we translate decision problems into instances of the SAT problem [17]. SAT is the problem of deciding whether a function f has an assignment x for which $f(x) = 1$. Such an assignment is called a *satisfying assignment*. If f has a satisfying assignment it is said to be *satisfiable*. Otherwise, f is said to be *unsatisfiable*.

In general, SAT is NP-complete [18], [19]. SAT solvers are algorithms that can solve SAT problems and, while worst-case exponential, are nonetheless very efficient for many practical problems. SAT solvers also return a satisfying assignment if the instance is satisfiable. Most of the state-of-the-art SAT solvers are conflict-driven and clause-learning [20]. In *incremental* SAT one asks whether f is satisfiable under the assumption of some variable assignments. These assignments are only temporarily assumed, making it possible to reuse the SAT instance and learned information when solving a sequence of similar SAT problems. In the remainder of the paper, we refer to instances of SAT as if they were calls an incremental SAT solver. $\mathrm{SAT?}(f, \alpha)$ is true if f is satisfiable under the assumptions α, and $\mathrm{UNSAT?}(f, \alpha)$ is true if f is unsatisfiable under the assumptions α.

III. SAT-based Dependency Computation

This section presents the SAT-based algorithm to compute the functional dependencies of a circuit. We first describe the encoding into SAT, then an implementation of the algorithm, and finally possible optimizations.

A. SAT Encoding

We encode the test for functional dependence and unateness as an instance of the SAT problem using the following theorem.

Theorem 1: Let $f(x_1, \ldots, x_n)$ be a Boolean function. Then

1) f is functionally dependent in x_i, if and only if $f_{\bar{x}_i} \oplus f_{x_i}$ is satisfiable,
2) f is positive unate in x_i, if and only if $f_{\bar{x}_i} \wedge \bar{f}_{x_i}$ is unsatisfiable, and
3) f is negative unate in x_i, if and only if $f_{x_i} \wedge \bar{f}_{\bar{x}_i}$ is unsatisfiable.

Proof: We only show the direction of "if"; the "only if" direction follows immediately from the definition of functional dependency and unateness.

1) Let x be a satisfying assignment to $f_{\bar{x}_i} \oplus f_{x_i}$. Then, we have $f_{\bar{x}_i}(x) \neq f_{x_i}(x)$.
2) Assume the function was satisfiable and let x be a satisfying assignment. Then $f_{\bar{x}_i}(x) = 1$ while $f_{x_i}(x) = 0$ which contradicts Equation (2).
3) Analogously to 2). ∎

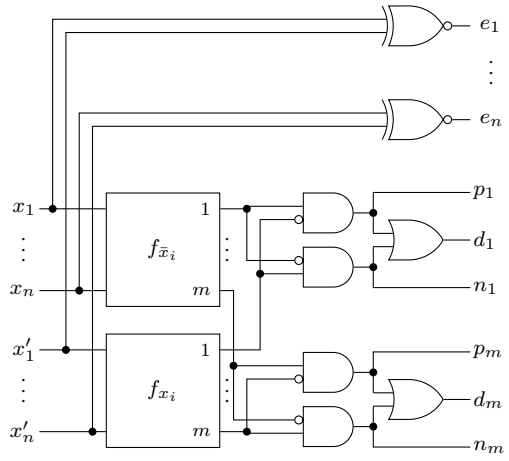


Fig. 1. Generic miter to encode functional dependency as SAT instance

In the implementation, we make use of the following immediate consequence of the theorem.

Corollary 1: f is functionally independent in x_i, if and only if $f_{\bar{x}_i} \oplus f_{x_i}$ is unsatisfiable.

In the following we consider multi-output functions $f : \mathbb{B}^n \to \mathbb{B}^m$, where each output is a function f_j. In order to compute the full dependency matrix which contains the dependency for each input-output pair, we transform the problem to a sequence of SAT instances as illustrated by the generic miter in Fig. 1. The two boxes represent two copies of f. The upper copy, with inputs $x_1, \ldots, x_n$, is used as the negative co-factor, while the lower copy, with inputs $x'_1, \ldots, x'_n$, is used as the positive co-factor of f. The groups of three gates on the lower right hand side realize the XOR operation which connect the outputs of the two copies. The signals of the AND gates are exposed as outputs and will be used to encode the unateness problems. The XNOR gates in the upper right of the figure are used to force all but one of the inputs, to have equal values.

Let $\Pi(f_j)$ be the characteristic Boolean function which is obtained by encoding the miter in Fig. 1 for the function f_j using encodings such as Tseytin [21] or EMS [22]. Also let

$$E_i = \{x_i = 0,\ x'_i = 1\} \cup \{e_k = 1 \mid k \neq i\} \tag{5}$$

be assignments that lead to a correct interpretation of the miter for input x_i, i.e., x_i is set to 0, x'_i is set to 1 and all the other inputs need to have the same value. We define three problems on top of the incremental SAT interface:

$$\begin{aligned} &\mathrm{DEP}(f_j, x_i) \\ &\quad = \mathrm{SAT?}(\Pi(f_j), E_i \cup \{d_j = 1\}) \end{aligned} \tag{6}$$

$$\begin{aligned} &\mathrm{POS\_UNATE}(f_j, x_i) \\ &\quad = \mathrm{UNSAT?}(\Pi(f_j), E_i \cup \{p_j = 1\}) \end{aligned} \tag{7}$$

$$\begin{aligned} &\mathrm{NEG\_UNATE}(f_j, x_i) \\ &\quad = \mathrm{UNSAT?}(\Pi(f_j), E_i \cup \{n_j = 1\}) \end{aligned} \tag{8}$$

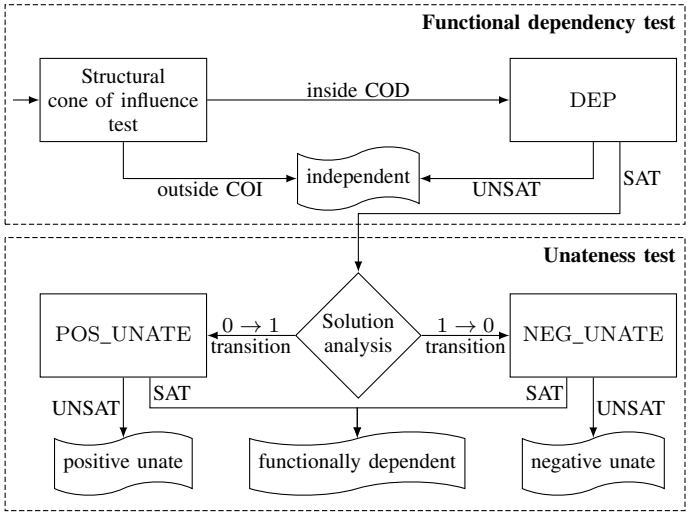


Fig. 2. Function Dependency and Unateness Computation Flow

Then the problems described in Theorem 1 and Corollary 1 can be solved as follows. The function f_j functionally depends on x_i, if $\mathrm{DEP}(f_j, x_i)$ holds. And the function f_j is positive (negative) unate in x_i, if $\mathrm{POS\_UNATE}(f_j, x_i)$ $(\mathrm{NEG\_UNATE}(f_j, x_i))$ holds.

B. Algorithm

Fig. 2 displays the general flow of the algorithm. For each pair of an input $x = x_i$ and an output $y = f_j$ the algorithm starts with a simple structural dependency check. If x is outside of y's structural cone of influence, it can be concluded that y is independent of x. This is a very efficient check. Otherwise, the algorithm proceeds with a functional dependency check $\mathrm{DEP}(y, x)$ as defined in Equation (6). We omit the arguments from the boxes.

If the instance is unsatisfiable, y is independent from x as no assignment exists that results in different logic values for y under the side constraint of $y_{\bar{x}} \oplus y_x$. In case the instance is satisfiable, x and y are at least functionally dependent. Additionally, the SAT solver returns a satisfying assignment which is analyzed for the logic value of y. In case $y_{\bar{x}}$ is 1 (and therefore y_x is 0), y cannot be positive unate in x as a counter example for Equation (2) is found. Likewise, negative unateness can be falsified if $y_{\bar{x}}$ is 0. Note that one of the two cases must hold as the original instance requires a difference between $y_{\bar{x}}$ and y_x.

In a last step, the algorithm specifically checks for unateness with an additional call to the SAT solver, unless it has been ruled out previously. If this SAT call is unsatisfiable, unateness can be concluded, otherwise the algorithm returns functional dependence.

C. Optimizations

As discussed above, we use incremental SAT solving because many of the calls to the SAT solver are very similar. Hence, instead of encoding a miter-like structure as illustrated in Fig. 1 for each input-output pair in an individual instance, we encode the complete output cone of a target input x_i in a single instance to profit from incremental SAT solving. We enforce the co-factors of x_i as unit clauses in this instance. As we target combinational circuits, the direction of the co-factors does not influence the satisfiability of the instance. Hence, we can restrict the search space by enforcing x_i to logic 1 and x_i' to logic 0 without loss of generality. Furthermore, XOR relations are encoded for each output to allow to enforce of a difference using an assumption.

On the output side, we iteratively run through each output in x's cone of influence and enforce a difference between $f_{\bar{x}_i}$ and f_{x_i} using an assumption. If the resulting instance is UNSAT we can conclude independence. Otherwise, the input-output pair is at least functionally dependent. By observing the direction of the difference at the output, we consider the pair either as an candidate for positive or negative unateness and run the respective check as described earlier.

Additionally, we perform a forward looking logic analysis of each satisfiable SAT instance to extract incidental information for future solver calls. In our experiments we found that quite often, the difference not only propagates to the target output, but also to multiple other outputs. Hence, we check the logic values of all following outputs as well. Additionally, an output may incidentally show differences in both directions and hence unateness can be ruled out without additional SAT calls.

The described SAT instances are very similar to detecting a stuck-at-1 fault at the input x. Hence, we employ encoding based speed-up techniques that are known from solving such ATPG problems. By adding D-chains [23] to the instance, the SAT solver can propagate the differences more easily. Additionally, we tuned the SAT solver's internal settings towards the characteristics of the circuit-based SAT instances which are dominated by a large number of rather simple SAT calls. For instance, we do not use preprocessing techniques on the SAT instances.

IV. Experimental Results

We implemented the proposed approach in C++ on top of the ATPG framework PHAETON [14] and the SAT solver *antom* [24]. All experiments were carried out on a single core of an Intel Xeon machine running at 3.3 GHz, 64 GB main memory running Linux 3.13. For the evaluations, we used combinational arithmetic benchmarks from EPFL[1] as well as sequential benchmarks from the ITC'99 [25] benchmark suite and industrial benchmarks provided by NXP (starting with 'b' and 'p' followed by a number, respectively). Finally, we applied the method to the Opencore benchmarks from the IWLS'05 [26] family. Sequential benchmarks were translated to combinational ones by expressing each latch as I/O pair. In order to keep the section compact, we skipped the benchmarks that had either negligible runtime or that could not be solved within a timeout of 2 hours.

[1] lsi.epfl.ch/benchmarks

TABLE I
EXPERIMENTAL EVALUATION

		Dependencies					Statistics		Runtimes	
Circuit	Inputs / Outputs	Structural	Functional	Pos. unate	Neg. unate	Incidental	Instances	Solves	Unateness	Total
adder	256 / 129	0	16512	256	0	16512	256	17024	4.07	4.40
bar	135 / 128	0	896	16384	0	777	135	33670	1.08	17.70
divisor	128 / 128	0	12220	66	2082	13026	128	9519	4502.32	4871.53
log2	32 / 32	0	1022	0	2	979	32	686	3746.98	3750.90
max	512 / 130	32512	32512	1024	512	15271	512	82241	37.13	110.24
sin	24 / 25	0	577	22	0	441	24	472	17.86	31.23
square	64 / 128	0	6041	68	3	5956	64	3926	447.34	450.51
b14	277 / 299	11	21803	705	68	18403	277	22368	20.74	27.71
b15	485 / 519	19504	40704	3338	292	33017	485	57931	77.33	231.29
b17	1451 / 1511	44054	135906	11271	1225	109432	1451	177747	234.55	637.20
b18	3307 / 3293	1084	331223	22367	2250	266523	3307	326342	439.45	690.52
b20	522 / 512	5298	51508	1212	370	42795	522	52742	120.06	232.61
b21	522 / 512	5310	51508	1238	136	42171	522	53808	94.69	203.93
b22	735 / 725	5313	78254	1740	359	65388	735	76254	163.15	280.46
p35k_s	2861 / 2229	0	140676	10802	10697	123321	2861	147727	628.73	1090.66
p45k_s	3739 / 2550	409	24910	13638	860	14256	3739	62307	20.84	59.95
p78k_s	3148 / 3484	377	52338	6032	0	48970	3148	50197	47.34	73.73
p81k_s	4029 / 3952	1380	387839	11724	18737	324849	4029	308638	419.38	897.82
p100k_s	5557 / 5489	756	77829	24347	3406	51415	5557	147954	4959.58	5236.16
des_area	367 / 192	0	11328	288	0	9756	367	5623	2.78	6.16
spi	272 / 273	16256	4205	982	117	1977	272	24649	1.54	10.03
systemcdes	312 / 255	592	2341	590	8	1580	312	4222	0.61	1.57
wb_dma	747 / 748	1195	10880	2364	842	6674	747	19879	2.25	4.54
tv80	372 / 391	9452	15265	1917	218	11906	372	28468	9.71	21.49
systemcaes	928 / 799	14613	17158	939	44	13273	928	29743	7.43	19.21
ac97_ctrl	2253 / 2247	10	7940	7083	831	5315	2253	25510	0.87	2.67
pci_bridge32	3517 / 3559	9906	59360	12179	2512	38839	3517	108600	57.35	106.20
aes_core	788 / 659	0	7290	541	29	6285	788	5636	2.17	4.43
wb_conmax	1899 / 2186	1976	46132	23968	16346	37884	1899	125420	39.40	98.11
des_perf	9041 / 8872	0	29344	7448	0	18627	9041	51718	16.63	68.60

All times are in seconds.

A. Dependency Computation

Table I lists the results of the evaluation. The first three columns list the name of the circuit as well as the number of inputs and outputs. The next four columns list the dependencies that were found, including the number of structural dependencies (only the strongest dependency is counted). The next three columns list statistics of the proposed SAT-based approach: The number of functional dependencies that were found incidentally followed by the number of generated SAT instances and calls to the SAT solver. The final two columns list the runtime for unateness checking and the total runtime in seconds.

As can be seen, our approach is able to completely compute the dependency table on a wide range of mid-sized circuits taken from various academic and industrial benchmark circuits within a maximum computation time of 2 hours (7200 seconds).

Interestingly, the number of input-output pairs that are positive unate are roughly an order of magnitude higher than those that are negative unate. This is most prominent for the barrelshifter circuit ‘bar’ from the EPFL benchmarks that contains mostly positive unate pairs but no negative one.

The effect of the optimizations described in Section III-C can be witnessed by the high number of dependencies identified incidentally as well the high ratio between the number of instances as well as the calls to the SAT solvers. Hence, these methods effectively keep the runtimes in check.

TABLE II
COMPARISON TO THE BDD-BASED APPROACH FROM ABC [7]

		Runtimes		ABC [7]	
Circuit	In / Out	Unate.	Total	Unate.	Total
adder	256 / 129	4.07	4.40	0.01	0.54
bar	135 / 128	1.08	17.70	18.96	19.05
divisor	128 / 128	4502.32	4871.53	TO	TO
log2	32 / 32	3746.98	3750.90	TO	TO
max	512 / 130	37.13	110.24	TO	TO
sin	24 / 25	17.86	31.23	0.15	866.99
square	64 / 128	447.34	450.51	TO	TO
b14	277 / 299	20.74	27.71	74.17	120.07
b15	485 / 519	77.33	231.29	199.45	368.25
b17	1451 / 1511	234.55	637.20	MO	MO
b18	3307 / 3293	439.45	690.52	MO	MO
b20	522 / 512	120.06	232.61	MO	MO
b21	522 / 512	94.69	203.93	MO	MO
b22	735 / 725	163.15	280.46	MO	MO

All times are in seconds; MO: memory out; TO: timeout ($\geq$ 7200 s)

B. Comparison to Existing Approach

We compared our approach to the BDD-based implementation in ABC [7] where identical circuit definitions readable for both tools were available. We listed the results in Table II.

The proposed SAT-based approach shows superior performance for the rather complex benchmark sets of the EPFL as well as the ITC'99 benchmarks where the approach does not suffer from excessive memory usage. For complex circuits, the BDD-based approach did not terminate due to insufficient memory requirements.

For the EPFL benchmarks, the BDD-based approach did not terminate due to a timeout which we set to 7200 seconds. 7 of the 10 arithmetic EPFL benchmarks can be solved using the SAT-based approach, and for 6 of them the SAT-based approach found the solution faster. The three remaining benchmarks cannot be solved within 7200 seconds by both approaches. It is worth to note that for benchmarks that are rather small or structurally simple (such as the adder) the BDD-based approach performs faster than the SAT-based approach.

V. Conclusions

We presented a SAT-based algorithm to compute functional dependence properties of combinational Boolean functions. We inspect which outputs in a multi-output function are functionally dependent on which inputs. Furthermore, the algorithms checks whether the input-output pair is unate if it is dependent, which is a stronger property. Furthermore, incremental encoding techniques known from ATPG problems are employed to speed up the algorithm.

In experimental studies on different benchmarks suites we detailed the robustness of the algorithms especially for hard combinational benchmarks. Additionally, out methods show better performance compared to previously presented BDD-based approaches with which many of the instances cannot be solved due to memory limitations or timeouts.

Acknowledgments

The authors wish to thank Robert Brayton, Alan Mishchenko, Bernd Becker, and Giovanni De Micheli for many helpful discussions. We also thank the anonymous reviewers for their valuable feedback. This research was supported by H2020-ERC-2014-ADG 669354 CyberCare and by the Baden-Württemberg Stiftung gGmbH Stuttgart within the scope of its IT security research programme.

References

[1] R. McNaughton, "Unate truth functions," *IRE Trans. on Elec. Comp.*, vol. 10, no. 1, pp. 1–6, 1961.

[2] J. Mohnke, P. Molitor, and S. Malik, "Limits of using signatures for permutation independent Boolean comparison," *Formal Methods in System Design*, vol. 21, no. 2, pp. 167–191, 2002.

[3] H. Katebi and I. L. Markov, "Large-scale Boolean matching," in *Design, Automation and Test in Europe*, 2010, pp. 771–776.

[4] C. A. J. van Eijk and J. A. G. Jess, "Exploiting functional dependencies in finite state machine verification," in *European Design and Test Conference*, 1996, pp. 9–14.

[5] J. R. Jiang and R. K. Brayton, "Functional dependency for verification reduction," in *Computer Aided Verification*, 2004, pp. 268–280.

[6] M. Soeken, B. Sterin, R. Drechsler, and R. K. Brayton, "Reverse engineering with simulation graphs," in *Formal Methods in Computer-Aided Design*, 2015, pp. 152–159.

[7] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.

[8] B. T. Murray and J. P. Hayes, "Test propagation through modules and circuits," in *Int'l Test Conf.*, 1991, pp. 748–757.

[9] M. Marhöfer, "An approach to modular test generation based on the transparency of modules," in *IEEE CompEuro 87*, 1987, pp. 403–406.

[10] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[11] C. Lee, J. R. Jiang, C. Huang, and A. Mishchenko, "Scalable exploration of functional dependency by interpolation and incremental SAT solving," in *Int'l Conf. on Computer-Aided Design*, 2007, pp. 227–233.

[12] J. R. Jiang, C. Lee, A. Mishchenko, and C. Huang, "To SAT or not to SAT: scalable exploration of functional dependency," *IEEE Trans. Computers*, vol. 59, no. 4, pp. 457–467, 2010.

[13] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Eén, "Improvements to combinational equivalence checking," in *Int'l Conf. on Computer-Aided Design*, 2006, pp. 836–843.

[14] M. Sauer, B. Becker, and I. Polian, "PHAETON: A SAT-based framework for timing-aware path sensitization," *IEEE Trans. Computers*, vol. PP, no. 99, pp. 1–1, 2015.

[15] M. Sauer, S. Reimer, I. Polian, T. Schubert, and B. Becker, "Provably optimal test cube generation using quantified Boolean formula solving," in *ASP Design Automation Conf.*, 2013, pp. 533–539.

[16] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 4–15, 1992.

[17] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185.

[18] S. A. Cook, "The complexity of theorem-proving procedures," in *Symposium on Theory of Computing*, 1971, pp. 151–158.

[19] L. A. Levin, "Universal sequential search problems," *Problems of Information Transmission*, vol. 9, no. 3, 1973.

[20] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[21] G. Tseytin, "On the complexity of derivation in propositional calculus," *Studies in Constructive Mathematics and Mathematical Logic*, 1968.

[22] N. Eén, A. Mishchenko, and N. Sörensson, "Applying logic synthesis for speeding up SAT," in *Theory and Applications of Satisfiability Testing*, 2007, pp. 272–286.

[23] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 15, no. 9, pp. 1167–1176, 1996.

[24] T. Schubert and S. Reimer, "antom," in *https://projects.informatik.uni-freiburg.de/projects/antom*, 2013.

[25] F. Corno, M. Reorda, and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results," *Design & Test of Computers, IEEE*, vol. 17, no. 3, pp. 44–53, 2000.

[26] C. Albrecht, "IWLS 2005 benchmarks," in *International Workshop for Logic Synthesis (IWLS): http://www. iwls. org*, 2005.

Identifying Transparent Logic in Gate-Level Circuits

Yu-Yun Dai[1], Robert K. Brayton[2]
[1,2]Department of EECS, University of California, Berkeley, U.S.A.
{[1]yunmeow, [2]brayton} @berkeley.edu

ABSTRACT

Many reasons exist for high-level information to be unavailable for a design. Identifying high-level constructs, e.g. control paths and words, from gate-level circuits, can assist verification, equivalence checking, reverse engineering, etc.. Word-level identification can be done by structural methods, but we focus on functional approaches because they only depend on dependencies between signals of a circuit. We introduce *transparent logic*, based on *functional isomorphism*, and provide algorithms to recognize control signals, data paths, internal words, and boundaries between different types of logic. Experiments show that the proposed algorithms can re-assemble words effectively from unstructured or synthesized circuits.

1. INTRODUCTION

In hardware, control logic regulates the data flow and dictates circuit functionalities. Logic can be classified as that where data is simply moved from one part of a circuit to another part without modifying it. Such logic is referred to as *transparent*. Another category transforms data by some word-level operator, e.g. a bit-vector operator defined in Verilog. A third category, control, determines which data is moved and when, or which operation is applied and when. Efficient recognition of such logic can benefit circuit verification, e.g. [4] as a guide to abstraction. To identify word-level operators in a gate-level circuit, it is crucial to find words and locate the boundaries (inputs/outputs) of arithmetic operators [10].

The basic example of transparent logic is a multiplexer (MUX) structure, which selects from several data signals and forwards it unaltered towards the outputs. Identification of MUXes can be performed over gate-level circuits very quickly using structural matching, but can be unreliable, especially if synthesis has been applied.

In this paper, we focus on functional approaches which do not depend on the actual gate-level structure of the circuit. These can augment structure methods and provide a much more reliable technique as we show in the experiments.

In general, functional methods, which rely only on functional dependencies, have been used to augment structural approaches. Examples are,

- Li and Subramanyan et. al. [6, 11] identified internal words based on *bitslice aggregation* (functional approach) and *shapehashing* (structural approach.) The candidate words found were used as boundaries of operators for further recognizing.
- Li et. al. [6, 7] identified functional operators in gate-level circuits, based on an existing library of blocks. Word-level information at the primary inputs was assumed available, but in many applications this information is not known.
- Sterin et. al. [10] extracted word-level operators functionally, given a library of operators and a slice of logic containing inputs and outputs of such operators. Word-level information was not required nor was the possible location and ordering of the inputs and outputs of an operator.

We present methods to identify *functional transparent logic*. This is inherited from *functional isomorphism*. Using this, we propose an algorithm to identify words, word-level operator boundaries, and control logic in gate-level circuits and apply this to a variety of test cases. Once operator boundaries are located (roughly), techniques like those in Sterin et. al. [10] can be used to identify the more precise location of the operators as well as their functionalities.

The paper is organized as follows. Section 2 introduces *functional* isomorphism. In Section 3, we describe the definition and propagation of *transparent* logic. Proposed algorithms for identifying transparent logic are given in Section 4. Experimental results are shown in Section 5, while Section 6 concludes this paper.

2. PRELIMINARY

2.1 Npn Isomorphism

Two graphs, $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, are *isomorphic*, if there exists a bijective mapping, $\mathbb{M}_{12}$: $V_1 \rightarrow V_2$, such that any two vertices u and v are adjacent in G_1, if and only if $\mathbb{M}_{12}(u)$ and $\mathbb{M}_{12}(v)$ are adjacent in G_2 [2]. Two circuits, C_1 and C_2, are isomorphic to each other if their logic gates and connections form two isomorphic graphs, while any gate g of C_1 and the mapped gate $\mathbb{M}_{12}(g)$ in C_2 are the same type. The relation between C_1 and C_2 is called *structural isomorphism*, which has been applied to reverse engineering [5].

In contrast, *functional isomorphism* is a relation between two signals in a circuit. A signal f in a circuit, supported by a set of other signals, S^f, is a Boolean function of these inputs: $f : \mathbf{B}^{|S_f|} \rightarrow \mathbf{B}$, for $\mathbf{B} = \{0, 1\}$.

In the following sections, for a Boolean variable x_i with its polarity p_i, $(x_i)^{p_i}$ represents the function: $p_i = 0 \rightarrow (x_i)^{p_i} \equiv x_i$ and $p_i = 1 \rightarrow (x_i)^{p_i} \equiv inv(x_i)$.

Definition 1: A pair of Boolean functions $f(x_1, \ldots x_n)$ and $g(y_1, \ldots, y_n)$ are *npn isomorphic*[1], if there exists a permutation $\pi \in \mathbf{S}_n$ and polarities p_{out} and $\{p_1, \ldots, p_n\} \in \mathbf{B}^n$ such that

$$f(x_1, \ldots, x_n) = g^{p_{out}}(x^{p_1}_{\pi(1)}, \ldots, x^{p_n}_{\pi(n)}) \qquad (1)$$

i.e., g can be made equivalent to f by selectively negating inputs, permuting inputs, and negating the output. The implied isomorphic mapping between the supports of g and

[1]or Negation-Permutation-Negation (NPN) equivalent

f is $\{y_i, x_{\pi(i)}^{p_i}\}$ and p_i is said to be the relative polarity between inputs y_i and $x_{\pi(i)}$.

A set of signals in a circuit, in which every pair is functionally npn isomorphic is called an *npn isomorphism class*.

2.2 Composition of Npn Isomorphism

Although improved methods for computing npn equivalence can be found in Soekin et. al. [9], this still can be time-consuming. To reduce this effort, the following theorem shows that finding and proving npn isomorphism classes can be done incrementally by working on smaller logic blocks and composing proved classes to obtain larger ones. Larger classes help extend paths of transparency (discussed in Section 3) in a circuit and to more reliably find transparency boundaries, and hence the input/output boundaries of word-level operators.

Definition 2: (*polar consistency*) Let $(f(s), g(t))$ be a pair of npn isomorphic functions with sets of supports $s = \{s_i\}$ and $t = \{t_j\}$, respectively. Suppose each pair of mapped input supports, $s_i \leftrightarrow t_j$, are npn isomorphic functions, i.e. $s_i(x)$ is npn isomorphic to $t_j(y)$. Let p_{out}^{ij} be the relative output polarity between $s_i(x)$ and $t_j(y)$, and p_{ij} be the relative input polarity between inputs s_i and t_j in the npn isomorphism between $f(s)$ and $g(t)$. The compositions $f(s(x))$ and $g(t(y))$ are *polar consistent*, if $p_{out}^{i\pi(i)} = p_{i\pi(i)}$, where π is the permutation in the isomorphism mapping of $(f(s), g(t))$.

Theorem 1: The compositions of $(f(s(x)), g(t(y)))$ are polar consistent if and only if $f(s(x))$ and $g(t(y))$ are npn isomorphic.

3. TRANSPARENT LOGIC

We propose *transparent logic*, as a method for identifying control paths and input/output boundaries of arithmetic operators.

3.1 Transparent Words

Intuitively, a *transparent word* is a set of signals, $\{w^k\}$, with supports, $\{S^k\}$, where under some evaluation of $\cap^k S^k$ (*common control*), $\{w^k\}$ is equivalent to a subset (*data-word*) of $\cup^k S^k$. In other words, the control evaluation makes the word transparent from some input data-word.

For example, outputs of a set of 2-to-1 multiplexers (MUX) controlled by the same selector signal s,

$$C[n-1:0] = s?A[n-1:0] : B[n-1:0], \qquad (2)$$

comprises a transparent word C, where $\forall j \in [1, n]$, $(C[j] = sA[j] + s'B[j])$. For this case, word C is transparent from word A or word B, depending on the value assigned to s.

Definition 3: Functions $F = \{w_k | k \in [1, m]\}$ of an npn isomorphism class comprise an m-bit *transparent word*, if:

1. Each function $w_k : \mathbf{B}^{S_k} \rightarrow \mathbf{B}$, has support $S_k = (Control, Data_k)$, (i.e. *Control* is the set of common signals), and each bit of *Control* is isomorphically mapped into itself.

2. The following formula, (where m_c is a minterm of *Control*) is *True*,

$$(\exists_{m_c} \forall_k \exists_{d^i \in Data_k} \exists_{p^i} (w_k(m_c, Data_k) \equiv (d^i)^{p^i})) \qquad (3)$$

(Symbol $\equiv$ to denote functional equivalence between Boolean functions.)

3. For any $(w_x, w_y) \in F$, the associated isomorphic support mapping $\mathbb{M}_{xy}$, satisfies $\mathbb{M}_{xy}(Data_x) = Data_y$.

Thus a transparent word is conditionally (by m_c) combinationally equivalent to a subset of its supports ($\{d^i\}$) or negations. Based on the above definition, the vector of conditionally equivalent data supports, $D^i = \{d_1^i, ..., d_m^i\}$ is called an *input word*.

Given a transparent word, $W = \{w_k | k \in [1, m]\}$, with the corresponding support partitions $\{(Control, Data_k) | k \in [1, m]\}$, the entire support set of W can be partitioned into **Control** and $\mathbf{Data}^W$, where $\mathbf{Data}^W = \bigcup^k Data_k$. The definition of transparent words can be restated as follows:

Definition 4: A transparent word W is a set of npn isomorphism functions supported by control supports **Control** and data supports $\mathbf{Data}^W = \bigcup^i D^i$, such that the following formula is *True*:

$$\forall_{D^i \in \mathbf{Data}^W} \exists_{m_c} \exists_{P^i} (W(m_c, \mathbf{Data}^W) \equiv (D^i)^{P^i}), \qquad (4)$$

where P^i set of polarity bits for D^i.

Although, for a D^i, there could be multiple assignments of $Control$ satisfying the Formula (4), the assignments of $m_c \in Control$ for different D^is are disjoint.

To illustrate these definitions, consider Equation(1): for each $C[j]$, the support set $\{s, A[j], B[j]\}$ can be partitioned into $Data_j = \{A[j], B[j]\}$ and $Control = \{s\}$, such that $(s = 1) \Rightarrow (C[j] = A[j])$ and $(s = 0) \Rightarrow (C[j] = B[j])$. Hence a common (control) assignment applied to all bits of the transparent word, makes all bits transparent from the corresponding supports simultaneously. The supports of C can be partitioned into $\mathbf{Data}^C$ and **Control**, where $\mathbf{Data}^C \equiv \{A[n-1:0], B[n-1, 0]\}$, and **Control** $\equiv \{s\}$.

Since negations of some bits of transparent words might occur during synthesis, it seems reasonable to consider the logic still as "transparent". Consider the example: $C[j] = sA[j] + s'B[j]$. The negation of a bit $C[j]$ can be done by inserting invertors after $A[j]$ and $B[j]$:

$$\begin{aligned} inv(C[j]) &= inv(sA[j] + s'B[j]) \\ &= s\ inv(A[j]) + s' inv(B[j]). \end{aligned} \qquad (5)$$

Thus some bits of C appear in negated form, but C (with some phase changes) can still be considered transparent from A and B. Note that the assignments to the control bits is unchanged by the negations on the data bits.

In Section 3.2 and 3.3, we will need to resolve cases where only some bits of a transparent word are negated. However, for composing transparencies to find larger ones, it is required that the polarities of the inputs and outputs are consistent. The theorem below addresses this issue.

Theorem 2: Given a transparent word W, the negation of each bit w_k can be done by negating each data support of w_k, without changing any control assignment.

3.2 Composition of Transparency

Similar to the composition of npn isomorphism, transparent functions are frequently created by composing smaller transparent blocks.

For example, in Figure 1, word C is transparent from A and B under the control of s_1, while word E is transparent from C and D under the control of s_2. Thus $(s_1 = 1, s_2 = 1) \rightarrow E \equiv A$, while $(s_1 = 0, s_2 = 1) \rightarrow E \equiv B$ i.e. transparency of E from A and B is obtained by composing of smaller transparent blocks.

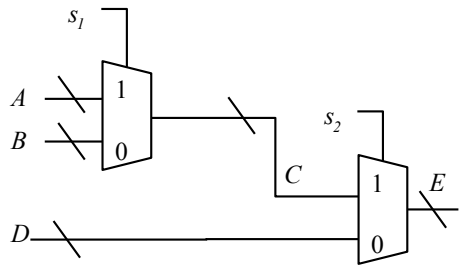


Figure 1: A transparent word can be implemented by composing smaller transparent words.

Definition 5: Let $\mathbf{W} = \{W^k(X)|k = [1,n]\}$ be a set of n m-bit transparent words, and let $Y = \{y_j|j = [1,m]\}$ be another transparent word with support $Data^Y = \mathbf{W} \bigcup V$ and $Control^Y$. Suppose each input word of Y is exactly one transparent word in $\mathbf{W}$ or one word in V. The compositions,

$$Z = \{z_j\} = \{y_j(\mathbf{W}(X), V, Control^Y)\} \quad (6)$$

form a *compound word*, and are denoted as $Z = Y \circ \mathbf{W}$.

Theorem 3: Assume Y is a transparent word and $\mathbf{W}$ is a set of transparent words. Let $\{\alpha_i^k\}$ be the set of minterms of $Control_k$, which enable W^k to be transparent from an input word $x_i^k \in Data_k$, and $\{\beta^k\}$ be the set of minterms of $Control^Y$ for $(Y \equiv W^k)$. Using the notation:

$$Control^Z = Control^Y \bigcup [\cup^k Control_k],$$
$$Data^Z = V \bigcup [\cup^k Data_k],$$

a compound word, $Z \equiv Y \circ \mathbf{W}$ is a transparent word controlled by $Control^Z$ if

$$\forall_k \forall_i (\{\hat{\alpha_i^k}\} \cap \{\hat{\beta^k}\} \neq \emptyset) \quad (7)$$

is *True*, where $\{\hat{\alpha_i^k}\}$ and $\{\hat{\beta^k}\}$ are $\{\alpha_i^k\}$ and $\{\beta^k\}$ extended to cubes of $Control^Z$, respectively.

Proof

1. Based on theorems 1 and 2, Z can be an npn isomorphism class by flipping the polarities of W^kwhenever its output polarity is not consistent with the input polarities of y^k.
2. Because Y is a transparent word, for each input word in V, there must exist an assignment of $Control^Y$ to enables the transparency from V.
3. Conditions satisfying Formula 7 imply that for each input word x_i^k of W^k, there exists an assignment of $Control^Z$ such that W^k is transparent from x_i^k, Y is transparent from W^k, and Y is transparent from x_i^k. Therefore, $Z \equiv Y \circ \mathbf{W}$ is a transparent word with $(Control^Z, Data^Z)$ as control and data supports.

3.3 Propagation of Transparency

Figure 2 illustrates how a deeper transparency can be composed from non transparent sections of logic. C is transparent from A when $s_1 = 1$, and D is transparent from B when $s_2 = 1$, but the logic block from C and D to the word E is not transparent (there is no common control support for each bit of E). However, E is transparent from A when $(s_1 = 1, s_2 = 0)$, while $(s_1 = 0, s_2 = 1)$ makes E transparent from B.

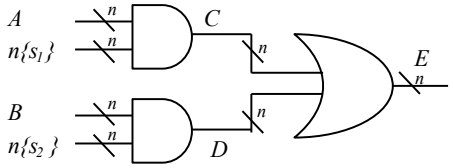


Figure 2: A transparent word can be decomposed into smaller transparent words and npn isomorphism classes.

We refer to this as the *propagation of transparency* when a transparent function block is composed of non-transparent sections. The conditions when this can happen are stated in the following.

Definition 6: Let $\mathbf{W}$ be a set of n m-bit transparent words, and let $Y(\mathbf{W}) = \{y_j(\mathbf{W})\}$ be an npn isomorphism class. Suppose each y_j is supported by exactly one bit of each W^k, and the isomorphically mapped supports of y_j are always from the same word of $\mathbf{W}$. The compositions, $Z = \{z_j\} = \{y_j(\mathbf{W}(X))\}$ are said to form a *proceeding word*.

Theorem 4: Assume Y and $\mathbf{W}$ are as in Definition 6 and the supports of W^k are $supp(W^k) = (Control_k, Data_k)$. Let $\{\alpha_i^k\}$ be the set of minterms of $Control_k$ which cause $(W^k \equiv x_i^k)$, and $\{\beta^k\}$ be minterms of $\cup^k Control_k$ which cause $(Y \equiv W^k)$. Using $Control^Z = \cup^k Control_k$, and $Data^Z = \cup^k Data_k$, a proceeding word, $Z \equiv Y \circ \mathbf{W}$, is a transparent word controlled by $Control^Z$ if

$$\forall_k \forall_i (\{\hat{\alpha}_i^k\} \cap \{\beta^k\} \neq \emptyset), \quad (8)$$

where $\{\hat{\alpha}_i^k\}$ refers to $\{\alpha_i^k\}$ extended to cubes of $Control^Z$.

Proof

1. Similar to the proof of Theorem 3, Z can be an npn isomorphism class by flipping the polarities of W^k if needed.
2. For each input word x_i^k in $Data^Z$, Formula 8 implies that there exists an assignment of $Control^Z$, such that $W^k \equiv x_i^k$, $Y \equiv W^k$, and thus, $Y \equiv x_i^k$, implying $Z \equiv Y \circ \mathbf{W}$ is a transparent word with $(Control^Z, Data^Z)$ as control and data supports.

For example, the circuit in Figure 2 satisfies the propagation of transparency, where the connections from C and E (W^k to y_j) meet the requirements of Definition 6. For C transparent to E, the assignment $(s_1 = 1, s_2 = 0)$ is compatible with the assignment for A transparent to C, and disjoint from the assignment $(s_1 = 0, s_2 = 1)$ which makes D transparent to E.

4. TRANSPARENCY IDENTIFICATION

The functional approach proposed for transparency identification relies only on dependencies among signals. It can be used to complement a structural approach, leading to a method that is more efficient with more reliable results.

In general, we want to identify transparent logic anywhere it occurs in the circuit - from inputs to internal words (forward), from internal words to outputs (backward), and between internal words. Two problems are formulated and solved in this paper: forward and backward transparency.

Forward Transparency: Given a boundary of input supports, e.g. primary inputs, find (1) input words, (2) transparent words in the fanout cones, (3) the corresponding partition of supports for each proved word and (4) partial assignments of **Control** for enabling the transparencies.

Backward Transparency: Given an output boundary of transparent words (candidates), e.g. primary outputs, find (1) transparent words on the boundary, (2) the boundary of supports in the fanin cones, (3) the corresponding partition of the supports and (4) partial assignments of **Control** for enabling the transparencies.

In both cases, we want to find the longest path for each transparency We provide details and discuss the common sub-problems for both forward and backward transparency algorithms and the challenges of this approach.

4.1 Proving Transparency of Sub-circuits

Both forward and backward transparency problems have common sub-problems: given a) the boundaries of supports b) targets (word candidates), and c) a set of proved words, find (1) transparent words on the target boundary, (2) support partitions and (3) partial assignments of **Control** for input words on the support boundary. Figure 3 outlines the steps for solving this sub-problem.

```
Algorithm: Find Transparent Words
Input:
  Circuit //one combinational gate-level circuit.
  Boundary = (Supports, Targets)
//two sets of signals in the circuit.
  ProvedWords // a set of proved words
Output:
  NewWords
  //a set of newly proved input and output words

01. NewWords = ∅
02. (NpnIsoClasses, SuppMaps) = NpnIsoClasses(Circuit,
...     Boundary)
03. For each c in NpnIsoClasses
04.   Words = splitClass( c, SuppMaps, ProvedWords)
05.   For each w in Words
06.     (Control, Data) = classifySupports(w, SuppMaps,
...       ProvedWords )
07.     If getAssignments(w, Control, Data, ProvedWords)
08.       addWords(NewWords, w, Data)
09. Return NewWords
```

Figure 3: Algorithm for proving transparency of a sub-circuit.

Given a sub-circuit specified by **Circuit** and **Boundary**, the function *NpnIsoClasses(...)* at Line 2 returns the npn isomorphism classes among signals in **Targets** with their isomorphic support mappings. Each npn isomorphism class can contain more than one word, so *splitClass(...)* works on those isomorphic signals, analyzes their supports and splits those signals into different candidate words. It decomposes each npn isomorphism class into sub-classes, such that all signals are driven by the same controls and each contain a bit from each of a common group of words.

Then *classifySupports(...)* at line 6 partitions supports into *Data* and *Control*, and groups the bits of each identified input word together (details are given in Section 4.2).

For each candidate word, the function *getAssignments(...)* at Line 7 formulates QBF problems as Equation (5) and apply a QBF solver to those problems to find assignments for control supports. Once the QBF solver proves the candidate word is indeed a transparent word, this word is kept in **NewWords** along with the corresponding assignments and input words.

For *compound words*, there exists a complete transparent block inside the sub-circuit, and all essential supports can be found on the boundary. Hence the QBF problems can be formulated only with the signals on the support boundary. In contrast, for *proceeding words*, the circuit defined by **Boundary** might only contain npn isomorphism classes, but no whole transparent words. Hence the QBF problems should consider the supports of input words on the boundary.

To illustrate how this is done for a *proceeding word*, consider Figure 2. The non-isomorphism class (the logic block from C and D to E) inside the sub-circuit is not a transparent block. For this case, the function *getAssignments(...)* uses the control supports of the proved words (s_1 and s_2 for C and D, respectively) to find the feasible assignments for the transparent condition.

4.2 Support Classification

Classifying supports into control and data types is critical for proving transparent words. According to Definition 2, all bits of a transparent word must each have support bits from the same set of input words, while mapped control supports must be identical, i.e. the same set of control bits appear in every support of the transparent word bits and they are mapped isomorphically to themselves.

Finding an ideal mapping cannot fully rely on the mappings found for npn functional isomorphism, because a legal mapping for two signals in the same npn isomorphism class could be illegal for transparent logic (see Figure 4).

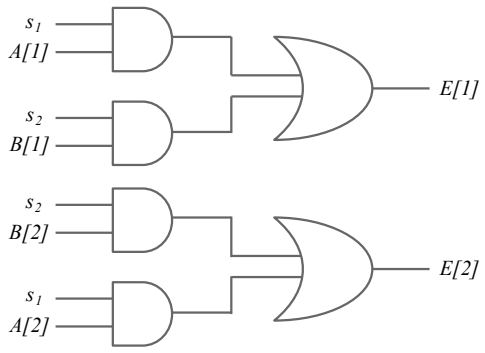


Figure 4: An example with inappropriate support mapping between signals in the same isormorphism class.

The circuit in Figure 4 is extracted from Figure 2, but with some input permutations. Given the support boundary, (A[1], A[2], B[1], B[2], s_1, s_2) and the targets, (E[1], E[2]), E[1] and E[2] are classified into the same npn isomorphism class. Due to symmetry of some Boolean functions, the support mapping is not unique.

For example, the support mapping, $(s_1, A[1], s_2, B[1])_{E[1]} \rightarrow (s_2, B[2], s_1, A[2])_{E[2]}$ satisfies the definition of isomorphism, i.e. when the same values are applied to the mapped inputs (i.e. s_1 and s_2), E[1] and E[2] should evaluate to the same value. However, this requirement blocks out any legal assignments of (s_1, s_2) for E being transparent from A and B; it conflicts with the control condi-

tion for transparency: $(s_1 = 1, s_2 = 0) \Rightarrow (E \equiv A)$ and $(s_1 = 0, s_2 = 1) \Rightarrow (E \equiv B)$.

For this case, the mapping of control supports can be revised easily, because the definition of transparent words requires that each control support is isomorphically mapped to itself. Moreover, if A and B have been proved already as words, the mapping issue can be resolved by forcing bits of the same word to be mapped to the same word.

Unfortunately, if input words have not been proved yet, it is necessary to enumerate all legal isomorphism mappings in order to determine correct mappings. This issue can be moderated by decomposing an input circuit into several sub-circuits properly. First, input words can be proved before being considered as supports of others. Also, as the sizes of the input circuits decrease, the number of feasible support mappings for npn isomorphism is reduced.

In the experiments for the results shown in Section 5, support classification only finds isomorphism mappings and uses the above heuristics, but does not enumerate all feasible mappings. Thus some transparent words could have been missed in the experiments.

4.3 Forward Transparency Algorithm

With the assistance of the algorithm outlined in Figure 3 , the algorithm for forward transparency is shown in Figure 5.

```
Algorithm: Forward Transparency
Input:
  Circuit
Output:
  ProvedWords

01. ProvedWords= ∅
02. InternalControls = findHighFanoutSignals(Circuit)
03. decideForwardOrder(InternalControls)
04. For each control in InternalControls
05.   Fanouts = collectImmediateFanouts(control)
06.   Supports = collectSupports(Fanouts, ProvedWords)
07.   Targets = collectFanouts(control, Supports)
08.   NewWords = findTransparentWords(Circuit,
...       Supports, Targets, ProvedWords)
09.   ProvedWords ⋃ = NewWords
10. ProceedingWords = propagateWords(ProvedWords)
11. ProvedWords ⋃ = ProceedingWords
12. Return ProvedWords
```

Figure 5: Algorithm for identifying transparent words in a gate-level circuit - forward transparency case

The function *findHighFanoutSignals(...)* in Line 2 uses the fact that all bits of a transparent word should be controlled by the same condition, so it collects all signals with more than 3 immediate fanouts as candidate controls. Also, *decideForwardOrder(...)*, sorts control signals in ascending topological order according to its greatest topologically immediate fanout.

Lines 5 to 9, are repeated for all control signals. In Line 5, the function *collectImmediateFanouts(...)* collects all immediate fanouts of the current *control*, while *collectSupports(...)* decides the boundary of supports for identifying npn isomorphism classes. In practice, this function backtracks from all signals in **Fanouts** and collects the nearest bits in **ProvedWords** or nearest primary inputs. The support boundary is a subset of the union of proved words and primary inputs. Then the function *collectFanouts(...)* collects all signals only driven by signals in **Supports** and saves them as the target boundary.

Based on the support and target boundaries, the function *findTransparentWords(...)* applies the algorithm of Figure 3 to find transparent words in the specified subcircuits. Those new words are added to **ProvedWords** in Line 9 and utilized in later procedures.

Finally, *propagateWords(...)* addresses the possible existence of proceeding words (Section 3.3) to enlarge transparent logic blocks. This searches for npn isomorphism classes in the fanout cones of the deepest transparent words and reuses the proved assignments based on Theorem 4. Those new words will be added to **ProvedWords**. This procedure is repeated until no new words are found.

Example: Consider Figure 1 as input **Circuit**. In the beginning, s_1 and s_2 are recognized as high-fanout signals. The greatest immediate fanout of s_2 is greater than that of s_1, hence signals in **InternalControl** are ordered as $s_1 \rightarrow s_2$. Based on controls in **InternalControl**, all bits of C are considered as isomorphism targets first, and C is proved as a transparent word from A and B, under the control of s_1. Here A, B, and C are added to **ProvedWords**. Then all immediate fanouts of s_2, bits of E, are identified as an isomorphism class, with the support boundary (s_2, C, D). The function *getAssignments(...)* finds assignments for (s_1, s_2), such that E is conditionally equivalent to C and D, and hence to A and B. The proved words, A, B, C, D, E, are returned as **ProvedWords**.

4.4 Backward Transparency Algorithm

The main differences between the forward and backward algorithms are (1) how to define support boundaries, (2) the challenge of classifying supports for sub-circuits and 3) all proved bits in the forward algorithm are transparent from certain primary inputs, but in the backward algorithm, only parts of the proved bits are transparent to the primary outputs. The proposed algorithm for backward transparency is shown in Figure 6.

```
Algorithm: Backward Transparency
Input:
  Circuit
Output:
  ProvedWords, TransBits

01. ProvedWords= ∅
02. InternalControls = findHighFanoutSignals(Circuit)
03. decideBackwardOrder(InternalControls)
04. For each control in InternalControls
05.   Fanouts = collectImmediateFanouts(control)
06.   Supports = collectFanins(Fanouts)
07.   Targets = collectFanouts(Supports)
08.   NewWords = findTransparentWords(Circuit,
...       Supports, Targets, ProvedWords)
09.   ProvedWords ⋃ = NewWords
10. ProceedingWords = propagateWords(ProvedWords)
11. ProvedWords ⋃ = ProceedingWords
12. TransBits = finalizeBackward(ProvedWords)
13. Return (ProvedWords, TransBits)
```

Figure 6: Algorithm for identifying transparent words in a gate-level circuit.

As in the forward algorithm, all high-fanout signals are collected and sorted by their greatest immediate fanouts, but sorted in descending topological order.

The immediate fanouts of each control signal are saved temporarily as **Fanouts**, while their immediate fanins form the support boundary. This is different than in *collectSupports(...)* in Figure 5, which traverses circuits until reaching primary inputs or proved words. At Line 7, all signals driven only by signals in **Support** are regarded as the target boundary. When *collectFanouts(...)* traverses forward from signals in **Support**, it might reach some signals which have been proved as input words for other transparent words. Those signals are included in **Targets**, while the traversal stops moving forward from them.

Lines 4 to 9 prove a set of disconnected transparent words, because they are proved in reverse topological order. Since there are no proved words in the support boundaries, when *findTransparentWords(...)* works on each subcircuit, the functions *splitClass(...)* and *classifySupports(...)* in Figure 3 only can use data dependencies and control signals inside the sub-circuit. In this case, the issues mentioned in Section 4.2 would arise.

Like the forward algorithm, *propagateWords(...)* enlarges the proved transparent blocks. Finally, the function *finalizeBackward(...)* at Line 12 composes the whole set of proved words into larger transparent logic blocks and saves those bits which are transparent to primary outputs as **TransBits**.

Example: Consider Figure 1 using the backward algorithm. Signals in **InternalControls** are ordered as $s_2 \rightarrow s_1$. Under control s_2, E is proved first as a transparent word from C and D. Then C is proved to be conditionally equivalent to A and B under control s_1. The function *finalizeBackward(...)* verifies the composition of assignments for (s_1, s_2), and hence the two proved words are concatenated into a larger transparent function block.

Example: When the backward algorithm is applied to the circuit in Figure 2, it can find $(s_1 = 1) \rightarrow (C \equiv A)$ and $(s_2 = 1) \rightarrow (D \equiv B)$ first. Then *propagateWords(...)* will do the same thing as in the forward algorithm. In the end, the function *finalizeBackward(...)* finds all proved words can be transparent to primary outputs.

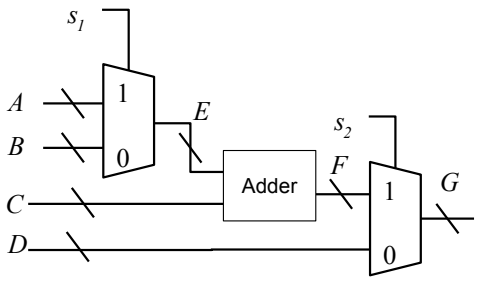


Figure 7: An example with disjointed transparent logic blocks.

Before *finalizeBackward(...)*, the proved words can be disjoint, and are not guaranteed to be reachable from the primary outputs. In the circuit in Figure 7, the backward algorithm finds two disconnected transparent words: $G = s_2?F : D$ and $E = s_1?A : B$, which are separated by an adder. *propagateWords(...)* will confirm that there is no way to connect the two words with transparent paths. Hence *finalizeBackward(...)* only returns one transparent word $G = s_2?F : D$, which is reachable from primary outputs.

5. EXPERIMENTAL RESULTS

The proposed algorithms were implemented in ABC [3]. All experiments were performed on a 16-core 2.60GHz Intel(R) Xeon(R) CPU with no time limit. All cases were processed as AIGs and analyzed for forward and backward transparency. Sequential circuits were converted into combinational designs by replacing flip-flops inputs and outputs with primary outputs and inputs respectively.

As a reference for the functional approach, we implemented a pure structural approach: 1) structural matching is used to locate all 2-to-1 MUXes in the AIGs, 2) signals with the same control are grouped into one word, and these connected words are collected into larger transparent blocks, and 3) words which are reachable from primary inputs (outputs) for forward (backward) transparency are returned.

We wanted to compare the efficiency and effectiveness of the structural algorithms versus our functional algorithms applied to highly-transparent cases. To select these, the functional forward algorithm was applied to all 230 cases of the single-output track in the Hardware Model Checking Competition 2014 [1]. For each case, some POs were proved conditionally equivalent to certain primary inputs. We computed the proportion of those POs to all POs and ran experiments on the top 10 cases with the highest percentages of transparent POs. Among the 230 cases, there are 20 cases with more than 50% transparent POs, while another 38 cases have more than 25% transparent POs. Table 1 shows the statistics of the selected cases after they were converted to combinational circuits. The last column of Table 1 indicates the percentages of transparent POs over all POs. The *6sxxx* cases are industrial problems from IBM and the *beem* examples come from different applications areas like protocols, planning, scheduling, communication, or puzzles.

Table 1: Statistics of the selected benchmarks from HWMCC'14 [1].

Case Name	PI #	PO #	AND #	Trans. PO %
6s195.aig	1344	1258	8046	87.1
beemfrogs1b1.aig	323	159	8493	86.0
6s171.aig	1357	1263	8074	84.6
beemloyd3b1.aig	237	118	3970	82.1
6s282b01.aig	1977	1934	10264	81.2
6s384rb024.aig	22367	14953	47933	79.0
6s206rb103.aig	37847	28644	103375	71.4
6s302rb09.aig	36962	27777	100571	70.3
6s348b53.aig	15797	15561	89567	70.1
beemldelec4b1.aig	2559	1215	34252	67.5

5.1 Experiments for Structural Approach

Table 2 shows the results for the structural approach coded for the experiments. Column 2 indicates the total number of signals, i.e. AIG nodes or primary inputs, that were classified as belonging to words. Column 3 lists the total number of structural MUXes recognized. Columns 4-7 (labeled *Forward Transparency*) give the statistics of the transparencies found using the forward algorithm. Column 4 shows the total number of transparent words reachable from primary inputs (including input words); Column 5 lists the number of AIGs plus inputs covered by all the transparent logic blocks found; Column 6 gives the (minimum, maximum) widths (the number of MUXes grouped together as a word) of found words, and Column 7 shows the (minimum, maximum) depths of transparent words on

boundaries. The depth of each word is the total number of AIG nodes between itself and the primary inputs, where one MUX is counted as depth 2.

Columns 8-11 (labeled *Backward Transparency*) show similar statistics for the backward case: Column 8 lists the total number of transparent words reachable from the primary outputs. Depth here is the number of AIGs between internal transparent words on boundaries and the primary outputs. The last column shows the combined run-time for the forward and backward structural approaches. Here we only identify 2-to-1 MUXes and MUXes with negation on outputs or inputs. We omit counting words with less than 4 bits

Discussion of Structural Results

Table 2 shows that most benchmarks contain wide transparent words. Some contain very deep transparent paths as well as some with only 1 or 2 levels of MUXes. The runtimes show that this approach is very efficient as expected.

Although these cases have high percentages of transparent POs, for some cases the structural approach cannot find any transparent words. Many MUXes are recognized but there are several reasons why the structural approach misses many transparent words:

1. structural matching only considers standard 2-to-1 multiplexers, while there are other types of transparent functions.
2. Many of the identified MUXes are controlled by different selection signals, and thus lead to words of less than 4 bits, which are excluded in the analysis.
3. Transparent words are required to be reachable from primary inputs through fully transparent paths. If a transparent word originates from the output word of an arithmetic operator (e.g. word G and F in Figure 7), it would not be reported, yet many MUXes would be involved in such a transparency.

Although quite fast, this approach itself is not enough for finding many of the whole transparent blocks that exist in these benchmarks as shown in the following section which shown the total words found by the functional approaches.

5.2 Experiments for Functional Approach

In the experiments for the functional approach, the function *NpnIsoClasses(...)* is created as an ABC command, *&iso* [8]. Table 3 shows the experimental results for both the forward and backward cases. The columns are similar Table 2.

Comparing Functional and Structural

We observe for the forward case:

1. The functional approach finds many more and wider transparent words in all cases. For example, in the last case, beemldelec4b1.aig, the functional approach finds many transparent words, while the structural approach finds none. One reason is the functional approach addresses all isomorphism classes and tries to prove transparency for them, while the structural method only considers 2-to-1 MUX cases.
2. The functional approach finds much more logic involved in transparent paths than the structural approach, on average about 2x more signals.
3. The functional approach finds deeper words than the structural method. As mentioned, the current structural approach cannot find any depth-1 transparent logic (a MUX has depth 2).
4. The runtime of the functional approach increases with circuit size, while the structural approach is much faster. The functional approach requires many circuit traversals and manipulations. In contrast, the structural approach only goes through an entire circuit once to collect MUXes, and then works on groups of MUXes as candidate words.

For backward transparency, we observe the following:

1. For the three cases where the structural approach cannot find any words, the functional approach finds several depth-1 transparent words.
2. Although the functional backward algorithm spends a lot of time proving internal transparent words, many are unreachable from the primary outputs.
3. Unlike the forward case, the functional backward approach seems to miss some words. The reason is that the present implementation of *NpnIsoClasses(...)* (*&iso* in ABC) only reports isomorphism classes in which the polarities of one word must be all the same. The backward case is more likely to have mixed polarity npn isomorphism classes, because some bits of input words for transparent logic blocks might have been synthesized with their supports, which are parts of operators. Therefore, some bits are excluded from our current implementation and this can have a more significant effect on the backward case.

Comparing the forward and backward functional approaches, we observe:

1. Due to the difficulties of support classification (mentioned in Section 4.2), the backward algorithm misses some transparent paths proved by the forward algorithm. We need to understand this better and improve the backard strategies.
2. For large cases, the backward algorithm takes much more time than the forward one because:
   - For some backward cases, there is no proved word on the support boundary that would be useful for classifying supports. Then it takes more time to prove transparent words. In contrast, the forward algorithm collects and proves candidate words in a topological order and the proved words can be used to prove candidates in their fanout cones.
   - The forward algorithm only checks each signal once, but the backward one needs to revisit some target signals to consider different support boundaries. The reason is, if the support boundary defined earlier cannot be used to prove transparent words (due to the support classification issue), those signals will be re-visited when a different support boundary is proposed.

Table 3 shows that the proposed algorithm finds a significant number of transparent words in the selected cases, even though the isomorphism algorithm used in these experiments is an early version largely limited by the structure of the circuit [8]. This may cause larger words to

Table 2: Experimental results of the structural approach on ten selected cases from HWMCC'14 [1]. N/A here means no transparent word found.

Case Name	Total Sig. #	MUX #	Forward Transparency				Backward Transparency				Runtime(s)
			Words#	Sig.#	Widths	Depths	Words #	Sig. #	Widths	Depths	
6s195.aig	9390	2357	18	4552	8, 512	2, 12	287	5005	4, 72	2, 6	0.056
beemfrogs1b1.aig	8816	2016	33	520	8, 8	32, 32	0	0	N/A	N/A	0.056
6s171.aig	9431	2362	11	4413	16, 512	2, 12	289	5034	4, 73	2, 6	0.058
beemloyd3b1.aig	4207	985	23	352	8, 8	22, 22	0	0	N/A	N/A	0.049
6s282b01.aig	12241	2472	65	1803	6, 66	2, 6	31	3643	6, 1031	2, 4	0.056
6s384rb024.aig	70300	14492	889	21278	4, 64	2, 4	992	45174	4, 4250	2, 8	0.122
6s206rb103.aig	141222	28684	2083	56295	4, 193	2, 4	2456	96678	4, 1398	2, 8	0.238
6s302rb09.aig	137533	27818	2054	55274	4, 191	2, 4	2421	94243	4, 1061	2, 8	0.235
6s348b53.aig	105364	28775	484	28850	4, 262	2, 12	304	58875	4, 734	2, 16	0.458
beemldelec4b1.aig	36811	8458	0	0	N/A	N/A	0	0	N/A	N/A	0.591

Table 3: Experimental results of the functional approach on ten selected cases from HWMCC'14 [1]

Case Name	Forward Transparency					Backward Transparency					
	Words#	Sig.#	Widths	Depths	Runtime(s)	Words#	Trans. W #	Trans. S #	Widths	Depths	Runtime(s)
6s195.aig	1003	7921	4, 512	3, 14	3.106	352	242	5025	4, 72	2, 6	4.130
beemfrogs1b1.aig	622	4153	6, 8	9, 40	3.753	464	19	151	7, 8	1, 1	2.689
6s171.aig	490	8036	4, 512	3, 14	3.471	407	190	4641	4, 67	1, 7	3.100
beemloyd3b1.aig	89	712	8, 8	2, 26	1.815	231	12	96	8, 8	1, 1	1.390
6s282b01.aig	268	6996	4, 966	2, 10	2.623	205	20	2744	4, 999	3, 4	2.989
6s384rb024.aig	2587	48311	4, 2308	3, 7	66.533	2295	1336	40036	4, 3338	1, 9	99.395
6s206rb103.aig	6552	105493	4, 1042	3, 12	297.935	6076	3472	79660	4, 305	1, 8	471.489
6s302rb09.aig	6583	103437	4, 872	3, 12	295.97	6071	3493	78696	4, 296	1, 8	451.814
6s348b53.aig	1879	58874	4, 367	3, 18	72.454	2644	715	57521	4, 598	1, 11	238.205
beemldelec4b1.aig	574	2688	4, 8	2, 3	27.368	2183	43	223	4, 21	1, 1	17.976

be broken down into smaller sub-words. A better isomorphism method, using npn isomorphism, has been developed but not yet integrated into the present implementation [9]. Also, run-time performance should improve as the implementation matures, e.g. run-time might be improved by skipping some intermediate levels of words, but then intermediate words might be missed. Also structural and functional methods can be inter-mixed. For real applications, the particular final usage of the found words will dictate a suitable balance between performance and the number of proved words.

6. CONCLUSIONS

This paper presented algorithms for finding transparent logic, which can be used to highlight word-level information in gate-level circuits. A functional approach was proposed to identify transparent logic in combinational circuits. Experimental results demonstrated that the proposed algorithms can be very effective in extracting words as well as some control logic.

Future work will include:

- new npn isomorphism functional methods will be integrated into the current approach,
- experiments will be conducted on examples from unrolled sequential circuits to try deeper circuits,
- a composite method combining both structural and functional approaches will be developed to achieve efficiency and effectiveness at the same time,
- an algorithm will be developed for finding internal transparency blocks, not just forward or backward transparencies, and appropriate examples will be created to measure its effectiveness,
- the method will be used to find all word -> operator and operator -> word boundaries in a gate-level design and integrated with reverse engineering functional approaches that can identify the operators.

The final goal is a fully functional approach to the reverse engineering of gate-level designs.

7. ACKNOWLEDGEMENTS

This work is supported in part by SRC contract 2265.001 and by the NSA via the TRUST grant. We also thank industrial sponsors of BVSRC, Altera, Atrenta, Cadence, Calypto, IBM, Intel, Mentor Graphics, Microsemi, Synopsys, and Verific for their continued support.

8. REFERENCES

[1] *Hardware Model Checking Competition 2014*. http://fmv.jku.at/hwmcc14cav/.

[2] S. Awodey. *Category theory*. Clarendon Press Oxford University Press, Oxford Oxford New York, 2006.

[3] R. Brayton and A. Mishchenko. Abc: An academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40. Springer, 2010.

[4] Y.-Y. Dai, K.-Y. Khoo, and R. Brayton. Sequential equivalence checking of clock-gated circuits. In *Design Automation Conference*. ACM, 2015.

[5] M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, (3):72–80, 1999.

[6] W. Li. Formal methods for reverse engineering gate-level netlists. Master's thesis, 2013.

[7] W. Li, G. Adria, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. Seshia. Wordrev: Finding word-level structures in a sea of bit-level gates. In *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, pages 67–74. IEEE, 2013.

[8] A. Mishchenko, N. Een, R. Brayton, M. Case, P. Chauhan, and N. Sharma. A semi-canonical form for sequential aigs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 797–802. EDA Consortium, 2013.

[9] M. Soeken, A. Mishchenko, A. Petkovska, B. Sterin, P. Ienne, R. K. Brayton, and G. D. Micheli. Heuristic npn classification for large functions using aigs and lexsat.

[10] M. Soeken, B. Sterin, R. Drechsler, and R. Brayton. Reverse engineering with simulation graphs. In *Formal Methods in Computer-Aided Design*, 2015.

[11] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Y. Tan, A. Tiwari, N. Shankar, S. Seshia, and S. Malik. Reverse engineering digital circuits using structural and functional analyses. *Emerging Topics in Computing, IEEE Transactions on*, 2(1):63–80, 2014.

Uninterpreted Function Abstraction and Refinement for Word-level Model Checking

Yen-Sheng Ho[1], Alan Mishchenko[1], Robert Brayton[1]
[1]Department of EECS, University of California, Berkeley, CA, USA
{ysho, alanmi, brayton}@eecs.berkeley.edu

***Abstract*—Methods for word-level model checking based on purely bit-level techniques have difficulties with heavy arithmetic logic. Word-level and SMT approaches often are limited by relying on (incomplete) bounded model checking. UFAR, a hybrid word- and bit-level approach, addresses these issues, taking advantage of modern bit-level sequential techniques while heavy arithmetic logic is addressed by word-level abstraction and the use of uninterpreted function (UF) constraints. The methods and efficiency improvements developed for UFAR enabled it to prove 2422 of a set of 2492 industrial sequential model checking problems within a 1-hour limit, while a bit-level model checker *super_prove* completed only 2115 of these within the same limit.**

I. Introduction

Model checking (MC) [20], [5], [22] on a Register-Transfer-Level (RTL) word-level netlist is a necessary verification task for applications involving sequential synthesis. In this, an RTL netlist is synthesized into another through retiming, clock-gating, pipelining etc., and MC is required for proving the correctness of the result. These problems are challenging if hard arithmetic operators such as multipliers, adders, and variable shifters are involved, and correspondences between flip flops are not known.

Previous methods in this domain fall into roughly three categories. One directly "bit-blasts" the problem and then uses bit-level sequential verification engines such as IC3/PDR [7], [16], interpolation [18], or BDDs [13]. Another translates the problems into SMT formulas (if possible) and then directly employs SMT solvers such as Boolector [11], Z3 [15], or CVC4 [3]. A third [2], [1], [8] applies *term-level abstraction*, replacing arithmetic operators with uninterpreted functions (UF), and then solving with SMT solvers. However, bit-level techniques are problematic when verifying circuits with heavy arithmetic logic. Most SMT-based approaches rely on (incomplete) bounded model checking (BMC) [6] or induction [21] and may not be applicable.

UFAR (Uninterpreted Function Abstraction and Refinement), is a hybrid word- and bit-level solver, which moderates the above issues. It takes advantage of modern sequential techniques such as PDR and BMC at the bit-level, while heavy word-level logic is tackled by abstraction and the use of uninterpreted function (UF) constraints.

Such techniques are not new, even at the word level. Conventional UF abstraction [2], [1], [8] methods implicitly enforce *all* possible UF constraints among the same functions. This becomes inefficient when the number of similar functions is large. Keys to UFAR's efficiency are how simulations and minimized counterexamples are used to refine abstractions, how constraints are added and removed lazily, which pairs of operators are constrained, and how UF constraints are applied between operators of the same type but with different bit widths. All this requires efficiently iterating between word-level Verilog and AIG representations as refinements are done. These techniques enable UFAR to prove problems containing hundreds of heavy word-level operators.

We prove that UFAR is a sound and complete framework for word-level counterexample guided abstraction and refinement (CEGAR) [14]. It starts with the extreme abstraction with all "problematic" word-level operators (e.g., multipliers, adders, etc) removed (i.e. operator outputs are replaced by unconstrained pseudo primary inputs). This is then bit-blasted and given to a sound and complete bit-level model checker. If a counterexample is returned, UFAR first simulates it on the original netlist to check if it is *real*. If so, UFAR terminates and reports it. Otherwise, the *spurious* counterexample is used to refine the current abstraction. Refinement is done in this context by 1) adding UF constraints between some pairs of chosen compatible operators, and 2) restoring one or more of the removed operators.

We experiment on 2492 industrial benchmarks for sequential RTL (word-level) model checking and show how different refinement methods and heuristics are complementary, each solving more problems in less time, and leading to a final algorithm which solves all but 70 of the benchmarks within a one hour time limit. We show detailed results on 19 examples having ranges of 4-475 multipliers, 21092-302277 AIG nodes, and 358-4785 flip-flops.

This paper first presents background material and formal settings in Section II. The UFAR algorithm is presented in Section III. Several optimization techniques for the algorithm are given in Section IV. Section V gives some details about the UFAR framework, including word-level representation and bit-blasting this into an AIG. Experimental results on an extensive set of industrial problems are presented in Section VI, comparing the effectiveness of the two optimizations and the overall UFAR algorithm. Some conclusions and future work are discussed in Section VII.

II. Bit-Vectors and UF Constraints

In the context of Verilog and its bit-vector operators, we need to be precise about applying UF constraints between pairs of operators. A UF constraint states that for two same-type functions, if their inputs are equal then their outputs are equal. Unfortunately, this is not at all straight-forward when bit-vector operators are involved. Incorrect application of UF constraints can lead to an unsound procedure on the one hand or to a too restrictive application on the other. In this section, we discuss bit-vector operators, define what it means to be the same function, state when and how to make UF constraints valid between two same-type operators, and prove the soundness of the derived methods.

A. The MC problem

We assume that the input RTL design is in *structural Verilog*. In structural Verilog, there are bit-vector (BV) signals including primary inputs (PIs), primary outputs (POs), flip flops (FFs), and internal signals. Flip flops have reset values as initial states. A bit-vector signal s can be either *signed* or *unsigned*, denoted by $signed(s)$. The *bit-width* of s is denoted by $bw(s)$. A design is modeled as a finite state machine (FSM).

Definition 1. A design in structural Verilog is a tuple $M = (I, O, S, S_0, T)$ where I is the set of inputs, O is the set of outputs, S is the set of state variables, S_0 is the set of initial states, and T is the set of (deterministic) transition relations where $T \subseteq I \times S \times S$. If $(i, s, s') \in T$, then there exists a transition from s to s'under i.

The input format is assumed to be *mitered* as a single FSM and a single output, out, representing the property to be checked. If the problem is to prove equivalence between two designs, a miter is created by merging all PIs and merging corresponding mapped FFs (if any). The output out is a Boolean signal, which is the OR of the pairwise XORs of the corresponding outputs of the two designs. Thus it is 1 if the two designs are different. Similarly for property checking, the output is a monitor which signals 1 if the property fails. In terms of linear temporal logic (LTL), the MC problem is formulated as $M \models \mathbf{G}\neg out$, meaning the miter M should never excite the signal out if the property holds.

B. Basics of word-level operators

We focus on abstracting problematic *word-level operators* in a design. The subset of operators considered are all word-level *binary* operators, such as $+, -, *, /, \%, <<, >>, <<<, >>>$. In Verilog, an operator is instantiated by a *structural statement* which only states the function type of the operator and the connection between signals[1]. An operator is modeled as a labeled node with a single output, up to two inputs, and its label of function type.

[1]Without loss of generality, we assume that each statement contains only 1 binary operator. Statements like `x = (a+b)*c` can always be rewritten to `y = a+b` and `x = y*c`.

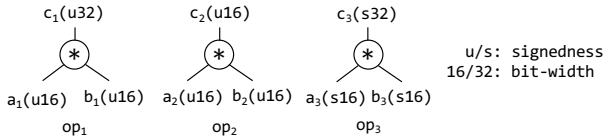


Fig. 1: Three multipliers with different functions.

Definition 2. An operator *op* is a tuple $op = (o, i_1, i_2, t)$ where o is the output signal, i_1 and i_2 are the input signals, and t is the label of function type.

For example, the Verilog statement, `c = a * b`, is modeled as $op = (c, a, b, *)$. Note that the inputs are *ordered* as specified in the Verilog statement. Note also that $*$ is a "function-type" and not a function, since the actual function that would be instantiated would depend on the properties of the signals to which its inputs and output are connected. The necessity of this important distinction will be clarified in the next section.

C. Functions of word-level operators

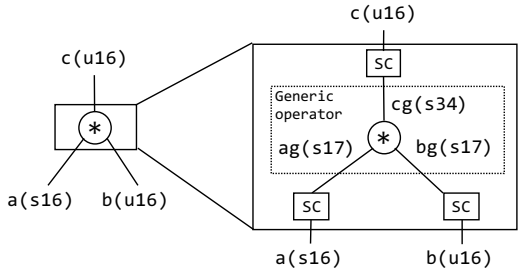
In Verilog, the actual *function* associated with an operator is determined by the bit-widths and signedness of its inputs, output, and function-type. Operators with the same function type do not necessarily have the same function; a function-type represents a set of functions. For example, the three multipliers in Figure 1 all represent different functions. Operators op_1 and op_3 are different since op_1 is *unsigned* multiplication while op_3 is *signed* multiplication. Operator op_2 is different because its output is only 16 bits.

To be precise in what follows, we need to explicitly model what a Verilog front end does when it bit-blasts a Verilog RTL design into a bit-level circuit. For this we need *generic operators* and *signal convertors*.

Definition 3 (Generic operator)**.** A generic operator is a bit-vector operator that agrees with the *integer function* of its function-type. That is, the bit-vector output, when evaluated as an integer, is consistent with the result using the integer function. It has the following properties.

- All of its inputs and output are signed.
- It is a pure arithmetic function parametrized with specified widths for its inputs and output.
- When a generic operation is implemented, its input widths should be compatible with the widths of the signals connected to them.
- The output width should be exactly large enough so as to not impose any restriction on the operation (such as truncation due to overflow).

In order to create signals used or produced by an instantiated generic function, they must be converted from unsigned to signed signals or vice versa. They also need to be converted by truncation, sign extension, or zero extension. We model this by emulating what Verilog does in its assignment operator (=) and concatenation operator ({}), called signal convertors.



(a) The relationship between a multiplier and its generic version. SC denotes signal converters.

```
wire signed [15:0] a;                    wire signed [15:0] a;
wire        [15:0] b;          Expose    wire        [15:0] b;
wire        [15:0] c = a * b;  ------->  wire        [15:0] c;
                                         wire signed [16:0] ag = {1'b0, a};
                                         wire signed [16:0] bg = {1'b0, b};
                                         wire signed [33:0] cg = ag * bg;
                                         assign             c = cg;
```

(b) A piece of Verilog code for exposing the generic operator.

Fig. 2: An example showing how generic operators are modeled and exposed.

The benefits of explicitly exposing generic operators include 1) it agrees with the arithmetic of not only *bit-vectors* but *integers*, and 2) it unifies unsigned and signed operators. For example, the original bit-vector multiplier, `c = a * b` in Figure 2, does *not* agree with integer arithmetic, meaning that if signals (`a`, `b`, `c`) are evaluated as integers (A, B, C), then they do *not* necessarily satisfy the relation $C = A \times B$ (here $\times$ is integer multiplication). To expose the generic multiplier in Figure 2, first we observe that the original one is unsigned[2] multiplication, so the original inputs are converted to signed generic inputs with leading zeros inserted. Then a generic output of 34 bits is created to prevent overflow. Finally the generic output is converted to the unsigned original one with some upper bits truncated. The generic multiplier, `cg = ag * bg`, agrees with the integer arithmetic by construction. This way, the original multiplier is represented by its generic version and some signal convertors on the inputs and output.

With generic operators, all same-type generic operators (e.g., multipliers) are considered to have the same function since they all agree with their integer functions (e.g. integer multiplication). This is important for uninterpreted function abstraction since uninterpreted function constraints are valid only for same-function classes.

D. Uninterpreted function constraints

The theory of uninterpreted functions (UF) states that given any *function* F with its input X, and any two instances of the same function, (x_1, f_1) and (x_2, f_2), then the Property (1) holds, stating that if the inputs are equal then the two outputs must be equal.

$$(x_1 = x_2) \Rightarrow (f_1 = f_2) \tag{1}$$

[2]In Verilog, an operation is unsigned if at least one input is unsigned.

This is called a UF constraint which is simply a relation implied by any pair of the same two functions.

For Verilog, we need to be more precise about "same function" and "equal inputs". By f and g being the same function we mean that f and g are instantiations of the same generic function-type. By two signals being equal, we will mean that they are signed and bit-wise equal *after* extension. Then Property (1) holds with these modifications. Thus,
a UF constraint is valid between any pair of same function-type generic operators (even if they have different bit widths).

Definition 4. Two signals, s_1 and s_2, are said to be *equal in Verilog* if the corresponding statement, `s1 == s2`, is evaluated to 1 in Verilog.

The precise Verilog semantics for comparing two signals is as follows. It does either zero- or sign-extension for the signal with the smaller bit-width depending on their signedness. If both signals are signed, then it does sign-extension. Otherwise, zero-extension is applied. Two signals are equal if they are bit-wise equal after extension.

Definition 5. For two same function-type generic operators, $op_1 = (o_1, i_{11}, i_{12}, t)$ and $op_2 = (o_2, i_{21}, i_{22}, t)$, the UF constraint, denoted as c, is either Constraint (2) or (3).

- If t is asymmetric:

$$c = (i_{11}{=}{=}i_{21}) \wedge (i_{12}{=}{=}i_{22}) \Rightarrow (o_1{=}{=}o_2) \tag{2}$$

- If t is symmetric:

$$c = \Big((i_{11}{=}{=}i_{21}) \wedge (i_{12}{=}{=}i_{22}) \Rightarrow (o_1{=}{=}o_2)\Big) \bigvee \Big((i_{11}{=}{=}i_{22}) \wedge (i_{12}{=}{=}i_{21}) \Rightarrow (o_1{=}{=}o_2)\Big) \tag{3}$$

We only apply UF constraints between generic instances of same function-type operators. The constraints are created as signals first and then treated as invariant constraints to the model checking problem (see Section III-C). Thus, abstractions are created by 1) using UF constraints and 2) replacing their outputs by new primary inputs (the generic operators are "black-boxed").

Definition 6. A generic instance is said to be *black-boxed* if its output is replaced by a fresh primary input consistent with the generic's output.

Thus the new primary input is signed and has the same width as the instance output being replaced. Note that a UF constraint may be added even though the two operators involved are both white-boxed. This can still be effective as it provides a relation between operators which may not be easy to derive using bit-level operations.

III. UFAR

In this section, the abstraction-refinement algorithm, UFAR, for solving word level model checking problems is described.

Algorithm 1 UFAR

Input: M — φM: the input miter
Input: $\mathcal{S}$ — $\varphi \mathcal{S}$: the set of problematic operators
Output: status $\in$ { SAT, UNSAT }
1: $\mathcal{B} \leftarrow \mathcal{S}$ — $\varphi \mathcal{B}$: the set of black-box operators
2: $\mathcal{P} \leftarrow \emptyset$ — $\varphi \mathcal{P}$: the set of UF constraints
3: $M \leftarrow$ EXPOSINGFUNCTIONS(M, $\mathcal{S}$)
4: **while true do**
5: $A \leftarrow$ CREATEABSTRACTION(M, $\mathcal{P}$, $\mathcal{B}$)
6: status, cex $\leftarrow$ MODELCHECKING(A)
7: **if** status = SAT **then**
8: **if** ISREALCEX(M, cex) **then**
9: **return** SAT
10: **else**
11: $\Delta\mathcal{P} \leftarrow$ REFINEUFPAIRS(A, $\mathcal{S}$, cex)
12: **if** $\Delta\mathcal{P} \neq \emptyset$ **then**
13: $\mathcal{P} \leftarrow \mathcal{P} \cup \Delta\mathcal{P}$
14: **continue**
15: **else**
16: $\Delta\mathcal{B} \leftarrow$ REFINEBLACK(M, $\mathcal{P}$, $\mathcal{B}$, cex)
17: $\mathcal{B} \leftarrow \mathcal{B} \backslash \Delta\mathcal{B}$
18: **else**
19: **return** UNSAT

A. The algorithm

Algorithm 1 provides a high level view of UFAR. It takes two inputs; one is a miter M in word-level structural Verilog and the other is $\mathcal{S}$, the set of *problematic* operators that we want to abstract (multipliers in most cases). UFAR will return SAT if a true counterexample is found; otherwise, it concludes that $M \models \mathbf{G}\neg out$ and returns UNSAT. We will prove that UFAR is a sound and complete algorithm in Section III-G.

There are two internal state sets in UFAR. The first is $\mathcal{B}$, the set of *black* operators that will be black-boxed in the abstraction. The second is $\mathcal{P}$, the set of operator pairs whose UF constraints will be added to the abstraction. Initially $\mathcal{B}$ = $\mathcal{S}$, thereby black-boxing all problematic operators, and $\mathcal{P}$ = $\emptyset$.

Algorithm 1 begins with the procedure of exposing generic operators (see Section III-B). It then operates in an abstraction-refinement loop (lines 4–19). Each iteration begins by creating an abstraction based on the current states of the algorithm, which will be discussed in Section III-C. The abstraction is then bit-blasted and solved by state-of-the-art bit-level engines concurrently (see Section III-D). If the solver returns UNSAT, the property is proven and UFAR terminates (line 19). Otherwise a counterexample to the abstraction (cex) exists. If cex is also a counterexample to the original miter, then the property is falsified and UFAR terminates (lines 8–9). Otherwise cex is spurious and UFAR analyzes it to refine the abstraction (lines 11–17).

Refinement is achieved in two phases. UFAR first tries to find new UF pairs that will block cex (see Section III-E). If such are found, UFAR adds them to $\mathcal{P}$ and starts a new iteration (lines 12–14). Otherwise, the second phase is started, where cex is analyzed to determine a set of critical operators ($\Delta\mathcal{B}$) that can block cex (see Section III-F). For the next iteration, UFAR will remove operators in $\Delta\mathcal{B}$ from $\mathcal{B}$ (lines 16–17) and hence these will be *white-boxed*.

B. Exposing generic operators

To expose the generic version of an operator, we modify the Verilog by inserting signed- or zero-extended signal convertors to ensure that it becomes signed and that the bit-width of its output is large enough. The procedure for each operator $op = (o, i_1, i_2, t)$ in the problematic set $\mathcal{S}$ is summarized below.

1) If one of the inputs is *unsigned*, then create zero-extension-by-1 signed signal convertors for both inputs. Denote two generic inputs as a_1 and a_2.
2) Create the generic operator $op_2 = (o_2, a_1, a_2, t)$ where o_2 is signed and has a large enough bit-width.
3) Replace the original output o with the statement $\circ$ = $\circ 2$. Note that this step creates the generic operator op_2, eliminating the original one op.

C. Creating abstractions

An abstraction (A) is created from the original miter (M), using $\mathcal{P}$ and $\mathcal{B}$, the two current states of Algorithm 1. CREATEABSTRACTION operates in two steps:

1) For each pair $p = (op_1, op_2)$ in $\mathcal{P}$, construct a Boolean signal c as defined in UF Constraints (2) or (3). Signal c = 1 implies that a UF constraint is active in M between op_1 and op_2. Signal c is then treated as an invariant constraint.
2) For each operator $op = (o, i_1, i_2, t)$ in $\mathcal{B}$, replace its output o with a fresh primary input ppi with the same signedness and bit-width, i.e. black-box it.

Note that an operator can be in a pair of $\mathcal{P}$ but not $\mathcal{B}$. For example, one benchmark contained a group of 3 multipliers where 2 UF constraints were used between them, but only one of the 3 was needed to be white-boxed for the final proof. Note also that $\mathcal{P}$ and $\mathcal{B}$ are monotone.

We claim that the model A is an abstraction of M.

Lemma 1. *Let N denote the model created after Step 1 (adding UF constraints) in* CREATEABSTRACTION. *N and M satisfy: ($\neg out$ denotes the property)*

$$N \models \boldsymbol{G}\neg out \Leftrightarrow M \models \boldsymbol{G}\neg out.$$

Proof. Consider any constraint signal c. We have $M \models \mathbf{G}c$ since the model M satisfies any valid UF constraint. Thus,

$$\begin{aligned} M \models \mathbf{G}\neg out &\Leftrightarrow M \models \mathbf{G}\neg out \wedge \mathbf{G}c \\ &\Leftrightarrow M, \mathbf{G}c \models \mathbf{G}\neg out \Leftrightarrow N \models \mathbf{G}\neg out \end{aligned}$$

□

Theorem 1. *The model A created by* CREATEABSTRACTION *is an abstraction of the miter M.*

Proof. From Lemma 1, N generated by Step 1 is equisatisfiable to the miter M. In Step 2, it creates the model A by replacing some internal signals in N with fresh primary inputs, which is a known procedure for producing an abstraction. □

D. Model checking using concurrency

To verify the current abstraction at the bit level, we could use a single engine like PDR since it is efficient, sound, and complete. Also, this procedure should be parallelized to take advantage of different engines. Running a BMC engine in parallel with PDR usually finds counterexamples to the current abstraction more efficiently and thus very effective in improving the algorithm. Also, various versions (based on different implementations and parameters) of PDR and BMC complement each other.

E. Refining UF pairs

This is the first phase of refinement. Given a (spurious) counterexample cex to the abstraction, we want to find new UF pairs $\Delta\mathcal{P}$ among operators in $\mathcal{S}$ that can block cex during the next iteration. REFINEUFPAIRS operates in two steps:

1) Simulate cex on the abstraction A to derive an assignment function $\kappa : S \times \mathbb{N} \rightarrow \mathbb{Z}$ that maps every signal in A at each time frame to a concrete value.
2) Identify pairs that violate UF constraints and add them to $\Delta\mathcal{P}$. For each time frame t and every pair of operators $(op_1, op_2) : op_1, op_2 \in \mathcal{S}, op_1 \neq op_2$, if the values of the inputs are equal but the outputs are different (Formula 4), then add (op_1, op_2) to $\Delta\mathcal{P}$. Note that we consider both input orders for symmetric operators although this is not shown in Formula 4 for simplicity.

$$(\kappa(i_{11}, t) = \kappa(i_{21}, t) \wedge \kappa(i_{12}, t) = \kappa(i_{22}, t)) \bigwedge \kappa(o_1, t) \neq \kappa(o_2, t) \tag{4}$$

Next, we discuss an upper bound for the size of $\mathcal{P}$.

Theorem 2. *The size of $\mathcal{P}$ in Algorithm 1 is bounded by $|\mathcal{S}|(|\mathcal{S}| - 1)$.*

Proof. Consider the worst case where the operators in $\mathcal{S}$ are all symmetric, then there are $\binom{|\mathcal{S}|}{2}$ pairs of operators with 2 possible permutations of binary inputs. Hence the number of pairs in the algorithm cannot exceed $|\mathcal{S}|(|\mathcal{S}| - 1)$. □

F. Refining black operators

In the second phase of refinement, we want to identify a subset of operators $\Delta\mathcal{B}$ in $\mathcal{B}$ such that if $\Delta\mathcal{B}$ is removed from $\mathcal{B}$, cex will be blocked for the next iteration. We call the procedure of removing elements from $\mathcal{B}$ *white-boxing* and the operators in $\mathcal{S} \setminus \mathcal{B}$ *white boxes*.

A straightforward way of identifying $\Delta\mathcal{B}$ is to simulate cex on the abstraction A and collect those operators in $\mathcal{B}$ that have input-output values inconsistent with their white-box values. However, this approach often finds an overly large $\Delta\mathcal{B}$, resulting in an unnecessarily large abstraction in the next round. Hence, we propose a *proof-based* approach that often obtains a much smaller $\Delta\mathcal{B}$.

The main idea is that if cex is spurious, then the BMC Formula (5) is UNSAT. Here the function $\nu(i, t)$ denotes the assignment of signal i at time t derived from cex being simulated on the original miter M, k is the depth of cex, and out is the miter signal.

$$I_M(\nu(s,0)) \wedge \bigwedge_{t=0}^{k-1} T_M(\nu(i,t), \nu(s,t), \nu(s,t+1)) \wedge \bigvee_{t=0}^{k} out(\nu(i,t), \nu(s,t)) \tag{5}$$

Next, multiplexers are introduced to select between the concrete version (white-box) and the abstracted version (black-box) of an operator. If assumptions are made such that all the concrete ones are selected initially, then the resulting BMC formula would still be UNSAT and a modern SAT solver like MiniSat [17] will return a subset of the assumptions that is sufficient for UNSAT. This is a variation of finding an *unsat core* and the subset returned is our candidate for $\Delta\mathcal{B}$.

The procedure REFINEBLACK operates in five steps.

1) For each pair in $\mathcal{P}$, construct a UF constraint signal and treat it as an invariant constraint on M.
2) For each operator $op = (o, i_1, i_2, t)$ in $\mathcal{B}$, introduce two fresh primary inputs, sel and ppi, where sel is a Boolean signal and ppi a bit-vector signal which is consistent with the output o_{gen} of the associated generic operator. Replace o_{gen} with $o^{\simeq}_{gen} = ITE(sel, o_{gen}, ppi)$ where ITE is the *if-then-else* operator. Depending on the value of sel, either the concrete operator (o_{gen}) or the abstracted one (ppi) flows to the new output $o^{\simeq}_{gen}$.
3) Denote the model created in Step 2 by N and unroll it with the values of cex plugged in, and keep sel and ppi as the remaining primary inputs. The cex values plugged in are initial states and PIs at each time frame, denoted by $\psi(s, 0)$ and $\psi(i, t)$ respectively.
4) Solve the BMC query (6), which is guaranteed to be UNSAT. Note that ψ is the assignment function of cex, X_t is the set of sel input signals at time t, PPI_t is the set of ppi input signals at time t, and x_{tn} is the sel signal for the n-th operator at time t. By propagating $x_{tn} = 1$ for all t and n, the query (6) is reduced to (5) by construction ($sel = 1$ means that the concrete version is chosen).

$$I_N(\psi(s,0)) \wedge \bigwedge_{t=0}^{k-1} T_N(\psi(i,t), X_t, PPI_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} out(\psi(i,t), X_t, PPI_t, s_t) \wedge \bigwedge_{t=0}^{k} \bigwedge_{n=0}^{|X_t|} x_{tn} \tag{6}$$

5) Derive a subset ΔX of X using the assumption interface of a modern SAT solver, and determine $\Delta\mathcal{B}$ from ΔX.

Theorem 3. *The set $\Delta\mathcal{B}$ found by* REFINEBLACK *is not empty ($|\Delta\mathcal{B}| > 0$).*

Proof. $|\Delta\mathcal{B}| = 0$ would mean that the formula (6) is SAT, which contradicts with the fact that cex is spurious. □

G. Analysis of the algorithm

Theorem 4. *Algorithm 1 is sound and complete.*

Proof. (sketch) Algorithm 1 is sound because it returns UNSAT only if the model A satisfies $\mathbf{G}\neg out$, which implies $M \models \mathbf{G}\neg out$ from Theorem 1. As for the completeness, the algorithm returns SAT only if a counterexample is real (line 8–9). Convergence follows because for each iteration (line 4–19), the following statements are true.

- $\mathcal{B}$ and $\mathcal{P}$ are monotone. Either $\mathcal{P}$ becomes strictly bigger (line 12–13) or $\mathcal{B}$ becomes strictly smaller (Theorem 3).
- $|\mathcal{P}|$ is upper bounded by $|\mathcal{S}|(|\mathcal{S}|-1)$ (Theorem 2) and $|\mathcal{B}|$ is lower bounded by 0 (empty set of black boxes).

Therefore the iteration must terminate implying that a definitive answer must have been found. □

IV. Optimization

In this section, we introduce two optimizations (counterexample minimization, and random simulation), each of which improves the basic version of UFAR, Algorithm 1.

A. Minimizing counterexamples

A counterexample can be minimized [19] in the sense that some inputs can be assigned as X (don't care), but the counterexample is still valid after ternary simulation. This way, the number of concrete assignments is minimized.

The main advantage of using minimized counterexamples is that Procedure RefineUFPairs in Algorithm 1 can return potentially fewer, but higher-quality pairs of constraints. This is done by modifying the condition (Formula 4) for identifying and adding a UF constraint, where we check if the inputs are equal and the outputs are different under concrete assignments. With minimized counterexamples, Xs might appear on the outputs of black-box operators (unconstrained pseudo primary inputs). We strengthen the condition by considering only *incompatible* outputs with X assignments. Two assignments are said to be *incompatible* if they have opposite values at some bit position, and *compatible* if they do not. For example, the assignments `XX01` and `X000` are incompatible while `10XX` and `100X` are compatible. With this strengthening, pairs that satisfy Formula 4 under concrete assignments might violate the new condition since their outputs become compatible after the minimization. For example, consider two operators with concrete assignments (o, in_1, in_2), (`0011`, `01`, `10`) and (`0101`, `01`, `10`), which satisfies Formula 4. After the minimization, if the assignments become (`0XX1`, `01`, `10`) and (`XXX1`, `01`, `10`), then the pair will not be added as UF constraints since it violates the strengthened condition with compatible outputs. Thus, it is likely that fewer constraints are added. Also, the constraints we drop are lower-quality in the sense that if they are added, then UFAR will still get similar counterexamples.

B. Performing random simulation

UFAR in Algorithm 1 only finds and applies UF constraints when a counterexample (CEX) is found. However, the CEX returned by a verification engine may not be unique. If UFAR were to get a different CEX, then it might find and apply a different set of UF constraints. This inherent randomness of counterexamples could cause UFAR to take a path where more white boxes are needed for a proof. Thus, random simulation is applied on the original miter to find candidates for "good" UF constraints. The idea is that if a UF constraint is useful for the final proof, then the corresponding pair of operators must be related in some way. This means that for some execution traces they would have identical input assignments.

The procedure of random simulation operates in 2 steps.

1) Determine the parameters: the number of patterns and the number of time frames. Run random simulation on the original miter.
2) For each time frame and for each pair of same function-type generic operators, count the number of times identical input patterns occur.

A threshold is then set for determining what are good candidates of UF constraints (a pair is considered good if its count is above the threshold). A threshold should be chosen carefully since there is a trade-off between the number and the quality of constraints; a lower threshold increases the chances of getting higher-quality UF constraints (in the sense that it is more difficult to find them with counterexamples), but a lower threshold also leads to a larger number of constraints.

V. The UFAR framework

UFAR involves an iteration of abstraction and refinement between two types of representations,

1) AIGs (bit-level circuit), and
2) an internal netlist format called WLC (word-level circuit), a new development in ABC [10] to represent word-level designs.

This capability includes 1) a very fast Verilog based bit-blaster, using Verilog semantics of the WLC box operators, to translate into an AIG, and 2) a duplication-based method to create different WLC netlists at the word level. These developments are critical in making UFAR efficient, to the extent that UFAR run-time is dominated by the SAT solving in the bit-level model checker.

A. Bit-blasting WLC with Verilog semantics

The framework starts with reading in a structural Verilog miter representing the model checking problem. This is translated into a WLC netlist (WLCm) using ABC's structural Verilog parser. Next, the generic operators of all designated "problematic" operators are exposed by creating a new WLC netlist, denoted as WLCg. More details of creating a new WLC netlist are described in the next subsection. It is important to note that WLCg needs to be created only once during the entire flow and represents the fully concretized problem. This is bit-blasted into an AIG, denoted by AIGg to be used later.

The next step is to create a WLC netlist, WLCa, for the current abstraction using WLCg and the state sets $\mathcal{P}$ and $\mathcal{B}$. WLCa is bit-blasted into an AIG, denoted as AIGa. During this, Verilog semantics are used to faithfully interpret the box operators of WLC netlists.

Typically the model checker, applied to AIGa, returns a counterexample which is simulated on AIGg to see if it is spurious. If so, the counterexample is first minimized, using AIGg as reference. This is analyzed to decide the state changes to $\mathcal{P}$ and $\mathcal{B}$, which will be used to block this counterexample. These are implemented by creating a new WLCa from WLCg and the current state sets. Then the next iteration proceeds.

B. Creating abstractions WLCa

In the iteration in the previous section, the next abstraction is constructed as a WLC netlist using inputs $\mathcal{P}$ and $\mathcal{B}$ and WLCg. This is achieved by constructing one intermediate netlist (WLCp) and the final netlist (WLCa). To activate the UF constraints in $\mathcal{P}$, WLCp is created by duplicating WLCg but attaching the UF constraints in $\mathcal{P}$ to the appropriate signals. The boxes listed in $\mathcal{B}$ need to be made black, so the outputs of each such box need to be replaced by new PIs. WLCa is built by duplicating WLCp but with the outputs of the boxes in $\mathcal{B}$ replaced by the new PIs.

VI. Experimental Results

In this section, we present the experimental results of our implementation of UFAR with different optimization methods enabled. The implementation is based on ABC [10] using its latest improvements to Verilog parsing and bit-blasting.

We ran UFAR on a set of 2492 industrial word-level Verilog designs that were synthesized by an industrial tool to be cycle-accurate with the original circuit. Multipliers are the targeted problematic operators for UFAR to abstract. All experiments were performed on a workstation of Intel Xeon E5504 CPUs clocked at 2.0 GHz with 24 GB of RAM.

Comparing our results against publicly available verification tools is difficult. To our knowledge, no tools exist that can parse such designs directly without requiring a major modification. Also, there is no standard format for sequential word level circuits, as there is for the combinational case with SMT-LIB [4]. Therefore we compared results of running a) `super_prove` [9] on bit-blasted designs against b) three UFAR versions with different optimization settings.

For `super_prove`, we simply bit-blasted an input miter and immediately called `super_prove` to solve it. For UFAR we used three versions in this comparison:

- `opt1` means the basic version.
- `opt2` means `opt1` plus counterexample minimization.
- `opt3` means `opt2` plus random simulation.

For all UFAR versions, four bit-level verification engines were run in parallel, 3 variants of PDR and one BMC implementation. BMC is much more efficient at finding counterexamples, while the 3 versions of PDR in combination are efficient at proving a problem UNSAT.

We present the results in Figure 3, where the horizontal axis represents wall-clock time and the vertical axis represents the cumulative number of solved instances. A time-out of 1 hour was enforced for each example. The result of `super_prove` is not shown in Figure 3 because its number of solved instances is 2115, well below the bottom scale of 2330.

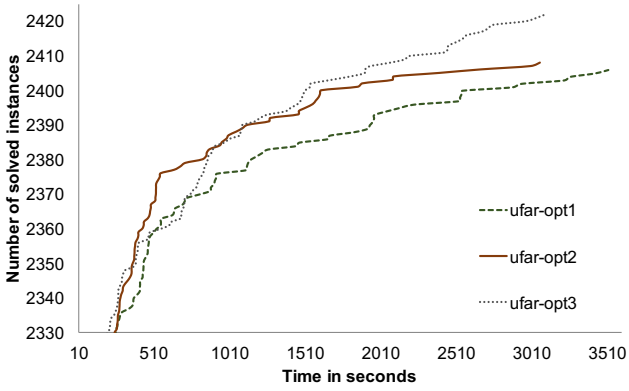


Fig. 3: Comparison of UFAR variants.

super_prove	ufar-opt1	ufar-opt2	ufar-opt3
2115	2398	2408	2422

TABLE I: The numbers of solved instances using different settings. 70 instances remain unsolved.

The `opt2` version is slightly better than `opt1` because the counterexample minimization prevents UFAR from applying too many constraints. The `opt3` version works best because the random simulation finds important UF constraints that can be missed by counterexamples. All solved instances are *unsat*.

Table I shows the numbers of instances finally solved by all versions within the 1-hour time-out. The three versions of UFAR outperform `super_prove`, which is often ineffective in solving problems with many arithmetic operators.

We selected 19 out of 2492 designs to present more detailed results in Table II. The selection is somewhat arbitrary but it does represent designs that are dissimilar and gives an idea of expected ranges of iterations needed, UF constraints used, and white box operators in the final abstractions. We observe the following from the table.

1) UFAR proves most cases with a relatively small number of white-box multipliers.
2) The number and quality of UF constraints are two important factors of performance. If the number is large, then UFAR generally needs more time to run, which is why counterexample-based constraint reduction is important. If the quality is good, then UFAR may prove a problem with fewer white boxes (or none). This supports the using of the random simulation to find good constraints.
3) It takes a nontrivial number of refinements for UFAR to converge, implying that UFAR builds up abstractions gradually. A major challenge is to figure out how to strike a good balance between the number and quality of UF constraints and the number of white boxes needed.
4) The main effect of counterexample minimization seems to be to make the overall algorithm more efficient but solves only 2 additional benchmarks.
5) Random simulation made UFAR faster and helped solve

					sp	ufar-opt1		ufar-opt2		ufar-opt3	
Design	#Mults	#AIGs	#FFs	Result	Time	Time	$i_p/i_w/n_p/n_w$	Time	$i_p/i_w/n_p/n_w$	Time	$i_p/i_w/n_p/n_w$
1	60	187303	2608	unsat	306.71	173.54	1/0/364/0	1127.1	3/0/12/0	3.39	1/0/874/0
2	11	72003	358	unsat			2/1/24/9	2661.2	3/1/30/9	1209.7	1/1/6/9
3	16	115888	550	unsat		20.75	2/0/37/0		3/1/17/10	18.69	1/0/12/0
4	12	49948	819	unsat		4.01	4/0/26/0	4.47	4/0/18/0	1.33	1/0/4/0
5	102	104272	1524	unsat		60.65	3/0/641/0	65.56	6/0/82/0	98.18	2/0/1252/0
6	144	161721	2456	unsat		1225.8	4/0/2366/0	2085.1	12/0/57/0	843.09	1/0/352/0
7	8	192143	5825	unsat		725.36	1/0/14/0	488.99	2/0/4/0	833.42	2/0/11/0
8	14	21092	400	unsat		474.51	3/1/30/10	316.95	3/1/28/8	272.7	2/1/33/8
9	32	46239	972	unsat	8.93		7/3/315/5		3/2/16/4	2428.6	3/3/276/9
10	43	302277	4065	unsat		157.14	3/0/597/0	106.83	4/0/40/0	66.38	1/0/447/0
11	4	25355	1552	unsat	156.16	1.49	1/0/2/0	1.10	1/0/2/0	0.99	1/0/2/0
12	15	49718	1707	unsat		25.33	3/1/40/7	48.46	8/1/34/7	24.93	3/1/52/7
13	21	65892	1997	unsat		40.18	2/1/154/7	50.42	5/1/40/9	102.24	6/1/90/7
14	223	292183	2649	unsat		2548.9	41/8/2398/50		34/9/674/55	1670.8	7/5/1401/37
15	63	91259	875	unsat		1967.4	10/4/915/29	517.86	10/4/142/37	2457.5	3/5/374/35
16	15	184859	4785	unsat		2939.8	1/1/68/4	535.18	2/1/7/3	2169.9	1/1/73/3
17	216	128137	1661	unsat		2231.2	4/0/1107/0		16/0/1943/0	401.76	1/0/394/0
18	253	199466	3751	unsat			118/0/23825/0	6.15	5/2/78/5	686.49	82/2/8638/3
19	475	274801	4204	unsat		470.71	5/0/10556/0	150.18	8/0/172/0	158.99	1/0/268/0

TABLE II: Detailed results of 19 unsat designs. The #Mults/#AIGs/#FFs means the number of multipliers/bit-level AIG nodes/bit-level flip flops. The $i_p/i_w/n_p/n_w$ means the number of iterations of applying new UF constraints/iterations of applying new white boxes/total UF constraints/total white boxes.

4 more benchmarks.

VII. Conclusion and Future Work

UFAR is an algorithm that abstracts (black-boxes) all problematic operators up front and refines them by applying UF constraints and/or white-boxing. We presented two optimization techniques for UFAR. We demonstrated UFAR's scalability on a large set of industrial problems.

For future work, we would like to understand a few of the anomalies in Table II (e.g., Designs 15 and 18) where an optimization caused quite a large slow-down in the solving. We also want to experiment on the 70 remaining unsolved benchmarks to find additional techniques to solve more problems. We plan to extend UFAR to use UF constraints across time frames and to perform refinement more gradually. For example, instead of white-boxing an entire operator, we might *grey-box* it. The under- and over-approximation method [12] can also be modified to fit into our work. An under-approximation is created by restricting bit-widths of primary inputs while an over-approximation can be built by abstracting problematic operators. Last, we plan to integrate modern SMT solvers to investigate possible advantages in this setting.

VIII. Acknowledgements

This work was supported in part by SRC contract 2265.001 as well as NSA under the TRUST project. We also thank industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Mentor Graphics, Microsemi, Synopsys, and Verific. for their continued support.

References

[1] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Reveal: A formal verification tool for verilog designs. In *Proc. of LPAR '08*.

[2] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of verilog models. In *Proc. of DAC'04*.

[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proc. of CAV'11*.

[4] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[5] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *Proc. of ICCD'06*.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proc. of TACAS'99*.

[7] A. R. Bradley. Sat-based model checking without unrolling. In *Proc. of VMCAI'11*.

[8] B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O'Leary. ATLAS: automatic term-level abstraction of RTL designs. In *Proc. of MEMOCODE'10*.

[9] R. Brayton, N. Een, and A. Mishchenko. Using speculation for sequential equivalence checking. In *Proc. of IWLS'12*.

[10] R. Brayton and A. Mishchenko. Abc: An academic industrial-strength verification tool. In *Proc. of CAV'10*.

[11] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proc. of TACAS'09*.

[12] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proc. of TACAS'07*.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of LICS'90*.

[14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV'00*.

[15] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. of TACAS'08*.

[16] N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Proc. of FMCAD'11*.

[17] N. Eén and N. Sörensson. An extensible sat-solver. In *Proc. of SAT'03*.

[18] K. L. McMillan. Interpolation and sat-based model checking. In *Proc. of CAV'03*.

[19] A. Mishchenko, N. Eén, and R. Brayton. A toolbox for counter-example analysis and optimization. In *Proc. of IWLS'13*.

[20] C. Pixley. A theory and implementation of sequential hardware equivalence. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 2006.

[21] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *Proc. of FMCAD'00*.

[22] C. A. van Eijk. Sequential equivalence checking based on structural similarities. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 2006.

Analysis of Incomplete Circuits using Dependency Quantified Boolean Formulas

Ralf Wimmer, Karina Wimmer, Christoph Scholl, Bernd Becker
Institute of Computer Science, Albert-Ludwigs-Universität Freiburg, Germany
{wimmer, wimmerka, scholl, becker}@informatik.uni-freiburg.de

***Abstract*—We consider Dependency Quantified Boolean Formulas (DQBFs), a generalization of Quantified Boolean Formulas (QBFs), and demonstrate that DQBFs are a natural calculus to exactly express the realizability problem of incomplete combinational and sequential circuits with an arbitrary number of (combinational or bounded-memory) black boxes. In contrast to usual approaches for controller synthesis, restrictions to the interfaces of missing circuit parts in distributed architectures are strictly taken into account. We present a solution method for DQBFs together with the extraction of Skolem functions for existential variables, which can directly serve as implementations for the black boxes. First experimental results are provided.**

I. Introduction

Solver-based techniques have proven to be successful in many areas in computer-aided design, ranging from formal verification of digital circuits [1], [2], [3], [4] over automatic test pattern generation [5], [6] to circuit synthesis [7]. While research on solving quantifier-free Boolean formulas (the famous SAT-problem [8]) has reached a certain level of maturity, designing and improving algorithms for quantified Boolean formulas (QBFs) is in the focus of active research. However, there are applications like the verification of partial circuits [4], [9], [10], the synthesis of safe controllers [7] for which QBF is not expressive enough to provide a compact and natural formulation. The reason is that QBF requires linearly ordered dependencies of the existential variables on the universal ones: Each existential variable implicitly depends on all universal variables in whose scope it is. Relaxing this condition yields so-called *dependency quantified Boolean formulas (DQBFs)*. DQBFs are strictly more expressive than QBFs in the sense that an equivalent QBF formulation can be exponentially larger than a DQBF formulation. This comes at the price of a higher complexity of the decision problem: DQBF is NEXPTIME-complete [11], compared to QBF, which is "only" PSPACE-complete. Encouraged by the success of SAT and QBF solvers and driven by the mentioned applications, research on solving DQBFs has started during the last few years [12], [13], [14], [15], yielding first prototypic solvers like iDQ [13] and HQS [14].

In this paper, we focus on the application of DQBF for analyzing *incomplete combinational and sequential circuits*. Such incomplete circuits appear in early design stages, when only a subset of the system's modules has already been implemented and verification is applied in order to find errors in the available parts as early as possible. Incomplete circuits also result if the complexity of the verification task is too high and therefore some parts, which are supposed not to influence the validity of some properties e. g., multiplier or memory modules, have been removed to make verification feasible. Analyzing incomplete circuits is also useful if a designer wants to localize errors (then one can remove parts of the design and if for all possible implementations of the removed parts the error does not disappear, the remaining parts must be erroneous). Therefore this problem has received considerable attention in the research community during the last 15 years, see, e. g., [4], [16], [17], [18], [19], [20], [21]. All solver-based approaches are restricted in the sense that they can either only handle a single black box or do not take the interfaces of the black boxes into account, allowing the black boxes to read signals which are not available to them in the actual design.

We show how the realizability problem for incomplete combinational and sequential circuits with an an arbitrary number of combinational or bounded-memory black boxes can be expressed as a DQBF. Here we show for the first time a DQBF-based solution for sequential circuits with several bounded-memory black boxes where the exact interface of the black boxes, i. e., the signals entering and leaving the black boxes, can be taken into account. We also show that solving a DQBF has the same complexity as deciding realizability. We not only sketch how DQBFs are solved in our DQBF solver HQS [14], [15], but also how so-called Skolem functions can be obtained from the solution process, provided that the formula is satisfied. These Skolem functions can directly serve as an implementation of the black boxes.

This paper builds on different sources: [9], [10] applies DQBF-based methods to incomplete combinational circuits with combinational black boxes. SAT- and QBF-based techniques for controller synthesis are considered in [7]; there a footnote give hints how DQBF can be used for that purpose. Due to the lack of efficient DQBF solvers at that time, this idea was not investigated further. However the method described there considers only a single black box which can read all primary inputs and the complete state information. The basic techniques implemented in our DQBF solver HQS have been described in [14], and [15] defines preprocessing techniques for DQBF, which speed-up the solution process considerably.

Structure of the paper: In the next section, we introduce dependency quantified Boolean formulas (DQBFs). In Section III we describe, how realizability of incomplete combinational and sequential circuits can be formulated as a DQBF. Section IV presents a method to solve DQBFs and to

obtain Skolem functions for satisfied DQBFs. In Section V we give preliminary experimental results, and conclude the paper in Section VI, pointing out challenges which need to be solved.

II. Foundations

Let φ, κ be quantifier-free Boolean formulas over the set V of variables and $v \lessdot V$. We denote by $\varphi[\kappa/v]$ the Boolean formula which results from φ by replacing all occurrences of v (simultaneously) by κ. For a set $V \gtrdot V$, we denote by $\mathcal{A}(V)$ the set of Boolean assignments for V, i. e., $\mathcal{A}(V) = \{\nu \mid \nu : V \to \{0,1\}\}$. For each formula φ over V, a variable assignment ν to the variables in V induces a truth value 0 or 1 of φ, which we call $\nu(\varphi)$.

Definition 1 (Syntax of DQBF): Let $V = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ be a set of Boolean variables. A *dependency quantified Boolean formula* (DQBF) ψ over V has the form $\psi := \forall x_1 \ldots \forall x_n \exists y_1(D_{y_1}) \ldots \exists y_m(D_{y_m}) : \varphi$, where $D_{y_i} \gtrdot \{x_1, \ldots, x_n\}$ for $i = 1, \ldots, m$ is the *dependency set* of y_i, and φ is a Boolean formula over V, called the *matrix* of ψ.

$V_\psi^\forall = \{x_1, \ldots, x_n\}$ denotes the set of universal and $V_\psi^\exists = \{y_1, \ldots, y_m\}$ the set of existential variables. We often write $\psi = Q : \varphi$ with the quantifier prefix Q and the matrix φ. $Q \setminus \{v\}$ denotes the prefix that results from removing a variable $v \lessdot V$ from Q together with its quantifier. If v is existential, then its dependency set is removed as well; if v is universal, then all occurrences of v in the dependency sets of existential variables are removed. Similarly we use $Q \cup \{\exists y(D_y)\}$ to add existential variables to the prefix. We sometimes assume that a DQBF $\psi = Q : \varphi$ as in Def. 1 with φ in conjunctive normal form (CNF) is given. A formula is in CNF if it is a conjunction of (non-tautological) *clauses*; a clause is a disjunction of *literals*, and a literal is either a variable v or its negation $\neg v$. As usual, we identify a formula in CNF with its set of clauses and a clause with its set of literals. For a formula φ (resp. clause C, literal l), $\mathrm{var}(\varphi)$ (resp. $\mathrm{var}(C)$, $\mathrm{var}(l)$) means the set of variables occurring in φ (resp. C, l), $\mathrm{lit}(\varphi)$ ($\mathrm{lit}(C)$) means the set of literals occurring in φ (C).

A quantified Boolean formula (QBF) (in prenex normal form) is a DQBF such that $D_y \gtrdot D_{y'}$ or $D_{y'} \gtrdot D_y$ holds for any two existential variables $y, y \lessdot V_\psi^\exists$. Then the variables in V can be ordered resulting in a linear quantifier prefix, such that for each $y \lessdot V_\psi^\exists$, D_y equals the set of universal variables which are to the left of y.

The semantics of a DQBF is usually defined by so-called Skolem functions.

Definition 2 (Semantics of DQBF): Let ψ be a DQBF as above. It is *satisfiable*, iff there are functions $s_y : \mathcal{A}(D_y) \to \mathbb{B}$ for $y \lessdot V_\psi^\exists$ such that replacing each $y \lessdot V_\psi^\exists$ by (a Boolean expression for) s_y turns ϕ into a tautology. The functions $(s_y)_{y \lessdot V_\psi^\exists}$ are called *Skolem functions* for ψ.

Example 1: Consider the following DQBF:

$$\forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2) : (x_1 \top \neg y_1) \wedge (x_2 \top y_1 \top y_2)$$

Here the variable y_1 depends only on x_1, but not on x_2; y_1 depends only on x_2, but not on x_1. It is satisfiable by using the Skolem functions $s_{y_1}(x_1) = x_1$ and $s_{y_2}(x_2) = \neg x_2$. Replacing y_1 and y_2 by their Skolem functions yields $(x_1 \top \neg x_1) \wedge (x_2 \top x_1 \top \neg x_2)$, which is obviously a tautology.

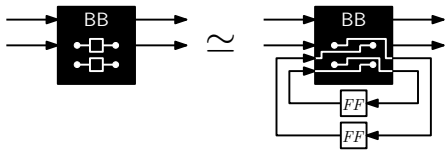


Fig. 1. Sequential circuits with extracted memory

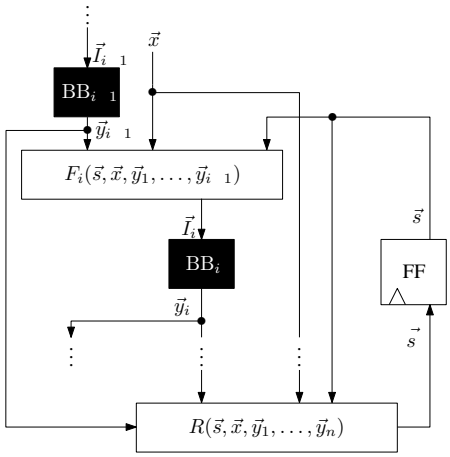


Fig. 2. Notation for incomplete sequential circuits

III. Analysis of Incomplete Circuits

In this section, we show how DQBFs can be used to analyze incomplete combinational and sequential circuits. In both cases we ask for realizability: Are there implementations of the missing parts ("black boxes") such that the complete circuit satisfies its specification.

We assume that the missing parts are either combinational or contain only a bounded amount of memory. In the latter case, we can put the flipflops of the black boxes into the available circuit part such that the incoming and outgoing signals of these flipflops are written and read only by the black boxes as sketched in Fig. 1.

Then the black boxes themselves are purely combinational. Note that the case of several black boxes with an *unbounded* amount of memory is undecidable [22].

We use the notation for incomplete sequential circuits as sketched in Fig. 2. The primary inputs are denoted by $\vec{x}$, the current state by $\vec{s}$, and the next state by $\vec{s}$. The missing parts are $\mathrm{BB}_1, \ldots, \mathrm{BB}_n$, whose interfaces, i. e., the signals entering and leaving the black boxes, are known. The input signals of black box BB_i are denoted by $\vec{I}_i$, its output signals by $\vec{y}_i$. The input cone of black box BB_i ensures the constraint $\vec{I}_i \equiv F_i(\vec{s}, \vec{x}, \vec{y}_1, \ldots, \vec{y}_i\ _1)$, the next state is described by $\mathrm{trans} := \vec{s}\ \equiv R(\vec{s}, \vec{x}, \vec{y}_1, \ldots, \vec{y}_n)$. We assume w. l. o. g. that no black box output is directly connected to an input of another black box or a flipflop, i. e., $\vec{y}_i \cap \vec{I}_j = \subset$for all i, j and $\vec{s}\ \cap \vec{y}_j = \subset$for

all j. Otherwise a buffer is inserted between the two black boxes without changing the functionality of the circuit. Additionally we assume that there are no cyclic dependencies between the combinational black boxes, i. e., that BB_i only depends on the outputs of $\mathrm{BB}_1, \ldots, \mathrm{BB}_{i-1}$.

We are considering invariant properties $\mathrm{inv}(\vec{s}, \vec{x}, \vec{y}_1, \ldots, \vec{y}_n)$, defined over the primary inputs $\vec{x}$, the current state $\vec{s}$ and the black box outputs $\vec{y}_1, \ldots, \vec{y}_n$, which are required to hold at any time.

A. Combinational Circuits

The same notation as introduced above is also used for combinational circuits. Here, the state and next state signals $\vec{s}$ and $\vec{s}'$ as well as the memory elements are omitted.

Definition 3: The *partial equivalence checking problem (PEC)* is defined as follows: Given an incomplete circuit C_{impl} and a (complete) specification C_{spec}, are there implementations of the black boxes in C_{impl} such that C_{impl} and C_{spec} become equivalent?

In the following we assume that (incomplete) implementation C_{impl} and specification C_{spec} are combined into a single circuit using a miter construction: Corresponding primary inputs are connected, corresponding outputs are connected via XOR gates. The outputs of the XOR-gates are combined via OR-gates into a single output signal. This output signal is constantly one iff, for some implementation of the black boxes, the two circuits are equivalent. This can be considered as a kind of invariant property, valid at the primary output of the combined circuit.

We now show how DQBF can be used to decide PEC.

Consider a PEC with black boxes $\mathrm{BB}_1, \ldots, \mathrm{BB}_n$. We first construct the quantifier prefix of the DQBF. The primary inputs $\vec{x}$ and the black box inputs $\vec{I}_1, \ldots, \vec{I}_n$ are universally quantified, all other variables are existentially quantified. The dependency set of black box outputs $\vec{y}_i$ contains exactly the inputs $\vec{I}_i$ of BB_i. Hence the quantifier prefix is

$$\forall \vec{x} \forall \vec{I}_1 \ldots \forall \vec{I}_n \exists \vec{y}_1(\vec{I}_1) \ldots \exists \vec{y}_m(\vec{I}_m) \ .$$

If the black boxes are not directly connected to the primary inputs but to internal signals we have to take into account that not all possible combinations of values may arrive at the inputs of the black boxes. Since we use a universal quantification for the black box inputs we have to ensure that our formula is satisfied if the value of the black box inputs $\vec{I}_i$ deviates from the values obtained as a function $F_i(\vec{x}, \vec{I}_1, \ldots, \vec{I}_{i-1})$.

$$\varphi(\vec{x}, \vec{I}_1, \ldots, \vec{I}_n, \vec{y}_1, \ldots, \vec{y}_n) := \left(\bigwedge_{i=1}^{n} \vec{I}_i \equiv \vec{F}_i \right) \simeq \mathrm{inv}(\vec{x}, \vec{y}_1, \ldots, \vec{y}_n) \, .$$

This formula is not necessarily given in CNF. By applying Tseitin transformation [23], which is essentially introducing auxiliary variables for the internal signals of the circuit, one can obtain a CNF φ' that is satisfiability equivalent to φ and whose size is linear in the size of φ. Let $\vec{a}$ be the vector of these auxiliary variables, which are existentially quantified in the quantifier prefix. As their values are implied by the values of the variables in their input cone, we can use as their dependency sets either the universal variables in their input cone or the set of all universal variables. We prefer the latter, because this typically leads to DQBFs that are easier to solve.

The resulting DQBF is:

$$\psi := \forall \vec{x} \forall \vec{I}_1 \ldots \forall \vec{I}_n \exists \vec{y}_1(\vec{I}_1) \ldots \exists \vec{y}_n(\vec{I}_n) \exists \vec{a}(\vec{x}, \vec{I}_1, \ldots, \vec{I}_n) : \varphi'(\vec{x}, \vec{I}_1, \ldots, \vec{I}_n, \vec{y}_1, \ldots, \vec{y}_n, \vec{a}) \ .$$

This formula ψ is satisfied iff we can replace all $\vec{y}_i(\vec{I}_i)$ with Skolem functions $s_i(\vec{I}_i)$ such that φ' becomes a tautology. The Skolem functions s_i exist if and only if there are implementations for the black boxes BB_i of the PEC, such that the PEC is satisfied. Therefore any PEC can be translated with linear effort into a DQBF and the PEC is satisfied iff the DQBF is satisfied.

It is easy to see that there is also the converse transformation [10]: Each DQBF can be turned into a PEC, having one black box for each existential variable such that the PEC is realizable iff the DQBF is satisfiable. This implies that PEC is NEXPTIME-complete.

B. Sequential Circuits

For incomplete sequential circuits with multiple combinational or bounded-memory black boxes, we investigate the following problem:

Definition 4: The *realizability problem for incomplete sequential circuits (RISC)* is defined as follows: Given an incomplete sequential circuit with multiple combinational (or bounded-memory) black boxes and an invariant property, are there implementations of the black boxes such that in the complete circuit the invariant holds at all times?

To decide RISC, one can apply a generalization of ideas described in [7] for the synthesis of controllers (which are in fact single black boxes with access to all state bits and all primary circuit inputs).

According to the notations introduced in the previous section, let $\vec{s}$ denote the variables encoding the current state of the circuit, $\vec{s}'$ the next state, and $\vec{x}$ the primary inputs. The formulas $F_1, \ldots, F_n$ describe the input cones of the black boxes, $\vec{I}_1, \ldots, \vec{I}_n$ their inputs, $\vec{y}_1, \ldots, \vec{y}_n$ their outputs, and R is the next state function of the circuit. Additionally we assume that init represents the circuit's initial state(s) and inv its states that satisfy the invariant.

Definition 5: A subset $W \supset S$ is a *winning set* if all states in W satisfy the invariant and, for all values of the primary inputs, the black boxes can ensure (by computing appropriate values) that the successor state is again in W.

A given RISC is realizable if there is a winning set that includes the initial state of the circuit. This can be formulated as a DQBF. Similar to the combinational case, we have to take into account that the black boxes are typically not directly connected to the primary inputs, but to internal signals. This is done by restricting the requirement that the successor state is again a winning state to the case when the black box inputs are assigned consistently with the values computed by their input cones.

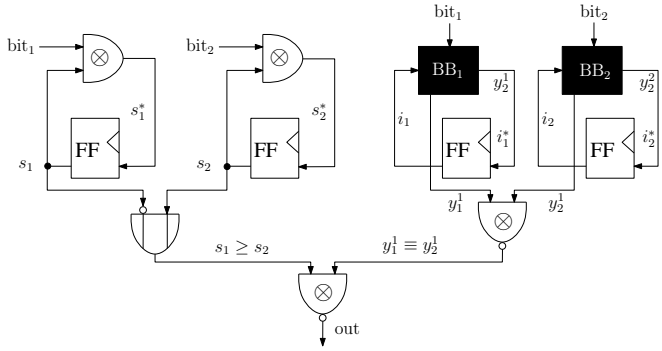


Fig. 3. Sequential circuit with two black boxes

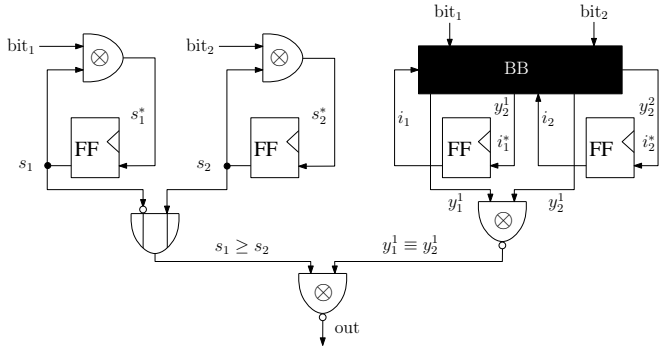


Fig. 4. The same sequential circuit as in Fig. 3, but with a single black box

Theorem 1: Given a RISC as defined above, the following DQBF is satisfied if and only if the RISC is realizable:

$$\forall \vec{s} \forall \vec{s}\ \forall \vec{x} \forall \vec{I}_1 \ldots \forall \vec{I}_n\ \exists \vec{y}_1(\vec{I}_1) \ldots \exists \vec{y}_n(\vec{I}_n)\ \exists w(\vec{s})\ \exists w\ (\vec{s}\ ) : \\ \big(\text{init} \simeq\ w\big) \wedge \big(w \simeq\ \text{inv}\big) \wedge \big(\vec{s} \equiv \vec{s}\ \simeq\ w \equiv w\ \big) \wedge \\ \Big(\big(w \wedge \bigwedge_{i=1}^{n} \vec{I}_i \equiv \vec{F}_i \wedge \text{trans}\big) \simeq\ w\ \Big).$$

Theorem 2: RISC is NEXPTIME-complete.

Proof: The reduction to DQBF above shows that RISC is in NEXPTIME. To show the hardness, we give a reduction from DQBF to RISC. First we transform the DQBF into an incomplete combinational circuit as shown in [9] such that the output of the circuit is constantly 1 iff the DQBF is satisfied. We now turn this combinational circuit into a sequential circuit with two states by storing the output of the combinational circuit in a 1-bit flipflop s. The initial state is $s \equiv 1$, the invariant is given by $s \equiv 1$. The original DQBF is satisfied iff the unsafe state 0 can be made unreachable by appropriate black box implementations. ■

Example 2: We illustrate the solution of RISC using two incomplete circuits in Fig. 3 and 4. The circuits are simple, but all the same illustrate the basic idea. We first start with the circuit in Fig. 3. The sequential circuit in Fig. 3 consists of two parts. The first part on the left can be seen as the specification for a simple sequential circuit: There are two bit streams applied to the inputs bit_1 and bit_2. The circuit computes the parities of the bit streams applied to bit_1 and bit_2 and outputs 1 iff the parity for bit stream bit_1 is smaller or equal to the parity for bit stream bit_2. The right hand side shows a given architecture for an implementation with two black boxes, one reading bit stream bit_1 and the other reading bit stream bit_2. The outputs of the black boxes are connected by an equivalence gate. Then the output of the overall circuit is computed by an equivalence gate connecting the outputs of specification and incomplete implementation. We require the invariant property that the output of the overall circuit is 1 at all times, i. e., that the black boxes are implemented in a way such that the implementation part agrees with the specification part. For this simple example it is easy to see that a corresponding implementation does not exist, even for black boxes with unbounded memory. Here we use our method where the number of flip flops for each black box is restricted to one. Fig. 3 already shows the transformed circuit where the memory is extracted from the black boxes.

Applying Theorem 1, we obtain the following formula parts:

- initial state:

$$\text{init} := (\neg s_1 \wedge \neg s_2 \wedge \neg i_1 \wedge \neg i_2)$$

- transition relation:

$$\text{trans} := (s_1 \equiv s_1 \otimes \text{bit}_1) \wedge (s_2 \equiv s_2 \otimes \text{bit}_1) \\ \wedge (i_1 \equiv y_2^1) \wedge (i_2 \equiv y_2^2)$$

- invariant:

$$\text{inv} := \big(\neg s_1 \top s_2\big) \equiv (y_1^1 \equiv y_2^1)\big)$$

Putting these parts together yields the following DQBF:

$$\forall \text{bit}_1 \forall \text{bit}_2 \forall s_1 \forall s_1 \forall s_2 \forall s_2 \forall i_1 \forall i_1 \forall i_2 \forall i_2 \\ \exists y_1^1(i_1, \text{bit}_1) \exists y_2^1(i_1, \text{bit}_1) \exists y_1^2(i_2, \text{bit}_2) \exists y_2^2(i_2, \text{bit}_2) \\ \exists w(s_1, s_2, i_1, i_2) \exists w\ (s_1, s_2, i_1, i_2) : \\ (\text{init} \simeq\ w) \wedge (w \simeq\ \text{inv}) \wedge \big((w \wedge \text{trans}) \simeq\ w\ \big) \\ \wedge\big(((s_1 \equiv s_1) \wedge (s_2 \equiv s_2) \wedge (i_1 \equiv i_1) \wedge (i_2 \equiv i_2)) \simeq\ (w \equiv w\ )\big)$$

By applying a DQBF solver, one can verify that this formula is unsatisfiable, meaning that the design in Fig. 3 is not realizable.

Now consider the circuit in Fig. 4. It differs from the design in Fig. 3 only in the black boxes: Both black boxes can read both input signals bit_1 and bit_2. Thus, the black boxes can equivalently be merged into one as shown in Fig. 4. It is easy to see that this implementation, which does not pay attention to the exact architecture by disregarding the interface of the black boxes, is now realizable. More precisely, it is realizable if we assume that the black box with bounded memory has at least 2 memory cells at its disposal. In Fig. 4 we depict the incomplete circuit with two memory cells extracted from the black box. Using our approach we can indeed prove realizability. The formula differs only in the dependency sets of the black box outputs: $D_{y_1^1} = D_{y_2^1} = D_{y_1^2} = D_{y_2^2} = \{\text{bit}_1, \text{bit}_2, i_1, i_2\}$. Now

the formula is satisfiable. The following Skolem functions turn the matrix into a tautology:

Variable	y_1^1	y_2^1	y_1^2	y_2^2	w	w
Skolem fct.	1	$\mathrm{bit}_1 \otimes i_1$	$\neg i_1 \top i_2$	$\mathrm{bit}_2 \otimes i_2$	1	1

Using these Skolem functions, the two flipflops in the right half store the same values as the two flipflops in the left half, i. e., $s_1 = i_1$ and $s_2 = i_2$. The equivalence $y_1^1 \equiv y_2^1$ corresponds to $1 \equiv (\neg i_1 \top i_2)$, which is the same as $i_1 \geq i_2$. Therefore the design is realizable.

For both incomplete circuits, our solver HQS [14] solved the DQBF in at most 0.1 seconds.

We can conclude that it is necessary to take the precise interfaces of the black boxes into account in order to obtain a valid answer whether the design is realizable.

IV. Solving DQBFs

Elimination-based DQBF solvers like HQS [14], [15] apply a series of transformation steps to the formula until a SAT or QBF problem results, which can be solved by an arbitrary SAT or QBF solver. As a pure yes/no answer is not satisfactory when solving analysis problems as presented in the previous section, we provide the main ideas how Skolem functions can be extracted from the solution process. More details can be found in [24]. This extraction proceeds in the reverse order of transformation, starting with (constant) Skolem functions for the final SAT problem, which correspond to a satisfying assignment.

A. Transformation steps

The central operation of elimination-based solvers is the elimination of existential and universal variables from the formula. QBF solvers can eliminate variables in the order given by the quantifier prefix (starting with the inner-most variable block). Because there is no linear order on the variables in a DQBF, this is typically no longer possible.

Lemma 1: Let $\psi = Q : \phi$ be a DQBF and $y \langle V_\psi^\exists$ an existential variable which depends on all universal variables. Then ψ is equivalent to $\psi := Q \setminus \{\exists y(D_y)\} : \phi[0/y] \top \phi[1/y]$.

If $s_z^{\psi'}$ for $z \langle V_{\psi'}^\exists$ are Skolem functions for the formula ψ, obtained by eliminating $y \langle V_\psi^\exists$, we set $s_y^\psi := \phi[1/y][s_z^{\psi'}/z]$ and $s_z^\psi := s_z^{\psi'}$ for $z \not\bowtie y$. This yields Skolem functions for ψ.

The elimination of universal variables in solvers like HQS [14] is done by *universal expansion* [25], [26], [27], [10]. This is applicable even if some existential variables depend upon the expanded universal one.

Lemma 2: For a DQBF $\psi = \forall x_1 \ldots \forall x_n \exists y_1(D_{y_1}) \ldots \exists y_m(D_{y_m}) : \varphi$ with $E_{x_i} = \{y_j \langle V_\psi^\exists \mid x_i \langle D_{y_j}\}$, the *universal expansion* w. r. t. variable $x_i \langle V_\psi^\forall$, is defined by

$$\big(Q \setminus \{x_i\}\big) \cup \big\{\exists y_j(D_{y_j} \setminus \{x_i\}) \,\big|\, y_j \langle E_{x_i}\big\} : \\ \varphi[1/x_i] \wedge \varphi[0/x_i][y_j/y_j \text{ for all } y_j \text{ with } x_i \langle D_{y_j}] \,.$$

That means, when eliminating a universal variable $x \langle V_\psi^\forall$, we have to copy all existential variables $y \langle V_\psi^\exists$ that depend upon x.

Assume that $s_z^{\psi'}$ for $z \langle V_{\psi'}^\exists$ are Skolem functions for ψ. Then $s_y^\psi := (x \wedge s_{y'}^{\psi'}) \top (\neg x \wedge s_y^{\psi'})$ for $y \langle V_\psi^\exists$ with $x \langle D_y$ and $s_z^\psi := s_y^{\psi'}$ for $z \langle V_\psi^\exists$ with $x \not\langle D_z$ are Skolem functions for ψ.

In principle, these two operations suffice to turn each DQBF into an (exponentially larger) SAT problem. In order to reduce computation time and memory consumption, pre- and inprocessing steps have turned out to be essential.

Standard operations are the detection of unit and pure literals. A literal ℓ is unit if (ℓ) is a clause in the formula. A literal ℓ is pure if $\neg\ell$ does not appear in the formula. In both cases $\mathrm{var}(\ell)$ can be replaced by a constant (which is also the Skolem function for that variable). Further preprocessing techniques like blocked clause elimination, the identification of equivalent variables, and structure extraction have been devised for DQBF [15]. All of them are supported when Skolem functions are to be computed.

B. Elimination sets

Since the expansion of all universal variables leads to an exponentially larger SAT instance, this is typically not feasible. Instead, the solver HQS eliminates variables only until a QBF is obtained, which can be solved by an arbitrary QBF solver. The central problem is to determine a minimum set of universal variables whose elimination turns the DQBF into a QBF [14]. To solve this, we can use the following dependency graph:

Definition 6: Let ψ be a DQBF. Its *dependency graph* $G_\psi = (V_\psi^\exists, E_\psi)$ is a directed graph with the existential variables as its nodes and edges $E_\psi = \{(y, z) \langle V_\psi^\exists \times V_\psi^\exists \mid D_y \not\bowtie D_z\}$.

It can be used to recognize if a DQBF is actually a QBF:

Lemma 3: Let ψ be a DQBF. Its dependency graph G_ψ is acyclic iff ψ has an equivalent QBF prefix.

That mean we have to find a minimum set $U \rangle V_\psi^\forall$ of universal variables whose elimination makes G_ψ acyclic. One can show that for making the graph acyclic by eliminating universal variables, it suffices to consider the cycles of length 2. The selection of variables can be done using a MAXSAT solver: for each universal variable x a variable m_x is created in the MAXSAT solver such that $m_x = 1$ means that x needs to be eliminated. Soft clauses are used to get an assignment with a minimum number of variables assigned to 1. Hard clauses enforce that for all $y, z \langle V_\psi^\exists$, $y \not\bowtie z$, either all variables in $D_y \setminus D_z$ or in $D_z \setminus D_y$ are eliminated.

The variables in U are then eliminated, ordered according to the number of existential variables that depend upon them.

For more details, including formal correctness proofs, we refer the reader to [14].

C. Solver overview

Fig. 5 shows the structure of the solver HQS. The input is a DQBF in CNF. After preprocessing, which is done on the CNF, gate detection is applied, essentially undoing Tseitin transformation and removing the existential variables introduced by the CNF transformation. The result is a representation of the formula as an And-Inverter graph (AIG), on which the further

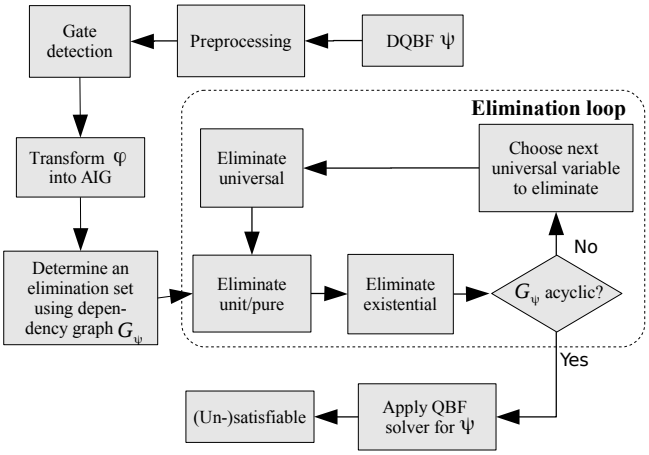


Fig. 5. Structure of the solver

steps are performed. Before the actual elimination loop starts, we determine a minimum elimination set as described above.

Within the elimination loop, we check for unit and pure variables, which can be replaced by constants. This is done on the AIG using syntactic checks. Additionally, all existential variables are eliminated for which this is possible. Otherwise they would double for each eliminated universal variable. Then we check if the dependency graph has already become acyclic. If this is the case we generate the corresponding QBF prefix and solve the formula using the QBF solver AIGsolve [28], which operates directly on AIGs. Otherwise we select the next universal variable to eliminate and expand it.

V. Experimental Results

In the following we present preliminary experimental results for incomplete combinational circuits. To solve the DQBFs, we use our elimination-based DQBF solver HQS [14], which was described briefly in the previous section.

We have extended HQS by the possibility to compute Skolem functions for satisfied DQBFs. The computation of Skolem functions works in two phases: During the solution process we collect the necessary data and store it on a stack. When the satisfiability of the formula has been determined, we free the other data structures of the solver and extract the Skolem functions from the collected data. We can apply don't-care minimization to the Skolem functions, based on Craig interpolants [29], and use the tool ABC [30] for further minimization of the Skolem functions' AIG representation.

All experiments were run on one Intel Xeon E5-2650v2 CPU core at 2.60 GHz clock frequency and 64 GB of main memory under Ubuntu Linux as operating system. We aborted all experiments which either took more than 1000 s CPU time or more than 8 GB (= 2^{30} bytes) of main memory. As benchmarks we used 4318 DQBF instances from different sources. Most of them are DQBFs resulting from equivalence checking of incomplete combinational circuits [10], [31], [13]. The remaining ones are controller synthesis problems [7].

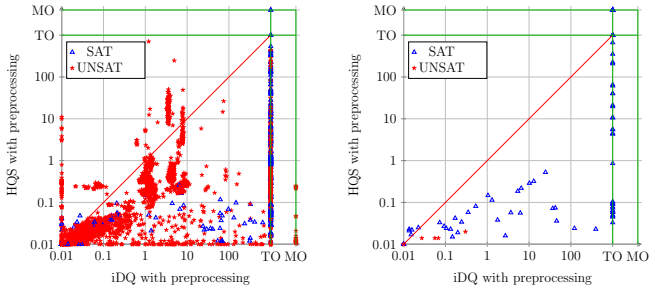


Fig. 6. Computation times (in seconds) of HQS and iDQ (both with preprocessing) on PEC instances [14], [31], [13] (left) and instances from controller synthesis [7] (right)

We first compare the efficiency of HQS with the only other available DQBF solver iDQ [13], which solves the formula by iteratively solving SAT instances generated from the DQBF. Both solvers were run after preprocessing the DQBFs. Since iDQ relies on a formula in CNF, while HQS does not, different preprocessing operations had to be applied: besides standard techniques like the detection of unit and pure literals and equivalent variables, preprocessing for HQS applies gate detection, which reverses Tseitin transformation in order to reconstruct the original circuit. This is not possible in case of iDQ. Instead, for iDQ, we apply blocked clause elimination and variable elimination by resolution, which are both not beneficial for HQS. We refer the reader to [15] for more details on DQBF preprocessing.

Figure 6 shows the results for incomplete combinational circuits (left, 3686 instances) and instances from controller synthesis (right, 89 instances) for those instances that could be solved by at least one solver; the remaining 543 instances could not be solved. The controller synthesis instances are incomplete sequential circuits with a single black box that can read all state bits and all primary inputs. We can observe in both cases, that HQS (with few exceptions) is more efficient and solves considerably more instances than iDQ.

In spite of the improvements made during the last few years, the size of the instances that can effectively be solved is smaller by roughly one to two orders of magnitude than solvable QBF instances – strongly dependent on the number of variable copies which are created to obtain an equivalent QBF.

The second set of experiments concerns the computation of Skolem functions for satisfiable DQBFs. We first measured the overhead of collecting the necessary data for computing Skolem functions during the solution process, i. e., until the truth value of the formula has been determined. The results are shown in Figure 7. We can observe that the overhead is in most cases negligible – in a very few cases, the memory consumption is even reduced. The reason for this behavior is that within the AIG package different optimizations like rewriting is triggered when certain thresholds are exceeded. This can lead to smaller AIGs and thus save memory (and computation time for the subsequent operations).

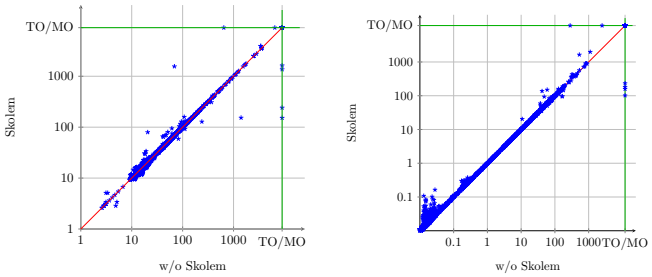


Fig. 7. Overhead of Skolem function computation on memory consumption (left) and running time (right)

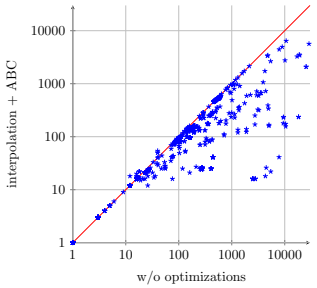


Fig. 8. Size of the computed Skolem functions with and without optimizations

For all 720 satisfiable instances we were able to solve without Skolem functions, we could also compute Skolem functions. For these instances we compare the sizes of the Skolem functions with and without optimizations using interpolation and ABC. Figure 8 displays the results. In many cases, we can reduce the sizes of the Skolem functions considerably, sometimes by up to two orders of magnitude.

Because QBFs are a special case of DQBFs, we can use HQS to compute Skolem functions for satisfied QBFs as well. In Fig. 9, we compare the sizes of the Skolem functions generated by HQS with those generated by the state-of-the-art QBF solver DEPQBF 5.0 [32], [33] for a set of satisfiable QBF instances from the QBF Gallery 2013[1] and from partial equivalence checking [4] (with a single black box). Since HQS (and in particular its preprocessor) is not optimized for solving QBF instances, we abstain from a detailed comparison of the running times of HQS and DEPQBF. DEPQBF is often (but not always) faster than HQS. In a few cases, the generation of Skolem functions with DEPQBF failed, because the necessary resolution proof became too large (we aborted DEPQBF when the size of the dumped resolution proof exceeded 20 GB).

Fig. 9 shows the sizes of the Skolem functions computed by DEPQBF and by HQS (with interpolation and ABC). To enable a fair comparison, we also applied ABC with the same commands to the Skolem functions generated using DEPQBF. We can observe that HQS' Skolem functions are in most cases smaller (often significantly) than those obtained from DEPQBF.

In summary, the experiments show that HQS is able to solve

[1]see http://www.kr.tuwien.ac.at/events/qbfgallery2013/

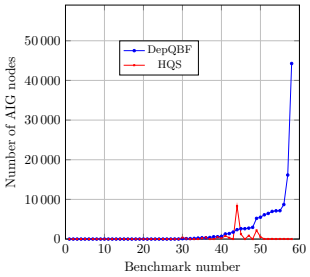


Fig. 9. Comparison of the sizes of Skolem functions from HQS and from DEPQBF on QBF instances. The instances are ordered according to the size of DEPQBF's Skolem functions

the DQBFs resulting from small to medium-sized circuits effectively. We can not only obtain a pure yes/no answer, but also Skolem functions for the satisfiable instances without significant overhead. On satisfiable *QBF* instances, the size of the Skolem functions computed by HQS is similar, in many cases smaller in comparison to Skolem functions computed by DEPQBF.

As a side remark, the example provided in Section III could easily be solved with a DQBF solver. Benchmarks dealing with the synthesis of multiple black boxes in sequential circuits currently do not exist, but would be interesting to have.

VI. CONCLUSION AND OPEN CHALLENGES

This paper has shown that DQBF formulations allow to express the realizability of invariant properties for incomplete combinational and sequential circuits with multiple black boxes in a natural way. First prototypic solvers allow not only to solve the resulting DQBFs for small to medium-sized circuits, but also to extract Skolem functions, which can serve as implementations of the missing parts.

Still, many challenges remain: The scalability of the solvers has to be improved and might be tuned towards specific applications. More powerful preprocessing techniques are necessary as well as improvements in the solver core. We hope that with the availability of solvers more applications of these techniques become feasible (distributed controller synthesis) or are newly discovered thereby inspiring further improvements of the solvers – just as it was for propositional SAT solving and is for QBF solving.

REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[2] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.

[3] P. Ashar, M. K. Ganai, A. Gupta, F. Ivancic, and Z. Yang, "Efficient SAT-based bounded model checking for software verification," in *Int'l Symp. on Leveraging Applications of Formal Methods (ISoLA)*, ser. Technical Report, T. Margaria, B. Steffen, A. Philippou, and M. Reitenspieß, Eds., vol. TR-2004-6. Paphos, Cyprus: Department of Computer Science, University of Cyprus, Oct. 2004, pp. 157–164.

[4] C. Scholl and B. Becker, "Checking equivalence for partial implementations," in *Proc. of DAC*. Las Vegas, NV, USA: ACM Press, Jun. 2001, pp. 238–243.

[5] S. Eggersglüß and R. Drechsler, "A highly fault-efficient SAT-based ATPG flow," *IEEE Design & Test of Computers*, vol. 29, no. 4, pp. 63–70, 2012.

[6] A. Czutro, I. Polian, M. D. T. Lewis, P. Engelke, S. M. Reddy, and B. Becker, "Thread-parallel integrated test pattern generator utilizing satisfiability analysis," *Int'l Journal of Parallel Programming*, vol. 38, no. 3-4, pp. 185–202, 2010.

[7] R. Bloem, R. Könighofer, and M. Seidl, "SAT-based synthesis methods for safety specs," in *Proc. of VMCAI*, ser. LNCS, K. L. McMillan and X. Rival, Eds., vol. 8318. San Diego, CA, USA: Springer, Jan. 2014, pp. 1–20.

[8] S. A. Cook, "The complexity of theorem-proving procedures," in *Proc. of STOC*. ACM Press, 1971, pp. 151–158.

[9] K. Gitina, S. Reimer, M. Sauer, R. Wimmer, C. Scholl, and B. Becker, "Equivalence checking for partial implementations revisited," in *Proc. of MBMV*, C. Haubelt and D. Timmermann, Eds. Rostock, Germany: Universität Rostock, ITMZ, Mar. 2013, pp. 61–70.

[10] ——, "Equivalence checking of partial designs using dependency quantified Boolean formulae," in *Proc. of ICCD*. Asheville, NC, USA: IEEE CS, Oct. 2013, pp. 396–403.

[11] G. Peterson, J. Reif, and S. Azhar, "Lower bounds for multiplayer non-cooperative games of incomplete information," *Computers & Mathematics with Applications*, vol. 41, no. 7–8, pp. 957–992, Apr. 2001.

[12] A. Fröhlich, G. Kovásznai, and A. Biere, "A DPLL algorithm for solving DQBF," in *Int'l Workshop on Pragmatics of SAT (POS)*, Trento, Italy, 2012.

[13] A. Fröhlich, G. Kovásznai, A. Biere, and H. Veith, "iDQ: Instantiation-based DQBF solving," in *Int'l Workshop on Pragmatics of SAT (POS)*, ser. EPiC Series, D. L. Berre, Ed., vol. 27. Vienna, Austria: EasyChair, Jul. 2014, pp. 103–116.

[14] K. Gitina, R. Wimmer, S. Reimer, M. Sauer, C. Scholl, and B. Becker, "Solving DQBF through quantifier elimination," in *Proc. of DATE*. Grenoble, France: IEEE, Mar. 2015.

[15] R. Wimmer, K. Gitina, J. Nist, C. Scholl, and B. Becker, "Preprocessing for DQBF," in *Proc. of SAT*, ser. LNCS, M. Heule and S. Weaver, Eds., vol. 9340. Austin, TX, USA: Springer, Sep. 2015, pp. 173–190.

[16] T. Nopper and C. Scholl, "Symbolic model checking for incomplete designs with flexible modeling of unknowns," *IEEE Trans. Computers*, vol. 62, no. 6, pp. 1234–1254, 2013.

[17] M. Herbstritt, B. Becker, and C. Scholl, "Advanced SAT-techniques for bounded model checking of blackbox designs," in *Proc. of MTV*. IEEE, 2006, pp. 37–44.

[18] W. Damm and B. Finkbeiner, "Automatic compositional synthesis of distributed systems," in *Proc. of FM*, ser. LNCS, C. B. Jones, P. Pihlajasaari, and J. Sun, Eds., vol. 8442. Singapore: Springer, May 2014, pp. 179–193.

[19] B. Finkbeiner and S. Schewe, "Bounded synthesis," *STTT*, vol. 15, no. 5-6, pp. 519–539, 2013.

[20] C. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, 1995.

[21] C. H. Seger, R. B. Jones, J. W. O'Leary, T. F. Melham, M. Aagaard, C. Barrett, and D. Syme, "An industrially effective environment for formal hardware verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, 2005.

[22] A. Pnueli and R. Rosner, "Distributed reactive systems are hard to synthesize," in *Annual Symp. on Foundations of Computer Science*. St. Louis, Missouri, USA: IEEE Computer Society, Oct. 1990, pp. 746–757.

[23] G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Studies in Constructive Mathematics and Mathematical Logic*, vol. Part 2, pp. 115–125, 1970.

[24] K. Gitina, R. Wimmer, C. Scholl, and B. Becker, "Skolem functions for DQBF," in *submitted for publication*, 2016.

[25] U. Bubeck and H. K. Büning, "Dependency quantified Horn formulas: Models and complexity," in *Proc. of SAT*, ser. LNCS, A. Biere and C. P. Gomes, Eds., vol. 4121. Seattle, WA, USA: Springer, Aug. 2006, pp. 198–211.

[26] U. Bubeck, "Model-based transformations for quantified Boolean formulas," Ph.D. dissertation, University of Paderborn, 2010.

[27] V. Balabanov, H. K. Chiang, and J. R. Jiang, "Henkin quantifiers and Boolean formulae: A certification perspective of DQBF," *Theoretical Computer Science*, vol. 523, pp. 86–100, 2014.

[28] F. Pigorsch and C. Scholl, "Exploiting structure in an AIG based QBF solver," in *Proc. of DATE*. IEEE, 2009, pp. 1596–1601.

[29] K. L. McMillan, "Applications of Craig interpolants in model checking," in *Proc. of TACAS*, ser. LNCS, N. Halbwachs and L. D. Zuck, Eds., vol. 3440. Edinburgh, UK: Springer, Apr. 2005, pp. 1–12.

[30] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Proc. of CAV*, ser. LNCS, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Edinburgh, UK: Springer, Jul. 2010, pp. 24–40.

[31] B. Finkbeiner and L. Tentrup, "Fast DQBF refutation," in *Proc. of SAT*, ser. LNCS, C. Sinz and U. Egly, Eds., vol. 8561. Vienna, Austria: Springer, Jul. 2014, pp. 243–251.

[32] F. Lonsing and A. Biere, "DepQBF: A dependency-aware QBF solver," *Journal on Satisfiability, Boolean Modelling and Computation*, vol. 7, no. 2-3, pp. 71–76, 2010.

[33] F. Lonsing, F. Bacchus, A. Biere, U. Egly, and M. Seidl, "Enhancing search-based QBF solving by dynamic blocked clause elimination," in *Proc. of LPAR*, ser. LNCS, M. Davis, A. Fehnker, A. McIver, and A. Voronkov, Eds., vol. 9450. Suva, Fiji: Springer, Nov. 2015, pp. 418–433.

Physical Design Factors That Contribute to Routing Congestion in Monolithic 3D Integrated Circuits

Yosef Borga, Daniel Limbrick
School of Electrical and Computer Engineering
North Carolina A&T State University
Greensboro, USA
yborga@aggies.ncat.edu, daniel.limbrick@ncat.edu

Sung Kyu Lim
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, USA
limsk@ece.gatech.edu

***Abstract*—Monolithic 3D integrated circuits (3D ICs) bond multiple silicon chips vertically in a single semiconductor manufacturing flow, which allows for finer integration, at the transistor level, than existing 3D ICs. Despite transistor-level monolithic 3D IC offering several performance benefits over existing technologies, it suffers from increased routing congestion, which inhibits it from exploiting these benefits. Congestion is such a problem that designs are not approaching optimal solution. This paper investigates technology-mapping and physical design factors that impact routing congestion. Results show congestion exacerbates designs and indicate that modifying existing physical design algorithms and layout rules can reduce routing congestion.**

I. Introduction

Integrated circuit technologies continue to shrink aggressively in order to improve performance and meet the need for increased functionality. Three-Dimensional Integrated Circuits (3D-ICs) provide a promising solution to continue the trend of increasing transistor density described in Moore's Law by bonding multiple silicon chips vertically to form an integrated stack [1]. Two current strategies for 3D integrated circuits are: (1) through-silicon-via (TSV) based technology where two dies are fabricated individually with their own metal layers and input/output pins and stacked on each other via through-silicon-vias [1], and (2) monolithic 3D integrated circuits that process all the functions in a single semiconductor manufacturing flow.

One of the design styles in monolithic 3D technology is transistor-level monolithic 3D integrated circuit (T-MIC). T-MIC, which involves building two-tier standard cells, has the benefits of reduced area, power and delay compared to 2D IC. Despite the benefits of T-MIC, it currently faces some design challenges that inhibits its adoption. One of the major problems in T-MIC designs is routing congestion. Routing congestion occurs when connections need to be made in an insufficiently small space causing longer routes and preventing design closure. Routing congestion has already been established as a growing bottleneck to transistor scaling in deep sub-micron transistor technologies due to the reverse scaling of wires [4]. This congestion curtails the performance, integration, yield, and cost of circuits. As shown in Fig. 2, the severity of routing congestion is greater in T-MIC designs due to the reduced cell area and densely packed cells. A highly compacted placement can result in routing congestion due to difficulties in accessing local pins and high number of nets crossing the routing area [5]. The same number of wires and pins available in 2D is available in T-MIC designs. Those wires and pins need to be routed within reduced cell area for T-MIC designs, which leads to increased routing demand that results in routing violations. In order to enable dense integration for all T-MIC designs, it is vital to mitigate routing congestion. Therefore, routing congestion is a severe problem in T-MIC designs that needs to be examined thoroughly.

Despite the severity of routing congestion in monolithic 3D designs, there are minimal works in solving this issue. Routing congestion is not perceptible as much in TSV-based 3D IC, because 2D placement and routing algorithms are employed prior to integrating the dies together. However, for monolithic 3D IC, the impact of 3D integration is felt at these design stages. The authors of [7] identified the severity of routing congestion and proposed ways to mitigate this issue by increasing the number of metal layers and reducing metal dimensions. This solution has the potential to significantly increase the fabrication cost. Increasing metal layers creates additional routing resources, but also increases the wirelength of the design due to detours required. This increase in wirelength minimizes the performance benefits gained from T-MIC designs, since most of the benefits come from reduction in cell area and wirelength. To date, there has been no work that analyzes the physical design factors that impact routing congestion in transistor-level monolithic 3D ICs at the physical design stage, as this technology is fairly new. However, such as analysis is necessary to facilitate the technology's adoption.

This paper investigates routing congestion in monolithic 3D IC across a typical automated design flow. Through this investigation, physical design factors (i.e., technology mapping, floorplanning, placement, and cell layout) that contribute to routing congestion are identified. Results indicate that T-MIC designs are not reaching optimal solution due to congestion. Even though the implementation of just certain cells and increasing row spacing is not a recommended solution, we show that congestion degrades performance to the point where implementing these options actually give a better result.

The remainder of the paper is organized as follows. Section II covers the description of monolithic 3D IC technology. Section III investigates technology-mapping

factors that contribute to routing congestion. This was imperative in determining how early in the design stage congestion arises. Section IV provides cell layout factors that contribute to routing congestion in 3D ICs. This was essential in determining how the layouts of a standard cell contribute to routing congestion. Section V investigates floorplanning factors that contribute to routing congestion. Finally, section VI provides a summary of this work.

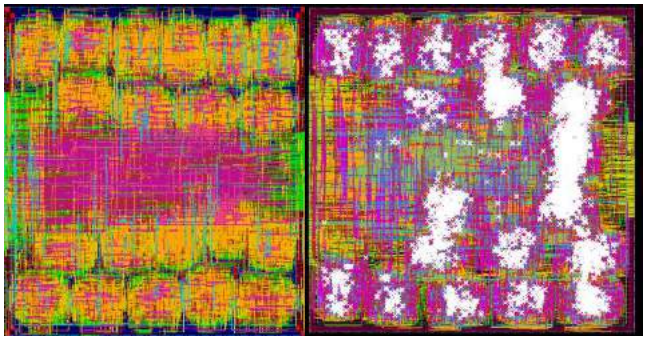
II. Monolithic 3D IC Technology

Monolithic 3D IC technology allows transistor logic to be built on multiple tiers, thus reducing both the cell and circuit footprint. Monolithic 3D ICs can be vertically integrated at the gate level, where gates are built on separate layers and stacked, or at the transistor level, where an individual gate is built on multiple tiers. Each case, allows for finer integration than existing 3D ICs [2]. Some of the advantages of T-MICs are [1, 2, 3]: (1) Reduction in die size, delay and power, (2) option to reuse design tools, (3) cost improvement, (4) monolithic-inter-vias (50nm) diameter are smaller than TSV (5μm), (5) alignment precision, and (6) heterogeneous integration.

The monolithic 3D IC technology library that was used in this study was taken from [7]. In the monolithic 3D technology, the standard cells are generated from the Nangate45nm [10] standard library. First, the original 2D IC cell is folded where the VDD and GND rails are aligned on top of each other to minimize area and avoid redesigning of internal connections. Then, the cell is cut in half and the NMOS side is placed on the top tier, while the PMOS part is placed on the bottom tier. The internal wires are carefully modified to preserve connectivity and to retain design rules. Then, the two tiers are connected using monolithic inter-tier vias (MIVs). Fig. 2 demonstrates the steps taken to create a transistor-level monolithic 3D IC. The method shown in Fig. 2 produces monolithic 3D standard cells that exhibit a 40% reduction.

In this work, Cadence Encounter Digital Implementation System was used to implement the RTL-to-GDSII design flow for monolithic 3D IC. First, the Register-transfer-level (RTL) codes of the benchmark circuits were synthesized using Cadence RTL Compiler. The resulting gate-level netlist was then used in the physical design process. For the physical design process, floorplanning, placement, routing and timing optimizations were performed using Cadence Encounter. Finally, the timing, power and congestion analysis were performed for each benchmark circuits.

The industrial circuits used as benchmark designs for this study are as follows: (1) advanced encryption standard (AES) and data encryption standard (DES) circuits are encryption engines used in security systems, (2) joint photographic experts group (JPEG) encoder circuit is a hardware compressor for digital images, and (3) VM256 is a partial-sum-add-based 256-bit integer multiplier. A summary of the T-MIC implementation of the benchmark circuits is presented in Table I. As shown, with T-MIC, the footprint reduction is around 40% and the wirelength reduction is between 15-30%.



(a) 2D (b) T-MIC

Fig. 1. Routing congestion difference between 2D and T-MIC for the AES benchmark circuit. The white X mark indicates the design rule violations caused by routing congestion.

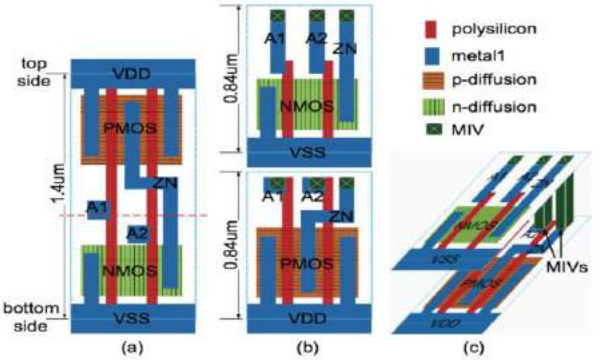


Fig. 2. Conversion of a 2D IC standard cell layout to a monolithic 3D IC design: a) Original 2D cell, b) After cut and flipping 2D cell, c) 3D cell [7]

Table I
Benchmark circuits and synthesis results using transistor-level monolithic technology. The parenthesis values represent the percentage difference of transistor-level monolithic technology over 2D technology.

Circuit	# Cells	# Nets	Total wirelength (μm)	Area (μm^2)
AES	14229	16313	185614 (-28.5%)	16212 (-41.7%)
DES	51370	61916	589760 (-21.5%)	66129 (-41.5%)
M256	192940	231376	6131934 (-15.6%)	547510 (-40.2%)
Jpeg	349729	447165	6550103 (-24.1%)	660794 (-39.9%)

III. Technology Mapping Factors Contributing to Routing Congestion

Routing congestion is the result of increased routing demand in a specific area. The routing supply and demand for that area is determined by the number and types of cell that are chosen for a given area. Complex standard cells are designed to improve the performance of a circuit by decreasing the footprint. Therefore, complex cells (e.g. AOI) tend to have fewer routing resources than simple cells (e.g. NAND2) because they are more compact. However, not all complex

cells increase routing congestion. Therefore, to evaluate the impact of complex cells, each design was synthesized, placed and routed using NAND-only implementation first, i.e. NAND and INV cells only, then compared with the 3D full standard cell library implementation. The NAND-only implementations resulted in a higher chip density and also significantly reduced congestion with the same design constraints. This result indicated that the type of cells used in the design process plays a major role in routing congestion. However, the NAND-only implementation comes with some performance penalties and an increase in cell usage, which results in a larger footprint.

The gains made by using these complex cells are limited by the ability to route the signals of the cells in a smaller space. The NAND-onl implementation was used as a base-function to identify which standard cells contributes the most to routing congestion when included in the design process. All the standard cells, in the 3D library, are introduced to the base-function implementation one by one and each benchmark circuit was then synthesized, placed and routed. Table II interprets the result after analyzing the contribution of each standard cell to congestion once introduced to the base-function design. All the standard cells in the congestion category do not exhibit congestion in all the benchmark circuits. This proved that the type of cells used in the design contribute to routing violations.

All the logic gates categorized under congestion-free in Table II can be synthesized, placed and routed together without incurring congestion. This new implementation is not a recommended solution; however, it indicates that 3D designs are not reaching optimal solution due to congestion. In table III, it shows that using congestion free gates enable denser integration and also lower cell area in some designs.

This new design can be implemented in the same placement utilization as the base-function implementation, which results in a finer integration. However, if any of the cells from the congestion category is introduced, the design will exhibit routing violations. And when this occurs, the utilization must be lowered to accommodate the routing demand, lowering the circuit density integration. Therefore, to utilize all the standard cells in the library, it is vital to identify the cause of congestion in those certain cells.

Table II.
Standard cells under "Congestion-free" can be used together in higher integration designs without incurring routing congestion. While, standard cells under "Congestion" are logic gates that mostly contribute to routing congestion.

Congestion	Congestion-free
AOI	NAND2
OAI	INV
XOR	HA
NOR3	MUX
NOR4	XNOR
AND2	OR
AND4	NOR2X4
NAND3	
NAND4	

IV. Cell Layout Factors Contributing to Routing Congestion

In order to identify the cause of congestion in certain cells, it is necessary to examine the difference between the two categories shown in Table II. Routing violations in T-MIC designs occurs due to limited routing resources, which is dependent on the type of cell being used. The routing resource can be exacerbated by higher number of pins in a specific area and limited access to those pins.

There are two factors that play significant role in routing access: (1) location of the pin and (2) area of the standard cell. Elaborating on the two cell categories shown in Table V, the standard cells categorized under congestion-free tend to follow the recommended guideline for the creation of pins and vias. In the process to allow easier access to pins, it is recommended that the location of pins be staggered horizontally [11]. Despite the guideline being set for 2D standard cells, it is not enforced due to the abundant routing area. However, in T-MIC designs, the cell area are reduced by 40%; thus, making pin staggering necessary for 3D standard cells to maintain routability. Following this 2D pin and via creation guideline for all 3D standard cells will offer higher degree of flexibility for routing. However, pin staggering might not be possible in all cases, therefore, it is recommended to spread out the pins to minimize the pin density within an area. Standard cells in the congestion-free category either have a staggered horizontally pins and or have a larger cell area where the pins are spread out to ensure easier access and routability of wires. From these study, it can be concluded that most complex cells in T-MIC need to be redesigned to allow greater utilization.

An example of how the routing resources are reduced for monolithic 3D IC is shown in Fig. 3. In part b of the figure, the NAND gate is folded, which decrease the number of routing tracks within the cell area from 9 horizontal tracks to 6 horizontal tracks (the number of vertical tracks are the same). Additionally, the input pins A1 and A2 are located on the same horizontal track, which means that one of the pins would have to use a vertical routing track in order to switch to another horizontal track. Staggering the pins would regain a vertical routing track.

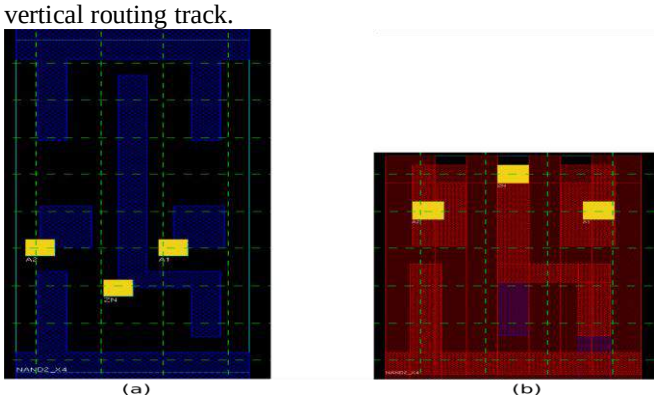


Fig.3. Abstract layout view of a 2-input NAND gate with 4X drive strength: (a) 2D version [10] and (b) 3D version [7]. The green dashed lines represent

the routing resources available over the top of each cell. The blue polygons represent the metal 1 layer and the red polygons represent the metal 2 layer.

Table III.

Cell selection can be used to overcome congestion and achieve finer integration. "CF" indicates the use of congestion free gates shown above.

Circuit	Utilization	Area (μm^2)	Leakage Power (mW)	Total Power (mW)
AES	71%	16212	0.44	14.82
AES CF	89%	16080	0.54	15.1
DES	80%	66219	1.72	180.1
DES CF	90%	67047	1.7	163.8
M256	40%	578535	5.95	230
M256 CF	41%	706872	8.9	330
JPEG	60%	660794	9.83	861
JPEG CF	68%	691213	13	947

V. Floorplanning Factors That Impact Routing Congestion

This paper proposes a way to alleviate congestion, during floorplanning, by manipulating the row spacing. The row spacing parameter determines the space between two adjacent rows of cells. The further the rows are from each other, the greater the available area for routing of cells. Therefore, row spacing can be used to alter the routability of a design. By manipulating the row spacing, higher density designs can be achieved without incurring congestion. While altering the row spacing increases the chip area, it also increases the chip utilization which can mitigate the area penalty. Actually, in some cases (e.g., AES, M256), the chip area of the implementation that uses row spacing is smaller than the implementation that does not. These results, shown in Table IV, further illustrate the point that the algorithms used to route 2D IC do not perform well for monolithic 3D IC. However, in most cases, increasing row spacing comes with slight performance penalties. As shown in Table IV, row spacing minimizes the performance benefits gained by T-MIC designs.

VI. Summary and Future Work

In this paper, it was shown that routing congestion significantly impacts the design of monolithic 3D IC by restricting the feasible solution space. Routing congestion is a major problem in transistor-level monolithic 3D ICs that curtails the performance, integration, yield, and cost of the circuit. Decreasing the placement utilization, increasing the row spacing or adding more metal layers can mitigate routing congestion. These options can relieve congestion without redesigning the cell but have performance penalties that can be prohibitive. Routing congestion could potentially be improved be redesigning the cells, since most complex standard cells in the T-MIC 3D IC library had limited access to pins. In future work, the complex cells will be redesigned to improve their routability.

Table IV.

Row spacing can be used to overcome congestion and achieve finer integration. R0.1 implies there is a 0.1µm row spacing between adjacent row of cells.

Circuit	Utilization	Area (μm^2)	Leakage Power (mW)	Total Power (mW)
AES	71%	16212	0.44	14.82
AES R0.1	86%	16155	0.44	14.85
DES	80%	66219	1.72	180.1
DES R0.1	90%	71055	1.67	181.8
M256	40%	578535	5.95	230
M256 R0.2	65%	490825	5.92	230.7
JPEG	60%	660794	9.83	861
JPEG R0.3	69%	969131	9.65	871

References

[1] Bobba, S.; Chakraborty, A.; Thomas, O.; Batude, P.; Ernst, T.; Faynot, O.; Pan, D.Z.; De Micheli, G., "CELONCEL: Effective design technique for 3-D monolithic integration targeting high performance integrated circuits," Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific , vol., no., pp.336,343, 25-28 Jan. 2011

[2] Chang Liu; Sung-Kyu Lim, "A design tradeoff study with monolithic 3D integration," *Quality Electronic Design (ISQED), 2012 13th International Symposium on* , vol., no., pp.529,536, 19-21 March 2012R

[3] Or-Bach, Z., "The monolithic 3D advantage: Monolithic 3D is far more than just an alternative to 0.7x scaling," *3D Systems Integration Conference (3DIC), 2013 IEEE International* , vol., no., pp.1,7, 2-4 Oct. 2013M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989

[4] Sylvester, D.; Keutzer, K., "Getting to the bottom of deep submicron," *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on* , vol., no., pp.203,211, 8-12 Nov. 1998

[5] Xiang Qiu; Marek-Sadowska, M., "Routing Challenges for Designs With Super High Pin Density,"*Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* , vol.32, no.9, pp.1357,1368, Sept. 2013

[6] Young-Joon Lee; Limbrick, D.; Sung Kyu Lim, "Power benefit study for ultra-high density transistor-level monolithic 3D ICs," *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE* , vol., no., pp.1,10, May 29 2013-June 7 2013

[7] Young-Joon Lee; Sung Kyu Lim, "Ultrahigh Density Logic Designs Using Monolithic 3-D Integration," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* , vol.32, no.12, pp.1892,1905, Dec. 2013

[8] Bo Hu; Marek-Sadowska, M., "Congestion minimization during placement without estimation," *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on* , vol., no., pp.739,745, 10-14 Nov. 2002

[9] N. Selvakkumaran, P. N. Parakh, and G. Karypis, "Perimeter-degree: a priori metric for directly measuring and homogenizing interconnections complexity in multilevel placement," in ACM/IEEE international Workshop on System Level Interconnect Prediction, 2003m pp 53-59

[10] Nangate, “Nangate 45nm open cell library,” [online]. Availble: http://www.nangate.com/openlibrary

[11] Cadence. Cadence Abstract Generator User Guide, Product Version 6.1.5, January 2011

A Fast Analytic Approach to the Collapsing and Verification of Threshold Logic Circuits

Nian-Ze Lee, Hao-Yuan Kuo, Yi-Hsiang Lai, and Jie-Hong R. Jiang
Department of Electrical Engineering / Graduate Institute of Electronics Engineering
National Taiwan University, Taipei 10617, Taiwan

ABSTRACT

Threshold logic circuits gain increasing attention due to their feasible realization with emerging technologies and their strong bind to neural network applications. In this paper, for logic synthesis we formulate the fundamental operation of collapsing threshold logic gates, not addressed by prior efforts. A necessary and sufficient condition of collapsibility is obtained for linear combination of two threshold logic gates, and an analytic approach is proposed for fast circuit transformation. On the other hand, for equivalence verification we propose a linear time translation from threshold logic circuits to pseudo-Boolean constraints, in contrast to prior exponential translation costs. Experimental results demonstrate the effectiveness of circuit transformation by the collapse operation and the memory efficiency of equivalence verification by our pseudo-Boolean translation.

1. INTRODUCTION

While the continuation of Moore's law slows down, different computation paradigms have been considered as possible substitutions for CMOS technologies. Among these possibilities, threshold logic (TL) circuits gain increasing attention due to their strong bind to neural network applications [11, 7] and feasible realization with emerging technologies, such as spintronics, memristors, resonant tunneling devices (RTD), quantum cellular automata (QCA), and single electron transistors (SET) [1, 4, 15].

As the technologies realizing threshold logic become more viable than before, the synthesis and verification of TL circuits are important topics of research to support large scale system construction. Among prior synthesis endeavors, [21] splits a Boolean network into unate nodes and applies linear programming to derive corresponding weights and thresholds; [19] proposes a decomposition algorithm that works directly on truth tables and applies a binate splitting heuristic; a tree matching method is used in [5] to synthesize TL circuits; in [17], implicant-implicit algorithms are proposed to improve synthesis performance; [14] starts from the and-inverter-graph (AIG) and utilizes the well-developed technology mapper in [23] to map the AIG with cuts that are threshold functions. Other efforts on fast identification of TL functions [16], rewiring [10] TL circuits, merging TL gates [2], among others, have also been investigated.

For formal verification of threshold logic circuits, prior work [6] and [22] proposed translation techniques from threshold logic gates to binary decision diagrams (BDD) and conjunctive normal form (CNF) formulas, respectively. They suffer from translation complexity exponential in the fanin size of a threshold logic gate and are not scalable to large designs.

In this paper, the fundamental operation of collapsing (or composing) two threshold logic gates is formulated, which subsumes the restricted merging operations in [2]. Building upon a necessary and sufficient condition of linear composition, an analytic approach is proposed for fast circuit transformation. The collapse (or composition) operation may be combined with other existing techniques to form useful scripts for threshold logic optimization. Moreover, a linear time translation from threshold logic circuits to pseudo Boolean (PB) constraints amenable for formal verification on threshold logic circuits is proposed. Unlike most previous synthesis and verification methods impose restriction on the fanin size of a threshold logic gate, our proposed collapse operation and circuit-to-PB-constraint conversion are not limited to threshold gate sizes. Experiments show promising benefits of our proposed methods.

Collapsing threshold logic gates may potentially result in circuits with high-fanin gates, whose practicality might seem unclear. However we mention a key application to justify the usefulness of collapse operation. In machine learning and brain emulation applications, a neuron in a neural network can have a large number of fanins. For example, a neuron of a deep convolutional neural network for image classification may have 2,048 inputs [9]. It is therefore common for neuromorphic chips to support high-fanin neurons, e.g., the IBM TrueNorth chip [12] implements programmable neurons with up to 256 inputs and may serve as a feasible platform to realize threshold gates with large fanin sizes. As prior work on threshold logic synthesis mostly considers threshold gates up to a few fanins, e.g., 8 inputs in [14], the collapse operation may help relax the fanin-size restriction for circuits mapped into neuromorphic architectures. On the other hand, the collapse (namely, composition) operation can be used as an elementary command, dual to the decomposition operation, in a synthesis script to transform threshold logic circuits for optimization, similar to its Boolean counterpart used in conventional logic synthesis, such as command `eliminate` in `SIS` [18].

The rest of this paper is organized as follows: preliminaries are given in Section 2; the proposed collapse operation and linear translation technique are formulated in Section 3 and 4. Two applications, threshold network collapsing and equivalence checking, are discussed in Section 5. Experimental results in Section 6 show the effectiveness of collapse operation and the memory efficiency of equivalence verification by the proposed linear translation. Section 7 concludes this paper.

2. PRELIMINARIES

A *threshold function* $f : \mathbb{B}^n \to \mathbb{B}$ over variables $(x_1, \ldots, x_n)$ specified by a vector of constant weights $(w_1, \ldots, w_n) \in \mathbb{Z}^n$ and a constant threshold value $T \in \mathbb{Z}$ is a Boolean function that satisfies

$$f(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} w_i x_i \geq T, \\ 0, & \text{otherwise.} \end{cases}$$

A *threshold logic gate* (TLG) v with n inputs $(x_1, \ldots, x_n)$ and one output z_v is a logic unit that realizes some threshold

function $f_v(x_1, \ldots, x_n)$ with weights $(w_1, \ldots, w_n)$ and threshold value T_v. The valuation of its output variable z_v is determined by $f_v(x_1, \ldots, x_n)$. In the sequel, a threshold logic gate v with weights $(w_1, \ldots, w_n)$ and threshold T_v is denoted as $v = [w_1, \ldots, w_n; T_v]$.

A *threshold logic circuit* (abbreviated *TL circuit*, or *TLC*) $G = (V, E)$ is a directed acyclic graph (DAG), where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. An edge $e = (u, v)$ signifies that vertex v refers to vertex u as an input; u is called a *fanin* of v; v is called a *fanout* of u. V_I (resp. V_O) is a nonempty subset of V such that every vertex in V_I (resp. V_O) has no fanins (resp. fanouts). We assume V_I and V_O are disjoint. A vertex $v \in V_I$ (resp. V_O) is referred to as a *primary input* (PI) (resp. *primary output* (PO)). A vertex $v \in V \backslash V_I$ represents a threshold logic gate. In the sequel, we shall not distinguish between a vertex and its corresponding threshold logic gate.

3. COLLAPSING THRESHOLD LOGIC

In this section, we formulate the collapse operation on TL circuits. We derive the feasibility conditions of collapsing in threshold logic. The collapse operation can be seen as the opposite operation of decomposition, which decomposes a TLG into a composite of multiple TLGs.

3.1 Problem Formulation

Problem Formulation 1 (Collapsing in General). *Given a threshold logic circuit $G(V, E)$, let vertices $u, v \in V$ with $(u, v) \in E$ and assume $u = [a_1, \ldots, a_n; T_u]$ with fanins $(x_1, \ldots, x_n)$ and $v = [b_1, \ldots, b_m; T_v]$ with fanins $(y_1, \ldots, y_m)$. Let $y_1 = z_u$. The* TLG collapsing problem of u to v *asks whether there exists a TLG $\tilde{v} = [c_1, \ldots, c_{n+m-1}; T_{\tilde{v}}]$ with fanins $(x_1, \ldots, x_n, y_2, \ldots, y_m)$ such that $f_{\tilde{v}}(x_1, \ldots, x_n, y_2, \ldots, y_m) = f_v(f_u(x_1, \ldots, x_n), y_2, \ldots, y_m)$ for all assignments to variables $(x_1, \ldots, x_n, y_2, \ldots, y_m)$.*

We remark that the general TLG collapsing problem formulated above involves $n+m$ parameters $c_1, \ldots, c_{n+m-1}, T_{\tilde{v}} \in \mathbb{Z}$ to search for legitimate $\tilde{v}$. The large search space $\mathbb{Z}^{n+m}$ imposes expensive computation. To overcome this obstacle, we consider the collapsing of u to v in the special form of a linear combination of u and v as the following formulation states.

Problem Formulation 2. (Collapsing via Linear Combination). *Given a threshold logic circuit $G(V, E)$, let vertices $u, v \in V$ with $(u, v) \in E$ and assume $u = [a_1, \ldots, a_n; T_u]$ with fanins $(x_1, \ldots, x_n)$ and $v = [b_1, \ldots, b_m; T_v]$ with fanins $(y_1, \ldots, y_m)$. Let $y_1 = z_u$. The* TLG collapsing problem of u to v via linear combination *asks whether there exists two parameters $k > 0$, $l > 0$ such that the TLG $\bar{v}(k, l) = [k \cdot a_1, \ldots, k \cdot a_n, l \cdot b_2, \ldots, l \cdot b_m; k \cdot T_u + l \cdot (T_v - b_1)]$ with fanins $(x_1, \ldots, x_n, y_2, \ldots, y_m)$ satisfies $f_{\bar{v}}(x_1, \ldots, x_n, y_2, \ldots, y_m) = f_v(f_u(x_1, \ldots, x_n), y_2, \ldots, y_m)$ for all assignments to variables $(x_1, \ldots, x_n, y_2, \ldots, y_m)$.*

Figure 1 illustrates the above second formulation. Note that by collapsing u to v via their linear combination we can effectively reduce the dimensions of parameter space from $n + m$ to 2. Below we discuss the feasibility conditions of the special collapsing under linear combination. We first assume $b_1 > 0$ and $\{x_1, \ldots, x_n\} \cap \{y_2, \ldots, y_m\} = \emptyset$ (i.e., fanin variables of u and v are disjoint). In Section 3.3, the cases of $b_1 < 0$ and $\{x_1, \ldots, x_n\} \cap \{y_2, \ldots, y_m\} \neq \emptyset$ (i.e., u and v share common fanin variables) will be discussed.

Example 1. *For a concrete example of the problem formulation, consider TLGs $u = [4, 3; 5]$ with fanins (x_1, x_2) and $v = [2, 1; 3]$ with fanins (y_1, y_2), where $y_1 = z_u$. We ask whether there exist feasible parameters (k, l) to collapse u to v via their linear combination $\bar{v}(k, l) = [4k, 3k, l; 5k + l]$. The answer to this question is to be obtained in Section 3.2.*

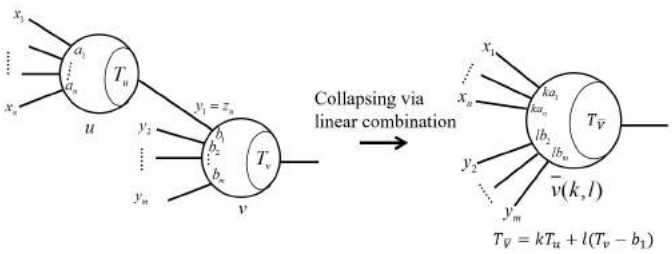


Figure 1: Collapse two TLGs via linear combination.

3.2 Collapsing Feasibility

To facilitate discussion, we define three functions

$$f_u^+(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \sum_{i=1}^{n} a_i x_i,$$

$$f_v^-(y_2, \ldots, y_m) \stackrel{\text{def}}{=} \sum_{j=2}^{m} b_j y_j,$$

$$f_{v \circ u}(x_1, \ldots, x_n, y_2, \ldots, y_m) \stackrel{\text{def}}{=} f_v(f_u(x_1, \ldots, x_n), y_2, \ldots, y_m),$$

and four sets of Boolean assignments

$$\phi_1 \stackrel{\text{def}}{=} \{(\beta_2, \ldots, \beta_m) \mid f_v^-(\beta_2, \ldots, \beta_m) \leq T_v - b_1 - 1\},$$

$$\phi_2 \stackrel{\text{def}}{=} \{(\beta_2, \ldots, \beta_m) \mid f_v^-(\beta_2, \ldots, \beta_m) \geq T_v\},$$

$$\phi_3 \stackrel{\text{def}}{=} \{(\beta_2, \ldots, \beta_m) \mid T_v - b_1 \leq f_v^-(\beta_2, \ldots, \beta_m) \leq T_v - 1\},$$

$$\phi_4 \stackrel{\text{def}}{=} \{(\alpha_1, \ldots, \alpha_n) \mid f_u^+(\alpha_1, \ldots, \alpha_n) \leq T_u - 1\},$$

for $(\alpha_1, \ldots, \alpha_n) \in \mathbb{B}^n$ and $(\beta_2, \ldots, \beta_m) \in \mathbb{B}^{m-1}$. Observe that if $\phi_3 = \emptyset$, then $f_{v \circ u} = f_v$ for all assignments to variables $(x_1, \ldots, x_n, y_2, \ldots, y_m)$, i.e., the valuation of z_v does not depend on that of z_u. In the following derivation, this futile situation is excluded.

The necessary and sufficient conditions for collapsing u to v by a linear combination of weights are stated in Theorem 1.

Theorem 1. *Given two TLGs $u = [a_1, \ldots, a_n; T_u]$ with fanins $(x_1, \ldots, x_n)$ and $v = [b_1, \ldots, b_m; T_v]$, $b_1 > 0$, with fanins $(y_1, \ldots, y_m)$, $y_1 = z_u$, and $\{x_1, \ldots, x_n\} \cap \{y_2, \ldots, y_m\} = \emptyset$, collapsing u to v via linear combination under parameter (k, l) yielding $\bar{v} = [ka_1, \ldots, ka_n, lb_2, \ldots, lb_m; kT_u + l(T_v - b_1)]$ if and only if parameters (k, l) satisfy the following inequalities whose side conditions are met.*

$$l(T_v - b_1 - \max_{\beta \in \phi_1}\{f_v^-\}) \geq k(\max\{f_u^+\} - T_u) + 1, \qquad \text{if } \phi_1 \neq \emptyset$$

$$l(\min_{\beta \in \phi_2}\{f_v^-\} + b_1 - T_v) \geq k(T_u - \min\{f_u^+\}), \qquad \text{if } \phi_2 \neq \emptyset$$

$$k(T_u - \max_{\alpha \in \phi_4}\{f_u^+\}) \geq l(\max_{\beta \in \phi_3}\{f_v^-\} + b_1 - T_v) + 1,$$

We remark that it is difficult to compute the coefficients in the inequalities of Theorem 1. For example, to compute $\max_{\beta \in \phi_1}\{f_v^-\}$, one needs to solve the *subset sum problem*, which is NP-complete [3]. This difficulty may be addressed by considering Theorem 2, which states a set of sufficient conditions and is much more efficient to compute.

Theorem 2. *Given two TLGs $u = [a_1, \ldots, a_n; T_u]$ with fanins $(x_1, \ldots, x_n)$ and $v = [b_1, \ldots, b_m; T_v]$, $b_1 > 0$, with fanins $(y_1, \ldots, y_m)$, $y_1 = z_u$, and $\{x_1, \ldots, x_n\} \cap \{y_2, \ldots, y_m\} = \emptyset$, collapsing u to v via linear combination under parameter*

(k,l) *yielding* $\bar{v} = [ka_1, \ldots, ka_n, lb_2, \ldots, lb_m; kT_u + l(T_v - b_1)]$ *if parameters* (k,l) *satisfy the following inequalities whose side conditions are met.*

$$\begin{aligned} & l \geq k(\max\{f_u^+\} - T_u) + 1, && \text{if } \phi_1 \neq \emptyset \\ & lb_1 \geq k(T_u - \min\{f_u^+\}), && \text{if } \phi_2 \neq \emptyset \\ & k \geq l(b_1 - 1) + 1, \end{aligned}$$

Notice that, unlike those in Theorem 1, the max and min operations in Theorem 2 range over all input assignments and do not need to be constrained by ϕ_1 and ϕ_2. Hence, the coefficients, $\max\{f_u^+\}$ and $\min\{f_u^+\}$, in these inequalities of Theorem 2 can be computed in time linear to the fanin size of the TLG, which enhances computational efficiency of collapse operation. A trade-off between the identification power of threshold functions and computational complexity is provided by Theorems 1 and 2.

PROOF. To prove the above two theorems, we search for the conditions such that $f_{v \circ u}$ equals $f_{\bar{v}}$ under all assignments $(\alpha_1, \ldots, \alpha_n, \beta_2, \ldots, \beta_m)$ for variables $(x_1, \ldots, x_n, y_2, \ldots, y_m)$. There are three cases to discuss: First, $(\beta_2, \ldots, \beta_m) \in \phi_1$, i.e., $f_v^-(\beta_2, \ldots, \beta_m) \leq T_v - b_1 - 1$. Second, $(\beta_2, \ldots, \beta_m) \in \phi_2$, i.e., $f_v^-(\beta_2, \ldots, \beta_m) \geq T_v$. Third, $(\beta_2, \ldots, \beta_m) \in \phi_3$, i.e., $T_v - b_1 \leq f_v^-(\beta_2, \ldots, \beta_m) \leq T_v - 1$. Their corresponding derivations are obtained in Lemmas 1 to 3. ■

LEMMA 1. *For every assignment* $(\alpha_1, \ldots, \alpha_n, \beta_2, \ldots, \beta_m)$ *to* $(x_1, \ldots, x_n, y_2, \ldots, y_m)$ *with* $(\beta_2, \ldots, \beta_m) \in \phi_1$*, i.e.,* $f_v^-(\beta_2, \ldots, \beta_m) \leq T_v - b_1 - 1$*,*

iff-condition *:* $f_{v \circ u} = f_{\bar{v}} \iff l(T_v - b_1 - \max_{\beta \in \phi_1}\{f_v^-\}) \geq k(\max\{f_u^+\} - T_u) + 1$

if-condition *:* $f_{v \circ u} = f_{\bar{v}} \Longleftarrow l \geq k(\max\{f_u^+\} - T_u) + 1$

PROOF. For any assignment in ϕ_1, we have $f_{v \circ u} = 0$. In the following derivation, $f_u^+ > T_u$ is assumed, since if $f_u^+ \leq T_u$, $f_{\bar{v}} = f_{v \circ u}$ trivially.

$$\begin{aligned} f_{\bar{v}} = 0 \iff & kf_u^+ + lf_v^- < kT_u + l(T_v - b_1) \\ \iff & k(f_u^+ - T_u) < l(T_v - b_1 - f_v^-) \\ \iff & \frac{k}{l} < \frac{T_v - b_1 - f_v^-}{f_u^+ - T_u} \\ \iff & \frac{k}{l} < \min\{\frac{T_v - b_1 - f_v^-}{f_u^+ - T_u}\} \\ \iff & \frac{k}{l} < \frac{T_v - b_1 - \max_{\beta \in \phi_1}\{f_v^-\}}{\max\{f_u^+\} - T_u} \\ \Longleftarrow & \frac{k}{l} < \frac{1}{\max\{f_u^+\} - T_u} \end{aligned}$$

From the above derivation, we have

$$f_{v \circ u} = f_{\bar{v}} \iff l(T_v - b_1 - \max_{\beta \in \phi_1}\{f_v^-\}) \geq k(\max\{f_u^+\} - T_u) + 1$$
$$f_{v \circ u} = f_{\bar{v}} \Longleftarrow l \geq k(\max\{f_u^+\} - T_u) + 1$$

■

EXAMPLE 2. *Continue Example 1. By* $\phi_1 \neq \emptyset$*, the conditions in Lemma 1 should be satisfied by parameter* (k,l)*. The conditions are*

$$f_{v \circ u} = f_{\bar{v}} \iff l(3 - 2 - 0) \geq k(7 - 5) + 1$$
$$f_{v \circ u} = f_{\bar{v}} \Longleftarrow l \geq k(7 - 5) + 1$$

Suppose the condition $l \geq 2k + 1$ *is violated, e.g., by setting* $l = 2k$*. The resulting TLG* $\bar{v}$ *becomes* $[4, 3, 2; 7]$*. As predicted by Lemma 1, a discrepancy must occur. Indeed, we can find* $(x_1, x_2, y_2) = (1, 1, 0)$ *such that* $f_{\bar{v}} = 1$ *and* $f_{v \circ u} = 0$*.*

LEMMA 2. *For every assignment* $(\alpha_1, \ldots, \alpha_n, \beta_2, \ldots, \beta_m)$ *to* $(x_1, \ldots, x_n, y_2, \ldots, y_m)$ *with* $(\beta_2, \ldots, \beta_m) \in \phi_2$*, i.e.,* $f_v^-(\beta_2, \ldots, \beta_m) \geq T_v$*:*

iff-condition *:* $f_{v \circ u} = f_{\bar{v}} \iff l(\min_{\beta \in \phi_2}\{f_v^-\} + b_1 - T_v) \geq k(T_u - \min\{f_u^+\})$

if-condition *:* $f_{v \circ u} = f_{\bar{v}} \Longleftarrow lb_1 \geq k(T_u - \min\{f_u^+\})$

PROOF. For any assignment in ϕ_2, we have $f_{v \circ u} = 1$. In the following derivation, $f_u^+ < T_u$ is assumed, since if $f_u^+ \geq T_u$, $f_{\bar{v}} = f_{v \circ u}$ trivially.

$$\begin{aligned} f_{\bar{v}} = 1 \iff & kf_u^+ + lf_v^- \geq kT_u + l(T_u - b_1) \\ \iff & k(f_u^+ - T_u) \geq l(T_v - b_1 - f_v^-) \\ \iff & \frac{k}{l} \leq \frac{f_v^- + b_1 - T_v}{T_u - f_u^+} \\ \iff & \frac{k}{l} \leq \min\{\frac{f_v^- + b_1 - T_v}{T_u - f_u^+}\} \\ \iff & \frac{k}{l} \leq \frac{\min_{\beta \in \phi_2}\{f_v^-\} + b_1 - T_v}{T_u - \min\{f_u^+\}} \\ \Longleftarrow & \frac{k}{l} \leq \frac{b_1}{T_u - \min\{f_u^+\}} \end{aligned}$$

From the above derivation, we have

$$f_{v \circ u} = f_{\bar{v}} \iff l(\min_{\beta \in \phi_2}\{f_v^-\} + b_1 - T_v) \geq k(T_u - \min\{f_u^+\})$$
$$f_{v \circ u} = f_{\bar{v}} \Longleftarrow lb_1 \geq k(T_u - \min\{f_u^+\})$$

■

EXAMPLE 3. *Continue Example 2. Because* $\phi_2 = \emptyset$*, the conditions in Lemma 2 need not be imposed for parameter* (k,l)*.*

LEMMA 3. *For every assignment* $(\alpha_1, \ldots, \alpha_n, \beta_2, \ldots, \beta_m)$ *to* $(x_1, \ldots, x_n, y_2, \ldots, y_m)$ *with* $(\beta_2, \ldots, \beta_m) \in \phi_3$*, i.e.,* $T_v - b_1 \leq f_v^-(\beta_2, \ldots, \beta_m) \leq T_v - 1$*:*

iff-condition *:* $f_{v \circ u} = f_{\bar{v}} \iff k(T_u - \max_{\alpha \in \phi_4}\{f_u^+\}) \geq l(\max_{\beta \in \phi_3}\{f_v^-\} + b_1 - T_v) + 1$

if-condition *:* $f_{v \circ u} = f_{\bar{v}} \Longleftarrow k \geq l(b_1 - 1) + 1$

PROOF. For any assignment in ϕ_3, we have $kf_u^+ + lf_v^- \geq kT_u + l(T_v - b_1) \iff f_u^+ \geq T_u$. In the following derivation, $f_u^+ < T_u$ is assumed, since if $f_u^+ \geq T_u$, $f_{\bar{v}} = f_{v \circ u}$ trivially. Under the assumption of $f_u^+ < T_u$, $f_{v \circ u} = 0$.

$$\begin{aligned} f_{\bar{v}} = 0 \iff & kf_u^+ + lf_v^- < kT_u + l(T_v - b_1) \\ \iff & k(f_u^+ - T_u) < l(T_v - b_1 - f_v^-) \\ \iff & \frac{k}{l} > \frac{f_v^- + b_1 - T_v}{T_u - f_u^+} \\ \iff & \frac{k}{l} > \max\{\frac{f_v^- + b_1 - T_v}{T_u - f_u^+}\} \\ \iff & \frac{k}{l} > \frac{\max_{\beta \in \phi_3}\{f_v^-\} + b_1 - T_v}{T_u - \max_{\alpha \in \phi_4}\{f_u^+\}} \\ \Longleftarrow & \frac{k}{l} > b_1 - 1 \end{aligned}$$

From the above derivation, we have

$$f_{v \circ u} = f_{\bar{v}} \iff k(T_u - \max_{\alpha \in \phi_4}\{f_u^+\}) \geq l(\max_{\beta \in \phi_3}\{f_v^-\} + b_1 - T_v) + 1$$
$$f_{v \circ u} = f_{\bar{v}} \Longrightarrow k \geq l(b_1 - 1) + 1$$

■

EXAMPLE 4. *Continue Example 3. Because $\phi_3 \neq \emptyset$, the conditions in Lemma 3 should be satisfied by parameter (k,l). The conditions are*

$$f_{v \circ u} = f_{\bar{v}} \iff k(5-4) \geq l(1+2-3)+1$$
$$f_{v \circ u} = f_{\bar{v}} \Longleftarrow k \geq l(2-1)+1$$

The condition $k \geq 1$ should not be violated since we consider $k > 0, l > 0$ in Problem Formulation 2. From the derived iff-constraints (i.e., one from Lemma 1 and the other from Lemma 3) in Examples 2 to 4, the two iff-conditions can be satisfied, e.g., by $(k,l) = (1,3)$, which asserts that u can be collapsed to v yielding $\bar{v} = [4,3,3;8]$. On the other hand, the derived if-constraints (one from Lemma 1 and the other from Lemma 3) in Examples 2 to 4 together yield no solution due to their under-approximation of (k,l) solutions.

We remark that the merging operations proposed in [2] are special cases of the collapse operation formulated above. For example, the AND *gate-based merging* in [2], which considers to "merge" an AND gate v with one of its fanins u, is equivalent to collapsing u to v in our formulation. Theorem 1 and 2 can be applied to derive feasible parameters to collapse u to v via their linear combination.

Due to the linearity of the inequality constraints, an advantage of the proposed collapsing method lies in that, after coefficients in the inequalities are computed, analytic solutions for (k,l) can be obtained. We remark that, in general, there may be multiple feasible solutions for (k,l), and those with smaller magnitude for k and l are preferred since the collapsed TLG would have smaller weights and threshold values. A possible approach to derive feasible solutions analytically is to find the intersecting point of these constraints and rounding it up to integers. For example, suppose two inequalities $l \geq k(\max\{f_u^+\} - T_u) + 1$ and $k \geq l(b_1 - 1) + 1$ are derived based on the if-conditions, one can solve for the intersecting point of the two lines analytically. Let the coordinates of the intersecting point be (l^*, k^*). A feasible (k,l) combination can be derived by first rounding l^* up to $\lceil l^* \rceil$ and substituting $\lceil l^* \rceil$ into $k = l(b_1 - 1) + 1$ to obtain the corresponding k coordinate. Such analytic search results in a fast collapsing computation. Also, observe that in Examples 2 to 4, if u and v are canonicalized to $[1,1;2]$ before collapsing, both the iff-conditions and the if-conditions can yield the solution $(k,l) = (1,1)$ to collapse u to v resulting in $\bar{v} = [1,1,1;3]$. Applying canonicalization in [10] may enhance the feasibility of collapse operation.

3.3 Extension to Negative Weight and Non-disjoint Fanin

Theorems 1 and 2 assume $b_1 > 0$ and $\{x_1, \ldots, x_n\} \cap \{y_2, \ldots, y_m\} = \emptyset$. We show how to handle the cases when $b_1 < 0$ and $\{x_1, \ldots, x_n\} \cap \{y_2, \ldots, y_m\} \neq \emptyset$ by an example. The basic operations to complement a fanin variable and invert a TLG can be found in [13].

EXAMPLE 5. *Let $u = [1,1;2]$ with fanins (x_1, x_2) and $v = [-1,-1,0]$ with fanins (y_1, y_2), where $y_1 = z_u$ and $y_2 = x_2$. We first make the negative weight positive by complementing y_1, resulting in $v^* = [1,-1;1]$. The inverter generated by complementing y_1 is then combined with u, yielding $u^* = [-1,-1;-1]$. To collapse u to v, it is equivalent to collapse u^* to v^*, where the weight of y_1 is inverted to 1, and Theorem 1 may be applied by first assuming that x_2 and y_2 are different variables. In this example, one can verify that $(k,l) = (1,2)$ is a feasible solution. As a result, the collapsed TLG is $\bar{v} = [-1,-1,-2;-1]$ with fanins (x_1, x_2, y_2). Since x_2 and y_2 are actually the same variable, their weights should be summed up and the resulting TLG is $\bar{v} = [-1,-3;-1]$ with fanins (x_1, x_2).*

4. TRANSLATING TL CIRCUIT TO PSEUDO BOOLEAN CONSTRAINTS

As an analogy to Tseitin's conversion translating Boolean logic circuits to conjunctive normal form (CNF) formulas [20], we propose a linear-time translation from TL circuits to pseudo Boolean (PB) constraints. Our translation converts every gate in a TL circuit to exactly two PB constraints, both of which are of length linear to the fanin size of the TLG. The conjunction of all the PB constraints gives the consistency condition of variable assignments on the entire circuit. Our translation may facilitate formal reasoning on TL circuits with PB constraint solving, similar to reasoning on Boolean logic circuits with satisfiability (SAT) solving. The conversion is detailed as follows.

Given a TLG $u = [a_1, \ldots, a_n; T]$ with input variables $(x_1, \ldots, x_n)$ and output variable y, we would like to derive PB constraints stating the relation $y \leftrightarrow (\sum_{i=1}^n a_i x_i \geq T)$, which is unfortunately not in the right format of PB constraints. The following theorem provides a solution.

THEOREM 3. *Given a TLG $u = [a_1, \ldots, a_n; T]$ with input variables $(x_1, \ldots, x_n)$ and output variable y, a truth assignment to variables $x_1, \ldots, x_n, y$ satisfies $y \leftrightarrow \sum_{i=1}^n a_i x_i \geq T$ if and only if it satisfies the following two pseudo Boolean constraints:*

$$(M - T - 1)y + (T - 1 - \sum_{i=1}^{n} a_i x_i) \geq 0 \quad (1)$$

$$(\sum_{i=1}^{n} a_i x_i - T) + (T - m)(1 - y) \geq 0, \quad (2)$$

where M (resp. m) is the sum of all positive (resp. negative) weights among $\{a_1, \ldots, a_n\}$. M (resp. m) is defined to be 0 if all weights are non-positive (resp. non-negative).

PROOF. Let $A \subseteq \mathbb{B}^{n+1}$ be the set of valuations to variables $(x_1, \ldots, x_n, y)$ that satisfy $y \leftrightarrow \sum_{i=1}^n a_i x_i \geq T$; let $B \subseteq \mathbb{B}^{n+1}$ be the set of valuations to variables $(x_1, \ldots, x_n, y)$ that satisfy both Eq. (1) and (2). We show $A = B$ by establishing $A \subseteq B$ and $B \subseteq A$ as follows.

To show $A \subseteq B$, consider an arbitrary $\tilde{z} = (\alpha_1, \ldots, \alpha_n, \beta) \in A$. There are two cases: When $\sum_{i=1}^n a_i \alpha_i \geq T$ and $\beta = 1$, Eq. (1) is satisfied by $\tilde{z}$ because by substituting $\tilde{z}$ into Eq. (1) we have $M - \sum_{i=1}^n a_i \alpha_i \geq 0$, which holds since M is defined to be the summation of all positive a_i. Also Eq. (2) is satisfied by $\tilde{z}$ because by substituting $\tilde{z}$ into Eq. (2) we have $\sum_{i=1}^n a_i \alpha_i \geq T$, which is our assumption. When $\sum_{i=1}^n a_i \alpha_i \leq T - 1$ and $\beta = 0$, Eq. (1) is satisfied by $\tilde{z}$ because by substituting $\tilde{z}$ into Eq. (1) we have $\sum_{i=1}^n a_i \alpha_i \leq T - 1$, which is our assumption. Eq. (2) is satisfied by$\tilde{z}$ because by substituting $\tilde{z}$ into Eq. (2) we have $\sum_{i=1}^n a_i \alpha_i - m \geq 0$, which holds since m is defined to be the summation of all negative a_i. From the above two cases, we have shown that $\tilde{z} \in B$, and thus $A \subseteq B$.

To show $B \subseteq A$, consider an arbitrary $\tilde{z} = (\alpha_1, \ldots, \alpha_n, \beta) \in B$. There are two cases: When $\sum_{i=1}^n a_i \alpha_i \geq T$, Eq. (1) implies $y = 1$ because $T - 1 - \sum_{i=1}^n a_i x_i < 0$, and under $y = 1$ Eq. (2) is automatically satisfied. Hence, $\tilde{z} \in A$. When $\sum_{i=1}^n a_i x_i \leq T - 1$, Eq. (2) implies $y = 0$ because $\sum_{i=1}^n a_i x_i - T < 0$, and under $y = 0$ Eq. (1) is satisfied automatically. Hence, $\tilde{z} \in A$. From the above two cases, we have shown that $\tilde{z} \in A$, and thus $B \subseteq A$.

With the cases analyzed above, the theorem follows. ■

EXAMPLE 6. *Consider a TLG $u = [1,1,2;2]$ with input variables (x_1, x_2, x_3) and output variable y. u can be translated into two PB constraints $y + 1 - x_1 - x_2 - 2x_3 \geq 0$ and $x_1 + x_2 + 2x_3 - 2 + 2(1 - y) \geq 0$ by Theorem 3.*

Prior work [6] translates a TLG into its maximally factored form and then converts it into Boolean expression diagrams

(BED) [8]. The converted BED is then transformed to BDD for further Boolean reasoning. However, BDD representation tends to suffer from the memory explosion issue when the number of primary inputs is large. The method may not be scalable. Prior work [22] applies a path search approach to enumerate the product terms for a sum-of-product (SOP) expression of a TLG. The SOP expression is then converted to a CNF formula for SAT solving based Boolean reasoning. It also has the memory explosion issue, especially when the fanin size of a TLG is large, the SOP expression can blow up. In contrast, our method can translate a TLG in time linear to the fanin size, resulting in a more efficient translation and compact representation of TLG.

5. APPLICATIONS

We apply the proposed collapsing technique and PB constraint translation technique to TL circuit synthesis and verification, respectively.

5.1 Synthesis

Given a TL circuit $G = (V, E)$, we apply the collapse operation to minimize the number of vertices. A TLG u can be eliminated from the circuit if it can be collapsed into all of its fanouts. In fact, one can show the the minimization problem of TLG collapsing is NP-hard. In light of the NP-hardness, we seek a simple heuristic to collapse a TL circuit.

Figure 2 sketches the procedure *CollapseNtk*. When a TLG u can be collapsed to all of its fanouts, the procedure collapses them by calling *CollapseNode* in Figure 3, where parameters (k, l) are computed by *CalKL*, and a new TLG is created by *CreateNode*. In our implementation, we resort to the sufficient conditions stated in Theorem 2 due to its low computation cost. Our empirical experience suggests that the iff-conditions in Theorem 1 offer negligible improvement in gate count reduction, and thus the if-conditions already provide good approximation to the sufficient and necessary criteria for collapse. A TLG is marked if none of its fanin can be collapsed to it; *CollapseNtk* terminates when all of the TLGs are marked.

CollapseNtk has an additional parameter B that constrains the fanout size of a gate when it is considered to be collapsed into its fanouts, in line05 of Figure 2. We apply a strategy of iteratively incrementing the bound in *CollapseIter*. As to be seen in the experiments, compared to collapsing without limitation on the fanout size, i.e., running *CollapseNtk* with $B = \infty$ directly, *CollpseIter* has better performance in terms of minimizing gate count. This phenomenon can be explained by the fact that if no limitation is imposed from the beginning, a gate with large fanout size would be collapsed to all its fanouts earlier if collapsible, resulting in a large number of collapsed gates, which are more difficult to be collapsed with other gates. Instead, if we set a bound to the fanout size and iteratively increases the bound, we can mitigate the increasing non-collapsibility.

5.2 Verification

Given two TL circuits, we may apply our TLG-to-PB-constraints conversion for their equivalence checking. A miter can be built to assert the equivalence relation between two circuits by disjuncting the XORs of corresponding output pairs. For each TLG in the miter, Theorem 3 is applied to translate it into PB constraints. For the added output equivalence circuitry, it can be first translated by Tseitin conversion into a set of clauses in CNF and then further translated for each clause to its PB constraint. Having all the PB constraints are obtained, we set the objective function to maximize the value of the miter output variable. The maximum value is 0 if and only if the two TL circuits under verification are equivalent.

```
CollapseNtk
  input: threshold logic network G = (V, E) and a bound B
  output: collapsed network G' = (V', E')
  begin
  01   unmark every v ∈ V;
  02   while (not all v ∈ V are marked)
  03     foreach v ∈ V
  04       foreach fanin u of v
  05         if |fanouts(u)| ≤ B
  06           if u can be collapsed to all of its fanouts
  07             foreach fanout t of u
  08               w := CollapseNode(u,t);
  09               unmark w;
  10               V := V\{t} ∪ {w};
  11             V := V\{u};
  12             continue; //to next v
  13         if every fanin of v cannot be collapsed to v
  14           mark v;
  15   return (V, E);
  end
```

Figure 2: Algorithm: Collapse TL circuit.

```
CollapseNode
  input: two TLGs u and v
  output: collapsed TLG w if collapsible, or NULL otherwise
  begin
  01   (k, l) := CalKL(u,v);
  02   if k > 0 and l > 0
  03     w := CreateNode(u,v,k,l);
  04     return w;
  05   else
  06     return NULL;
  end
```

Figure 3: Algorithm: Collapse two TLGs.

6. EXPERIMENTAL RESULTS

The collapsing-based synthesis and equivalence verification algorithms discussed in Section 5 were implemented in the ABC environment [23]. The experiments were conducted on a Linux machine with a Xeon 2.3 GHz CPU and 200 GB RAM. ISCAS and ITC benchmark suits were selected for experiments.

The proposed collapse operation can be directly applied to AIGs as an AIG node can be easily translated to a TLG. For example, an AND gate is translated to $[1, 1; 2]$ and a NAND gate is translated to $[-1, -1; -1]$. We experimented on collapsing a functionally reduced AIG (synthesized by command fraig in ABC) into a TL circuit, and the results are shown in Table 1. Columns 4 and 5 report the numbers of AND gates and logic levels of the AIG circuits, respectively; Columns 6, 7, and 8 report the number of TLGs, the number of logic levels of the TL circuits, and the CPU time, respectively, for direct collapsing without fanout bounds (as discussed in Section 5.1); Columns 9, 10, and 11 report the number of TLGs, the number of logic levels of the TL circuits, and the runtime, respectively, for iterative collapsing with incrementing fanout bounds. The numbers in parentheses listed in Columns 6 and 9 (resp. Columns 7 and 10) are the ratios of gate counts (resp. level counts) of the collapsed TL circuits to those of original AIG circuits. Without the increment strategy of fanout bounds, the collapse operation achieves an average of 50% reduction in gate count and 28% reduction in logic level; with the increment strategy, the reduction ratios are further enhanced to 55% in gate count and 30% in logic level.

To compare the synthesis quality with the methods proposed in [2], we also apply our iterative collapsing technique to five largest benchmarks experimented in [2]. The results are shown in Table 2. Columns 2 and 3, with data repeated from [2], report the numbers of AND gates in the benchmark

Table 1: Statistics of collapsing AIG circuits

benchmarks profile			statistics of AIG		statistics of collapsed TLC (w/o ite)			statistics of collapsed TLC (ite)		
circuit	#pi	#po	#AND	#level	#TLG	#level	time (s)	#TLG	#level	time (s)
c3540	50	22	1028 (1.00)	40 (1.00)	514 (0.50)	33 (0.83)	0.00	418 (0.41)	29 (0.73)	0.03
c5315	178	123	1741 (1.00)	38 (1.00)	736 (0.42)	27 (0.71)	0.00	684 (0.39)	23 (0.61)	0.05
c6288	32	32	2334 (1.00)	120 (1.00)	1407 (0.60)	93 (0.78)	0.01	1404 (0.60)	95 (0.79)	0.17
c7552	207	108	1961 (1.00)	29 (1.00)	991 (0.51)	22 (0.76)	0.01	846 (0.43)	19 (0.66)	0.08
s5378	35	49	1362 (1.00)	19 (1.00)	578 (0.42)	14 (0.74)	0.00	537 (0.39)	11 (0.58)	0.03
s9234.1	36	39	1804 (1.00)	33 (1.00)	835 (0.46)	18 (0.55)	0.01	762 (0.42)	18 (0.55)	0.06
s13207	31	121	2605 (1.00)	33 (1.00)	1213 (0.47)	21 (0.64)	0.01	1190 (0.46)	20 (0.61)	0.08
s15850	14	87	3330 (1.00)	46 (1.00)	1601 (0.48)	34 (0.74)	0.01	1479 (0.44)	32 (0.70)	0.10
s35932	35	320	10124 (1.00)	14 (1.00)	5078 (0.50)	9 (0.64)	0.03	4758 (0.47)	8 (0.57)	0.19
s38417	28	106	9062 (1.00)	30 (1.00)	4747 (0.52)	21 (0.70)	0.03	4388 (0.48)	19 (0.63)	0.32
s38584	12	278	11646 (1.00)	34 (1.00)	4981 (0.43)	22 (0.65)	0.04	4639 (0.40)	20 (0.59)	0.32
b04	11	8	534 (1.00)	23 (1.00)	300 (0.56)	17 (0.74)	0.00	284 (0.53)	17 (0.74)	0.02
b12	5	6	996 (1.00)	17 (1.00)	498 (0.50)	12 (0.71)	0.00	438 (0.44)	15 (0.88)	0.03
b14	32	54	5609 (1.00)	65 (1.00)	2866 (0.51)	49 (0.75)	0.02	2565 (0.46)	53 (0.82)	0.26
b15	36	70	8158 (1.00)	65 (1.00)	4028 (0.49)	47 (0.72)	0.05	3667 (0.45)	44 (0.68)	0.29
b17	37	97	26389 (1.00)	93 (1.00)	13379 (0.51)	66 (0.71)	0.17	12027 (0.46)	73 (0.78)	1.25
b18	37	23	77757 (1.00)	132 (1.00)	39733 (0.51)	98 (0.74)	0.42	35343 (0.45)	87 (0.66)	4.70
b19	24	27	156224 (1.00)	136 (1.00)	80621 (0.52)	105 (0.77)	0.87	70765 (0.45)	85 (0.63)	10.18
b20	32	22	11552 (1.00)	66 (1.00)	5929 (0.51)	47 (0.71)	0.05	5284 (0.46)	58 (0.88)	0.51
b21	32	22	11728 (1.00)	70 (1.00)	6017 (0.51)	54 (0.77)	0.06	5381 (0.46)	59 (0.84)	0.53
b22	32	22	17614 (1.00)	68 (1.00)	9071 (0.51)	57 (0.84)	0.09	8028 (0.46)	60 (0.88)	0.94
geomean			(1.00)	(1.00)	(0.50)	(0.72)		(0.45)	(0.70)	

CollapseIter
input: threshold logic network $G = (V, E)$ and a bound B
output: iteratively collapsed network $G' = (V', E')$
begin
01 $i := 1$;
02 **while** $i \leq B$;
03 $(V,E) := CollapseNtk(G(V,E),i)$;
04 $i := i + 1$;
05 **return** (V,E);
end

Figure 4: Algorithm: Iterative collapsing.

circuits and the numbers of TLGs in the synthesized circuits. Columns 4 and 5 report the numbers of AND gates in the benchmark circuits (synthesized by `resyn2` script in `ABC`, as described in [2]) and the numbers of TLGs in the collapsed TL circuits. Apart from the slight mismatch between the numbers in Columns 2 and 4, an average of 43% reduction in gate count was achieved in [2] while an average of 56% reduction was achieved by our collapsing-based synthesis technique, confirming the generality and effectiveness of our formulation.

Table 2: Comparison between methods in [2] and iterative collapsing

	statistics in [2]		statistics of collapsed TLC	
circuit	#AND	#TLG	#AND	#TLG
aes_core	20509	10057	19875	7505
wb_conmax	41070	21956	41163	19626
ethernet	57205	35243	43549	19110
des_perf	71327	42719	69500	27316
vga_lcd	88854	55402	90880	50734
geomean	(1.00)	(0.57)	(1.00)	(0.44)

We further experiment on collapsing the TL circuits synthesized by [14] to evaluate the room for collapsing on optimized TL circuits. The overall experimental flow is as follows.First, threshold logic synthesis method (`ABC &if` mapper) used in [14] is applied to transform benchmark circuits into TL circuits. Second, the algorithm *CollapseIter* is applied to the synthesized TL circuits to collapse TLGs. Two collapse strategies were applied: one with fanout bound B set to infinity and the other with B incrementing from 1 up to 100. After collapsing, equivalence checking based on the proposed TLG-to-PB conversion is applied to verify the equivalence between TL circuits before and after collapsing. A miter is built to assert the equivalence between the collapsed network and the original one. Moreover, we reimplemented the state-of-the-art method, *path search with weight ordering* [22], called `weight order` in our discussion, to compare and evaluate our PB translation method. The miter is translated into a set of clauses by `weight order` and into a set of PB constraints by our translation. SAT solver `Minisat` and PB solver `Minisat+` were adopted to solve the CNF and PB constraints, respectively. In addition to the two methods, we also applied command `cec`, a state-of-the-art combinational equivalent checking (CEC) tool, in `ABC` for equivalence checking the AIGs translated by `weight order` from TLGs to compare.

Table 3 shows the results of collapse operation on TL circuits optimized by [14]. Columns 4, 5, and 6 report the number of TLGs, the number of levels of the TL circuits synthesized by [14], and the CPU time, respectively; Columns 7, 8, and 9 report the number of TLGs, the number of logic levels of the TL circuits, and the CPU time, respectively, for direct collapsing without fanout bounds; Columns 10, 11, and 12 report the number of TLGs, the number of logic levels of the TL circuits, and the CPU time, respectively, for iterative collapsing with incrementing fanout bounds.

The numbers in parentheses listed in Columns 7 and 10 (resp. Columns 8 and 11) are the ratios of gate counts (resp. level counts) of the collapsed TL circuits to those of TL circuits optimized by [14]. On top of the synthesized circuits by [14], the collapse operation obtains an average of 16% reduction in the number of TLGs with runtime less than 7 seconds for all benchmarks. The results show the effectiveness of the proposed collapse operation based on linear composition. The efficiency lies in fast analytic computation of collapsing conditions. However, the collapse operation achieves no reduction in logic level. This phenomenon could be attributed to the well-optimized `ABC &if` mapper applied in [14]. Since the technology mapper has optimized the circuits in terms of delay, it is hard to further reduce logic levels.

Table 4 (resp. Table 5) shows the results of equivalence verification between circuits before and after collapsing without (resp. with) the iterative fanout bound increment. The three verification techniques mentioned above, i.e., CNF-based, AIG-based (CEC), and PB-based, were evaluated in terms of

Table 3: Statistics of collapsing TL circuits

benchmarks profile			statistics of TLC synthesized by [14]			statistics of collapsed TLC (w/o ite)			statistics of collapsed TLC (ite)		
circuit	#pi	#po	#TLG	#level	time (s)	#TLG	#level	time (s)	#TLG	#level	time (s)
c3540	50	22	465 (1.00)	13 (1.00)	1.90	397 (0.85)	13 (1.00)	0.00	399 (0.86)	13 (1.00)	0.03
c5315	178	123	732 (1.00)	10 (1.00)	2.58	643 (0.88)	10 (1.00)	0.00	644 (0.88)	10 (1.00)	0.04
c6288	32	32	1424 (1.00)	29 (1.00)	6.04	1105 (0.78)	29 (1.00)	0.00	1102 (0.77)	29 (1.00)	0.11
c7552	207	108	950 (1.00)	10 (1.00)	4.91	716 (0.75)	10 (1.00)	0.00	713 (0.75)	10 (1.00)	0.06
s5378	35	49	578 (1.00)	5 (1.00)	1.64	489 (0.85)	5 (1.00)	0.00	489 (0.85)	5 (1.00)	0.03
s9234.1	36	39	744 (1.00)	7 (1.00)	2.23	635 (0.85)	7 (1.00)	0.00	631 (0.85)	7 (1.00)	0.04
s13207	31	121	1257 (1.00)	8 (1.00)	2.10	1046 (0.83)	8 (1.00)	0.00	1049 (0.83)	8 (1.00)	0.05
s15850	14	87	1630 (1.00)	10 (1.00)	3.24	1360 (0.83)	10 (1.00)	0.00	1356 (0.83)	10 (1.00)	0.08
s35932	35	320	5878 (1.00)	5 (1.00)	1.84	4299 (0.73)	5 (1.00)	0.01	4299 (0.73)	5 (1.00)	0.14
s38417	28	106	4857 (1.00)	8 (1.00)	5.14	4248 (0.87)	8 (1.00)	0.01	4220 (0.87)	8 (1.00)	0.28
s38584	12	278	4391 (1.00)	8 (1.00)	6.24	4000 (0.91)	8 (1.00)	0.01	3999 (0.91)	8 (1.00)	0.19
b04	11	8	270 (1.00)	9 (1.00)	1.67	235 (0.87)	9 (1.00)	0.00	235 (0.87)	9 (1.00)	0.01
b12	5	6	490 (1.00)	5 (1.00)	2.33	445 (0.91)	5 (1.00)	0.00	445 (0.91)	5 (1.00)	0.03
b14	32	54	2680 (1.00)	13 (1.00)	11.51	2253 (0.84)	13 (1.00)	0.01	2218 (0.83)	13 (1.00)	0.21
b15	36	70	4077 (1.00)	16 (1.00)	14.99	3616 (0.89)	16 (1.00)	0.01	3566 (0.87)	16 (1.00)	0.30
b17	37	97	13009 (1.00)	22 (1.00)	37.48	11279 (0.87)	22 (1.00)	0.05	11131 (0.86)	22 (1.00)	0.90
b18	37	23	36370 (1.00)	43 (1.00)	88.31	30732 (0.84)	43 (1.00)	0.14	30180 (0.83)	43 (1.00)	2.90
b19	24	27	72362 (1.00)	46 (1.00)	181.76	61320 (0.85)	46 (1.00)	0.29	60061 (0.83)	46 (1.00)	6.67
b20	32	22	5660 (1.00)	15 (1.00)	18.82	4679 (0.83)	15 (1.00)	0.02	4630 (0.82)	15 (1.00)	0.42
b21	32	22	5660 (1.00)	16 (1.00)	19.29	4730 (0.84)	16 (1.00)	0.02	4628 (0.82)	16 (1.00)	0.44
b22	32	22	8429 (1.00)	16 (1.00)	24.42	7000 (0.83)	16 (1.00)	0.03	6846 (0.81)	16 (1.00)	0.67
geomean			(1.00)	(1.00)		(0.84)	(1.00)		(0.84)	(1.00)	

CPU time and memory usage. Columns 2, 3, and 4 show the runtime in seconds spent in generating files for different techniques; Columns 5, 6, and 7 show the size in KB of the generated files; Columns 8, 9, and 10 show the runtime for equivalence checking. A time (resp. memory) limit of two hours (resp. 1GB) is imposed, and a `TO` (resp. `MO`) in the tables indicates timeout (resp. memory out).

As can be seen from the two tables, TLG-to-PB translation always generates constraints of size linear to the number of TLGs in the TL circuit. This property results in a fast and compact translation. In contrast, the CNF formula size and AIG size resulted from `weight order` translation are sensitive to the SOP representations of TLGs. These translations suffer from memory explosion. On average, the size of CNF formula (resp. AIG) files is 70 (resp. 7) times the size of PB files. In Table 4 the `weight order` translation exceeds the memory limit for benchmarks `b14` to `b22`, whereas in Table 5 it can generate files within memory limit except for `b18` and `b19`. This phenomenon suggests that collapsing with iterative fanout bound increment could synthesize TL circuits with simpler TLGs.

The proposed PB-based equivalence checking solved six large benchmarks (`b14-17`, `b20-22`) uniquely in Table 4, demonstrating the unique value of the linear translation. On the other hand, not surprisingly, CEC achieves the best scalability among the three methods due to its powerful exploitation of circuit similarities for verification reduction. A compact translation from TLG to AIG would greatly benefit the usage of CEC and improve the scalability of threshold logic verification, which is left as our future work.

7. CONCLUSIONS AND FUTURE WORK

In this paper, the collapse operation of threshold logic gates has been formulated, and the sufficient and necessary conditions of collapsibility using linear composition have been derived. Moreover, a linear-time translation from a threshold logic gate to PB constraints has been proposed for equivalence verification. Experimental results showed fast and effective TL circuit collapsing as well as memory efficient equivalence checking.Specifically, on top of the optimized TL circuits, an average of 16% gate count reduction is achieved by the collapse operation, and the proposed translation uniquely solves six benchmarks in our experiments. For future work, we would like to study how to combine the collapse operation with other TL operations to form useful scripts for TL circuit optimization, to develop compact translation from TLG to AIG for verification, and to apply TL synthesis for neural network applications.

REFERENCES

[1] V. Beiu, J. Quintana and M. Avedillo. VLSI implementations of threshold logic-a comprehensive survey. *IEEE Trans. on Neural Networks, 14(5)*, pp. 1217-1243, 2003.

[2] Y.-C. Chen, R. Wang and Y.-P Chang. Fast synthesis of threshold logic networks with optimization. In *Proc. Asia and South Pacific Design Automation Conf.*, pp. 486-491, 2016.

[3] T. Cormen, C. Leiserson, R. Rivest and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[4] L. Gao, F. Alibart and D. Strukov. Programmable CMOS/memristor threshold logic. *IEEE Trans. on Nanotechnology, 12(2)*, pp. 115-119, 2013.

[5] T. Gowda, S. Leshner, S. Vrudhula and G. Konjevod. Synthesis of threshold logic circuits using tree matching. In *Proc. European Conf. on Circuit Theory and Design*, pp. 850-853, 2007.

[6] T. Gowda, S. Vrudhula and G. Konjevod. Combinational equivalence checking for threshold logic circuits. In *Proc. ACM Great Lakes Symp. on VLSI*, pp. 102-107, 2007.

[7] G.-B. Huang, Q.-Y. Zhu, K.-Z. Mao, C.-K. Siew and P. Saratchandran. Can threshold networks be trained directly? *IEEE Trans. on Circuits and Systems II: Express Briefs, 53(3)*, pp. 187-191, 2006.

[8] H. Hulgaard, P. Williams and H. Andersen. Equivalence checking of combinational circuits using Boolean expression diagrams. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 18(7)*, pp. 903-917, 1999.

[9] A. Krizhevsky, I. Sutskever and G. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. Advances in Neural Information Processing Systems*, pp. 1097-1105, 2012.

[10] P.-Y. Kuo, C.-Y. Wang and C.-Y. Huang. On rewiring and simplification for canonicity in threshold logic circuits. In *Proc. Int'l Conf. on Computer-Aided Design*, pp. 396-403, 2011.

[11] R. Lippmann. An introduction to computing with neural nets. In *IEEE ASSP Magazine, 4(2)*, pp. 4-22, 1987.

[12] P. Merolla *et al.* A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science vol.345*, pp. 668-673, 2014.

[13] S. Muroga. Threshold logic and its applications. New York: Wiley-Interscience, 1971.

[14] A. Neutzling, J. Matos, A. Reis, R. Ribas and A. Mishchenko. Threshold logic synthesis based on cut pruning. In *Proc. Int'l Conf. on Computer-Aided Design*, pp. 494-499, 2015.

[15] N. Nukala, N. Kulkarni and S. Vrudhula. Spintronic threshold logic array (STLA)-a compact, low leakage, non-volatile gate array architecture. *Journal of Parallel and Distributed Computing, 74(6)*, pp. 2452-2460, 2014.

[16] A. Palaniswamy and S. Tragoudas. An efficient heuristic to

Table 4: Comparison of CNF, AIG and PB-based equivalence verification (without iterative collapsing)

Statistics for equivalence verification between TLC before and after collapsing (w/o ite)									
	file generation time (s)			verification file size (KB)			verification time (s)		
circuit	CNF	CEC	PB	CNF	CEC	PB	CNF	CEC	PB
c3540	0.05	0.02	0.00	5304	504	112	44.58	0.35	1.76
c5315	0.07	0.03	0.00	7652	712	176	4.46	0.35	0.88
c6288	0.21	0.09	0.00	24252	2336	316	TO	2.03	TO
c7552	0.07	0.03	0.00	7636	764	208	2.71	0.44	1.89
s5378	0.07	0.03	0.00	7948	840	184	0.04	0.18	0.35
s9234.1	0.24	0.10	0.00	27732	2988	224	0.12	0.86	0.49
s13207	0.15	0.07	0.00	17492	2008	492	0.08	0.43	1.82
s15850	0.45	0.18	0.00	60716	4576	536	0.25	1.17	2.44
s35932	0.11	0.05	0.01	10276	1224	1524	18.78	1.15	25.20
s38417	0.57	0.24	0.01	72196	6360	1504	85.90	2.03	31.28
s38584	5.09	1.90	0.01	891944	76172	1512	3.57	22.66	28.97
b04	0.05	0.03	0.00	5740	836	72	13.62	1.61	0.12
b12	0.06	0.02	0.00	6556	720	160	3.47	0.16	0.24
b14	MO	MO	0.01	MO	MO	712	MO	MO	111.95
b15	MO	MO	0.01	MO	MO	1180	MO	MO	17.83
b17	MO	MO	0.03	MO	MO	4016	MO	MO	160.67
b18	MO	MO	0.09	MO	MO	11060	MO	MO	TO
b19	MO	MO	0.18	MO	MO	22368	MO	MO	TO
b20	MO	MO	0.01	MO	MO	1460	MO	MO	1795.46
b21	MO	MO	0.01	MO	MO	1492	MO	MO	2846.92
b22	MO	MO	0.02	MO	MO	2156	MO	MO	2524.79

Table 5: Comparison of CNF, AIG and PB-based equivalence verification (with iterative collapsing)

Statistics for equivalence verification between TLC before and after iterative collapsing									
	file generation time (s)			verification file size (KB)			verification time (s)		
circuit	CNF	CEC	PB	CNF	CEC	PB	CNF	CEC	PB
c3540	0.04	0.01	0.00	3208	316	108	15.58	0.29	1.58
c5315	0.04	0.01	0.00	3052	324	176	1.52	0.29	1.51
c6288	0.12	0.04	0.01	9216	960	304	TO	1.60	TO
c7552	0.06	0.02	0.00	4960	512	208	1.55	0.39	2.18
s5378	0.10	0.03	0.00	7996	832	180	0.04	0.17	0.44
s9234.1	0.27	0.09	0.00	24340	2608	224	0.13	0.67	0.62
s13207	0.14	0.05	0.01	11728	1408	484	0.06	0.29	2.20
s15850	MO	2.84	0.01	MO	112484	532	MO	0.47	2.81
s35932	0.15	0.05	0.03	10276	1224	1524	15.83	1.16	18.66
s38417	1.81	0.55	0.03	190724	15248	1488	246.33	3.10	23.53
s38584	1.78	0.50	0.03	213720	18552	1500	0.81	4.97	26.33
b04	0.06	0.03	0.00	5716	832	72	8.30	1.66	0.12
b12	0.11	0.03	0.00	9356	1056	160	3.36	0.18	0.29
b14	1.05	0.27	0.01	111928	8336	672	1650.61	4.52	100.18
b15	0.80	0.25	0.02	79404	6740	1140	242.50	4.38	17.17
b17	MO	2.39	0.08	MO	93956	3676	MO	70.25	188.12
b18	MO	MO	0.24	MO	MO	10096	MO	MO	TO
b19	MO	MO	0.45	MO	MO	20464	MO	MO	TO
b20	3.38	0.99	0.03	393620	37684	1404	2506.10	17.81	1401.99
b21	2.19	0.61	0.03	245672	15728	1396	1240.13	7.87	1977.18
b22	5.84	1.58	0.05	668384	64828	2060	3634.66	39.72	3274.91

identify threshold logic functions. *ACM Journal on Emerging Technologies in Computing Systems, 8(3)*, 19, 2012.

[17] A. Palaniswamy and S. Tragoudas. Improved threshold logic synthesis using implicant-implicit algorithms. *ACM Journal on Emerging Technologies in Computing Systems, 10(3)*, 21, 2014.

[18] E. Sentovich *et al.* Sequential circuit design using synthesis and optimization. In *Proc. Int. Conf. on Computer Design*, pp. 328-333, 1992.

[19] J. Subirats, J. Jerez and L. Franco. A new decomposition algorithm for threshold synthesis and generalization of Boolean functions. *IEEE Trans. on Circuits and Systems I: Regular papers, 55(10)*, pp. 3188-3196, 2008.

[20] G. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, pp. 466-483, 1983.

[21] R. Zhang, P. Gupta, L. Zhong and N. Jha. Synthesis and optimization of threshold logic networks with application to nanotechnologies. In *Proc. Conf. on Design, Automation and Test in Europe-Volume 2*, pp. 904-909, 2004.

[22] Y. Zheng, M. Hsiao and C. Huang. SAT-based equivalence checking of threshold logic designs for nanotechnologies. In *Proc. ACM Great Lakes Symp. on VLSI*, pp. 225-230, 2008.

[23] Berkeley Logic Synthesis and Verification Group. ABC: a system for sequential synthesis and verification. Available: `http://www.eecs.berkeley.edu/~alanmi/abc/`

Criticality and Sensitivity Analysis for Incremental Performance Optimization of Asynchronous Pipelines

Chun-Hong Shih and Jie-Hong R. Jiang
Department of Electrical Engineering / Graduate Institute of Electronics Engineering
National Taiwan University, Taipei 10617, Taiwan

ABSTRACT

Asynchronous methodologies gain increasing adoption in modern IC design to overcome synchronization limitations. There has been recent work optimizing asynchronous pipeline performance based on static performance analysis (SPA). Despite its linear-time complexity, SPA remains inefficient for large designs that undergone an excessive number of incremental optimization iterations. In this paper, we investigate the performance criticality and sensitivity of asynchronous pipelines, and propose incremental SPA for iterative optimization with buffer insertion. Experimental results show that our method achieves average runtime improvement by two orders of magnitude. For circuits with a wide range of delay distributions among their pipeline modules, our method can reduce pipeline cycle time to a level not achievable by prior methods while inserting significantly fewer buffers.

1. INTRODUCTION

Having the potential to overcome synchronization issues in modern nanometer integrated circuits, asynchronous design methodologies draw recent attention. Among various asynchronous delay models, the quasi-delay insensitive (QDI) model is robust and promising for automatic asynchronous circuit synthesis compatible to synchronous design flow [6, 7, 9]. Despite recent advances in asynchronous synthesis tools, various analysis and synthesis algorithms awaits further improvement.

In this paper, we address the performance analysis and optimization of QDI circuits, including the well-known precharged half buffer (PCHB) [13], weak-conditioned half buffer (WCHB) [13], pre-charged full buffer (PCFB) [11], and null convention logic (NCL) [7] circuits. The performance analysis and optimization for asynchronous circuits can be more challenging than those for synchronous circuits. To give a counterintuitive example, increasing the delay of a pipeline module in an asynchronous circuit may possibly improve the circuit cycle time. Unlike the monotonicity of synchronous circuit timing optimization, where reducing a gate delay should not increase the entire circuit delay, asynchronous circuit performance optimization do not enjoy such a nice property.

There are various prior methods proposed for asynchronous circuit performance analysis, e.g., [5, 11, 16, 14, 21]. Among prior methods, static performance analysis [5] is the most scalable technique. Despite the linear-time efficiency of SPA, it remains impractical for large designs undergone excessive optimization iterations. In this work, we investigate the performance criticality and sensitivity of asynchronous pipelines. Thereby, we propose an incremental approach to static performance analysis, which allows local update of timing information without reanalyzing an entire circuit under some delay perturbation.

Performance optimization for asynchronous circuits are under active development, e.g., [1, 2, 8, 12, 15, 22]. Slack matching using mixed integer linear programming (MILP) or relaxed linear programming was proposed in [1, 2]. In [8], heuristic optimization methods were proposed based on pipeline structures. Recent work [12, 15, 22] focused on NCL circuit synthesis and optimization. An overview of recent advances can be found in [18]. In contrast to the prior work, we extend the efficient SPA [5] and the general four-phase time mark graph model [10] for scalable performance optimization of various types of acyclic[1] pipeline circuits. We apply our developed incremental SPA to iterative performance improvement via buffer insertion and removal. Experimental results show significant speedup by two orders of magnitude on average over prior work [5]. Moreover, for libraries with increased range of cell delays, our optimization may achieve cycle times not attainable by prior method and use far fewer buffers. It suggests the robustness of our method against library migration issues.

2. PRELIMINARIES

Among a number of timing models in asynchronous design, the *quasi-delay insensitive* (QDI) model is robust without any timing assumption except for the *isochronic fork* assumption [24]. Well-known QDI pipeline templates include PCHB, WCHB, PCFB and NCL pipelines. The operations of PCFB, PCHB, WCHB and NCL circuits follow a four-phase handshake protocol between a sender and receiver as shown in the timing diagram of Fig. 1, taken from [10]. A transaction cycle in the protocol consists of four phases: In the request phase (denoted rq), the sender initiates a request to send data by raising its request signal; in the acknowledge phase (denoted ak), the receiver acknowledges the receipt of the sender's data by raising its acknowledge signal; in the de-request phase (denoted dr), the sender resets its request signal; in the de-acknowledge phase (denoted da), the receiver resets its acknowledge signal.

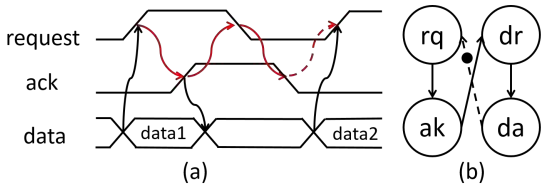


Figure 1: (a) Timing diagram of four-phase communication; (b) transaction cycle of four phases ([10]).

[1]In this work, we focus on acyclic pipelines, which, though restricted, may arise naturally in the design flow converting synchronous design to asynchronous implementation, and are not uncommon in high performance designs. Extension to cyclic pipelines, as our future work, should be possible in view of the recent advancement [23].

In this paper, we consider an asynchronous circuit as a network of pipeline modules. We focus on acyclic pipeline networks. We abstract an asynchronous pipeline circuit as a *delay graph* $G(V, E)$, where a node $v \in V$ represents a PCFB, PCHB, WCHB or NCL pipeline module and an edge $e = (u, v) \in E$ represents the connection from node u to v. Each node $v \in V$ is associated with some delay attributes. Two disjoint subsets $I \subseteq V$ and $O \subseteq V$ of nodes are identified as the primary inputs (PIs) and primary outputs (POs). The set of fanin and fanout nodes of v are denoted $FI(v)$ and $FO(v)$, respectively. The set of transitive fanin (resp. fanout) nodes of v, but excluding v, is denoted $TFI(v)$ (resp. $TFO(v)$).

A delay graph receives input data from its environment through its PIs and produces output data to its environment through its POs. The data processing of a delay graph can be viewed as a stream of data tokens flowing through the pipeline structure defined by the delay graph. Therefore the performance of a delay graph can be characterized by its *cycle time* (the reciprocal of *throughput*).

According to the four-phase (*rq*, *ak*, *dr*, and *da*) operation of a delay graph, each node $v \in V$ is associated with four operation phases, denoted $v.rq$, $v.ak$, $v.dr$, and $v.da$. In processing the first input data, the start times of $v.rq$, $v.ak$, $v.dr$, and $v.da$ are denoted as $t(v.rq)$, $t(v.ak)$, $t(v.dr)$, and $t(v.da)$, respectively; the (local) cycle time of node v is denoted as τ_v. Depending on the detailed protocol of PCFB, PCHB, WCHB, and NCL, a node needs to wait for other nodes to enter their certain phases before the node can enter some phase. In the sequel, we use $\delta(u.p_1, v.p_2)$ to denote $|t(u.p_1) - t(v.p_2)|$ for $u, v \in V$ and $p_1, p_2 \in \{rq, ak, dr, da\}$. Also, we let $v.d_1$ denote $\delta(v.rq, v.ak)$, which corresponds to the *evaluation time* of v, and $v.d_2$ denote $\delta(v.dr, v.da)$, which corresponds to the *precharge time* of v. Fig. 2 shows a delay graph example, where each node (box) is specified with its d_1 and d_2 values. For PI (resp. PO) nodes in the graph, only their d_1 values are specified while we don't care the delay returning the acknowledge signal from the PIs to the environment (resp. from the environment to the POs).

2.1 Static Performance Analysis

Static performance analysis (SPA) for PCFB, PCHB, WCHB, and NCL circuits has been proposed in [5, 10]. Given a delay graph $G(V, E)$, SPA computes $t(v.rq)$, $t(v.ak)$, $t(v.dr)$, $t(v.da)$ and τ_v for each node $v \in V$. Below we only list the PCHB rules for reference due to space limitation, while those for PCFB, WCHB, and NCL can be found in [10].

$$t(v.rq) = \max_{u \in FI(v)} \{t(u.ak) + \delta(u.ak, v.rq)\} \quad (1)$$

$$t(v.ak) = t(v.rq) + \delta(v.rq, v.ak) \quad (2)$$

$$t(v.dr) = \max\{t(v.ak) + \delta(v.ak, v.dr), \max_{u \in FO(v)} \{t(u.ak) + \delta(u.ak, v.dr)\}\} \quad (3)$$

$$t(v.da) = t(v.dr) + \delta(v.dr, v.da) \quad (4)$$

$$\tau_v = \max\{t(v.da) + \delta(v.da, v.rq), \max_{u \in FO(v)} \{t(u.da) + \delta(u.da, v.rq)\}\} - \max_{u \in FI(v)} \{t(u.ak) + \delta(u.ak, v.rq)\}. \quad (5)$$

SPA can be done by two traversals in a topological order over the delay graph. It computes $t(v.rq)$ and $t(v.ak)$ of each node v in the first traversal, and $t(v.dr)$ and $t(v.da)$ in the second traversal. Finally, the local cycle time of every node can be determined by Eq. (5), and the cycle time of the delay graph is determined by $\max_v \tau_v$.

For example, Fig. 2 shows a delay graph with nodes $V = \{v_1, \ldots, v_{11}\}$ denoted by the boxes. Each box contains d_1 and d_2 values, except PIs $\{v_1, v_7\}$ and POs $\{v_6, v_{11}\}$ containing only their d_1 values. Under the PCHB protocol, the values $t(v.rq)$, $t(v.dr)$, and τ_v for each node v computed by SPA are shown above the corresponding box.

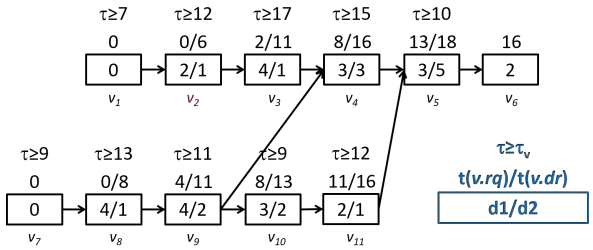


Figure 2: Delay graph of an asynchronous pipeline circuit.

3. INCREMENTAL STATIC PERFORMANCE ANALYSIS

Due to the large size of modern integrated circuits, logic synthesis approaches heavily rely on incremental circuit restructuring. The circuits before and after each incremental modification step can be very similar. Exploiting their similarities may greatly simplify performance analysis computation. We study below how SPA can be made incremental.

In incremental analysis, it is important to determine the range of impact due to a delay change in a circuit. We define the notion of *local slack* to characterize the situation of data stall.

DEFINITION 1. *Given a delay graph $G(V, E)$, let the* local slack*, denoted $\sigma(e)$, of an edge $e = (u, v) \in E$ be*

$$\sigma(e) = t(v.rq) - t(u.ak).$$

If $\sigma(e)$, for $e = (u, v)$, is greater than 0, then node u has to wait exactly $\sigma(e)$ time units upon the moment that u finishes its data evaluation to pass its data token to v. In view of the delay graph, it means $t(v.rq) > t(u.ak) + \delta(u.ak, v.rq)$ under the PCFB, PCHB, WCHB and NCL protocols as they have the same formula of $t(v.rq)$ in [10].

With the local slack definition, we derive incremental performance analysis for the effects of inserting or removing a pipeline module from an asynchronous pipeline circuit. By the similarities of their working behaviors, we divide the four considered pipeline templates into two groups: one with PCHB and PCFB and the other with WCHB and NCL. (According to the rules of static performance analysis [10], the firing time $t(u.dr)$ in PCHB and PCFB does not affect $t(v.da)$, for $v \in FO(u)$, whereas $t(u.dr)$ in WCHB and NCL does affect $t(v.da)$. We discuss incremental performance analysis under the effects of buffer insertion and buffer removal, and the effects of evaluation and precharge delay time changes on a node in a delay graph.

3.1 Incremental Analysis for Buffer Insertion

Buffer insertion is an effective technique for performance optimization of asynchronous pipeline circuits. We study how to perform incremental SPA for circuits under buffer insertion. Given a delay graph $G(V, E)$, consider inserting a new node, $w \notin V$, to G on edge $e = (u, v) \in E$ as shown in Fig. 3. We define the following four sets of nodes with respect to u

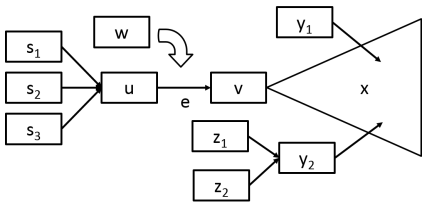


Figure 3: Delay graph under buffer insertion.

and v to ease our discussion of Theorems 1 to 4.

$$\begin{aligned} S &= \{s \mid s \in FI(u)\} \\ X &= \{x \mid x \in TFO(v)\} \\ Y &= \{y, z \mid x \in TFO(v), y \in FI(x)\backslash\{v\}, z \in FI(y)\backslash\{v\}\} \\ R &= \{r \mid r \notin S \cup X \cup Y \cup \{u, v\}, r \in V\} \end{aligned}$$

For example, in Fig. 3 we have $S = \{s_1, s_2, s_3\}$, X is signified by the cone, $Y = \{y_1, y_2, z_1, z_2\}$, and R consists of the rest of the nodes (not shown in the graph).

As notational convention, in the sequel we let the primed version v' of a node v denote the corresponding node of v in the modified delay graph. The following four theorems can be established. (We only show the proof of Theorem 1 while the other nineteen can be proved similarly.)

THEOREM 1. *Given a delay graph $G(V, E)$ of a PCHB or PCFB circuit, let $w \notin V$ to be inserted on edge $e = (u, v) \in E$ yielding a new graph $G'(V \cup \{w\}, E \cup \{(u, w), (w, v)\}\backslash\{e\})$ and let $w.d_1 = \delta(w.rq, w.ak)$. If $\sigma(e) \geq w.d_1$, then $\tau_s \leq \tau_s$, $\tau_u \leq \tau_u$, $\tau_w \leq \tau_u$, $\tau_v \leq \tau_v$, $\tau_x = \tau_x$, $\tau_y = \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$.*

PROOF. For PCHB and PCFB circuits, because $\sigma(e)$ is greater than $w.d_1$, inserting w does not affect $t(v.rq)$ and $t(v.ak)$. So $t(v.rq)$ and $t(v.ak)$ remain intact after buffer insertion, i.e., $t(v'.rq) = t(v.rq)$ and $t(v'.ak) = t(v.ak)$. Because the values of $t(rq)$ and $t(ak)$ of the nodes in the fanin cone of s do not change after inserting w, it makes $t(s.rq)$ and $t(s.ak)$ unaffected by the insertion. Also since $t(u.rq)$ is determined by $t(s.ak)$ by Eq. (1), $t(u.rq)$ remains unaffected, too.

Because $\sigma(e) \geq w.d_1$, the value of $t(w.ak)$ is not greater than $t(v.rq)$. Also $t(w.rq) \leq t(w.ak)$ according to Eq. (2), hence $t(w.rq) \leq t(v.rq)$. By the inequality $t(v.ak) \geq t(v.rq)$, we have $t(w.ak) \leq t(v.ak)$. And $t(w.rq) \geq t(u.ak)$ by Eq. (1). We have $t(w.ak) \geq t(u.ak)$ according to Eq. (2). Therefore, we conclude $t(u'.rq) = t(u.rq)$, $t(u'.ak) = t(u.ak)$, $t(w.rq) \leq t(v'.rq)$, $t(w.ak) \leq t(v'.ak)$, and $t(w.ak) \geq t(u.ak)$.

According to the SPA equations of PCHB and PCFB circuits, $t(u.dr)$ is determined by $t(v.ak)$, and the insertion of w makes $t(u'.dr)$ be determined by $t(w.ak)$. By $t(w.ak) \leq t(v'.ak)$ shown above, we have $t(u'.dr) \leq t(u.dr)$. Also we have $t(u'.da) \leq t(u.da)$ by Eq. (4). $t(v'.ak) = t(v.ak)$ makes $t(x'.rq) = t(x.rq)$ and $t(x'.ak) = t(x.ak)$. And $t(v.dr)$ being determined by $t(x.ak)$ according to Eq. (3). So we have $t(v'.dr) = t(v.dr)$ and $t(v'.da) = t(v.da)$. By Eq. (3), the value of $t(w.dr)$ is determined by $t(v'.ak)$ and $t(w.ak)$. By Eq. (3), the value of $t(v'.dr)$ is determined by $t(v'.ak)$ and $t(x'.ak)$. By the obtained $t(w.ak) \leq t(v'.ak)$ and $t(v'.ak) \leq t(x'.ak)$, we have $t(w.dr) \leq t(v'.dr)$. Also $t(w.da) \leq t(v'.da)$ according to Eq (4). Consequently, we conclude $t(u'.dr) \leq t(u.dr)$, $t(u'.da) \leq t(u.da)$, $t(v'.dr) = t(v.dr)$, $t(v'.da) = t(v.da)$, $t(w.dr) \leq t(v'.dr)$, and $t(w.da) \leq t(v'.da)$.

After we insert a buffer, the changing of four essential values of a node in a PCHB or PCFB pipeline is shown above. The values of the rest of the nodes remain the same. For node $s \in S$, the three conditions $t(s'.da) = t(s.da)$, $t(u'.da) \leq t(u.da)$, and $t(t'.ak) = t(t.ak)$, where $t \in FI(s)$, make $\tau_s \leq \tau_s$ according to Eq. (5). For node u, the three conditions $t(u'.da) \leq t(u.da)$, $t(w.da) \leq t(v.da)$, and $t(s.ak) = t(s.ak)$ make $\tau_u \leq \tau_u$ according to Eq. (5). For node v, the three conditions $t(x'.da) = t(x.da)$, $t(v'.da) = t(v.da)$, and $t(w.ak) \geq t(u.ak)$ make $\tau_v \leq \tau_v$ according to Eq. (5). Finally, we arrive

$$\begin{aligned} \tau_s &\leq \tau_s, \qquad \forall s, s \in FI(u) \\ \tau_u &\leq \tau_u, \\ \tau_v &\leq \tau_v, \end{aligned}$$

and the cycle times of the rest of the nodes remain the same. ■

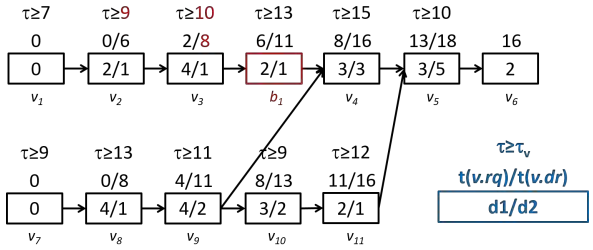


Figure 4: Delay graph with node b_1 inserted to the circuit of Fig. 2.

EXAMPLE 1. *Fig. 4 shows the result after node b_1 is inserted into the original PCHB circuit in Fig. 2. We have $\delta(b_1.rq, b_1.ak) = 2$, and $\sigma(e) = 2$ for $e = (v_3, v_4)$ in the original circuit. Note that $\sigma(e)$ equals $\delta(b_1.rq, b_1.ak)$. According to Theorem 1, $S = \{v_2\}$, $X = \{v_5, v_6\}$, $Y = \{v_{10}, v_{11}\}$, $R = \{v_1, v_7, v_8, v_9\}$ and the changed time values, including $t(v_3.dr), \tau_{v_3}, \tau_{v_2}$, are in red in Fig. 4. All other values (in black) remain the same. Hence the circuit cycle time decreases from 17 to 15.*

THEOREM 2. *Given a delay graph $G(V, E)$ of a WCHB or NCL circuit, let $w \notin V$ to be inserted on edge $e = (u, v) \in E$ yielding a new graph $G'(V \cup \{w\}, E \cup \{(u, w), (w, v)\}\backslash\{e\})$ and let $w.d_1 = \delta(w.rq, w.ak)$. If $\sigma(e) \geq w.d_1$, then $\tau_s \leq \tau_s$, $\tau_u \leq \tau_u$, $\tau_x \geq \tau_x$, $\tau_y \geq \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$. (The cycle time change of v is undetermined.)*

THEOREM 3. *Given a delay graph $G(V, E)$ of a PCHB or PCFB circuit, let $w \notin V$ to be inserted on edge $e = (u, v) \in E$ yielding a new graph $G'(V \cup \{w\}, E \cup \{(u, w), (w, v)\}\backslash\{e\})$ and let $w.d_1 = \delta(w.rq, w.ak)$. If $\sigma(e) < w.d_1$, then $\tau_s \leq \tau_s$, $\tau_u \leq \tau_u$, and $\tau_r = \tau_r$, where $s \in S$ and $r \in R$. (The cycle time changes of v, x, and y are undetermined.)*

THEOREM 4. *Given a delay graph $G(V, E)$ of a WCHB or NCL circuit, let $w \notin V$ to be inserted on edge $e = (u, v) \in E$ yielding a new graph $G'(V \cup \{w\}, E \cup \{(u, w), (w, v)\}\backslash\{e\})$ and let $w.d_1 = \delta(w.rq, w.ak)$. If $\sigma(e) < w.d_1$, then $\tau_s \leq \tau_s$, and $\tau_r = \tau_r$, where $s \in S$ and $r \in R$. (The cycle time changes of u, v, x, and y are undetermined.)*

Table 1 summarizes the cycle time changes of nodes $s \in S, u, v, x \in X, y \in Y, r \in R$ after buffer insertion under the four cases discussed in Theorems 1 to 4. Rows 2 to 5 correspond to the cases covered by Theorems 1 to 4, respectively. An entry with symbol "$\leq$" in the column of τ_a in the table indicates cycle time decreasing, i.e., $\tau_a \leq \tau_a$; symbol "$\geq$" indicates cycle time increasing, i.e., $\tau_a \geq \tau_a$; symbol "$=$"

Table 1: Cycle Time Change due to Buffer Insertion

		τ_s	τ_u	τ_v	τ_x	τ_y	τ_r
$\sigma(e) \geq w.d_1$	PCHB, PCFB	$\leq$	$\leq$	$\leq$	$=$	$=$	$=$
	WCHB, NCL	$\leq$	$\leq$	?	$\geq$	$\geq$	$=$
$\sigma(e) < w.d_1$	PCHB, PCFB	$\leq$	$\leq$	?	?	?	$=$
	WCHB, NCL	$\leq$	?	?	?	?	$=$

indicates unchanged cycle time, i.e., $\tau_a = \tau_a$; symbol "?" indicates undetermined cycle time change.

By inserting node w on edge $e = (u, v)$ with $\sigma(e) \geq w.d_1$, it causes $t(v'.rq) > t(w.ak) + \delta(w.ak, w.rq)$. Thus, $t(v'.rq)$ equals $t(v.rq)$, and $t(x'.rq)$ equals $t(x.rq)$. The above theorems suggest that compared to the condition $\sigma(e) \geq w.d_1$, under $\sigma(e) \geq w.d_1$ the effects of buffer insertion are more predictable and less calculation is needed for incremental analysis. On the other hand,by inserting w, because the right hand side of Eq. (3) for $t(v.dr)$ does not refer to any time values of the fanins of v, we conclude $t(v.dr) = t(v'.dr)$ and also $t(x.dr) = t(x'.dr)$ for $x \in X$, in $TFO(v)$. This property holds for PCFB as well by the SPA rules. In contrast, the change affects $t(v.dr)$ in WCHB or NCL circuits. This fact makes the cycle time reduction of inserting a buffer in a PCHB or PCFB circuit better than that in a WCHB or NCL circuit.

3.2 Incremental Analysis for Buffer Removal

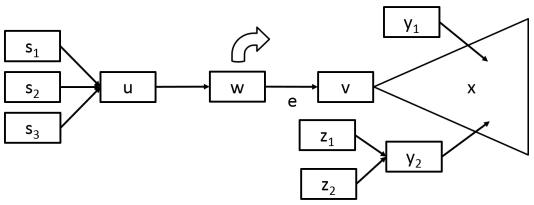


Figure 5: Delay graph under buffer removal.

Buffer removal removes a buffer node from a delay graph. Given a delay graph $G(V, E)$, consider removing a buffer node $w \in V$ from G with $(u, w), (w, v) \in E$ as shown in Fig. 5. We define the following four sets of nodes to be used in Theorems 5 to 8.

$$\begin{aligned} S &= \{s \mid s \in FI(u)\} \\ X &= \{x \mid x \in TFO(v)\} \\ Y &= \{y, z \mid x \in TFO(v), y \in FI(x)\backslash\{v\}, z \in FI(y)\backslash\{v\}\} \\ R &= \{r \mid r \notin S \cup X \cup Y \cup \{u, w, v\}, r \in V\} \end{aligned}$$

For example, in Fig. 5 we have $S = \{s_1, s_2, s_3\}$, X is signified by the cone, $Y = \{y_1, y_2, z_1, z_2\}$, and R consists of the rest of the nodes (not shown in the graph).

The following four theorems can be established.

THEOREM 5. *Given a delay graph $G(V, E)$ of a PCFB or PCHB circuit, let w be the removed node and edge $e = (w, v) \in E$ yielding a new graph $G'(V\backslash\{w\}, E\cup\{(u, v)\}\backslash\{(u, w), (w, v)\})$. If $\sigma(e) > 0$, then $\tau_s \geq \tau_s$, $\tau_u \geq \tau_u$, $\tau_v \geq \tau_v$, $\tau_x = \tau_x$, $\tau_y = \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$.*

THEOREM 6. *Given a delay graph $G(V, E)$ of a WCHB or NCL circuit, let w be the removed node and edge $e = (w, v) \in E$ yielding a new graph $G'(V\backslash\{w\}, E\cup\{(u, v)\}\backslash\{(u, w), (w, v)\})$. If $\sigma(e) > 0$, then $\tau_s \geq \tau_s$, $\tau_u \geq \tau_u$, $\tau_x \leq \tau_x$, $\tau_y \leq \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$. (The cycle time change of v is undetermined.)*

THEOREM 7. *Given a delay graph $G(V, E)$ of a PCHB or PCFB circuit, let w be the removed node and edge $e = (w, v) \in E$ yielding a new graph $G'(V\backslash\{w\}, E\cup\{(u, v)\}\backslash\{(u, w), (w, v)\})$. If $\sigma(e) = 0$, then $\tau_s \geq \tau_s$, $\tau_u \geq \tau_u$, and $\tau_r = \tau_r$, where $s \in S$ and $r \in R$. (The cycle time changes of v, x, y are undetermined.)*

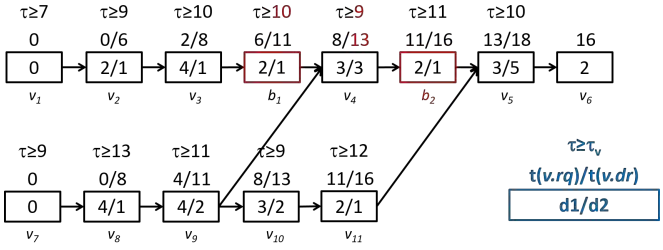


Figure 6: Delay graph with node b_2 inserted to the circuit of Fig. 4.

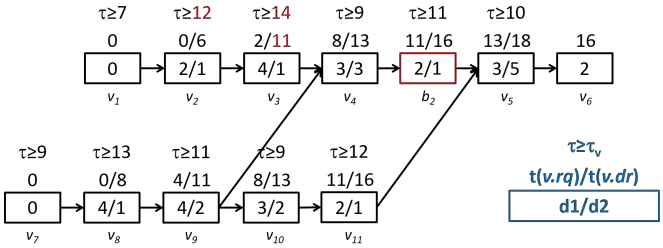


Figure 7: Delay graph with node b_1 removed from the circuit of Fig. 6.

EXAMPLE 2. *Fig. 7 shows the result after we remove a node, b_1, from the PCHB circuit in Fig. 6. The $\sigma(e)$ is 0, where $e = (b_1, v_4)$ in Fig. 7. According to Theorem 7, $S = \{v_2\}$, $X = \{v_5, v_6\}$, $Y = \{v_{11}\}$, $R = \{v_1, v_7, v_8, v_9, v_{10}\}$ and the changed values are in red in Fig. 7. The rest values still remain the same. The cycle time increases from 13 to 14.*

THEOREM 8. *Given a delay graph $G(V, E)$ of a WCHB or NCL circuit, let w be the removed node and edge $e = (u, v) \in E$ yielding a new graph $G'(V\backslash\{w\}, E\cup\{(u, v)\}\backslash\{e\})$. If $\sigma(e) = 0$, then $\tau_s \geq \tau_s$, and $\tau_r = \tau_r$, where $s \in S$ and $r \in R$. (The cycle time changes of u, v, x, y are undetermined.)*

Table 2 summarizes the cycle time changes of nodes $s \in S, u, v, x \in X, y \in Y, r \in R$ after buffer removal under the four cases discussed in Theorems 5 to 8. Rows 2 to 5 correspond to the cases covered by Theorems 5 to 8, respectively.

Table 2: Cycle Time Change due to Buffer Removal

		τ_s	τ_u	τ_v	τ_x	τ_y	τ_r
$\sigma(e) > 0$	PCHB, PCFB	$\geq$	$\geq$	$\geq$	$=$	$=$	$=$
	WCHB, NCL	$\geq$	$\geq$	?	$\leq$	$\leq$	$=$
$\sigma(e) = 0$	PCHB, PCFB	$\geq$	$\geq$	?	?	?	$=$
	WCHB, NCL	$\geq$	?	?	?	?	$=$

The following facts can be observed from the table. First, comparing the corresponding entries between Tables **??** and **??**, the inequality signs are reversed. This phenomenon is due to the inverse operations between buffer insertion and removal. Second, compared to the condition $\sigma(e) = 0$, under $\sigma(e) > 0$ the effects of buffer removal are more predictable and less calculation is needed for incremental analysis. Third, because

the entries with symbol $\leq$ in Table **??** appear only in WCHB and NCL circuits at $\sigma(e) > 0$ for nodes x and y, WCHB and NCL circuits have better chance to reduce the cycle time by buffer removal than PCHB and PCFB circuits.

3.3 Incremental Analysis for Evaluation Time Change

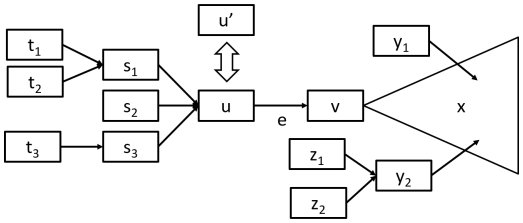


Figure 8: Delay graph under node replacement for evaluation time change.

Given a delay graph $G(V, E)$, consider replacing a node, $u \in V$, with $u' \notin V$. We define the following four sets to be used in Theorems 9 to 16.

$$\begin{aligned} S &= \{s, t \mid s \in FI(u), t \in FI(s)\} \\ X &= \{x \mid x \in TFO(v)\} \\ Y &= \{y, z \mid x \in TFO(v), y \in FI(x)\backslash\{v\}, z \in FI(y)\backslash\{v\}\} \\ R &= \{r \mid r \notin S \cup X \cup Y \cup \{u, v\}, r \in V\} \end{aligned}$$

For example, in Fig. 8 we have $S = \{s_1, s_2, s_3, t_1, t_2, t_3\}$, X is signified by the cone, $Y = \{y_1, y_2, z_1, z_2\}$, and R consists of the rest nodes (not shown in the graph).

The following eight theorems can be established.

THEOREM 9. *Given a delay graph $G(V, E)$ of a PCHB or PCFB circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_1 = \delta(u.rq, u.ak)$ and $u'.d_1 = \delta(u'.rq, u'.ak)$. If $u.d_1 > u'.d_1$ and $\sigma(e) > 0$, where $e = (u, v) \in E$, then $\tau_s \leq \tau_s$, $\tau_u \leq \tau_u$, $\tau_v \geq \tau_v$, $\tau_x = \tau_x$, $\tau_y = \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$.*

THEOREM 10. *Given a delay graph $G(V, E)$ of a PCHB or PCFB circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_1 = \delta(u.rq, u.ak)$ and $u'.d_1 = \delta(u'.rq, u'.ak)$. If $u.d_1 > u'.d_1$ and $\sigma(e) = 0$, where $e = (u, v) \in E$, then $\tau_s \leq \tau_s$, $\tau_u \leq \tau_u$, and $\tau_r = \tau_r$, where $s \in S$ and $r \in R$. (The cycle time changes of v, x, y are undetermined.)*

THEOREM 11. *Given a delay graph $G(V, E)$ of a PCHB or PCFB circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_1 = \delta(u.rq, u.ak)$ and $u'.d_1 = \delta(u'.rq, u'.ak)$. If $u.d_1 < u'.d_1$ and $\sigma(e) > 0$, where $e = (u', v) \in E$, then $\tau_s \geq \tau_s$, $\tau_u \geq \tau_u$, $\tau_v \leq \tau_v$, $\tau_x = \tau_x$, $\tau_y = \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$.*

THEOREM 12. *Given a delay graph $G(V, E)$ of a PCHB or PCFB circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_1 = \delta(u.rq, u.ak)$ and $u'.d_1 = \delta(u'.rq, u'.ak)$. If $u.d_1 < u'.d_1$ and $\sigma(e) = 0$, where $e = (u', v) \in E$, then $\tau_s \geq \tau_s$, $\tau_u \geq \tau_u$, and $\tau_r = \tau_r$, where $s \in S$ and $r \in R$. (The cycle time changes of v, x, y are undetermined.)*

Table 3 summarizes the cycle time changes of nodes $s \in S, u, v, x \in X, y \in Y, r \in R$ after evaluation time change of a node in PCHB or PCFB circuits under the four cases discussed in Theorems 9 to 12. Rows 2 to 5 correspond to the cases covered by Theorems 9 to 12, respectively.

Table 3: Cycle Time Change due to Evaluation Time Change (PCHB/PCFB)

PCHB, PCFB		τ_s	τ_u	τ_v	τ_x	τ_y	τ_r
$u.d_1 > u\ .d_1$	$\sigma(e) > 0$	$\leq$	$\leq$	$\geq$	$=$	$=$	$=$
	$\sigma(e) = 0$	$\leq$	$\leq$	?	?	?	$=$
$u.d_1 < u\ .d_1$	$\sigma(e) > 0$	$\geq$	$\geq$	$\leq$	$=$	$=$	$=$
	$\sigma(e) = 0$	$\geq$	$\geq$	?	?	?	$=$

THEOREM 13. *Given a delay graph $G(V, E)$ of a WCHB or NCL circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_1 = \delta(u.rq, u.ak)$ and $u'.d_1 = \delta(u'.rq, u'.ak)$. If $u.d_1 > u'.d_1$ and $\sigma(e) > 0$, where $e = (u, v) \in E$, then $\tau_s \leq \tau_s$, $\tau_u \leq \tau_u$, $\tau_v \leq \tau_v$, $\tau_x \leq \tau_x$, $\tau_y \leq \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$.*

THEOREM 14. *Given a delay graph $G(V, E)$ of a WCHB or NCL circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_1 = \delta(u.rq, u.ak)$ and $u'.d_1 = \delta(u'.rq, u'.ak)$. If $u.d_1 > u'.d_1$ and $\sigma(e) = 0$, where $e = (u, v) \in E$, then $\tau_s \leq \tau_s$, $\tau_u \leq \tau_u$, and $\tau_r = \tau_r$, where $s \in S$ and $r \in R$. (The cycle time changes of v, x, y are undetermined.)*

THEOREM 15. *Given a delay graph $G(V, E)$ of a WCHB or NCL circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_1 = \delta(u.rq, u.ak)$ and $u'.d_1 = \delta(u'.rq, u'.ak)$. If $u.d_1 < u'.d_1$ and $\sigma(e) > 0$, where $e = (u', v) \in E$, then $\tau_s \geq \tau_s$, $\tau_u \geq \tau_u$, $\tau_x \geq \tau_x$, $\tau_y \geq \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$. (The cycle time change of v is undetermined.)*

THEOREM 16. *Given a delay graph $G(V, E)$ of a WCHB or NCL circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_1 = \delta(u.rq, u.ak)$ and $u'.d_1 = \delta(u'.rq, u'.ak)$. If $u.d_1 < u'.d_1$ and $\sigma(e) = 0$, where $e = (u', v) \in E$, then $\tau_s \geq \tau_s$, $\tau_u \geq \tau_u$, and $\tau_r = \tau_r$, where $s \in S$ and $r \in R$. (The cycle time changes of v, x, y are undetermined.)*

Table 4 summarizes the cycle time changes of nodes $s \in S, u, v, x \in X, y \in Y, r \in R$ after evaluation time change of a node in WCHB or NCL circuits under the four cases discussed in Theorems 13 to 16. Rows 2 to 5 correspond to the cases covered by Theorems 13 to 16, respectively.

Table 4: Cycle Time Change due to Evaluation Time Change (WCHB/NCL)

WCHB, NCL		τ_s	τ_u	τ_v	τ_x	τ_y	τ_r
$u.d_1 > u\ .d_1$	$\sigma(e) > 0$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$=$
	$\sigma(e) = 0$	$\leq$	$\leq$	?	?	?	$=$
$u.d_1 < u\ .d_1$	$\sigma(e) > 0$	$\geq$	$\geq$	?	$\geq$	$\geq$	$=$
	$\sigma(e) = 0$	$\geq$	$\geq$	?	?	?	$=$

Analyzing circuit performance after the evaluation time of a node is changed is similar to that after buffer insertion and buffer removal. We consider the value change of $t(v.rq)$ according to the local slack of $e = (u, v)$ or $e = (u', v)$. First, we consider the case that the evaluation time of a node decreases. If the local slack of edge $e = (u, v)$ is greater than 0, it means $t(v.rq) > t(u.ak) + \delta(u.ak, v.rq)$. The decrease of evaluation time of u will not affect $t(v.rq)$. In contrast, if the local slack of edge $e = (u, v)$ is 0, it means $t(v.rq) = t(u.ak) + \delta(u.ak, v.rq)$, and we know the decrease of evaluation time of u cannot increase $t(v.rq)$. Second, we consider the

case that the evaluation time of a node increases. By replacing node u with u', if the local slack of edge $e = (u', v)$ is greater than 0, it means $t(v'.rq) = t(v.rq) > t(u'.ak) + \delta(u'.ak, v.rq)$. On the other hand, if the local slack of edge $e = (u', v)$ is 0, it means $t(v'.rq) = t(u'.ak) + \delta(u'.ak, v'.rq)$ and we know the value of $t(v.rq)$ must increase.

After deducing the change of $t(x.rq)$ according to $t(v.rq)$, we can get the change of $t(v.dr)$, $t(x.dr)$ and $t(y.dr)$. With all the information, we can compute the cycle times of v, x and y. According to the SPA equations of $t(v.dr)$ of different pipeline templates and the change of $t(v.rq)$, we can compute the cycle time change of a node after circuit modification.

3.4 Incremental Analysis for Precharge Time Change

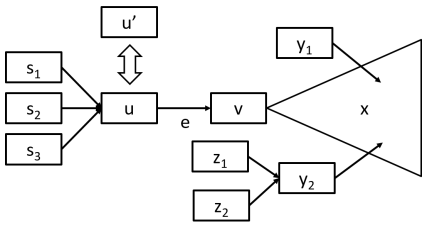


Figure 9: Delay graph under node replacement for precharge time change.

Given a delay graph $G(V, E)$, consider replacing a node, $u \in V$, with $u' \notin V$. We define the following four sets to be used in Theorems 17 to 20.

$$\begin{aligned} S &= \{s \mid s \in FI(u)\} \\ X &= \{x \mid x \in TFO(v)\} \\ Y &= \{y, z \mid x \in TFO(v), y \in FI(x)\backslash\{v\}, z \in FI(y)\backslash\{v\}\} \\ R &= \{r \mid r \notin S \cup X \cup Y \cup \{u, v\}, r \in V\} \end{aligned}$$

For example, in Fig. 9 we have $S = \{s_1, s_2, s_3\}$, X is signified by the cone, $Y = \{y_1, y_2, z_1, z_2\}$, and R consists of the rest of the nodes (not shown in the graph).

The following four theorems can be established.

THEOREM 17. *Given a delay graph $G(V, E)$ of a PCHB or PCFB circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_2 = \delta(u.dr, u.da)$ and $u'.d_2 = \delta(u'.dr, u'.da)$. If $u.d_2 > u'.d_2$, then $\tau_s \leq \tau_s$, $\tau_u \leq \tau_u$, $\tau_v = \tau_v$, $\tau_x = \tau_x$, $\tau_y = \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$.*

THEOREM 18. *Given a delay graph $G(V, E)$ of a PCHB or PCFB circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_2 = \delta(u.dr, u.da)$ and $u'.d_2 = \delta(u'.dr, u'.da)$. If $u.d_2 < u'.d_2$, then $\tau_s \geq \tau_s$, $\tau_u \geq \tau_u$, $\tau_v = \tau_v$, $\tau_x = \tau_x$, $\tau_y = \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$.*

THEOREM 19. *Given a delay graph $G(V, E)$ of a WCHB or NCL circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_2 = \delta(u.dr, u.da)$ and $u'.d_2 = \delta(u'.dr, u'.da)$. If $u.d_2 > u'.d_2$, then $\tau_s \leq \tau_s$, $\tau_u \leq \tau_u$, $\tau_v \leq \tau_v$, $\tau_x \leq \tau_x$, $\tau_y \leq \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$.*

THEOREM 20. *Given a delay graph $G(V, E)$ of a WCHB or NCL circuit, let $u \in V$ to be replaced by $u' \notin V$ yielding a new graph $G'(V \cup \{u'\}\backslash\{u\}, E \cup \{(s, u'), (u', v)\}\backslash\{(s, u), (u, v)\})$ and let $u.d_2 = \delta(u.dr, u.da)$ and $u'.d_2 = \delta(u'.dr, u'.da)$. If $u.d_2 < u'.d_2$, then $\tau_s \geq \tau_s$, $\tau_u \geq \tau_u$, $\tau_v \geq \tau_v$, $\tau_x \geq \tau_x$, $\tau_y \geq \tau_y$, and $\tau_r = \tau_r$, where $s \in S$, $x \in X$, $y \in Y$, and $r \in R$.*

Table 5 summarizes the cycle time changes of nodes $s \in S, u, v, x \in X, y \in Y, r \in R$ after precharge time change of a node under the four cases discussed in Theorems 17 to 20. Rows 2 to 5 correspond to the cases covered by Theorems 17 to 20, respectively.

Table 5: Cycle Time Change due to Precharge Time Change

		τ_s	τ_u	τ_v	τ_x	τ_y	τ_r
$u.d_2 > u\ .d_2$	PCHB, PCFB	$\leq$	$\leq$	$=$	$=$	$=$	$=$
	WCHB, NCL	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$=$
$u.d_2 < u\ .d_2$	PCHB, PCFB	$\geq$	$\geq$	$=$	$=$	$=$	$=$
	WCHB, NCL	$\geq$	$\geq$	$\geq$	$\geq$	$\geq$	$=$

We observe that the influence of changing the precharge time of a node on the cycle time of a circuit is not as huge as that of changing the evaluation time of a node. It is because after the precharge time of a node is modified, the values of $t(p.rq)$ of all the nodes p in a delay graph remain the same. We only need to consider the value changes of $t(p.dr)$ of all the nodes p according to the protocol of the pipeline templates.

4. INCREMENTAL PERFORMANCE OPTIMIZATION

We exploit incremental SPA for performance optimization of QDI pipeline circuits. In particular, buffer insertion and removal are applied for cycle time reduction and area recovery, respectively. According to the theorems in Sections 3.1, the cycle time of a circuit may be reduced via buffer insertion. For a PCHB, WCHB, PCFB, or NCL circuit, the above theorems assert that inserting a buffer w on an edge e with $\sigma(e) \geq w.d_1$ reduces the cycle times of certain nodes. Also we know exactly which nodes should be updated for their cycle times after inserting a buffer.This incrementality reduces the cycle time of the target node while the number of nodes with their local cycle times affected is minimized.After each buffer insertion step, we can use the incremental analysis proposed in Section 3.1 to analyze the affected part of the circuit. We only update the local cycle times of the affected nodes rather than reanalyzing the whole circuit again.Given a target cycle time, buffer insertion can be repeatedly performed at the output of the most critical node until the target cycle time is met or the current cycle time cannot be reduced further. On the other hand, with the repeated buffer insertion it may be possible that some of the earlier inserted buffers become non-crucial in achieving the final cycle time and can be removed for area recovery. Fig. 10 sketches a computation flow implementing the above procedure.

EXAMPLE 3. *Consider the circuit of Fig. 2, which has cycle time 17. Suppose our target cycle time is 14. First, we identify the most critical node v_3 and insert a buffer on its fanout edge e with $\sigma(e) = 2 \geq w.d_1$. The modified circuit is shown in Fig. 4. The cycle time of the original critical node is reduced, and the cycle time of the modified circuit is 15. To further reduce the cycle time, we repeat the process and identify node v_4 to be the most critical node. We insert a second buffer and the new circuit is shown in Fig. 6. The cycle time of this new circuit is 14, achieving the target. To remove redundant buffers, we check if any of the inserted buffers can be removed without affecting the target cycle time. Indeed, the first inserted buffer becomes non-crucial after the insertion of*

```
PerformanceOptimizationViaBufferInsertionRemoval
  input: a QDI pipeline circuit P and target cycle time τ*
  output: an optimized QDI pipeline network Q
  begin
  01  Q := P; τ† := τ*;
  02  τ := StaticPerformaceAnalysis(Q);
  03  LocalSlackAnalysis(Q);
  04  V' := V;
  05  while V' not empty and τ > τ†
  06    choose v ∈ V' with maximum τ_v;
  07    Q := InsertBuffer&UpdateTiming(Q, v);
  08    V' := V'\{v};
  09  if τ > τ†
  10    τ† := τ;
  11  foreach inserted buffer b
  12    τ_b := IncrementalAnalysis(Q, b, REMOVE);
  13    if τ_b ≤ τ†
  14      Q := RemoveBuffer&UpdateTiming(Q, b);
  15  return Q;
  end
```

Figure 10: Algorithm: Performance optimization via buffer insertion and removal using incremental analysis.

the second buffer and can be removed. This removal is possible due to the second inserted buffer affecting the most critical node in the original circuit of Fig. 2.

5. EXPERIMENTAL RESULTS

The proposed incremental analysis and optimization algorithms were implemented in the C++ language under the ABC synthesis and verification environment [3]. All experiments were conducted on a Linux machine with a Xeon 2.3GHz CPU and 32GB RAM. Large benchmark circuits from ISCAS and ITC were selected for the experiments. To apply our incremental static performance analysis on acyclic pipelines, the circuits were made combinational before synthesized and mapped into asynchronous pipelines by the flow in [5]. In the following, only PCHB results were shown as similar results were observed for PCFB, WCHB, and NCL circuits.

We compared our incremental performance optimization algorithm, which relies on incremental SPA, against the prior slack matching method [5] by level balancing, which relies on SPA. Three experiments were conducted under three timing library settings for different distributions of evaluation delays and precharge delays among the library cells. The first experiment was conducted based on the library of [5] created from the Predictive Technology Model (PTM) 180 nm technology file [20]. The second and third experiments were conducted based on the libraries modified by gradually increasing the delay distributions of the library cells to study the robustness of the compared optimization algorithms against varied library characteristics. The results are shown in Table 6 with three parts for ranges 75-125 (ps), 75-225 (ps), and 75-525 (ps). In the table, Column 2 shows the number of pipeline modules in a circuit; Columns 3 and 4 show the original cycle time τ and the target cycle time 0.8τ, respectively; Columns 5-7 show the results of [5] in terms of the achieved cycle time, the number of inserted buffers, and the CPU time in seconds, respectively; Columns 8-10 show the results of our method in terms of the achieved cycle time, the number of inserted buffers, and the CPU time in seconds, respectively. The numbers printed in bold face indicate that the produced cycle times do not meet the targeted 0.8τ.

As evident from Table 6, our method outperforms prior work [5] in runtime with an average of two orders of magnitude speedup. On the other hand, when cycle time and the number of inserted buffers were compared, the compared two methods yielded results with varied qualities. Under the 75-125 range, the results of the two methods are comparable. For circuits `s38417`, `b17`, `b21`, prior work inserted fewer buffers than our method. Under the 75-225 range, our method yielded substantial savings on buffers inserted in certain circuits, such as `s35932`, where the target cycle time was not attained by [5] and thus yielding a large number of inserted buffers. This non-optimality of prior work amplifies along the increase of library cell delay range. Under the 75-525 range, the target cycle times of a large portion of the circuits cannot be attained by prior work, but all were attained by our method. The above phenomenon can be understood in that, when the delay range is small, balancing pipeline levels corresponds well to the reduction of data stall time in the pipeline network. In contrast, when the timing range is large, balancing pipeline levels does not necessarily balance the token arrival times at pipeline modules. Therefore, our method is more scalable and effective for performance optimization, and more robust against library migration.

We note that there are other slack matching methods for asynchronous circuit performance optimization, such as [4, 8, 17, 19, 25]. However, most prior methods cannot directly analyze half buffer pipelines. Different performance optimization methods may rely on different performance analysis methods that may impose different timing constraints. Moreover, other performance analysis methods are not as scalable as SPA. As was studied in [26], an experimental comparison between mixed integer linear programming (MILP) based slack matching [1] and SPA based slack matching [5] suggested the superior scalability of the latter method. Therefore in this work we focused on the comparison with [5].

6. CONCLUSIONS AND FUTURE WORK

We have proposed an incremental approach to static performance analysis of asynchronous pipeline circuits. Cycle time changes due to local circuit modifications, including buffer insertion, buffer removal, node evaluation delay change, and node precharge delay change, have been analyzed. In addition, our incremental SPA have been applied for iterative performance optimization of asynchronous pipeline circuits. Experimental results have demonstrated significant speedup and notable cycle time improvement, especially for circuits with wide delay distributions of pipeline modules, compared to prior SPA-based performance optimization. Our approach is promising for scalable optimization of large asynchronous designs. For future work, we plan to extend our method to cyclic pipeline circuits based on [23].

REFERENCES

[1] P. A. Beerel, G. D. Dimou, and A. M. Lines. Proteus: An ASIC Flow for GHz Asynchronous Designs. *IEEE Design & Test of Computers*, 28(5): 36-51, 2011.

[2] D. Bufistov, J. Julvez, and J. Cortadella. Performance Optimization of Elastic Systems Using Buffer Resizing and Buffer Insertion. In *Proc. Int'l Conf. on Computer-Aided Design*, pp. 442-448, 2008.

[3] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification.* `http://www.eecs.berkeley.edu/~alanmi/abc/`

[4] P. Beerel, A. Lines, and M. Davies. Logic Synthesis of Multi-Level Domino Asynchronous Pipelines. U.S. Patent No. 8,051,396, 2011.

[5] C.-C. Chuang, Y.-H. Lai, and J.-H. R. Jiang. Synthesis of PCHB-WCHB Hybrid Quasi-Delay Insensitive Circuits. In *Proc. Design Automation Conference*, pp. 192:1-192:6, 2014.

[6] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Tran. Information and Systems*, 80(3): 315-325, 1997.

[7] K. Fant and S. Bandt. *Null Convention Logic^{TM} System.* US patent 5,305,463, 1994.

[8] G. Gill and M.Singh. Bottleneck Analysis and Alleviation in Pipelined Systems: A Fast Hierarchical Approach. In *Proc. Int'l*

Table 6: Experimental Results of Performance Optimization Using Static Analysis

circuit	library cell delay range: 75-125								
	# node	original τ (ns)	target τ (ns)	[5]			Ours		
				τ (ns)	# buffer	run time (s)	τ (ns)	# buffer	run time (s)
c1355	147	0.694	0.555	0.554	96	0.021	0.554	96	0.001
c5315	803	1.394	1.115	1.110	622	2.000	1.110	622	0.065
c6288	573	3.164	2.531	2.512	38	6.500	2.485	36	0.029
c7552	893	1.158	0.926	0.924	162	0.530	0.924	162	0.018
s35932	6723	0.675	0.540	0.535	4736	37.000	0.535	4736	2.000
s38417	6313	1.402	1.122	1.120	1899	67.000	1.120	1903	1.480
s38584	7020	1.512	1.210	1.209	603	78.000	1.209	602	1.085
b17	13354	4.194	3.355	3.355	2116	9600.000	3.355	2225	90.000
b20	5541	2.920	2.336	2.336	750	510.000	2.336	746	2.500
b21	5789	2.928	2.342	2.342	628	520.000	2.341	639	1.925
b22	8247	3.054	2.443	2.443	1073	2200.000	2.443	1070	4.800
average ratio		1.000	0.800	0.798	1.000	1.000	0.797	1.001	0.023
circuit	library cell delay range: 75-225								
	# node	original τ (ns)	target τ (ns)	[5]			Ours		
				τ (ns)	# buffer	run time (s)	τ (ns)	# buffer	run time (s)
c1355	147	1.179	0.943	0.941	56	0.008	0.941	56	0.001
c5315	803	2.092	1.674	1.673	329	1.726	1.673	329	0.082
c6288	573	4.714	3.771	3.745	48	6.459	3.766	45	0.089
c7552	893	1.844	1.475	1.471	110	0.526	1.470	104	0.024
s35932	6723	1.135	0.908	**0.922**	5472	8.486	0.908	895	0.505
s38417	6313	2.194	1.755	1.755	1331	65.457	1.755	1333	1.227
s38584	7020	2.288	1.830	1.830	245	150.414	1.830	245	1.008
b17	13354	6.042	4.834	4.833	2015	12365.600	4.833	1961	193.810
b20	5541	4.376	3.501	3.500	408	976.437	3.500	400	1.144
b21	5789	4.548	3.638	3.637	487	948.253	3.637	487	1.289
b22	8247	4.577	3.662	3.661	682	1918.500	3.661	669	2.736
average ratio		1.000	0.800	0.800	1.000	1.000	0.799	0.908	0.029
circuit	library cell delay range: 75-525								
	# node	original τ (ns)	target τ (ns)	[5]			Ours		
				τ (ns)	# buffer	run time (s)	τ (ns)	# buffer	run time (s)
c1355	147	2.435	1.948	**2.159**	128	0.004	1.944	86	0.003
c5315	803	4.679	3.743	**3.860**	2953	1.461	3.743	339	0.308
c6288	573	10.094	8.075	**8.186**	6756	6.074	8.075	60	0.179
c7552	893	3.936	3.149	**3.484**	1291	0.432	3.146	108	0.082
s35932	6723	2.425	1.940	**2.237**	5472	8.202	1.940	597	1.179
s38417	6313	4.608	3.686	**3.804**	14398	43.887	3.686	1381	3.509
s38584	7020	5.076	4.061	**4.272**	19369	88.535	4.060	179	2.726
b17	13354	13.352	10.682	10.681	2150	10760.500	10.681	2012	422.831
b20	5541	9.265	7.412	7.412	796	615.613	7.412	786	8.381
b21	5789	9.232	7.386	7.384	864	704.918	7.384	844	15.216
b22	8247	9.613	7.690	7.689	1378	1999.840	7.689	1375	36.046
average ratio		1.000	0.800	0.836	1.000	1.000	0.800	0.454	0.148

Symp. on Asynchronous Circuits and Systems, pp. 195-205, 2009.

[9] A. Kondratyev and K. Lwin. Design of Asynchronous Circuits Using Synchronous CAD Tools. *IEEE Design & Test of Computers*, 19(4): 107-117, 2002.

[10] Y.-H. Lai, C.-C. Chuang and J.-H. R. Jiang. A General Framework for Efficient Performance Analysis of Acyclic Asynchronous Pipelines. In *Proc. Int'l Conf. on Computer-Aided Design*, pp. 736-743, 2015.

[11] A. Lines. Pipelined Asynchronous Circuits. M.S. thesis, California Institute of Technology, 1995.

[12] M. T. Moreria, M. E. Arendt, R. A. Guazzelli, and N. L. V. Calazans. A New CMOS Topology for Low Voltage Null Convention Logic Gates Design. In *Proc. Int'l Symposium on Asynchronous Circuits and Systems*, pp. 93-100, 2014.

[13] A. Martin and M. Nyström. Asynchronous Techniques for System-on-chip Design. *Proc. of the IEEE*, 94(6): 1089-1120, 2006.

[14] P. McGee and S. Nowick. An Efficient Algorithm for Time Separation of Events in Concurrent Systems. In *Proc. Int'l Conf. on Computer-Aided Design*, pp. 180-187, 2007.

[15] M. Moreira, A. Neutzling, M. Martins, A. Reis, R. Ribas, N. Calazans. Semi-Custom NCL Design Commercial EDA Frameworks: Is it Possible? In *Proc. Int'l Symposium on AsynchronouCircuits and Systems*, pp. 53-60, 2014.

[16] J. Magott. Performance Evaluation of Concurrent Systems Using Petri Nets. *Information Processing Letters*, 18: 7-13, 1984.

[17] M. Najibi and P. A. Beerel. Slack Matching Mode-Based Asynchronous Circuits for Average-Case Performance. In *Proc. Int'l Conf. on Computer-Aided Design*, pp. 219-225, 2013.

[18] S. M. Nowick and M. Singh. Asynchronous Design - Part 1: Overview and Recent Advances. *IEEE Design & Test*, 32(3): 5-18, 2015.

[19] P. Prakash and A. Martin. Slack Matching Quasi Delay-insensitive Circuits. In *IEEE Intl. Symp. on Asynchronous Circuits and Systems*, pp. 195-204, 2006.

[20] Nanoscale Integration and Modeling Group. *Predictive Technology Model*. `http://ptm.asu.edu/`

[21] C. Ramamoorthy and G. S. Ho. Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets. *IEEE Trans. Software Eng.*, SE-6(5): 440-449, 1980.

[22] R. B. Reese, S. C. Smith, and M. A. Thornton. Uncle - An RTL Approach to Asynchronous Design. In *Proc. Int'l Symposium on Asynchronous Circuits and Systems*, pp. 65-72, 2012.

[23] C.-H. Shih, Y.-H. Lai and J.-H. R. Jiang. SPOCK: Static Performance Analysis and Deadlock Verification for Efficient Asynchronous Circuit Synthesis. In *Proc. Int'l Conf. on Computer-Aided Design*, pp. 442-449, 2015.

[24] J. Sparso and S. Furber. *Principle of Asynchronous Circuit Design - A Systems Perspective*, Springer, 2002.

[25] G. Venkataramani and S. C. Goldstein. Leveraging Protocol Knowledge in Slack Matching. In *Proc. Int'l Conf. on Computer-Aided Design*, pp. 724-729, 2006.

[26] Withheld publication for blind review.

SystemCDG: AI Based Coverage Driven Stimuli Generation for SystemC

Jannis Stoppe[1,2] Arved Friedemann[1] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Group of Computer Architecture, University of Bremen, 28359 Bremen, Germany

***Abstract*—Due to the ever-increasing complexity of hardware systems, the design process is progressively being transferred from the *Register-Transfer Level* (RTL) to the *Electronic System Level* (ESL). On the ESL, system prototypes are developed using existing high-level software languages, resulting in earlier results and shorter development iterations.**

The current de-facto standard for ESL designs is SystemC, a C++ library for hardware design. C++, while being a fast and established language, is hard to analyse, giving tools much less information to work with than for RTL designs.

Stimuli generation tools suffer from this problem as they are unable to retrieve detailed knowledge about the internal connections of SystemC modules and thus cannot directly be transferred from RTL to ESL. In this paper, we propose a data mining approach that hooks into a given SystemC design, learns its structures during its simulation and generates stimuli to set signals within the system.

I. Introduction

As the amount of transistors per surface area keeps rising, so does the complexity of hardware.

While traditional hardware development on the Register Transfer Level (RTL) relies on methods such as the re-use of existing parts in a modular fashion to be able to create more complex designs [2], the ability to keep up with the potential that is offered by the technological advance is still limited. This discrepancy is also known as the *design gap* [21], describing the divergence between what can be manufactured and what can be designed.

Hardware Design Languages (HDLs) traditionally define a system in a way that allows it to be translated into hardware. Hence, with an increasing hardware complexity, the complexity of the description rises as well. Another approach to handle this issue is to increase the abstraction of the design language. A language that allows for hardware to be described in a way that is not necessarily synthesizeable (i.e. translatable into hardware) but that still describes the behaviour of the system appropriately could allow designers to quickly describe a system on a broad scale and then iteratively refine it in order to be manufactured. As this approach differs fundamentally from the way hardware is described traditionally, it is placed on a conceptually different layer: the Electronic System Level (ESL).

Design on the ESL usually relies on existing, high-level programming languages that allow engineers to rapidly describe a certain behaviour. One way to expand existing languages with the required behaviour is to provide libraries for these languages that come with structures that can be used to model a hardware design (such as signals or modules) and a simulation kernel that runs the design.

The de-facto standard [19] for ESL hardware design is SystemC. It is a C++ library that allows designers to build simulated hardware prototypes using C++, facilitating them with the ability to rely on a vast amount of existing C++ libraries to describe the system's behaviour while at the same time providing them with a language that provides good performance and the ability to handle memory as required.

Building ESL designs using SystemC/C++ also results in issues concerning the analysis of a given design though. While traditional hardware models are static and provide tools with a structure to base other methods on, a compiled SystemC design is optimized by the compiler to run fast, removing any information that is not needed for its execution. This means that methods that rely on the analysis of a given design are hard to apply to ESL designs in general and SystemC designs in particular.

One such method is Coverage Directed test Generation (CDG) [22]: with a system's structure being unavailable, generating stimuli directed at certain signals becomes a non-trivial task.

This paper describes an approach to use machine learning to learn about a given SystemC design while it is being simulated in order to generate stimuli that target signals at arbitrary positions within the design.

The remainder of this paper is divided into sections describing SystemC and the idea of machine learning in sections II and III. It then describes the approach and the issues it solves in Section IV, provides an evaluation in Section V and ends with a short conclusion, wrapping up the contribution.

II. SystemC

SystemC is a C++ library for hardware design on the Electronic System Level (*ESL*).

It provides classes to model hardware systems such as modules and signals. SystemC also comes with a simulation kernel that takes care of scheduling and signal propagation, allowing designers to simulate the design they prototyped.

The execution of a SystemC design is divided into two parts, the *elaboration phase* and the *simulation phase*. The former is used to create a design instance based on the previously written classes. A class representing a certain module can be instanciated any number of times, representing modules that are re-used throughout the design. This creation of all needed parts of the design is itself part of it and thus also written in C++, allowing any needed construct to be used to describe the hardware model. This means that even the creation of a design can be described using e.g. complex loop structures or dependencies on user-input, making it impossible to deduce a system's structure from its source code. The latter is used to simulate the system, running the design that was created in the previous phase.

By being built on top of C++, SystemC designs allow the designers to utilize a vast amount of existing libraries. This

way, a module's functionality can quickly be implemented without taking care of its hardware structure to ensure that a design works as intended.

This principle enables the Hardware/Software Co-Design at this level: As any C++ constructs can be used to implement a module's logic, the designer does not need to decide whether a module's functionality should be implemented in hardware or software or a mixture of both. Instead, the implemented module is simulated using its C++ implementation which may represent hardware, software or a mixture of both.

However, as C++ compilers are following the principle of "what you don't use, you don't pay for" [26], they heavily optimize the resulting executable. While this may result in a speed advantage in certain cases, it also means that the result is harder to analyse. As the compiled program does not contain anything that is not needed for execution, any information that is not required for running the program (such as names of methods, fields or (most) classes) is removed from the program and even the structure of the code may be altered. This means that a lot of techniques that are available in other high-level languages such as C# or Java cannot be used in C++. Most importantly, C++ does not support reflection or introspection beyond a very fundamental level, making the retrieval of meta data about a program that is being executed far from trivial.

Especially the modules' internals are black boxes. Due to being implemented in plain C++, there is no universal way to retrieve their internal connections. This inability to retrieve a static netlist-like representation of a system represents the major barrier for porting hardware analysis approaches from the RTL to the ESL.

One such application that cannot be trivially translated to the ESL is stimuli generation. Methods such as e.g. [28] on the RTL rely on the knowledge about a system's internal connections: In order to trigger certain signals, the system is analysed and the stimuli are generated accordingly. Without the ability to retrieve a system's correct interconnections, any such method is unable to generate any meaningful output.

Current SystemC analysis methods focus on the difficulties of extracting the data from a description language that may be highly dynamical and written using various dialects, usually attempting to either parse the source code directly [8], [4], [1] or hook into a particular compiler [18], [11] to retrieve the information. This kind of approach has the advantage of getting a detailed code analysis via the tools that are handling the code anyway, but it essentially limits the available code foundation to a supported subset. If the existing projects are written using different dialects, this quickly becomes a major issue concerning the application of any such approach.

III. AI Analysis

If the system itself cannot easily be analysed to collect the required knowledge, the computer can still learn how the system works by watching it.

Instead of attempting to retrieve a complete model using these methods, other, less detailed approaches are valid courses of action. One such approach is to accept the modules as impenetrable black boxes and learn about their structure by observing their accessible interfaces. This basic idea is the foundation of data mining or AI approaches that analyse given systems solely based on their observable behaviour.

This shift in paradigm has the important distinction that the resulting models may be incomplete or even wrong. Still, methods that do not require complete models but work well even using incomplete or incorrect data may benefit from this approach.

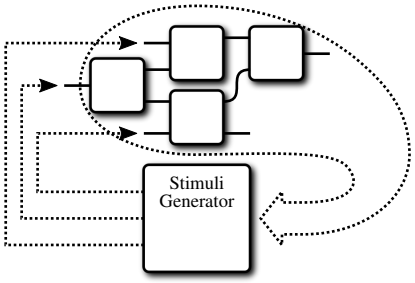


Fig. 1. The machine learning feedback loop: a stimuli generator observes the system and generates new input based on the generated model.

One such case is *Coverage Driven Stimuli Generation* (CDG). The idea of CDG is to generate stimuli that specifically aim at improving certain coverage values. If e.g. a certain signal should be switched to increase its toggle coverage (a value that measures how often a signal was switched from 0 to 1 and vice versa), specific input stimuli need to be generated to achieve this.

For RTL designs, the required input values can be calculated by going "backwards" through the netlist (i.e. from the considered signal towards the input signals), calculating possible signal assignments. On the ESL, this is not possible as the modules' internal dependencies cannot easily be calculated.

The proposed approach using an AI that learns how the modules work internally is especially valid for CDG though: when generating stimuli, making a wrong guess does not necessarily result in a worse result. Instead, the AI can use the new, unpredicted case to refine its model and generate the successive stimuli accordingly, thus improving with each mistake as illustrated in Figure 1. This way, the model is refined along the way, getting better with each unpredicted case, which in turn results in an improved ability do steer the simulation into any desired direction.

We therefore propose a scheme that uses an AI as a stimuli generation module for SystemC designs in order to generate inputs directed at improving certain coverage values during the simulation of the ESL prototypes.

IV. ESL AI Based Stimuli Generation

The basic idea of the ESL CDG scheme is to have an AI analyse the system's structures in order to be able to generate stimuli that trigger certain signals within the system that would otherwise be hard to trigger.

A. State of the Art

Several algorithms have been developed to generate test patterns for a system simulation.

However, only few focus on SystemC in particular: most CDG approaches focus on the Register-Transfer-Level or Gate-Level, generating stimuli for systems that are guaranteed to be synthesizeable.

These approaches all follow the same pattern of establishing a feedback loop that connects the observations of a system's behaviour to new stimuli that feed the system itself. This loop

is then used to verify assumptions about the system and refine the model that is generated from the observations [13].

Currently, there are three major classes of approaches to generate test patterns for the devices using machine learning:

- Evolutionary Algorithms (e.g. [29], [5]). These are based on methods that simulate evolutionary processes to optimize a set of individuals against a fitness criterion. The advantage of this approach is that it corresponds nicely to a given coverage value: individuals (corresponding to a certain set of stimuli) that have already gained a better coverage are the ones that are used to generate the next generation of individuals, resulting in an improving performance.
- Probabilistic models such as Bayesian Networks or Markov Models (e.g. [9], [27]). These rely on probabilistic connections between nodes in directed graphs, modelling the probabilities how a system changes its state as labels for transitions between a graph's nodes.
- Data Mining approaches that attempt to build comprehensive models from a vast set of recorded data (e.g. [16])

The resulting models are then used to generate new stimuli to test the simulated device.

SystemC approaches in contrast focus on the generation of the stimuli themselves, specifically not targeting signals within the model. Frameworks such as CRAVE [14] allow for a detailed specification of how the stimuli are supposed to be generated but are not able to generate stimuli that target a specific signal within the model.

While the general approach of watching the execution and generating stimuli based on this performance is portable, algorithms for other levels of abstractions of course use the information available from the according descriptions.

One defining feature of development on the ESL is that the designer does not have to specify a synthesizeable system but may as well leave decisions concerning implementation details for later. Hence, any algorithm focusing on CDG for the ESL can not rely on information such as the final hardware structure and its connections as this information does not exist yet.

To summarize, existing approaches either target synthesizeable languages in order to be able to set signals within the design or focus on SystemC but do not support setting signals within the system.

B. CDG for SystemC

One major design aspect is that due to the difficulties in analysing SystemC designs, no a priori information should be required for the approach to be applied. This point is rather closely connected to non-intrusiveness, which is a desired trait of any approach handling SystemC designs. As the stimuli generator should work on all given designs without any further alteration being required, no knowledge can be taken for granted that cannot be extracted from the standard interfaces that are available anywhere.

The foundation for the given approach is the *C4.5* algorithm [23], which is widely used in diverse applications as computer vision [10], language processing [17], medical diagnosis [30], financial analysis [20], general game playing [24], robotics [3] and others.

C4.5 takes a given set of data that consists of an arbitrary amount of input variables, each providing n stimuli and a single target variable that provides n results, and splits it into more "tidy" sub-sets with regard to a variable, based on its assignments.

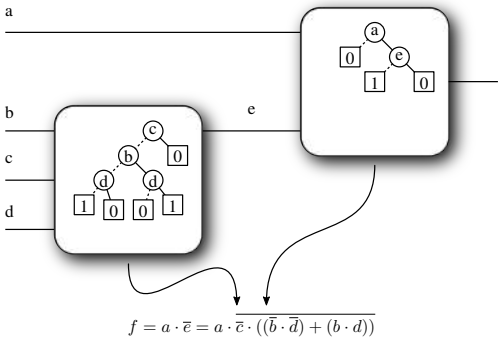


Fig. 2. Several trees may be merged over module borders to retrieve the formula for a single output.

All possible splits are evaluated by means of the entropy function $-(p_0 \log_2(p_0) + p_1 \log_2(p_1))$, where p_0 (p_1) denotes the probability of the considered signal being set to 0 (1) according to the data set. The set is then divided to minimize the entropy value for the resulting sub-sets (weighed by their size) and the algorithm starts again for each newly created subset. If the entropy value can no longer be improved, the algorithm terminates, returning the according decision tree.

There are some issues when applying an AI approach to generic SystemC designs though. These issues and the approaches to overcome them are outlined below.

a) SystemC's modularity: this is a concept that differs from the approach of building a single decision tree for a given set of data.

SystemC designs usually contain several modules that interact via ports and signals. While it would be possible to build a single, large decision tree for the signal that is supposed to be set (which would have to take all connections to the inputs into account), building smaller models for the modules results in a better division of labour, allowing the machine learning approach to work on small, independent problems and re-using results for modules that are used repeatedly.

The approach therefore is to use this pre-defined modularity to first divide the learning problem into smaller parts, generate models for each individual class of modules and later merge these parts to be able to generate the inputs that are needed for a particular result. Figure 2 illustrates this method, building decision trees for each module which can then serve as a foundation to calculate a general formula for any given output signal.

The proposed implementation takes the trees for the desired result, translates them into boolean statements and merges these. The resulting formula is then given to a SAT solver [7] to determine a set of input stimuli that sets the output in the desired way (if the current assumptions about the design hold).

b) No pre-defined point in time for the extraction: a SystemC simulation does not have to follow the traditional clock scheme.

While SystemC allows for clocks to be used, they are not required for a system to work. Instead, the designer may e.g. simply tell the simulation kernel to wake up a given thread

after an arbitrary amount of simulation time and generate new input for the design.

This makes it hard for any algorithm to determine any point in time when a system's status should be observed and logged for further analysis.

When a clock is attached to the stimuli generation module, it uses the clock's period information to invoke itself a δ-cycle (which denotes the minimum amount of time that may pass in a SystemC simulation) before the next clock edge is set in order to wait for the system to have reached a stable state with all signals and variables being set coherently.

If the system does not rely on a clock to update its modules, the stimuli generator needs to be invoked manually (i.e. via a method call) each time the system has reached its next state.

c) Time delays: these may result in a given signal to affect the output an arbitrary amount of cycles after it has been set.

A module that is connected to certain signals may have any of its methods set to be *sensitive* to any of these signals, meaning that the given method will be executed if the given signal alters its value. This is usually used to calculate a module's outputs when a certain input changes. As an example, several modules may be connected in a straight line, with all of them being connected to a clock. The method that recalculates a module's output can be set to be sensitive to the signals a_n and b_n or the clock, triggering its behaviour when the respective signal is set to a new value. Assume that all signals, a_n, b_n and the clock, are set at time $t = 0$. If the modules are set to be recalculated upon a change of signals a_n or b_n, a change in the respective signal will trigger a chain of recalculations that travel through the system as each module will recalculate its output as soon as the result of the previous module is available. If, however, the modules only react to changes in the clock signal, the result will take one clock cycle per intermediate module to travel through the system as the method will not be triggered when a previous module has finished its calculations.

As this means that an arbitrarily large delay may be introduced in a given system, the machine learning algorithm may not assume that a result is available once all calculations have been performed.

However, unless a module stores a value internally, it cannot delay the calculation and propagation for longer than a single time step.

Our solution is therefore to feed the machine learning algorithm for a module with the current and the previous signal assignments as inputs. If the machine learning algorithm detects any dependency to the previous time step, the formulas are generated with the additional variables that represent the previous time step. The delay is propagated backwards when merging the trees into formulas, rising with each required delay.

If the stimuli generator then targets a given signal and attempts to assign a given value, it needs to start the according amount of steps in advance, assigning the according variables at the given steps until the result can be read and compared to the expected one.

d) Internal states: SystemC programs may store values at arbitrary locations in the program. Modules may contain internal variables, the program may contain global variables that are accessed from anywhere and objects may be used out being referenced at all times.

The simplest solution is to require the program to adhere to certain standards. If all internal values were stored using signals instead of native variables, the stimuli generator could find them via the SystemC-API and retrieve the according values without any further intervention.

If, however, this approach cannot be applied (e.g. because an existing design relies heavily on the usage of field variables and was not meant to be refactored), the information can instead be gathered by reading the object's state from memory. While this approach works in many cases, it is easy to come up with corner cases that are hard to cover appropriately: In order to read e.g. a pointer's value instead of merely the memory address, the stimuli generator needs to know the object's structure. While this information can be gathered from the debug symbols [25], the question still remains which of the information is required. Iterating through memory, gathering all states that are somehow accessible for a given module, regardless or being logically related or not, results in a tremendous amount of data that could theoretically influence a module's output, increasing both, the memory footprint and the computation time for the machine learning algorithm.

e) User-defined datatypes: As the designer may define any arbitrary type to be used for communication on a signal, the stimuli generator may be required to handle these.

While the stimuli generator could provide the required signal assignment by creating an object that simply has the required bits set to a specific value, the current implementation does not offer this feature. There are several reasons we so far decided against the automatic generation of bitmasks for user-defined types:

- The creation of new instances of these types from bitmasks is an inherently unsafe operation as the resulting object may not adhere to the contraints required by operations that work on this object.
- Pointers cannot easily be detected, so types that contain references to other parts in memory can neither be analysed nor created automatically.
- The size of types may not be known at runtime as C++ is not type-safe and the type information of objects is usually discarded by the compiler.

We therefore suggest to supply the stimuli generator with methods to handle the according types. This way, while requiring the designer to write additional code, the underlying SystemC code base does not need to be adapted. As the different amount of types that are used on signals should be manageable, we consider this an acceptable trade-off to avoid the issues that arise from an automatic translation of arbitrary types.

f) Circles: SystemC designs may contain circles.

This may pose a problem especially with regard to the idea of iterating through the modules and combining their learned models into one large formula over the consecutive time steps.

In order to handle circles appropriately, we propose a method that builds the model in two steps, illustrated in Figure 3:

First, the machine learning algorithm builds its models per module and iterates backwards through the system to calculate the time delays for its model, *avoiding* any circles. In this step, no circles are handled. While this algorithm calculates a potentially incomplete model, it is guaranteed to terminate. The amount of timesteps for calculating the given output is stored for the next step.

Second, the same algorithm is executed but circles are unrolled over time just like the remaining circuit to a depth that matches the one calculated in the previous step. This means

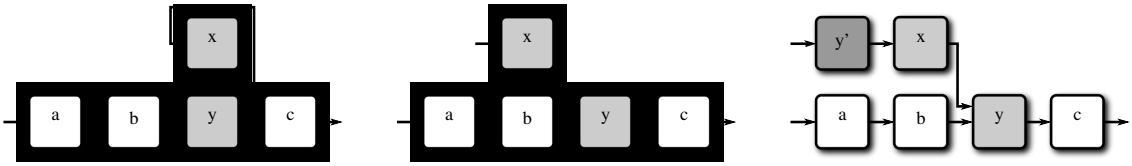


Fig. 3. Circles (gray modules, left) are not handled in the machine learning algorithm's first analysis iteration, with the process stopping as soon as any element would appear the second time (center). They are unrolled in a second step as far as required to be on par with the elements going the furthest back in time (dark gray module, right).

that for a design that requires the stimuli generator to plan n steps ahead, circles are unrolled exactly n steps, enabling the generator to take circular dependencies into account for its dependencies.

With the given methods combined, a stimuli generator can be written that properly handles a given SystemC design. This generator is implemented as a module that reads all signals of the ESL design it is part of and hooks into all unused input ports before the simulation starts in order to provide stimuli to the system. When a target is met (or missed) it extends the model with the newly added information from the behaviour since the last learning process and generates the next input, which is then fed to the system (over an arbitrary amount of time) until the output can again be observed. It is able to target *specific signals deep within the system* in order to e.g. boost coverage of certain signals *without any a priori knowledge about the system itself*.

The proposed solution currently does not handle Transaction Level Modelling (TLM) designs. The TLM framework that has been incorporated into the SystemC standard and its reference implementation not only features generic payloads to travel along the modules' connections but more importantly offers timing modifications in order to be able to more quickly calculate certain values [12]. This leads to designs that "drift apart" concerning the timing, with parts of the model waiting for other parts to catch up. It also is a problem for the suggested time delay computation which requires the model to provide intermediate results at the according points in the simulation time. Extending the algorithm to be able to handle designs using the TLM framework properly therefore remains an open question that may hopefully be handled in a future work.

The usage of a machine learning algorithm that recursively splits a given data set means that the input values need to have some kind of natural order. C4.5 works best when the input data is an enumeration or binary input (e.g. a boolean value or a set of just a few values that are not connected in any way) and is able to handle e.g. floating point values as they can be ordered, values such as strings essentially cannot be handled well. However, due to the domain of hardware design, we argue that signals and states usually rely less on such parameters than other programs, allowing the approach to still be applied to a wide variety of designs.

Additionally, the approach relies on being able to actually learn a module's behaviour. A module that implements a function that is supposed to be tough to guess or that contains a lot of inputs, outputs and according interdependencies is inherently hard to learn and thus will not work well with the presented approach. Especially trapdoor functions such as hash value computation or encryption methods are basically incompatible to the given approach: as a hash function serves the purpose of not allowing a specific output to be generated at will, trying to make an AI learn to do just that is a rather difficult task at best.

Still, the proposed approach represents an easy-to-use implementation that is close to a "fire and forget" application, allowing the designer to add the given stimuli generator to a design that connects itself as required and generates stimuli to boost a certain fitness criterion as long as it is either connected to a clock or triggered when the system has reached its next state.

TABLE I. EXPERIMENTAL RESULTS

benchmark	rnd cycles / s	ml cycles / s
treeFunctional2	3,184 / **0,095109**	**1,008** / 0.377791
treeFunctional4	9,759 / **1,13084**	**1,021** / 2.08185
treeFunctional6	40,680 / 18,114	**1,181** / **11.0289**
treeFunctional8	153,046 / 300,413	**1,478** / **62.2137**
treeDelayed2	3,282 / **0,097361**	**2,482** / 0.968074
treeDelayed4	10,207 / **1,24202**	**4,713** / 6.8692
treeDelayed6	40,680 / **19,374**	**9,208** / 59.4449
treeDelayed8	167,795 / **329.512**	**14,210** / 413.925
treeInternal2	2,278 / **0,063881**	**2,051** / 0.904289
treeInternal4	4,330 / **0,465483**	**3,827** / 7.155
treeInternal6	9,374 / **3,94015**	**6,051** / 51.6372
treeInternal8	14,470 / **24,3296**	**8,043** / 305.583

Amount of simulated cycles / real time until the model's outputs were toggled 1000 times using either random stimuli generation or the proposed method.

V. EVALUATION

To test the given approach, we wanted scalable circuits that could be increased in size in order to see how the approach would perform.

treeFunctional$n$ is a tree of SystemC modules, each realizing an arbitrary boolean gate with inverters inserted at random locations. The amount of inputs available to the stimuli generator is 2^n, with the tree consisting of n levels of modules. The test terminated as soon as the output signal on the top of the tree was toggled 1000 times.

treeDelayed$n$ has each module (including the not gates) connected to a clock and only refreshes its outputs upon a clock tick, resulting in varying amounts of ticks until the signal passed the tree and the result is set.

treeInternal$n$ enriches this test with modules that have an internal state that is toggled with each clock cycle and switches its behaviour between being a NOR and an OR gate. For the machine learning algorithm, this results in a circular dependency, with the signal depending on its own previous state (or the clock, which also depends on its own previous state) that need to be handled.

While the machine learning algorithm always needs less simulated cycles than the random stimuli generator to reach the toggle goal, it is slower in real time when the chip has dependencies over multiple timestamps. However, the example data set shows that from a certain model size, applying

the suggested algorithm to functional models results in an increased performance compared to random generation.

The toggle coverage, which the algorithm is supposed to increase, grows faster when the stimuli generator is applied to the system – on signals that are only indirectly connected to the generator.

The CDG approach is able to target certain signals but does so at a cost. Signals that are e.g. toggled regularly by a random stimuli generator may be toggled more quickly by the machine learning module concerning the system's clock, but the computation of the stimuli results in a slower simulation speed, which may result in a real time performance decrease when relying solely on the given approach. The sensible approach thus is to use the proposed method only for signals that are not easily toggled, starting with a broad random generation first, as it is common practice on the RTL [15].

The bottom line is that the approach is able to target given stimuli at will, bypassing the issue of black boxes inbetween and setting signals withing a given SystemC model by learning the modules' behaviour alone. It thus provides an easily applicable solution to analyse a system and trigger a certain behaviour without requiring any manually made system descriptions.

VI. Conclusion

We have presented an approach to apply a machine learning algorithm to SystemC designs, allowing for stimuli generation directed at signals deep within the model without any further knowledge about the system itself. This allows for a stimuli generator to target signals in a SystemC design without any deeper analysis of the underlying source code or model, resulting in an easily applicable approach to boost the coverage of a given SystemC simulation.

References

[1] Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. Scoot: A tool for the analysis of systemc models. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–470. Springer, 2008.

[2] Pierre Bricaud. *Reuse methodology manual: for system-on-a-chip designs*. Springer Science & Business Media, 2012.

[3] James Brusey and Lin Padgham. Techniques for obtaining robust, real-time, colour-based vision for robotics. In Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*, volume 1856 of *Lecture Notes in Computer Science*, pages 63–73. Springer Berlin / Heidelberg, 2000.

[4] Javier Castillo Villar and Pablo Huerta. SystemC to Verilog Synthesizable Subset Translator. http://opencores.org/project,sc2v, 2010. Accessed 2014-12-23.

[5] Adriel Cheng and Cheng-Chew Lim. Optimizing System-on-Chip Verifications with Multi-Objective Genetic Evolutionary Algorithms. *Journal of Industrial and Management Optimization*, 10(2):383–396, 2014.

[6] Rolf Drechsler and Daniel Große. Reachability analysis for formal verification of systemc. In *Digital System Design, 2002. Proceedings. Euromicro Symposium on*, pages 337–340. IEEE, 2002.

[7] Niklas Een and Niklas Sörensson. Minisat: A sat solver with conflict-clause minimization. *Sat*, 5, 2005.

[8] Görschwin Fey, Daniel Große, Tim Cassens, Christian Genz, Tim Warode, and Rolf Drechsler. ParSyC: an efficient SystemC parser. In *Workshop on Synthesis And System Integration of Mixed Information technologies*, 2004.

[9] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Design Automation Conference (DAC)*, pages 286–291. IEEE, 2003.

[10] Mark A. Friedl, Douglas K. McIver, John C.F. Hodges, X.Y. Zhang, D. Muchoney, Alan H. Strahler, Curtis E. Woodcock, Sucharita Gopal, Annemarie Schneider, Amanda Cooper, Alessandro Baccini, Feng. Gao, and Crystal Barker Schaaf. Global land cover mapping from modis: algorithms and early results. *Remote Sensing of Environment*, 83(1-2):287–302, 2002.

[11] Christian Genz and Rolf Drechsler. Overcoming Limitations of the SystemC Data Introspection. In *Design, Automation and Test in Europe (DATE)*, pages 590–593. European Design and Automation Association, 2009.

[12] Frank Ghenassia et al. *Transaction-level modeling with SystemC*. Springer, 2005.

[13] Charalambos Ioannides and Kerstin I Eder. Coverage-directed test generation automated by machine learning–a review. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 17(1):7, 2012.

[14] Hoang M Le and Rolf Drechsler. Crave 2.0: The next generation constrained random stimuli generator for systemc. *DVCon Europe*, 2014.

[15] Lingyi Liu, David Sheridan, William Tuohy, and Shobha Vasudevan. Towards coverage closure: Using goldmine assertions for generating design validation stimulus. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[16] Lingyi Liu, David Sheridan, William Tuohy, and Shobha Vasudevan. A technique for test coverage closure using goldmine. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(5):790–803, 2012.

[17] Inderjeet Mani and George Wilson. Robust temporal processing of news. In *Meeting on Association for Computational Linguistics*, ACL '00, pages 69–76, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[18] Kevin Marquet and Matthieu Moy. Pinavm: a systemc front-end based on an executable intermediate representation. In *International Conference on Embedded Software (EMSOFT)*, pages 79–88. ACM, 2010.

[19] Kevin Marquet, Matthieu Moy, and Bageshri Karkare. A theoretical and experimental review of SystemC front-ends. pages 124–129, 2010.

[20] Edmar Martinelli, André de Carvalho, Solange Rezende, and Alberto Matias. Rules extractions from banks' bankrupt data using connectionist and symbolic learning algorithms. In *Int'l Conf. on Computational Finance*, pages 515–533, New York, NY, USA, 1999. MIT Press.

[21] Benjamin Menhorn and Frank Slomka. Confirming the design gap. In *Advances in Computational Science, Engineering and Information Technology*, pages 281–292. Springer, 2013.

[22] Gilly Nativ, S Mittennaier, Shmuel Ur, and Avi Ziv. Cost evaluation of coverage directed test generation for the ibm mainframe. In *Test Conference, 2001. Proceedings. International*, pages 793–802. IEEE, 2001.

[23] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[24] Xinxin Sheng and David Thuente. Using decision trees for state evaluation in general game playing. *KI - Künstliche Intelligenz*, 25:53–56, 2011.

[25] Jannis Stoppe, Robert Wille, and Rolf Drechsler. Data extraction from systemc designs using debug symbols and the systemc api. In *VLSI (ISVLSI), 2013 IEEE Computer Society Annual Symposium on*, pages 26–31. IEEE, 2013.

[26] Bjarne Stroustrup. Abstraction and the c++ machine model. In *Embedded Software and Systems*, pages 1–13. Springer, 2005.

[27] Ilya Wagner, Valeria Bertacco, and Todd Austin. Microprocessor verification via feedback-adjusted markov models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(6):1126–1138, 2007.

[28] Shuo Yang, Robert Wille, Daniel Große, and Rolf Drechler. Coverage-driven stimuli generation. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 525–528. IEEE, 2012.

[29] Xiaoming Yu, A. Fin, F. Fummi, and E.M. Rudnick. A fenetic testing framework for digital integrated circuits. In *International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 521–526, 2002.

[30] Zhi-Hua Zhou and Yuan Jiang. Medical diagnosis with c4.5 rule preceded by artificial neural network ensemble. *Information Technology in Biomedicine, IEEE Transactions on*, 7(1):37 –42, march 2003.

Graphene Logic Synthesis Using a Constructive Approach

Mayler G. A. Martins
ECE Department
Carnegie Mellon University
Pittsburgh, PA 15213
mayler@cmu.edu

***Abstract*—It is well-known that the graphene possesses extraordinary electrical, mechanical, and optical capabilities. These properties are being explored to develop more efficient switching devices. One of these switching devices is called Reconfigurable Gate, which behaves like a traditional multiplexer gate. It possesses smaller area, better performance, and lower power consumption when compared to its CMOS implementation. This work focuses on the synthesis of RG-MUX gate and its variation, the RG-EXP gate. This paper proposes a modification of the traditional CMOS mapping flow to utilize both graphene logic styles efficiently. This modification consists of a specialized algorithm for the synthesis of functions using graphene that consider the characteristics of the logic to generate efficient implementations. Results compare the number of MUX gates generated by the proposed approach and by the translation of a BDD, showing a reduction of more than 35% of the number of graphene devices when using RG, compared to the use of the BDD.**

***Index Terms*—Digital Integrated Circuit, Graphene, Emerging Technologies, Standard-Cell Library, Functional Composition**

I. Introduction

The CMOS scaling is the primary contributor to the continued improvement of circuit performance. However, the increasing difficulties on this subject recently forced Intel to extend the tick-tock cycle (which comprises in a new process node for 'tick' and a new architecture for 'tock'). It is clear that the process challenges that appeared in the FinFET fabrication are now affecting the industry timescale. As a result, considerable effort has been put forth to develop new devices that may allow further progress in computation capability, the so-called Emerging Technologies. These new technologies include single electron transistors (SET) [1], nanowires [2] quantum cellular automata (QCA) [3], resonant tunneling diodes (RTD) [4], memristors [5], devices based on electron spin as spin-diodes [6],some spin transfer torque-magnetic tunneling junction (STT-MTJ) [7], and graphene structures [8]–[10].

The majority of cited technologies have different primitive gates, compared to CMOS. While CMOS logic typically uses (N)AND, (N)OR and NOT gates as basic operations, the primitive gates in these technologies differ considerably. For instance, QCA, TPL and SET and some STT-MTJ technologies have the majority (and minority) gate as the basic logic element. Memristors and other types of magnetic tunneling junctions (MTJs) can be used in implication logic-based design while Si double-gate nanowire FETs can efficiently implement the XOR function [11]. Spin-diodes use a current-based logic having operators with different costs [12]. Unfortunately, the state-of-the-art academic and industrial tools do not have support for the technologies above. However, the necessary effort from adapting a whole set of logic/physical synthesis algorithms to test a new technology turns to be almost unfeasible. The possibility to leverage algorithms and methodologies already used in CMOS design for new technologies is desirable, as it would simplify the initial integration process. Still, existing algorithms designed for CMOS cannot handle the peculiarities of each technology. There are works trying different strategies, from proposing sets of base functions [13], [14], use of standard libraries [8], [15], and finally, post-synthesis strategies [10].

In the last years, the graphene drew much attention. The electric properties (especially superconductivity) attracted many researchers from the device community. However, the absence of a valence band presents a challenge for digital devices, since the separation from low and high signals are unclear [16]. The use of electrostatic doping allowed to implement a p-n junction in graphene, circumventing the valence band problem. The developed device can be considered a "graphene switch", but it has multiplexer logic behavior. This switch is called RG-MUX, and it was demonstrated its superior performance and smaller area [17], [18]. However, its power consumption is higher than the traditional CMOS switches [19]. Miryala et al. propose an adiabatic computation to reduce the power consumption using RG-MUX switches which operate two order of magnitude less than the CMOS counterpart [9].

This paper proposes algorithms that handle the presented graphene logic style (using an MUX as the primitive gate). Previous approaches take advantage of an ROBDD structure to implement an MUX circuit [8], [10]. Still, the ROBDD structure has drawbacks. One of them is the fixed variable ordering, which guarantees the canonicity of a Boolean function but it restricts the synthesis of MUX circuits. Also, in this translation, the select input will always be a variable. This restriction imposes an overhead in an MUX circuit. These drawbacks show that there is a space for specialized algorithms. The proposed algorithm uses a bottom-up approach to performing the synthesis of functions using MUX gates, considering

the characteristics of the technology to generate efficient implementations. To evaluate the quality of these algorithms, we will use these algorithms to transform a CMOS library into an MUX library.

The rest of the paper is organized as follows. The graphene properties are described in Section II. The algorithms to synthesize functions using graphene are presented in Section III and an analysis of algorithms and a technology mapping exercise is shown in Section IV. Finally, conclusions and future work are outlined in Section V.

II. Graphene Properties

The graphene is a two-dimensional carbon honeycomb lattice, in which one atom forms each vertex, and it is present in another carbon materials, as graphite and carbon nanotubes. The graphene has been theorized for decades. However, it was only isolated and characterized in 2004 by Andre Geim and Konstantin Novoselov at the University of Manchester [20], resulting in a Nobel Prize in 2010. Graphene extraordinary properties include being 100 times stronger than the strongest steel with the same thickness of graphene sheet; excellent heat/electricity conduction and nearly transparent, which is an impressive property to photonics [21]. The use of Chemical Vapor Deposition enables the graphene mass production [22].

Two important properties of materials for the VLSI industry are the high carrier mobility and high saturation velocity, in which graphene possesses both. However, the absence of the valence band turns it harder to use, since it turns harder to distinct the ON and OFF signals. Moreover, the zero band-gap energy distribution results in high leakage currents and unsatisfactory I_{on}/I_{off} ratio.

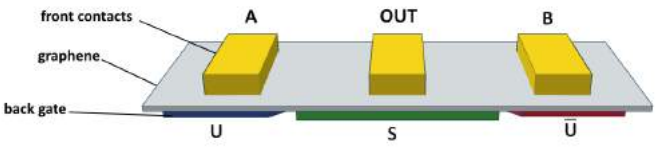


Figure 1. Structure of a Reconfigurable Gate.

One approach to circumvent the band-gap problem is to use a strategy based om electrostatic doping, which is used to implement a device that "emulates" a P-N junction, implementing a switch that has a complex gate behavior. This switch is called reconfigurable gate (RG) [23], having an optoelectronic property that allows steering carriers from input to output. Figure 1 presents the structure of a graphene RG. It consists of a thin graphene layer, the front contacts, containing the inputs (A and B) and the output of the MUX and the back contacts, which implements the MUX selector (S) and the split-gate (U and $\overline{U}$). The U input can be configured fixed in logic 1, implementing the MUX function (RG-MUX) or it can be a third input. If U is a logic input, the behavior is an MUX with an XNOR(S, U) in the selector pin, increasing the functionality of the gate [10], being called RG-EXP. Figure 2 show the logic implementations of both RG-MUX and RG-EXP gates.

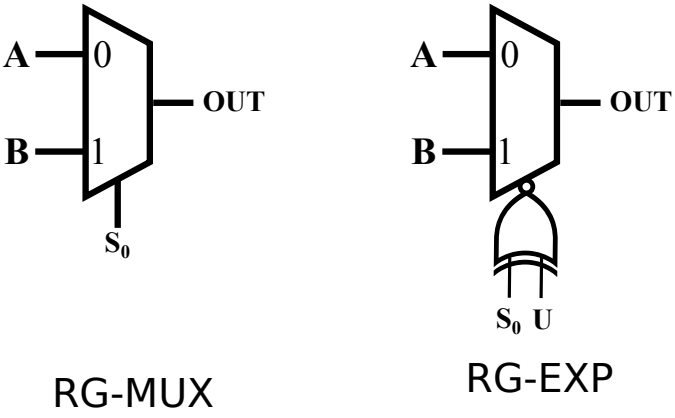


Figure 2. RG-MUX and RG-EXP logic representations.

III. Synthesis of Boolean Functions Using RG-MUX and RG-EXP

As presented in Section II, the basic logic gates for graphene technology are RG-MUX or RG-EXP. Developing an algorithm that can synthesize Boolean functions using the basic gates of the graphene technology will impact directly in the circuit area. These implementations will be employed as the reference to building area efficient standard cells.

In this paper we will use Functional Composition (FC) [24]. The first reason to use this paradigm is its utilization of a functional and structural representation, which avoids the structural bias dependence [25] and makes FC a Boolean method. Moreover, it can perform a bottom-up association of Boolean functions as opposed to the more traditional top-down functional decomposition approach. Such associations enable an efficient use of the primitive operators.

The rest of this section will present the setup of Functional Composition for the graphene technology, both for the synthesis of RG-MUX and RG-EXP gates. A more detailed information about the FC principles can be found in [24].

A. Functional Composition Setup

FC uses bonded-pairs to represent logic functions. The bonded-pair is a data structure that contains one functional and one structural representation of the same Boolean function. The bonded-pair representation used to represent an MUX network circuit is composed by the tuple <truth-table, MUX network>. The truth table is represented as an integer. Storing the truth table data as a computer word or an array of words, basic Boolean operations can be done in constant time by parallel operation over their truth tables. The RG network is defined by an expression representing an RG logic tree. The initial bonded-pairs are the variables, which are stored in a zero cost set since these functions do not need RG to be implemented.

The algorithm to synthesize RG-MUX (MUX) gates is presented in Algorithm 1. The method CREATE_INITIAL_FUNCTIONS (line 3) generates the set of all 0-cost functions. These functions are input variables in directed forms as well as constants true and false, which

are in optimal form. The method ASSOCIATE (line 9-15) represents the partial order in FC. The method to synthesize MUX gates is as follows (line 11-15). GET_INDECES_MUX (line 11) is implemented according to the cost function. The cost function is chosen in this paper as the number of MUX (MUX-E) gates. Although any other cost function, such as logic depth can be selected as well. The term k-cost function is used hereafter to refer to a function with cost k, i.e., the implementation of f requires k multiplexer gates.

Since the multiplexer is not symmetrical, it is necessary to consider the different ordering for f_1, f_2 and f_3. In order to generate k-cost functions, all (k-1)-cost functions must be already known. When an implementation for a function f is first found with a cost c, the optimal implementation cost for f is c. In other words, f is a c-cost function. The general composing rule is that when (C+1)-functions are generated, three functions f_1, f_2 and f_3 are combined such that the sum of their costs is C (line 14).

Algorithm 1 FC-MUX Algorithm

```
1:  function CREATE_ALL_FUNCTIONS (n, type)
2:      i ← 1
3:      B[0] ← CREATE_INITIAL_FUNCTIONS(n)
4:      while any function is not synthesized do
5:          if then type = MUX
6:              B[i] ← ASSOCIATE_MUX(B, i)
7:          else
8:              B[i] ← ASSOCIATE_MUX_E(B, i)
9:          i ← i + 1
10:     return B
11:
12: function ASSOCIATE_MUX (B, i)
13:     S ← ∅
14:     indecesMux ← GET_INDECES_MUX (i)
15:     for each idx ∈ indecesMux do
16:         ⟨i, j, k⟩ ← idx
17:         S ← S ∪ COMB(B[i], B[j], B[k], MUX)
18:     return S
19: function ASSOCIATE_MUX_E (B, i)
20:     S ← ∅
21:     indecesMuxE ← GET_INDECES_MUX_E (i)
22:     for each idx ∈ indecesMuxE do
23:         ⟨i, j, k, l⟩ ← idx
24:         S ← S∪ COMB(B[i], B[j], B[k], B[l], MUX_E)
25:     return S
```

The approach used to synthesize MUX-E is very similar. The combinations are done three by three to implement to generate MUX implementations and four by four to produce the MUX-E implementations. Similar approaches were applied in different technologies [14], [15], [26], [27] and had interesting results. Table I shows implementations for functions up to 2-inputs, for RG-MUX and RG-EXP, respectively.

Table I
MINIMAL IMPLEMENTATIONS USING RG-MUX AND RG-EXP FOR FUNCTIONS UP TO 2 INPUTS.

Function	RG-MUX M(S,A,B)	RG-EXP M-E(S,U,A,B)
0	0	0
$\overline{a} \cdot \overline{b}$	M(M(b,a,1),1,0)	M-E(0,M-E(0,b,1,a),0,1)
$\overline{a} \cdot b$	M(a,b,0)	M-E(0,a,0,b)
$\overline{a}$	M(a,1,0)	M-E(0,a,0,1)
$a \cdot \overline{b}$	M(b,a,0)	M-E(0,b,0,a)
$\overline{b}$	M(b,1,0)	M-E(0,b,0,1)
$a \oplus b$	M(b,a,M(a,b,0))	M-E(b,a,1,0)
$\overline{a} + \overline{b}$	M(M(b,0,a),1,0)	M-E(0,M-E(0,b,a,0),0,1)
$a \cdot b$	M(b,0,a)	M-E(0,b,a,0)
$a \ominus b$	M(b,M(a,1,0),a)	M-E(b,a,0,1)
b	b	b
$\overline{a} + b$	M(a,1,b)	M-E(0,a,b,1)
a	a	a
$a + \overline{b}$	M(b,1,a)	M-E(0,b,a,1)
$a + b$	M(b,a,1)	M-E(0,b,1,a)
1	1	1

B. Multi-threaded Functional Composition

To speed-up the execution time, spreading the combinations between multiples threads can be a solution. As the FC can be thought as a "Cartesian product" between sets of functions, it can be highly parallelizable. One strategy that seems to be highly efficient (but more resource hungry) is the MapReduce [28]. MapReduce consists primarily of each thread receiving a data set, working in this data set (e.g. generating the combinations) and being allowed to read the implementation hash table. If the implementation does not exist or if the thread created a better implementation (e.g. multi-objective optimization [29]), it stores in a temporary hash table. Only at the end of the computation, the thread is allowed to merge the results with the main thread hash table. This fact implies in an almost non-existent concurrent scenario since the threads only write once in the main thread, and as the threads finish at different times, the chance of 2 threads writing in the same resource is very low.

An experimental version of the algorithm was implemented using the technique above and in Table II we have the execution times for the multi-thread version of the algorithm. The server used has 2 Intel Xeon 2.3 GHz processors, each one with six cores and 12 threads, totaling 12 cores and 24 threads. Opening more than 24 threads started to give worse results. Despite being able to handle 24 threads, the Hyper-Threading is only efficient when different computations are happening (e.g. instruction in a thread and ALU operation in another one). In this sense, we can observe that we achieved almost the theoretical parallelization for 12 cores.

C. Advantages of Proposed Method over BDDs

The algorithm to compute MUX networks using BDDs in [8] converts the BDD nodes into RG-MUX gates. Albeit being a very straightforward method, there are three main

Table II
PERFORMANCE ANALYSIS FOR FC-MUX MULTI-THREAD ALGORITHM.

#Threads	Execution Time	Improvement (compared with single thread)
1	515 min	-
2	287 min	1.79x
4	158 min	3.25x
8	91 min	5.65x
16	49 min	10.5x
24	47 min	10.95x
32	48 min	10.72x

drawbacks. The first one is that converting a BDD into an MUX network implies that the 'select' input will always have a variable. This impacts negatively in the area, since some functions achieve the minimum implementation using this input with more complex logic (e.g. another MUX). The second case is that MUX network generated is strongly based in the BDD order. In some cases, this order will impact negatively, using more MUX gates than necessary. Finally, there are several strategies to reduce the number of nodes in a BDD, including sifting and dynamic reordering. However, not all nodes in a BDD need to converted to an MUX. For instance, every node that has negative edge points to a logic zero and a positive edge that points to logic one can be replaced by a simple variable, reducing the number of nodes. Still, one needs in the worst case to test all possible BDD orders to identify the best one (e.g., the order that has the least number of nodes that are not variables).

One example of how about the ordering affects the number of nodes is presented in Figure 3. As example a simple function was chosen: $(d+\bar{c})\cdot(c+a)\cdot(\bar{d}+\bar{a})\cdot(\bar{c}+b)$. The BDD presented in (a) was provided by CUDD. It has the best order, but it needs 4 MUX gates to implement it. The BDD presented in (b), was generated by brute-force, trying all orders. We observe that a different order provided better results, needing only 3 MUX gates to implement the function. Finally, in (c) we observe an implementation of the FC-MUX algorithm. It needs just two gates due the use of the select pin for more complex logic than variables.

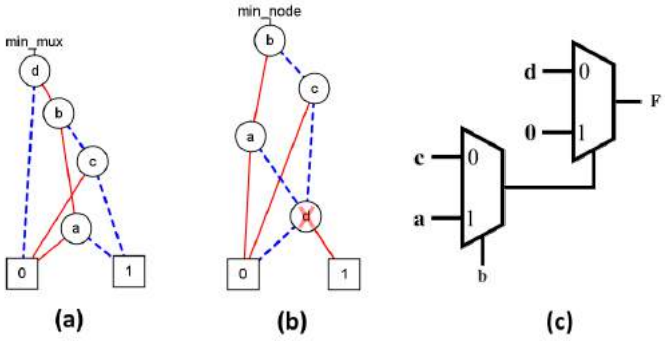


Figure 3. Comparison between BDD orders and FC-MUX implementation.

IV. RESULTS

In order to evaluate the proposed algorithms, all functions up to 4 inputs were implemented using RG-MUX and RG-EXP.

In this section, the quality of the proposed MUX logic algorithms is evaluated, in comparison to the translation of BDDs proposed in [8]. Also, a technology mapping study using the standard cell methodology is performed.

The first step was the generation of an optimal fanout free MUX networks for all functions with up to four inputs, stored in a look-up table (LUT). Two types of MUX were generated, the MUX present in RG-MUX and the MUX-E in RG-EXP gate. It is worth to mention that all 4-input functions were synthesized using at most eight MUX or five MUX-E gates. For the BDD generation, the ABC tool [30] was used, using the bdd command, which uses the CUDD package internally. The results are presented in Figure 4. Compared to CUDD results, FC-MUX can reduce in 28% the number of multiplexers. Moreover, FC-MUXE can reduce 48% the number of devices.

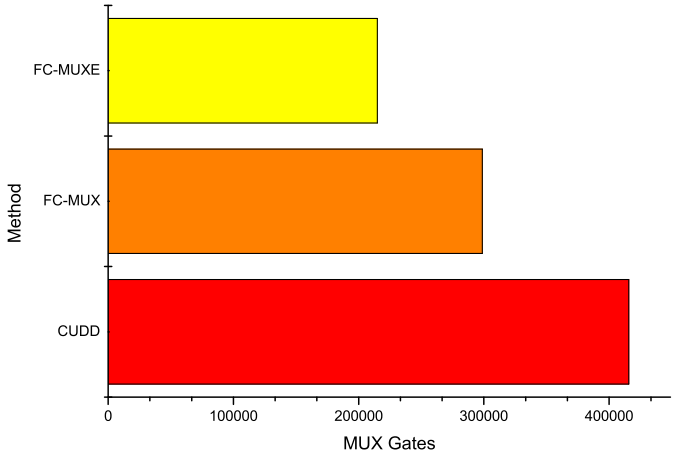


Figure 4. Histogram of the differences from CUDD compared to FC.

In orde to confirm the impact of the proposed algorithm, a simple technology mapping was performed. The ABC tool was used to do the technology mapping of MCNC benchmarks. A 65 nm commercial library was used as a template. As the library uses a considerable number of cells with more than four inputs, a modified version of the algorithm was implement to compute just the read-once functions. The algorithm was able to enumerate all read-once functions with 5 and six inputs. From a set of 85 combinational cells (functions), 82 were synthesized with FC-MUX algorithm. The other three functions (having more than six inputs) were converted using the BDD algorithm. In this experiment, we are ignoring power and timing information and trying just to optimize area.

The results of the technology mapping performed in ABC for each benchmark, compared with the same library converted using BDDs is presented in Table III. It is interesting to observe that the MUX-E improvements reduced considerably since the implementations of the gates that are found in the library usually have the same number of devices compared to MUX implementations. Nevertheless, an average reduction of almost

40% using MUX-E gates was achieved. In the final version of this paper, results using commercial tools using OpenCore benchmarks will also be presented.

V. Conclusion

In this paper an algorithm was introduced for synthesizing circuits using MUX and MUX-E (RG-MUX and RG-EXP, respectively) logic gates, using functional composition, for the graphene technology. This algorithm generates an optimal structure composed of MUX or MUX-E gates for a given a Boolean function up to 4 inputs. In order to evaluate the quality of the algorithm, a CMOS library was converted into an MUX library to perform technology mapping. The results over MCNC benchmarks show that there is a combinational area reduction of almost 40%, compared to a library generated using BDDs. Future works include synthesis and analysis to graphene-based Pass-XNOR logic circuits [31]. Also, a study of more realistic delay and power estimation to the synthesis is being performed. Finally, it is being evaluated the exploration of other MUX-based technologies, as the m-logic [7].

References

[1] W. Wei, J. Han, and F. Lombardi, "Design and evaluation of a hybrid memory cell by single-electron transfer," *IEEE Trans. Nanotechnol.*, vol. 12, no. 1, pp. 57–70, 2013.

[2] K. Jansson, E. Lind, and L.-E. Wernersson, "Performance evaluation of iii–v nanowire transistors," *IEEE Trans. Electron Devices*, vol. 59, no. 9, pp. 2375–2382, 2012.

[3] D. Tougaw and M. Khatun, "A scalable signal distribution network for quantum-dot cellular automata," *IEEE Trans. Nanotechnol.*, vol. 12, no. 2, pp. 215–224, 2013.

[4] J. Lee, J. Lee, and K. Yang, "A low-power 40-gb/s 1: 2 demultiplexer ic based on a resonant tunneling diode," *IEEE Trans. Nanotechnol.*, vol. 11, no. 3, pp. 431–434, 2012.

[5] X. Zhu, X. Yang, C. Wu, N. Xiao, J. Wu, and X. Yi, "Performing stateful logic on memristor memory," *IEEE Trans. Circuits Syst. II*, vol. 60, no. 10, pp. 682–686, 2013.

[6] J. S. Friedman, B. W. Wessels, G. Memik, and A. V. Sahakian, "Emitter-coupled spin-transistor logic: Cascaded spintronic computing beyond 10 ghz," *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, vol. 5, no. 1, pp. 17–27, 2015.

[7] D. Morris, D. Bromberg, J.-G. J. Zhu, and L. Pileggi, "mlogic: Ultra-low voltage non-volatile logic circuits using stt-mtj devices," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 486–491.

[8] S. Miryala, A. Calimera, M. Poncino, and E. Macii, "Exploration of different implementation styles for graphene-based reconfigurable gates," in *IC Design & Technology (ICICDT), 2013 International Conference on*. IEEE, 2013, pp. 21–24.

[9] S. Miryala, A. Calimera, E. Macii, and M. Poncino, "Ultra low-power computation via graphene-based adiabatic logic gates," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*. IEEE, 2014, pp. 365–371.

[10] S. Miryala, V. Tenace, A. Calimera, E. Macii, M. Poncino, L. Amarú, G. De Micheli, and P.-E. Gaillardon, "Exploiting the expressive power of graphene reconfigurable gates via post-synthesis optimization," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. ACM, 2015, pp. 39–44.

[11] P.-E. Gaillardon, L. G. Amarù, S. Bobba, M. De Marchi, D. Sacchetto, and G. De Micheli, "Nanowire systems: technology and design," *Phil. Trans. R. Soc. A: Mathematical, Physical and Engineering Sciences*, vol. 372, no. 2012, p. 20130102, 2014.

[12] M. Martins, F. Marranghello, J. Friedman, A. Sahakian, R. Ribas, and A. Reis, "Enhanced spin-diode synthesis using logic sharing," in *Digital System Design (DSD), 2015 Euromicro Conference on*. IEEE, 2015, pp. 218–224.

[13] K. Kong, Y. Shang, and R. Lu, "An optimized majority logic synthesis methodology for quantum-dot cellular automata," *IEEE Trans. Nanotechnol.*, vol. 9, no. 2, pp. 170–183, 2010.

[14] M. G. Martins, V. Callegaro, F. S. Marranghello, R. P. Ribas, and A. I. Reis, "Majority-based logic synthesis for nanometric technologies," in *Nanotechnology (IEEE-NANO), 2014 IEEE 14th International Conference on*. IEEE, 2014, pp. 256–261.

[15] M. G. A. Martins, F. S. Marranghello, J. S. Friedman, A. V. Sahakian, R. P. Ribas, and A. I. Reis, "Spin diode network synthesis using functional composition," in *Integrated Circuits and Systems Design (SBCCI), 2013 26th Symposium on*. IEEE, 2013, pp. 1–6.

[16] C. Ahn, A. Bhattacharya, M. Di Ventra, J. Eckstein, C. D. Frisbie, M. Gershenson, A. Goldman, I. Inoue, J. Mannhart, A. J. Millis *et al.*, "Electrostatic modification of novel materials," *Reviews of Modern Physics*, vol. 78, no. 4, p. 1185, 2006.

[17] S. Miryala, A. Calimera, E. Macii, and M. Poncino, "Delay model for reconfigurable logic gates based on graphene pn-junctions," in *Proceedings of the 23rd ACM international conference on Great lakes symposium on VLSI*. ACM, 2013, pp. 227–232.

[18] S. Miryala, M. Montazeri, A. Calimera, E. Macii, and M. Poncino, "A verilog-a model for reconfigurable logic gates based on graphene pn-junctions," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 877–880.

[19] C. Pan and A. Naeemi, "Device-and system-level performance modeling for graphene pn junction logic," in *Quality Electronic Design (ISQED), 2012 13th International Symposium on*. IEEE, 2012, pp. 262–269.

[20] A. K. Geim and K. S. Novoselov, "The rise of graphene," *Nature materials*, vol. 6, no. 3, pp. 183–191, 2007.

[21] S. Stankovich, D. A. Dikin, G. H. Dommett, K. M. Kohlhaas, E. J. Zimney, E. A. Stach, R. D. Piner, S. T. Nguyen, and R. S. Ruoff, "Graphene-based composite materials," *nature*, vol. 442, no. 7100, pp. 282–286, 2006.

[22] C. Xu, H. Li, and K. Banerjee, "Modeling, analysis, and design of graphene nano-ribbon interconnects," *Electron Devices, IEEE Transactions on*, vol. 56, no. 8, pp. 1567–1578, 2009.

[23] S. Tanachutiwat, J. U. Lee, W. Wang, and C. Y. Sung, "Reconfigurable multi-function logic based on graphene pn junctions," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*. IEEE, 2010, pp. 883–888.

[24] M. G. A. Martins, R. P. Ribas, and A. I. Reis, "Functional composition: A new paradigm for performing logic synthesis," in *Quality Electronic Design (ISQED), 2012 13th International Symposium on* IEEE, 2012, pp. 236–242.

[25] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," *IEEE Trans. Comput.-Aided Design*, vol. 25, no. 12, pp. 2894–2903, 2006.

[26] A. Neutzling, M. G. Martins, R. P. Ribas, and A. I. Reis, "A constructive approach for threshold logic circuit synthesis," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 385–388.

[27] F. S. Marranghello, V. Callegaro, M. G. Martins, A. I. Reis, and R. P. Ribas, "Improved logic synthesis for memristive stateful logic using multi-memristor implication," in *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 181–184.

[28] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[29] M. G. A. Martins, L. Rosa, A. B. Rasmussen, R. P. Ribas, and A. I. Reis, "Boolean factoring with multi-objective goals," in *Computer Design (ICCD), 2010 IEEE International Conference on*. IEEE, 2010, pp. 229–234.

[30] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification, Release 20130425," in , 2013.

[31] V. Tenace, A. Calimera, E. Macii, and M. Poncino, "One-pass logic synthesis for graphene-based pass-xnor logic circuits," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 2015, pp. 1–6.

Table III
RESULTS OVER MCNC BENCHMARKS.

Benchmark	CUDD [8]	FC-MUX	Reduction (%)	FC-MUXE	Reduction (%)
9symml	296	188	36.5%	185	37.5%
alu2	558	361	35.3%	358	35.8%
alu4	1006	653	35.1%	635	36.9%
apex6	867	593	31.6%	591	31.8%
apex7	281	178	36.7%	178	36.7%
c8	139	91	34.5%	87	37.4%
cc	88	53	39.8%	53	39.8%
cht	234	148	36.8%	148	36.8%
cm150a	70	47	32.9%	48	31.4%
cm151a	35	21	40.0%	21	40.0%
cm152a	30	20	33.3%	20	33.3%
cm162a	58	34	41.4%	33	43.1%
cm163a	52	33	36.5%	33	36.5%
cm42a	37	18	51.4%	18	51.4%
cm85a	50	30	40.0%	29	42.0%
cmb	59	38	35.6%	38	35.6%
comp	143	86	39.9%	86	39.9%
cordic	74	48	35.1%	43	41.9%
count	183	117	36.1%	103	43.7%
cu	67	41	38.8%	41	38.8%
dalu	1511	893	40.9%	858	43.2%
decod	60	32	46.7%	32	46.7%
des	4766	2873	39.7%	2767	41.9%
example2	335	198	40.9%	194	42.1%
f51m	165	105	36.4%	102	38.2%
frg1	135	90	33.3%	90	33.3%
i1	56	34	39.3%	33	41.1%
i10	2586	1652	36.1%	1522	41.1%
i2	278	209	24.8%	209	24.8%
i3	164	126	23.2%	126	23.2%
i4	304	226	25.7%	226	25.7%
i5	218	144	33.9%	144	33.9%
i6	701	417	40.5%	417	40.5%
i7	839	533	36.5%	533	36.5%
i8	1457	935	35.8%	954	34.5%
i9	865	567	34.5%	557	35.6%
k2	1902	1185	37.7%	1184	37.7%
lal	114	72	36.8%	67	41.2%
my_adder	173	96	44.5%	48	72.3%
pair	2010	1313	34.7%	1294	35.6%
parity	35	30	14.3%	15	57.1%
pcle	77	51	33.8%	49	36.4%
pcler8	103	64	37.9%	65	36.9%
pm1	56	37	33.9%	36	35.7%
sct	87	53	39.1%	49	43.7%
tcon	32	8	75.0%	8	75.0%
term1	229	153	33.2%	146	36.2%
ttt2	226	144	36.3%	140	38.1%
unreg	162	98	39.5%	98	39.5%
vda	872	539	38.2%	533	38.9%
x1	400	264	34.0%	266	33.5%
x2	58	36	37.9%	36	37.9%
x3	991	644	35.0%	630	36.4%
x4	474	320	32.5%	319	32.7%
z4ml	41	25	39.0%	21	48.8%
Average:			**36.7%**		**39.8%**

Patent Interpretation using Boolean Logic and Venn Diagrams

Simone R. N. Reis[1], Andre I. Reis[2,3], Jordi Carrabina[3], Pompeu Casanovas [1,4]

[1]IDT
UAB
Bellaterra, Barcelona, Spain
simonerosa.nunes@e-campus.uab.cat

[2]Institute of Informatics
UFRGS
Porto Alegre, Brazil
andre.reis@inf.ufrgs.br

[3]CEPHIS,
UAB
Bellaterra, Barcelona, Spain
jordi.carrabina@uab.cat

[4]D2DCRC, Law School
Deakin University
Geelong, Australia
p.casanovasromeu@deakin.edu.au

***Abstract*—*Patents are a type of exclusivity rights that are based on a written description of an object. This paper describes how patents are interpreted, in order to know if a product is covered or not by a patent. Patent interpretation is made through the all-element rule, and this rule is strongly related to Boolean logic. This way, we describe the use of set theory and Venn diagrams to perform patent interpretation.*

Keywords — Set theory; Venn diagrams; patents; patent interpretation

I. INTRODUCTION

Patents are a form of intellectual property aimed to protect human inventions. Basically, the inventor discloses her invention through a patent document and in exchange he or she receives temporary exclusivity rights to practice (manufacture, sell, etc.) the invention. The patent document has several sections, namely: abstract, description, claims and drawings. The section that legally determines what is protected by a patent is the claim section [1] [2]. In this sense, the exact wording of the claims is very important and should receive much attention [3] [4]. The wording of the claims is firstly written by the inventor (or his/her attorney), but this is not necessarily the final writing. During the application process, claim wording will probably suffer modifications in the final text, as requested by an examiner associated with the government office that issues the patent rights if they granted. The modifications of the text are often necessary so that the claims represent an original invention for which the inventor receives exclusivity rights.

Several works have been proposed in claim interpretation, i.e. legally determine the invention that is protected by the claims of a patent. Patent interpretation is strongly based in a concept known as the all-element[1] rule [7, 8]; especially in the United States, but also in a large number of countries. The PhD thesis of R.G.Fabris [5] makes an analysis of claim interpretation in Europe and Brazil. The book edited E.D.Manzo [6] covers claim interpretation in several countries, through chapters written by local specialists. Neither Fabris nor Manzo cite the so called all-element rule. The work of Wang [7] describes how to legally avoid patent infringement, based on claim interpretation and explicitly describes the all element rule. The all-element rule is also explicitly described in the work of Schechter and Thomas [8]. Unfortunately, these works do not make an effort to visually analyze claim interpretation. Visual analysis of claims by using Venn diagrams [9] is proposed by Brainard [10], but only the relationship among claims is discussed, pointing out that dependent or derived claims are more restrictive. Brainard work lacks a more precise discussion of infringement by comparing the individual elements of an object to the individual elements of a claim.

In this work, we propose the representation of elements of a claim as sets visually represented as Venn diagrams. This is different of the representation proposed by Brainard, where only complete claims are represented as sets. We argue that by representing individual elements of a claim as sets, the all-element rule can better be visualized and the inventors can better understand what is protected and what is not protected by a specific claim. Also, we illustrate that the order in which dependent claims are made affect the protection provided by the set of claims. This is shown with the help of a simple didactic example created to illustrate the relationship between the words of the claims and the objects that are covered or not as the consequence of the specific writing order.

This paper is organized as follows. Section II presents basic concepts. The all element rule is discussed in section III. Section IV describes how to perform patent claim interpretation using Boolean logic and Venn diagrams. A general discussion is made in Section V. Conclusions are presented in section VI.

II. BASIC CONCEPTS

This section presents the basic concepts used in this work. Our analysis of patent coverage is based on performing patent interpretation based on set theory and representing it through Venn diagrams. These concepts are discussed in the following sub-sections.

A. Set Theory

Set theory has many applications; it is able to describe membership of elements to a set. A given element *e* can belong to a set or it may not belong to the set. For instance, if an

[1] The word element here is an inadequate but established name in patent literature, as it will be discussed later. From a set theory point of view this rule should be better named the all-properties rule.

element *e* belongs to a set *S*, we write $e \in S$. If an element *e* does not belong to a set *S*, we write $e \notin S$.

B. Venn Diagrams

Venn diagrams are a graphical representation of sets, which include the notions of elements pertaining or not pertaining to a set. For instance, Fig. 1 shows two sets S1={e2, e3} and S2={e3, e4} with a common element e3. In this case we say that the intersection between S1 and S2 is $S1 \cap S2 = \{e3\}$. Fig. 2 illustrates an additional (and different) example of Venn diagram where sets S1={e2, e3, e4} and S2={e3, e4} are in a configuration where S1 contains S2, which is expressed by $S1 \supset S2$. It is also possible to say that S2 is contained by S1, which is expressed by $S2 \subset S1$.

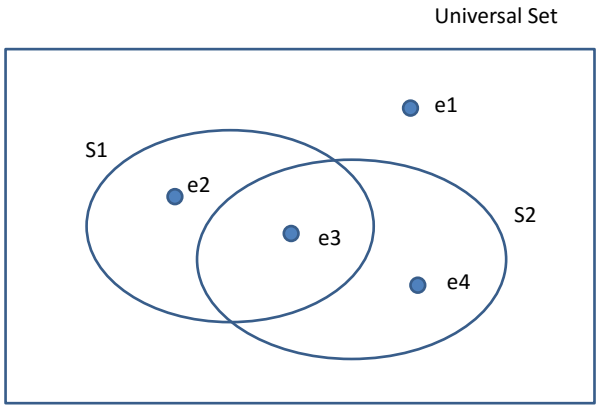


Fig. 1. Two sets S1={e2, e3} and S2={e3, e4} with a common element e3.

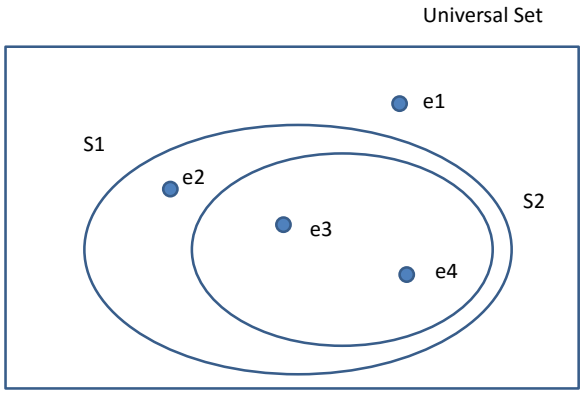


Fig. 2. Two sets S1={e2, e3, e4} and S2={e3, e4}, such that $S1 \supset S2$.

C. Patent Interpretation with Sets and Venn Diagrams

A patent is an intellectual property right that grants exclusivity to the patent owner. Patent rights are described with words and drawings. Patent interpretation consists in determining if a product is covered or not by a patent, through comparing: (1) a given product against the (2) description provided by the words and drawings in the patent. The result of patent interpretation can thus have two issues. It can be decided that a product is covered by the patent and therefore the patent owner has exclusivity rights for the patent. Or alternatively it can be decided that the product is not covered by the patent and therefore the patent owner has no say about commercial aspects involving the product.

In this paper we will represent patent coverage with Venn diagrams. The basic idea is shown in Fig. 3, where patent coverages are represented as the sets C1, C2 and C3, which can be broader (C1, with five elements) or narrower (C3, with two elements) depending on patent text and patent interpretation. The exact way patent interpretation is made will be described later on (Sections III and IV).

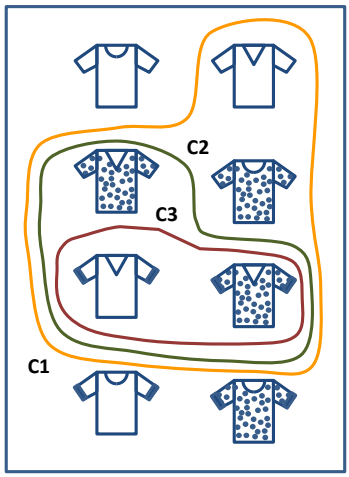


Fig. 3. Illustrating patent coverage with Venn diagrams.

III. The All-Element Rule

Patent coverage is determined by the so-called all-element rule. However, the meaning of the word element in the all-element rule is different from the meaning of the word element in Set theory. For this reason we will provide the two definitions below.

Definition 1: An **element** (Set Theory) is a member of a set.

Example 1: considering Fig. 1, e3 is both an element of sets S1 and S2.

Definition 2: An **element**[2] (all-element rule) is a constructive characteristic that may be present in an object.

From a set theory point of view, the more adequate name for definition 2 would be **property**, instead of element. So we will refer to **property** and **all-properties rule** in the following, in order to avoid confusion between definitions 1 and 2.

Example 2: Fig. 4 illustrates polka dots as a constructive property (definition 2) of T-shirts. Notice that the subset of T-shits with polka dots is composed of four T-shirts that are elements (definition 1) of the subset.

Definition 3: The **all-property rule** states that a product is covered by a patent when the product presents all the

[2] Better named as **property**, instead of element. However, the name element is well established in the patent law community.

constructive properties (definition 2) recited in a single claim of the patent.

Example 3: Consider a patent claim stating: "a T-shirt characterized by presenting polka dots, V-neck and folded sleeves". This claim considers three different constructive properties. Polka-dots as constructive property are shown in Fig. 4. The constructive property V-neck is illustrated in Fig. 5. Folded sleeves are depicted in Fig. 6. The relationship among the three constructive properties is presented in Fig. 7. Notice that the only one type of T-shirt at the intersection of the three sets contains the three constructive properties. This is the only type of T-shirt covered by the patent claim.

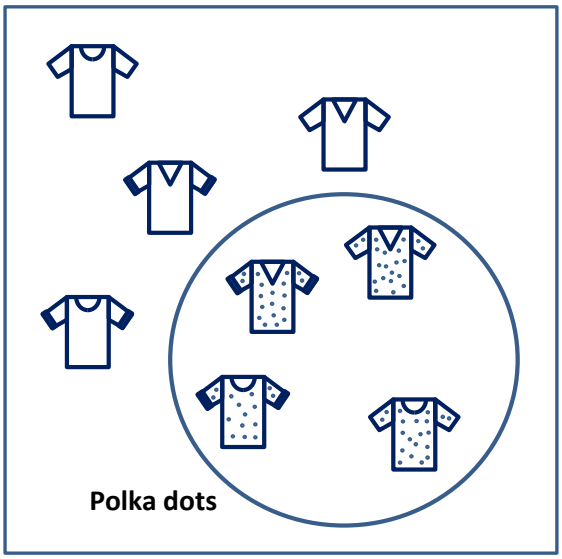


Fig. 4. The subset of T-shirts with polka dots.

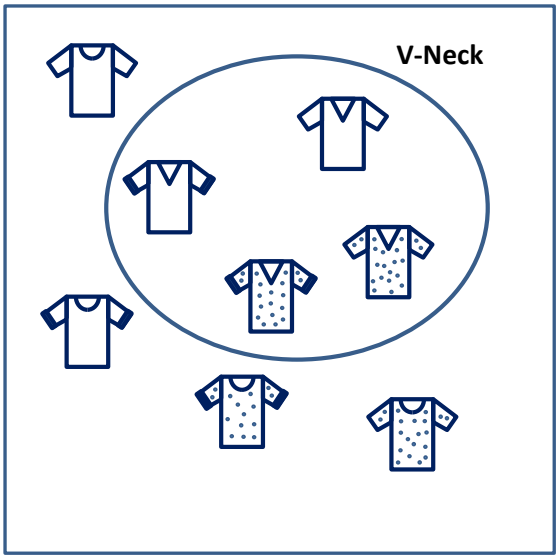


Fig. 5. The subset of T-shirts with V-necks.

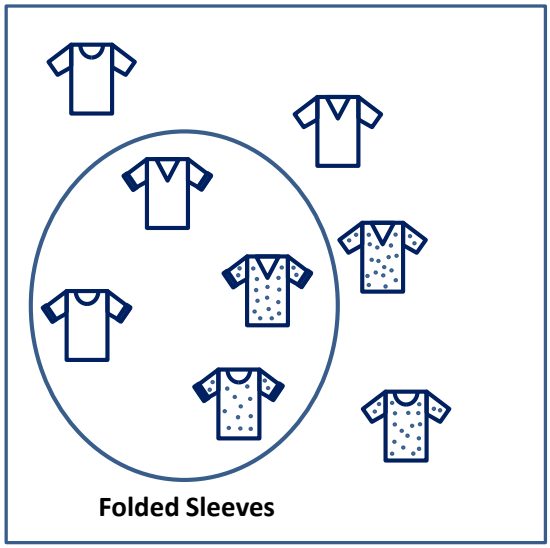


Fig. 6. The subset of T-shirts with folded sleeves.

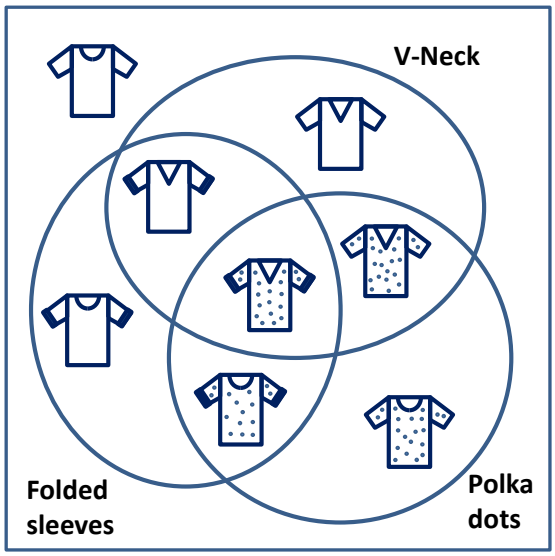


Fig. 7. The subsets of T-shirts with polka dots, V-neck and folded sleeves.

IV. Patent Interpretation Using Venn Diagrams

In this section we further explain the logic behind patent interpretation. The first step in interpreting patent claims is to understand the constructive properties (definition 2) listed in patent claims.

Once the constructive properties are understood, the patent interpreter should visualize that for *n* constructive properties, there are 2^n possible combinations. For instance, considering T-shirts having (or not) polka dots, V-necks and folded sleeves, eight (2^3) different combinations are possible. These combinations are listed in Table I.

For easier reference, a binary code is also assigned to each combination in Table I, representing the absence (0) or presence (1) of each constructive property. The binary codes

are used as reference to illustrate the drawings of each possible T-shirt combinations in Fig. 8.

TABLE I. POSSIBLE COMBINATIONS OF T-SHIRTS WITH POLKA DOTS, V-NECK AND FOLDED SLEEVES.

Code	Polka dots	V-Neck	Folded Sleeves
000	no	no	no
001	no	no	yes
010	no	yes	no
011	no	yes	yes
100	yes	no	no
101	yes	no	yes
110	yes	yes	no
111	yes	yes	yes

(a) 000 (b) 001 (c) 010 (d) 011

(e) 100 (f) 101 (g) 110 (h) 111

Fig. 8. Drawings of the 8 possible T-shirt combinations.

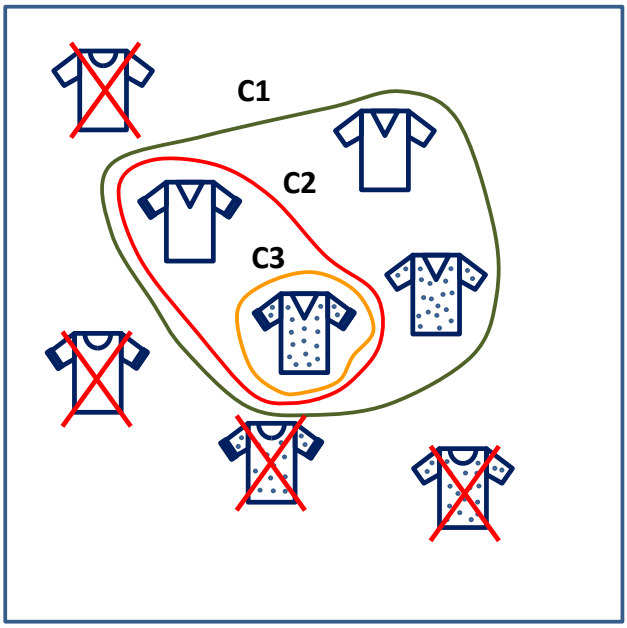


Fig. 9. Patent interpretation corresponding to example 4.

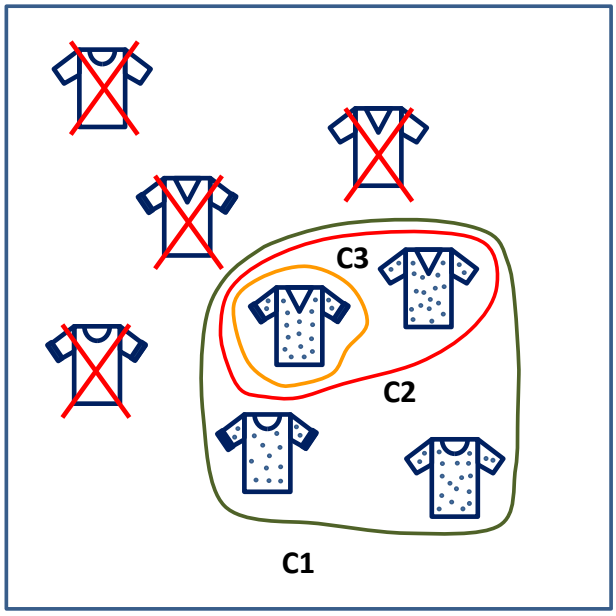


Fig. 10. Patent interpretation corresponding to example 5.

In the following, we will consider three different examples of coverage considering claims involving the three properties depicted in Table I and Fig. 8.

Example 4: consider a patent having the following set of claims:

"*1. A T-shirt characterized by having V-neck.*

2. A T-shirt according to claim 1 characterized by having folded sleeves.

3. A T-shirt according to claim 2 characterized by having polka dots".

The corresponding patent interpretation is shown in Fig. 9. In this case, claim 1 (C1) requires the constructive property V-neck, which corresponds to shirts from Table I labeled with binary codes of type ´-1-´, meaning all the codes presenting V-neck. The symbol ´-´ represents a don´t-care and can be set to 0 or 1. This corresponds to four T-shirts with code 010, 011, 110 and 111. Claim 2 (C2) further restricts T-shirts to have folded sleeves, which results in T-shirt codes of type ´-11´. This corresponds to codes 011 and 111. Claim 3 (C3) further restricts T-shirts to have polka dots, which results in the T-shirt code ´111´. Notice that every derived claim is a restriction of the coverage of the prior claims. This happens because each new derived claim is adding constructive properties (definition 2) that will be required by the all-property rule.

Example 5: consider a patent having the following set of claims:

"*1. A T-shirt characterized by having polka dots.*

2. A T-shirt according to claim 1 characterized by having V-neck.

3. A T-shirt according to claim 2 characterized by having folded sleeves".

The corresponding patent interpretation is shown in Fig. 10. In this case, claim 1 (C1) requires the constructive property polka-dots, which corresponds to shirts from Table I labeled with binary codes of type ´1--´, meaning all the codes presenting polka dots. This corresponds to four T-shirts with code 100, 101, 110 and 111. Claim 2 (C2) further restricts T-shirts to have V-neck, which results in T-shirt codes of type ´11-´. This corresponds to codes 110 and 111. Claim 3 (C3) further restricts T-shirts to have folded sleeves, which results in the T-shirt code ´111´. Again, as in the example 4, every derived claim is a restriction of the coverage of the prior claims.

Example 6: consider a patent having the following set of claims:

"*1. A T-shirt characterized by having folded sleeves.*

2. A T-shirt according to claim 1 characterized by having polka dots.

3. A T-shirt according to claim 2 characterized by having V-neck".

The corresponding patent interpretation is shown in Fig. 11. In this case, claim 1 (C1) requires the constructive property folded sleeves, which corresponds to shirts from Table I labeled with binary codes of type ´--1´, meaning all the codes presenting folded sleeves. This corresponds to four T-shirts with code 001, 011, 101 and 111. Claim 2 (C2) further restricts T-shirts to have polka dots, which results in T-shirt codes of type ´1-1´. This corresponds to codes 101 and 111. Claim 3 (C3) further restricts T-shirts to have V-neck, which results in the T-shirt code ´111´. Again, as in the examples 4 and 5, every derived claim is a restriction of the coverage of the prior claims.

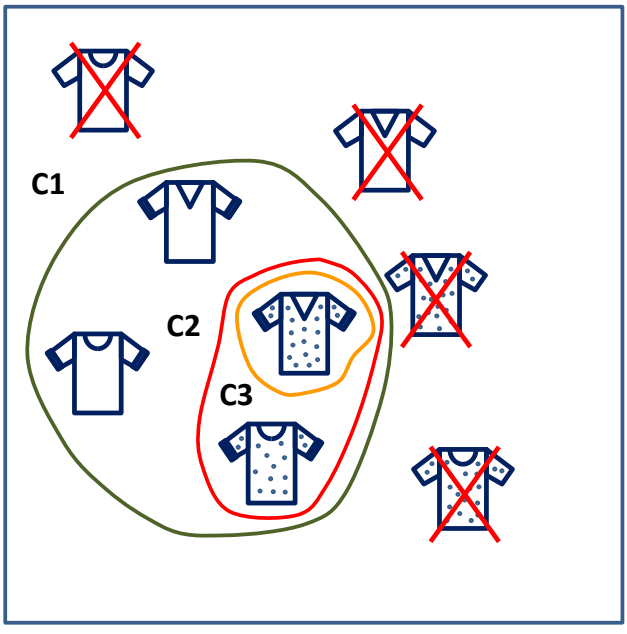


Fig. 11. Patent interpretation corresponding to example 4.

V. General discussion

In the following we provide some more general discussion about our framework for patent interpretation.

A. *Adding properties means adding restrictions*

The main consequence of the all-property rule is that adding constructive properties (definition 2) to a claim means to add restrictions. This is clear from examples 4, 5 and 6 where C1 ⊃ C2 ⊃ C3. As a consequence, good broader claims include few constructive properties.

B. *Different properties in the first claim mean different coverage*

It is also clear that the properties added to the first independent claim will determine the broadest coverage of the patent. This way, the properties to be added to the first claim must be carefully selected, in order to not restrict patent coverage.

C. *Importance of order when claiming*

The examples 4, 5 and 6 presented a situation where the constructive properties being added to each derived claims resulted in different coverage. Three different orders were shown in the examples. However, six different orders are possible, corresponding to the six distinct paths in the lattice shown in Fig. 12.

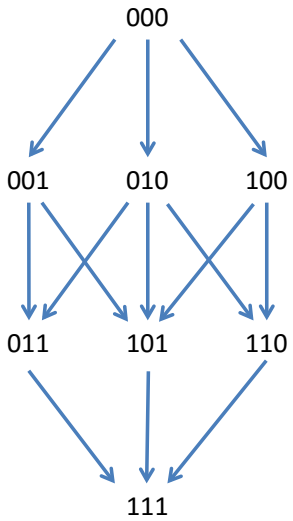


Fig. 12. Lattice containing the 6 possible different partial orders when combining 3 possible properties.

D. *Is it legally possible to claim an entire universal set?*

This could potentially be achieved through a claim in which no constructive property is recited, e.g. *´All T-shirts´*. This type of claims is not allowed from a legal point of view. In order to further clarify, we provide a definition and an example.

Definition 4: An **abstract claim** is a claim where no constructive property is recited.

Example 7: Abstract claims could be recited as: ´*1. A T-shirt*`, or in a more mathematical language as ´1. A T-shirt characterized by ∅´. In this case the symbol ∅ represents the empty set, meaning that no constructive property was recited in the claim. This abstract claim would cover T-shirts with code ´---´, meaning that each ´-´ can be substituted by 0 or 1 and the eight different codes for T-shirts are all covered by the claim, including the code ´000´. The T-shirt with code ´000´ can only be covered by abstract claims.

Fortunately, abstract claims are not allowed. In order to be able to claim an invention, it is necessary to list at least one constructive property that will work as a restriction.

Considering the eight possible codes in Table I, it is possible to produce a set of claims that covers seven out of the eight possibilities. This is illustrated in the example below.

Example 8: consider a patent having the following set of claims:

"*1. A T-shirt characterized by having V-neck.*

2. A T-shirt characterized by having folded sleeves.

3. A T-shirt characterized by having polka dots".

The set of claims above corresponds to the coverage shown in Fig. 7. Notice that each claim is an independent claim, and therefore the examiner could potentially require dividing the patent into three independent patent applications, as the inventions may be considered distinct.

E. Initial Nature of the Universal set is Important

From a more general perspective, the idea of claiming all T-shirts can be implemented by starting from a more general nature for the universal set. In our examples, we have been using T-shirts as the universal set and claiming subsets of T-shirts. To have a broader coverage, it is conceptually possible to start from a larger universal set (e.g. clothes) and then claim an intended universal set (e.g. T-shirts) as a subset.

F. Properties not recited in the claims

Property's that are not recited in the claims do not play a role in patent interpretation. For instance, consider again the claims from example 6. Consider the two T-shirts shown in Fig. 13, which have a front opening. The T-shirt in Fig. 13.a is covered by the claim number 1, as it has folded sleeves (which correspond to all-properties listed in claim 1). The T-shirt in Fig. 13.b is not covered by the claim number 1, as it has not folded sleeves (which are listed in claim 1).

G. Why patens have several claims?

Patents have a structure of several claims derived from one another. From a protective point of view, this is equivalent of having several subsequent walls or fences protecting a certain area. Under attack, the fences may fall. Similarly, under litigation, claims may be nullified. This way it is important to have several claims, even if they are protecting the same invention, as the more general claims may be nullified during litigation.

(a) With folded sleeves. (b) Without folded sleeves.

Fig. 13. Two T-shirts with front opening.

H. One Real Life Example

The following claim was extracted from a patent application [11]. The claim number eleven reads: "*11. System, according with claim 6, characterized to be used in information transmission and information reception, where information can be any kind of data, by any means in which the technique can be employed,* ***as example*** *optic fibers, wireless systems, radio systems, TV broadcasting, telemetry and data transmission system, breaking systems for car, train, airplanes, communications systems for satellites, rockets, memory access, missiles or spaceships.*"

Giving examples in the claims is not adequate nor allowed by the patent law, so the claim was modified to be acceptable for granting. The final claim in the granted patent [12] reads: "*19. The system according to claim 18, wherein the error correcting code is employed in a means selected from the group consisting of optic fibers, wireless systems, radio systems, television broadcasting, telemetry and data transmission systems, breaking systems for cars, trains, and airplanes, communications systems for satellites, rockets, memory access, missiles, and spaceships.*"

Notice that the text "*optic fibers, wireless systems, radio systems, television broadcasting, telemetry and data transmission systems, breaking systems for cars, trains, and airplanes, communications systems for satellites, rockets, memory access, missiles, and spaceships*" listed in the claim does not make it more general, but it makes it less general, as by the all-properties rule this is a mandatory property in the covered invention.

VI. CONCLUSIONS

We have discussed the logic behind patent interpretation. Our discussion considered the all-property rule and illustrated it through the use of set theory and Venn diagrams. This way, the basics of patent interpretation was described in terms of concepts familiar to the logic synthesis community. We also described how the recited properties in a claim are restrictions applied to an initial universal set.

ACKNOWLEDGMENT

André Reis is supported by a grant from CAPES - Brazil.

REFERENCES

[1] Corcoran, P., "It Is All in the Claims! [IP Corner]," Consumer Electronics Magazine, IEEE , vol.4, no.3, pp.83,89, July 2015

[2] Rackman, Michael I., "Inventors: Protect thyself: Careful attention to the claims section will go far toward establishing patent validity and extending the scope of protection," Spectrum, IEEE , vol.15, no.2, pp.54,60, Feb. 1978.

[3] Emma, Phil. 2005. Writing the claims for a patent. IEEE Micro 25, 6 (November 2005), 79-81.

[4] Osenga, Kristen. "Linguistics and patent claim construction." Rutgers Law Journal, Vol. 38 (2006), pp. 61-108.

[5] R. Guerra Fabris. La determination de l´objet du Brevet en Droit Bresilien et Europeen. Université de Strassbourg. PhD Thesis, June 22nd 2012.

[6] Edward Manzo. Patent Claim Interpretation - Global Edition, 2014-2015 ed. LegalWorks (October 24, 2014).

[7] Wang, Shyh-Jen. Designing around patents: a guideline. Nature biotechnology, v. 26, n. 5, p. 519-522, 2008.

[8] R. Schechter, J. Thomas, Principles of Patent Law (Concise Hornbook Series). West Academic Publishing, St. Paul MN, 2007.

[9] H. Parks, G. Musser, R. Burton and W. Siebler, Mathematics in Life, Society, and the World. Prentice Hall, Upper Saddle River, New Jersey, 2000.

[10] Brainard, Thomas D. "Patent Claim Construction: A Graphic Look." J. Pat. & Trademark Off. Soc'y 82 (2000): 670.

[11] L.V. Carginini and R.D.R.Fagundes. Method for Encoding and/or Decoding Multimensional and a System Comprising Such Method. United States Patent Application 20110083062

[12] L.V. Carginini and R.D.R.Fagundes. Method for encoding and/or decoding multimensional and a system comprising such method. United States Patent 8631307

Hybrid Synchronous-Asynchronous Tool Flow for Emerging VLSI Design

Filipp Akopyan, Carlos Tadeo Ortega Otero and Rajit Manohar
Computer Systems Laboratory
Cornell University
Ithaca, NY, U.S.A.
faa7@cornell.edu, {cto3,rajit}@csl.cornell.edu

***Abstract*—In the era of high-speed and low-power VLSI circuits, the question of which circuit family is best for a given application has become extremely relevant. From a designer's perspective, technology miniaturization brings increased parameter variation and decreased reliability, which lead to circuit malfunction. To mitigate the risks of undesirable circuit behavior, a designer has to make decisions not only at the micro-architectural scale, but also at the transistor-level scale. Various emerging technologies and non-conventional circuit families may help alleviate reliability problems and provide better performance.**

We have developed automated tools that allow designers to select the circuit family that yields the best results in terms of various design metrics for any application. Using our tools and techniques, the circuit choices can be made in a timely manner without in-depth knowledge of every circuit family under consideration. We demonstrate a tool flow that offers signi cant reduction in the design cycle time. We provide synchronous and asynchronous circuit libraries for designers to evaluate their circuit architectures and explain how this work can be extended to arbitrary types of circuit families for any given technology node. Finally, we evaluate the novel tools using ITC-99 benchmark suite and present simulation results for various circuit implementations in terms of throughput, power, process corners, and input statistics.

I. Introduction

The main purpose of modern Computer-Aided Design (CAD) tools is to aid designers combat the issues of power management and decreased reliability [1], as well as to decrease design cycle time. Contemporary industrial design flows are well understood, documented, constantly updated and improved by large-scale CAD corporations. However, the limitations of current design flows are also well-known [2], particularly in areas of low-power and high-speed VLSI. The increased complexity of VLSI designs paired with the challenges brought out by feature miniaturization have increased design cycle duration and time to market. One of the primary reasons for this increase is a lack of *fast and accurate* circuit level simulation tools for non standard cell-based circuit families. Presently, if a designer wants to perform gate-level optimizations of a non standard cell-based (or transistor-level optimizations of a standard cell-based) circuit family, he is forced to perform slow transistor-level simulations that can take several days to complete for a reasonable size VLSI design. Furthermore, these simulations are often infeasible, since many foundries do not reveal the backend structure of the gates in their standard cells to the designers due to confidential intellectual property agreements. In addition, *mixed* high- and low-level simulations are oftentimes complex or even impossible. Thus, designers are forced to perform most of their optimizations only at a high-level circuit scale, where no notions of transistors or even gates exist. This type of decision-making process renders crucial low-level design space optimizations unavailable. Emerging technologies are particularly impacted by the restriction on low-level design optimizations, since existing mapping from high-level description to a limited set of templated cells oftentimes negates the benefits of customized logic. Moreover, if a designer tries to perform low-level optimizations, he is required to have in-depth knowledge of multiple circuit families under consideration. As a result of such complexity, it becomes extremely difficult to predict what circuit family will be better for a given application in a particular environment under a given set of optimization criteria.

An efficient flow would potentially look as follows. A designer specifies his design using a high-level description language, such as Verilog. Then, an intelligent tool automatically synthesizes the description into multiple transistor-level netlists using various types of circuit families, which are then simulated at a transistor-level scale. After analyzing the data obtained from simulations, a designer has enough information to decide which circuit family is best-suited for their design, given their set of requirements.

There are currently dozens of circuit families to consider. Synchronous families include static CMOS, domino logic, differential signaling, etc. Many low-power/high-speed applications may benefit from self-timed (asynchronous) circuit families, which offer tradeoffs in terms of throughput and power consumption in comparison to synchronous circuits [3]. Some potential benefits of asynchronous logic include data-driven switching activity and absence of clock circuitry. However, these advantages come with the overhead of additional handshaking signals and potentially more complex data encoding (for example, dual-rail signals). Asynchronous circuits operate without global clocks and are comprised of many fine-grained hardware processes operating in parallel. Fig. 1 compares the progression in time of synchronous and asynchronous circuit processes. Asynchronous processes operate by communicating 'tokens' using handshake protocols. The data-driven nature of asynchronous circuits allows a circuit to idle without switching activity when there is no work to be done. Another advantage of asynchronous circuits is the capability for correct operation in the presence of continuous and dynamic changes in delays [4]. Sources of such local delay variations may include temperature, supply voltage fluctuations, process variations, noise, and radiation.

In order to allow designers to choose the best circuit family for a design, we have developed a tool flow for automatic synthesis of logic blocks into synchronous and asynchronous logic families. This automatic synthesis enables a systematic comparison between different circuit family implementations.

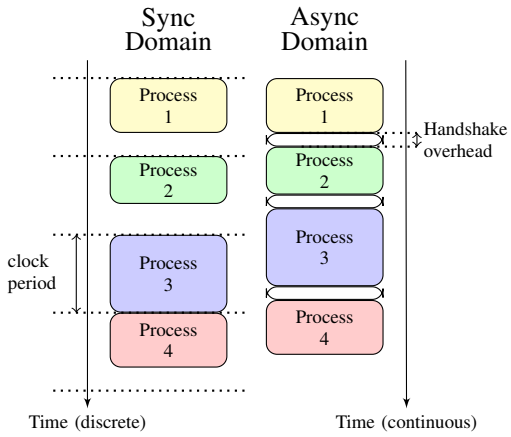


Fig. 1: Synchronous and Asynchronous Domains

After compilation of a given digital logic block into several synchronous and asynchronous implementations, we measure power, performance, and throughput. Using our flow, a designer can evaluate various circuit types and quantitatively determine under which conditions an asynchronous circuit would result in reduced delay or power consumption compared to its synchronous counterpart and vice versa. At this point one can decide which implementation should be used; prior to going through the entire synthesis/layout of all circuit blocks. Our tool flow also provides highly-optimized and pre-compiled cell libraries for different logic families, both synchronous and asynchronous. These libraries eliminate the requirement of thorough knowledge of all circuit families. All our tools are compatible with industrial standard cell libraries – a fact which gives a designer another degree of freedom. One has an option to pick the factory supplied standard cells, if they are sufficient for a given design, or to decide that another circuit family should be used instead.

This paper presents a novel hybrid synchronous-asynchronous tool flow with transistor-level libraries and all the necessary transformations to give a designer the flexibility of performing power/throughput/area optimizations at all levels of the design cycle. We evaluate our tools using the ITC-99 benchmarks, which are commonly used to assess CAD tools.

II. Related Work

Various CAD tools exist for VLSI synthesis using standard cells. Some examples are the industrial synthesis tool offerings from Cadence and Synopsys. Standard tools, such as Synopsys Design Compiler [5], mainly use synchronous static CMOS standard cell synthesis. These tools can take a RTL netlist and synthesize it into synchronous gate-level netlist using a supplied library of standard cells. Since oftentimes foundries do not supply transistor-level descriptions of the standard cells, the flexibility of performing detailed transistor-level optimizations may be taken away from a designer.

Asynchronous circuits can be synthesized using various methods. On one hand, Martin describes a system where CSP-type (Communicating Sequential Processes) programs [6] are decomposed recursively until they can be translated into corresponding transistor-level implementations [7]. On another hand, Tangram [8] and later Balsa [9] use a syntax directed approach to translate CSP to an abstract handshaking circuit, which is then mapped to a standard cell library. While Tangram and Balsa allow a designer to perform prototyping, the resulting circuits tend to be slower and area-inefficient, due to the limited selection of components supplied in the circuit libraries. Farhoodfar *et al.* also use syntax directed translation on hardware processes to synthesize a CSP-like program into a library of templated pipelined buffer cells (PCHB, PCFB) [10]. The resulting circuits compiled by this tool are highly-pipelined and very power hungry.

Another approach to synthesize asynchronous circuits is to leverage existing HDL and synthesis engines to generate asynchronous circuits. One such tool is Pipefitter, which takes an initial specification in Verilog and uses a commercial synthesis engine along with the asynchronous control synthesizer Petrify [11] to generate gate-level netlists. A delay line matched to the latency of the logic in each pipeline stage is added to ensure no race conditions exist. Law [12] presents a similar solution, except with more localized control circuits. Blunno [13] utilizes an approach that involves splitting a system into control and datapath blocks. In order to connect together those blocks, the synthesizable HDL is supplemented with mechanisms to implement asynchronous channels. Ellervee describes the techniques for automatic synthesis of asynchronous circuits for RTL-based design descriptions [14]. Lighthart [15] exploits HDL tools by having a functional library that implements 3-value logic. The resulting RTL undergoes syntactic transformations into CMOS-implementable delay insensitive 2-phase gates. Weaver [16] compiles a synchronous single-rail RTL description into a finely-pipelined QDI asynchronous implementation by replacing synchronous logic gates and registers with their QDI equivalents.

Beerel designed Proteus [17], which is a hybrid approach. The input to Proteus is specified in CSP; the tool goes through a series of syntactic transformations to get an RTL representation, which is then mapped into gates using a single-track full buffer template. Proteus offers significant advantages over other tools as it exploits the expressiveness of CSP and it is complemented with optimizations targeted for asynchronous circuits. The main disadvantage of Proteus is that it requires a complete rewrite of the design into CSP, which makes it unwieldy to perform quick prototyping and comparison against synchronous circuits. Furthermore, Proteus optimizations are limited on large blocks that perform complex algorithmic functions.

To the current knowledge of the authors, the results from previous work have not provided a flexible tool flow, which is evaluated using detailed simulations, analysis and comparison between synthesized asynchronous and synchronous circuits. The extensions of previous tools to utilize *arbitrary* transistor-level circuit families have not been discussed either.

III. Tool Flow Comparison

A. Industrial Tool Flow

A sample industry-standard tool flow for ASIC implementation is shown in Fig. 2. First, a designer creates a high-level Verilog RTL description of the circuits. Second, the RTL is synthesized into a gate-level netlist using a *restricted* set of standard cells with pre-layout timing estimates supplied by the foundry. Third, a designer works with automatic place and route tools to obtain a physical implementation (layout) of the circuit. In practice, the third step is not fully automatic and requires significant manual effort. The generated layout

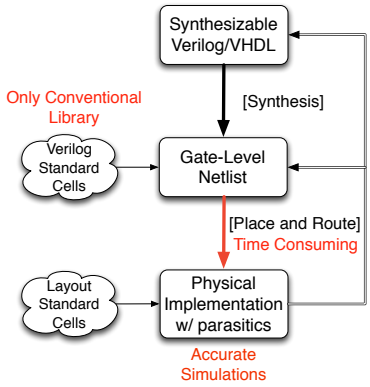


Fig. 2: Industrial Tool Flow

can then be extracted with parasitic effects, with a caveat that oftentimes the back-end contents (transistors) of the standard cells are not revealed. Only after these steps are completed, a designer can perform *accurate* analog-level simulations with the estimated parasitic elements from the layout — for the *first* time since the beginning of the design cycle. The place and route step takes a large portion of the design cycle time and needs to be partially/fully repeated after every modification to the circuits prior to having the ability to perform the next set of accurate simulations. With this flow, it takes designers a *long time* to get to the first (and all subsequent) set of accurate simulations, where many common problems, such as cross coupling, charge sharing, and signal swing issues are revealed.

B. Proposed Tool Flow

In order to reduce the design time and allow engineers to test various types of circuit families for a given implementation, we augment the industrial tool flow, as demonstrated in Fig. 3. Our tool flow contributions are highlighted using green font and surrounded with a dotted box.

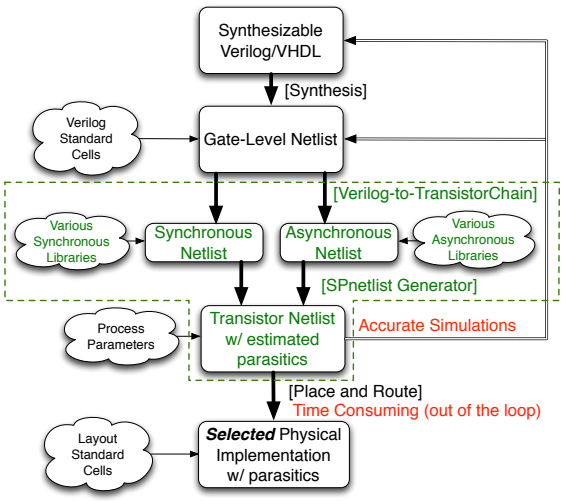


Fig. 3: Proposed Tool Flow

In this flow, a designer creates a high-level Verilog description of the architecture, which is then synthesized into a gate-level netlist using a set of supplied high-level (logic only) standard cells. Any industrial tool, such as Synopsys Design Compiler, may be used for this level of synthesis. At this point we use our custom tool, Verilog-to-TransistorChain (Sec V-A) to generate two netlists: asynchronous and synchronous. Here, instead of using only the "black-box" industrial cells, a designer has a choice of also using highly robust cells from various different circuit family libraries. The circuit family and block granularity is specified in our Verilog-to-TransistorChain tool based on architectural considerations and may be modified at any point. For example, if the selected circuit family for a given block is synchronous, the synthesized netlist is used directly with transistor-level libraries of various synchronous families. However, if the selected family is asynchronous, we perform several netlist transformations (described in later sections) to obtain a logically equivalent asynchronous gate-level netlist. In this case, asynchronous transistor libraries are attached to the generated gate-level netlist. Presently, our tools perform the transformation of synchronous gate-level netlists into Quasi-Delay Insensitive (QDI) [4] asynchronous netlists, but it is simple to perform a similar set of transformations to obtain other types of asynchronous netlists (e.g. bundled-data). A designer *does not* need to have an in-depth understanding of the operation of asynchronous circuits because our tools automatically perform semantic-preserving transformations of the original RTL. Afterwards, we use another custom tool, SPnetlist Generator (Sec V-B) to produce a transistor-level netlist with estimated parasitics for the desired circuit families, which can now be used for accurate simulations. To conduct a realistic simulation, our tools include wiring, fan-out and internal transistor capacitance models (Sec V-B). These models are used to compute parasitics associated with each gate. The advantage of our flow is that, at this *early* point in the design cycle, engineers can perform analog simulations using industrial simulators. These simulation results take into consideration most of the parasitic effects of a given design and aid in making a decision of which circuit family to use for the blocks.

Our proposed tool flow eliminates the iterative place-and-route step for all the preliminary design decisions and measurements. Once the transistor netlist is finalized and satisfies all the metrics, the place and route step is performed only *once* with some minor post-layout adjustments to account for placement related cross-talk, transmission line effects, etc.

C. Proposed Simulator Chain

In our proposed tool flow, a designer has much more flexibility simulating circuits at various pre-layout levels of development, as shown in Fig. 4. As in the industrial flow, the behavioral and RTL Verilog/VHDL code, as well as the gate-level netlist may be simulated with an industrial simulator, such as Synopsys VCS [5]. After we generate the intermediate synchronous and asynchronous netlists, the synchronous netlist may be simulated with the same simulator as before. In the asynchronous scenario, we use our custom digital simulator, PRSIM (Sec V-C). PRSIM may also be used to simulate synchronous netlists, which use custom libraries. We have also developed a simulator interface, Automatic Cosimulation and Environment Generator, to allow the cosimulation of synchronous and asynchronous circuits simultaneously (Sec V-D).

From the synchronous and asynchronous netlists we automatically generate a transistor netlist with estimated parasitics for the block. This netlist is simulated using any accurate

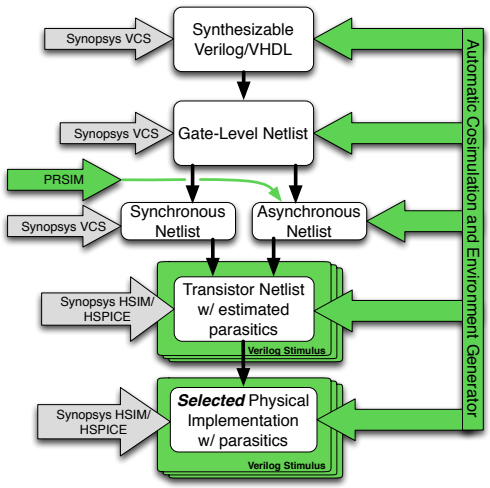


Fig. 4: Sample Proposed Simulator Chain

analog simulator (e.g. HSPICE, Ultrasim, HSIM, Spectre). Majority of the iterative implementation improvement and evaluation tasks are performed at this stage (prior to place and route). After layout is performed, the same analog simulator may be used for the final assessment of the *selected* physical implementation with exact parasitics.

IV. Industrial Tools Used in the Flow

Synchronous Digital Simulator. We use Synopsys VCS for behavioral, RTL, and gate-level simulations. VCS includes an Verilog Procedural Interface, which provides hooks to perform simulations at multiple levels of circuit abstractions.

Transistor-Level Analog Simulator. Synopsys HSPICE is our preferred simulator for accurate circuit simulation. However, we favored Synopsys HSIM for larger circuits [18]. HSIM drastically improves simulation speed because of its hierarchical approach to circuit modeling.

V. Novel Tools Overview

A. Verilog-to-TransistorChain

The Verilog-to-TransistorChain tool converts a Verilog netlist into an equivalent intermediate format gate-level netlist that hierarchically describes pull-up and pull-down transistor chains (called production rules) of each gate used in the design. This tool can be used to generate either a synchronous netlist or an asynchronous netlist. Generation of a synchronous netlist is straightforward. The synthesized Verilog netlist consists of instances of standard cells. Each standard cell can be described using production rules annotated with transistor sizing information. Combining this information with the instances in the Verilog netlist results in a synchronous netlist specified using production rules.

Generation of an asynchronous netlist uses a syntax-directed translation of the synchronous netlist. Each standard cell is replaced with an equivalent asynchronous standard cell, where all input and output wires are replaced by asynchronous handshake channels that carry one-bit data. Combinational logic is replaced by equivalent one-bit pipeline stages. Flip-flops are replaced by one-place token buffers that are initialized with the value of the flip-flop on reset. The main additional step necessary in the asynchronous case is to insert explicit circuits to support fan-out (one-to-many) connections by combining the handshake signals using completion gates. This results in an asynchronous netlist that is structurally equivalent to the synchronous netlist generated by standard logic synthesis. We used this approach to minimize the differences between the synchronous and asynchronous netlists.

The Verilog-to-TransistorChain can work with other asynchronous families (e.g. bundled data) with minimal tool modifications, if required by a designer. Eventually, for more efficient conversion, we would like to perform the synchronous-to-asynchronous transformation using a higher-level behavioral description (where applicable). However, that would require a more complex compiler to perform an optimized and semantic-equivalent translation. Once this high-level asynchronous transformation is implemented, a designer may use other automated methods, such as concurrent pipeline synthesis described by Teifel [19], to perform the circuit synthesis operation. Such high-level transformation is outside of the scope of this work.

B. SPnetlist Generator

This tool syntactically translates every production rule generated by the Verilog-to-TransistorChain into a transistor-level hierarchical SPICE netlist. A single production rule takes the form $G \mapsto S$, where G is a boolean expression called the guard and S is a boolean assignment. Each production rule corresponds to a pull-up or a pull-down transistor switching network, depending on whether the boolean assignment S is for an up-going or a down-going transition. The ordering of transistors is deterministically derived from the production rule in the following manner. Power rails are connected to the source terminal of the transistor generated form the left-most literal in the production rule guard, whereas the output is connected to the drain terminal derived from the right-most literal of the guard. A rule,

```
a & b -> c-
```

is translated into a transistor netlist starting from ground (GND). The '$-$' symbol means that the output describes a down-going transition. The SPICE netlist that corresponds to this production rule is:

```
M0_ GND a #3 GND nfet W=0.2U L=0.045U
M1_ #3 b c GND nfet W=0.1U L=0.045U
```

A configuration file controls multiple parameters such gate input and output capacitances, wiring loads, minimum p- and n- transistor size, source/drain area and perimeter, and spacing between two FETs in the same diffusion stack. From these parameters, the SPnetlist Generator automatically calculates parasitic capacitances and default areas and perimeters for transistor chains. This automatic calculation is crucial to accurately model the behavior of a synthesized circuit.

C. Asynchronous Digital Simulator (PRSIM)

The simulator we use for asynchronous and non-conventional synchronous circuits testing is a custom event-driven digital simulator. The input to PRSIM is an automatically generated (using Verilog-to-TransistorChain) netlist based on production rules. A set of production rules can be viewed as a sequence of events. All events are stored in a queue. When all pre-conditions of an event become true, a timestamp is attached to that event. Once the timestamp of an event coincides with PRSIM's running clock, the event

(production rule) is executed. Timestamps of events can be deterministic or can follow a probability distribution. The probability distributions may be random or long-tailed (i.e most events are scheduled in the near future, while some events - far in the future). Such flexibility allows PRSIM to simulate behaviors of asynchronous and synchronous circuits at random or deterministic timing. Whenever an event is executed, PRSIM performs multiple tests to verify correct circuit behavior, including: 1) Verify that all events are *non-interfering*; an event is non-interfering if it does not result in a short circuit under any allowed input conditions; 2) Verify proper codification on synchronous buses and asynchronous channels; 3) Verify the correctness of expected values of a channel or a bus (optional); 4) Verify that events are *stable*; in asynchronous context, an event is considered stable when all receivers acknowledge each signal transition before the signal changes its value again. We have extended PRSIM's functionality to evaluate energy, power and transient effects of temperature & supply voltage on gate delays, if desired.

D. Automatic Cosimulation and Environment Generator

The Automatic Cosimulation tool allows simultaneous cosimulation of an arbitrary mix of synchronous and asynchronous circuit-families at various levels of abstraction. This tool was developed using two main components: VPI-PRSIM and the custom Environment Generator (which can be also used as a stand-alone tool). The connectivity and interactions between our tools and all the simulators are shown in Fig. 5.

The first component of Automatic Cosimulation, VPI-PRSIM, allows dynamic bindings between the VCS and PRSIM simulations. We have built VPI-PRSIM using the Verilog Procedural Interface (VPI / PLI 2.0) as defined in the Verilog standard [20]. The VPI-PRSIM module provides an API that can be used within Verilog/VHDL to transfer signals and control between a C/C++ program and a Verilog/VHDL simulator. This module allows simulators to exchange timestamps and events through callbacks coordinated by the VPI interface. VPI-PRSIM registers callback functions that are exercised whenever an event occurs. The registered callback functions provided by the VPI interface allow our event-driven simulator PRSIM to process changes in Verilog interface signals. Any time the Verilog/VHDL simulator calls a PRSIM function, PRSIM checks for pending events (signal transitions), updates its event queue and any Verilog signals that are modified by PRSIM. PRSIM synchronizes the simulation timestamp between the two simulators before and after an event is evaluated. The bindings between VCS and HSIM are performed using Synopsys's VCS-HSIM cosimulation interface that also relies on the VPI (PLI 2.0) interface [18].

The second component of our Automatic Cosimulation is the Environment Generator, which automatically creates interfaces between the Verilog/VHDL simulator VCS, the PRSIM asynchronous simulator, and the HSIM transistor-level simulator. It generates the top-level Verilog to interconnect instances defined at various levels of abstraction including Verilog/VHDL, production rules, and transistor-level netlists. The Environment Generator has preloaded communication primitives: boolean signals, buses, and channels. Whenever a connection needs to be made between different levels of abstraction, our tool detects the type of the connection required, and automatically emits the Verilog code that performs the necessary connections. The Environment Generator also allows to back annotate parasitics, specify parameters, interfaces, and initial conditions (ic-s) for PRSIM and HSIM simulators. Furthermore, our tool has an extensive module library that allows designers to automatically send, receive, probe, and check correctness of communication channels and buses. These modules are configurable and can be added on demand using a configuration file.

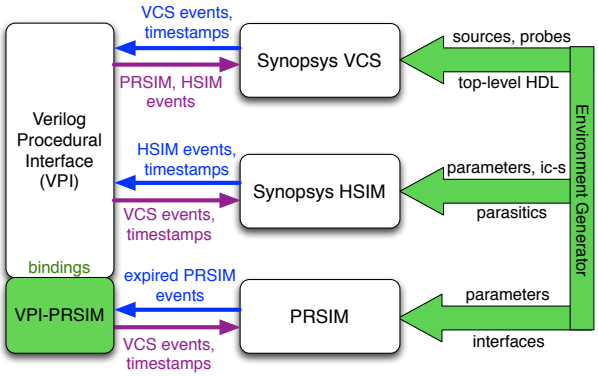


Fig. 5: Automatic Cosimulation Flow Chart

Automatic Cosimulation, with Environment Generator as its key component, allows an engineer to work with hybrid synchronous-asynchronous designs, while implementing the actual circuits using multiple families. A designer can cosimulate high-level behavioral, RTL, production rule, and transistor-level circuit descriptions with a mixture of digital and analog (transistor-level) simulators.

E. Circuit Family Libraries

Our tool flow supports multiple synchronous and asynchronous transistor-level libraries. This feature enables synthesis of circuit descriptions into corresponding transistor-level netlists for various circuit families. Post synthesis (pre-layout) analysis allows designers to select the best circuit family for each part of the architecture depending on the targeted metric, such as power consumption, throughput, latency, etc. For a new technology node it is straightforward to calibrate existing libraries. The parameters that change with a new node are process properties and transistor descriptions: minimum size, parasitic parameters, mobility values, etc. – used for the SPnetlist Generator configuration. As an example, we have created a static CMOS synchronous family and a QDI asynchronous family due to QDI's robustness to delay, temperature, and process variations [21] . We have implemented the QDI cells using PCEHB and HCHB type handshake reshufflings [22].

Layout for the cells in any logic family can be automatically generated by an on-demand std-cell generator, such as custom *cellTK* [23]; or an industrial place and route tool, such as Cadence Encounter. Unlike a traditional ASIC flow, which relies on a predefined and characterized gate-level netlist, *cellTK* generates highly efficient layout cells for an arbitrary transistor-level netlist.

VI. Tool Flow Evaluation

A. Benchmark Considerations

To evaluate our tools, we utilized our two aforementioned libraries: a static highly-robust synchronous library (industry-

based) and a QDI PCEHB-based asynchronous library. The circuits are implemented in a $45\,\mathrm{nm}$ technology node [24].

We have obtained ITC-99 benchmarks [25,26] and compiled them into synchronous and asynchronous netlists using our tool flow. We selected the ITC-99 benchmarks because of their variety in functionality, structure, and complexity as shown in Table I. The benchmarks range from simple sets of gates to designs with around 600 gates and dozens of flip-flops. Some benchmarks represent control processes, while others embody complex arithmetic and logical data paths. The behavioral and RTL design of the ITC-99 benchmarks is originally created in VHDL, which is also supported by our tool flow.

TABLE I: ITC-99 Benchmarks and Functionalities

Name	Description	Gates	Flip-Flops	I/O ports	VHDL lines
b01	Serial flow comparator	45	5	4/2	110
b02	BCD number recognizer	25	4	3/1	70
b03	Resource arbiter	150	30	6/4	141
b04	Min-Max search	480	66	13.8	80
b05	Memory	608	34	3/36	319
b06	Interrupt handler	66	9	4/6	128
b07	Count points on a straight line	382	51	3/8	92
b08	Numeric series	168	21	11/4	89
b09	Serializer/Deserializer	131	28	3/1	103
b10	Voting system	172	17	13/16	167

B. Throughput Comparison

To measure the throughput of the asynchronously implemented benchmarks, we run transistor-level simulations of the circuit with random input patterns for a sufficient amount of time to reach steady state and then capture the average throughput. All simulations are initially performed using the typical-typical (TT) transistor corner models. For the synchronous implementations, we initially run the clock at a low frequency for a long period of time and record the trace of output values, using random inputs. We then gradually increase the clock frequency and compare the obtained values with the trace recorded initially. Whenever we find a mismatch in the output values or internal state, we postulate that there was a timing violation somewhere in the circuit, i.e. setup or hold time of a flip-flop was violated. At that point the last correctly recorded frequency is regarded as the maximum circuit frequency under the given conditions. All the original synchronously-implemented ITC-99 benchmarks have *ideal* clocks, i.e. they do not account for clock distribution and clock uncertainties such as signal jitter and skew. As a result, to perform a fair comparison, we padded the synchronous operating frequency with a $20\,\%$ safety margin, which is consistent with studies found in the literature [27].

The average absolute throughput in MHz for both synchronous and asynchronous implementations is shown in Fig. 6, in the columns labeled *Sync-Nominal* and *Async-Nominal*. The synchronous implementations perform better for less complex benchmarks [b01, b06], while more complex benchmarks gain throughput using asynchronous realizations [b05, b07].

C. Process Variations

In order to analyze the impact of foundry process variations incurred due to typical device mismatch, we run the same set of synchronous and asynchronous simulations in two extreme process corners: slow-slow (SS) and fast-fast (FF). Fig. 6 also demonstrates the effect that process corners have on the behavior of synchronous and asynchronous circuits respectively.

As expected, the throughputs of both circuit families increase when both NFET and PFET devices get faster. On average, the designs running in the FF process corner deliver an average of 27% more throughput than in the TT corner. On the other hand, when transistors are running in the SS process corner, the benchmark performance degrades by an average of 25% in this process technology. Synchronous circuits tend be slightly less sensitive to process variation in terms of performance compared to their asynchronous counterparts, as demonstrated in Fig. 6. However, what is not captured by these results is that synchronous implementations must account for the worst possible variation, while asynchronous implementations *automatically* adjust to the wide range of process variations. The trends for the SS and FF corners exactly resemble the nominal scenario, considering the relative speed variation. Such behavior confirms that all the analysis, as well as all the conclusions drawn from our experiments with the nominal device models can also be used in the presence of process variations. The variations considered in this study are 3σ, represented by the process corners.

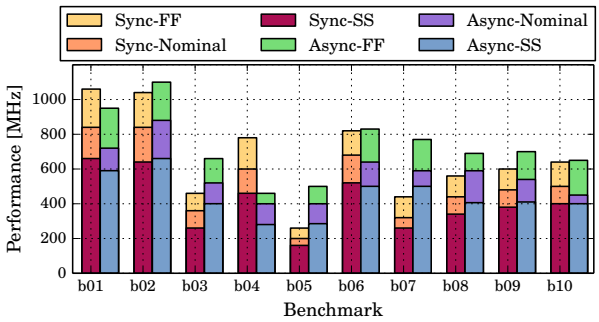


Fig. 6: Performance and Process Variations Analysis

D. Power Analysis

In our power experiments, we want to find the effective input signal duty cycle that results in equal power consumption of the synchronous and asynchronous implementations, while still maintaining the ability of processing bursty inputs at the maximum throughput. For that, we fix the throughput of both synchronous and asynchronous circuits (per benchmark) to the highest frequency that both implementations support nominally, i.e. lower value of the {Sync-Nominal, Async-Nominal} pair in Fig. 6. We then vary the input signal duty cycle by sending bursts of high-throughput inputs, followed by periods of input inactivity. Our analysis assumes that the power consumption is dominated by the dynamic power. This proves to be correct in low leakage technologies, where similar power consumption is observed for the presented synchronous and asynchronous implementations.

At the scale of benchmarks in our analysis (small and medium size circuits), the overheads of clock-gating control complexity offset the benefits from clock-gating itself. We, thus, compare the asynchronous implementations to non clock-gated synchronous implementations. Also, due to the fact that clock tree networks were not physically present in the original benchmarks, we have omitted the clock tree load and switching energy for all synchronous benchmarks.

Mathematically, the dynamic power estimated by Eq. 1, is used to derive Eq. 2 (for asynchronous circuits) and Eq. 3 (for synchronous circuits). In the following calculations f_{sw} is the circuit's switching frequency, which is estimated with the frequency of circuit operation in each scenario. In the asynchronous case, Eq. 2, α_{async} is the activity factor of the circuit, C_{async} represents the total load capacitance in the asynchronous implementation and f_{avg} is the input frequency averaged across all bursts and inactivity periods. f_{avg} is also representative of the average duty cycle of the inputs. The activity factor in a QDI asynchronous design is maximally equal to 1, since majority of the nodes in an asynchronous circuit switch twice per handshake by going to active phase and neutral phase. In the synchronous calculation, Eq. 3, there are two components contributing to the circuit's dynamic power: the combinational logic term and the clock term. Here, C_{cl} is the lumped load capacitance of the logic, and C_{clk} is the clock network capacitance. α also has two components: α_{cl}, which is the activity factor of the logic and, for consistency, α_{clk} - the activity factor of the clock. f_{clk} denotes the high-throughput signal switching within input bursts and, in the synchronous case, represents the clock frequency required to support this high throughput. f_{cl}, the switching frequency of combinational logic is equal to $\alpha_{cl} f_{avg}$, since the combinational logic needs to switch only when inputs change (which in the case of bursty inputs does not happen on every clock cycle). α_{cl} is maximally equal to 0.5 for combinational logic according to statistics and α_{clk} is equal to 1, since a clock has two edges per cycle.

$$P = C_{total} V dd^2 f_{sw} \tag{1}$$

$$\begin{aligned} P_{async} &\approx \alpha_{async} C_{async} V dd^2 f_{avg} \\ &= C_{async} V dd^2 f_{avg} \end{aligned} \tag{2}$$

$$\begin{aligned} P_{sync} &\approx C_{cl} f_{cl} V dd^2 + C_{clk} f_{clk} V dd^2 \\ &= (\alpha_{cl} C_{cl} f_{avg} + \alpha_{clk} C_{clk} f_{clk}) V dd^2 \\ &= (0.5 C_{cl} f_{avg} + C_{clk} f_{clk}) V dd^2 \end{aligned} \tag{3}$$

We can get a closed form solution for approximate value of f^*_{avg} (power break-even average input frequency for synchronous and asynchronous implementations) by equating Eq. 2 and Eq. 3 and solving for f_{avg}. The obtained result is given by Eq. 4 and shows that f^*_{avg} is directly proportional to the product of maximum clock rate f_{clk} and clock network capacitance C_{clk}; and inversely proportional to the difference of total asynchronous capacitance C_{async} and fraction of combinational logic capacitance C_{cl}.

$$f^*_{avg} = \frac{C_{clk} f_{clk}}{C_{async} - 0.5 C_{cl}} \tag{4}$$

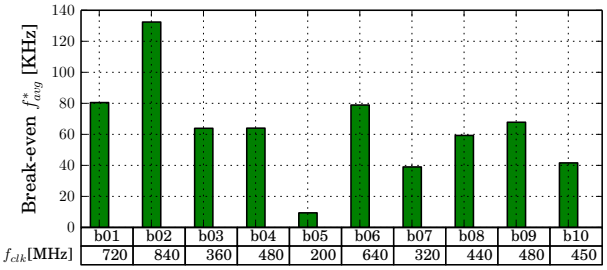


Fig. 7: Power Break-Even Average Input Frequency

Fig. 7 shows the power break-even average input burst frequency f^*_{avg} for each benchmark. Specifically, at f^*_{avg} the power consumption of the asynchronous implementation is equal its synchronous counterpart. f_{clk} in this figure is shown in MHz. In the analysis of Fig. 7, all the capacitances were extracted from the actual circuits (while omitting the clock distribution network). One can see that average break-even frequency occurs in the 10-s to 100-s of KHz range for different benchmarks (while still supporting 100-s of MHz maximum input throughput). For bursty, yet high-throughput inputs within the busts, asynchronous circuits provide a better tradeoff in terms of power consumption below this f^*_{avg}. Above f^*_{avg} synchronous circuits are more power efficient for our presented set of gate-level netlist transformations. To improve the power efficiency of asynchronous circuits in our tool flow we are looking into implementing several additional techniques, as outlined in Sec VI-F.

E. Design Space Investigation

Our tools demonstrate the tradeoffs of implementations in the synchronous and asynchronous design spaces. An example study is shown in Fig. 8, which presents a 3-dimensional view of the average input frequency vs. maximum supported throughput vs. calculated dynamic power consumption for benchmark b01. Fig. 8(a,b) demonstrates power consumption for the asynchronous and synchronous implementations respectively. Similar to previous experiments, f_{avg} denotes the input frequency averaged across bursts and inactivity periods on y-axis, f_{clk} denotes the maximum signal throughput within the bursts on x-axis, and the color gradient represents the power consumption. Fig. 8(c) shows the difference in dynamic power, P_{async} and P_{sync}, for the two implementations. One observes that synchronous realization has less power consumption in the presence of constant high input data rates represented by f_{avg}. In contrast, the asynchronous implementation results in a more efficient power consumption when maximum throughput is required (f_{clk}), but the average input frequency is low (f_{avg}). The graphs for the other benchmarks follow a similar trend. The only difference in the other benchmarks is the absolute value on the break-even f^*_{avg}, as shown in Fig. 7.

F. Analysis of Simulation Results

The obtained results agree with the original hypothesis that we have made while developing these synchronous-to-asynchronous transformations. In many cases, the obtained asynchronous implementations were able to achieve higher performance than their synchronous counterparts. Additionally, our initial prognosis, which turned out to be true, stated that due to the nature of the transformations (gate-level netlist conversion), the number of transistors in the asynchronous implementation would on average be higher than that of the corresponding synchronous implementation (the resulting area overhead of asynchronous circuits was at least twice the area of the synchronous circuits). Similarly, in the QDI 4-phase handshake circuits the signal activity factor is higher, since the circuit on every data token goes through the active phase and returns to the neutral phase. These facts lead to a higher power consumption of the asynchronous implementations in scenarios, where the inputs arrive at high frequency with no inactive periods. However, the asynchronous implementations automatically produced in this tool flow have proven to be advantageous in designs, where there are bursts of high-frequency

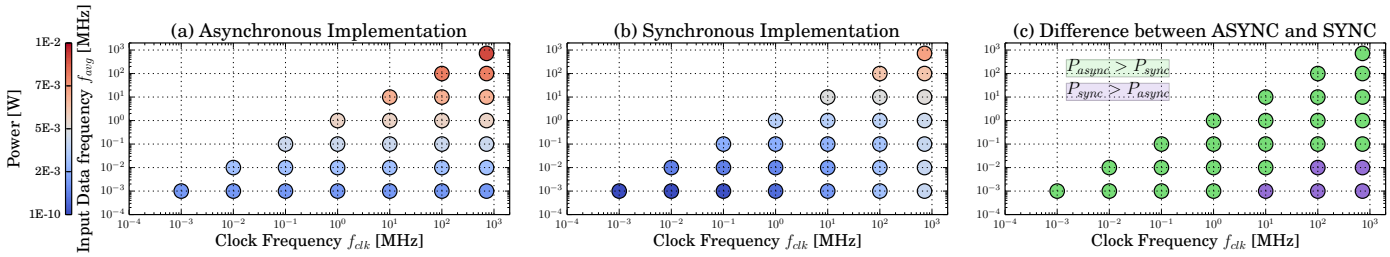


Fig. 8: b01 benchmark Power Consumption (in Watts) for sweeps of f_{avg} and f_{clk} frequencies

inputs followed by long quiescent periods. Such applications include speech processing, on-chip networks, neuromorphic circuits, etc. In these designs, the circuit irealizations must still be able to support maximum throughput, which means that the clocks would have to switch at the maximum input rate if implemented in a synchronous manner.

We are planning to implement several additional techniques to enhance out tool flow. Specifically, we are looking into optimizing the gate-level transformations and pipelining based on designer's metrics (area, throughput, power). To perform this effectively we would extend our tool flow and add libraries for other synchronous and asynchronous circuit families. Beyond that, as mentioned previously, we want to implement high-level transformations to allow implementation-specific optimizations earlier in the design cycle.

VII. Conclusion

In this paper, we proposed a novel way of designing complex VLSI circuits. This approach is especially useful in large-scale projects where accurate measurements and decisions early in the design cycle can drive many high-level architectural decisions. Our tool flow allows designers to select the ideal circuit family for each digital block in the design based on various metrics, without necessitating thorough expertise in all logic families. The tool flow may be extended with additional synchronous and asynchronous circuit family libraries without the necessity to modify the underlying tools.

As part of the novel flow, we also presented a method for integrated simulation framework of both synchronous and asynchronous circuits. This framework allows cosimulation using different circuit technologies and different levels of abstraction. To evaluate the framework, we provided quantitative results comparing multiple benchmark implementations using two logic families: QDI asynchronous and static synchronous. The presented gate-level pipelining approach to compile asynchronous circuits increases the throughput compared to the synchronous approach, while also increasing power consumption and area in many cases. However, for applications where the circuits are mostly idle and have bursts of high frequency activity, the power consumption of the asynchronous implementations is lower. In the future, among other enhancements, we would like to focus on performing not only gate-level syntactic transformations, but also higher-level, semantic-preserving circuit family-specific architectural optimizations and transformations.

References

[1] D. Sylvester and H. Kaul, "Future performance challenges in nanometer design," in *Proc. Design Automation Conference*, ACM Press, 2001.

[2] R. Bryant and et al., "Limitations and challenges of computer-aided design technology for CMOS VLSI," in *Proc. of the IEEE*, 2002.

[3] D. Fang, S. Peng, C. LaFrieda, and R. Manohar, "A three-tier asynchronous FPGA," in *International VLSI/ULSI Multilevel Interconnection Conference*, 2006.

[4] A. J. Martin, "The Limitations to Delay-Insensitivity in Asynchronous Circuits," in *ARVLSI*, 1990.

[5] "Synopsys tool manuals," in *https://solvnet.synopsys.com/*, 2010.

[6] C. van Berkel and R. Saeijs, "Compilations of communicating processes into delay-insensitive circuits," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1988.

[7] S. Burns and A. Martin, "Syntax-directed translation of concurrent programs into self-timed circuits," tech. rep., DTIC Document, 1988.

[8] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," in *EDAC*, IEEE, 1991.

[9] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis language," *The Computer Journal*, vol. 45, no. 1, 2002.

[10] H. P. K. Saleh, M. Naderi, M. H. Shafiabadi, H. Kalantari, and A. Farhoodfar, "Synthesis Tool for Asynchronous Circuits Based on PCFB and PCHB," in *Proc. Computer Society of Iran Computer Conference*, 2004.

[11] J.Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers,"

[12] C. Law, B. Gwee, and J. Chang, "Modeling and synthesis of asynchronous pipelines," *IEEE Trans. Very Large Scale Integr. Syst.*, 2011.

[13] I. Blunno and L. Lavagno

[14] J. Oberg, J. Plosila, and P. Ellervee, "Automatic synthesis of asynchronous circuits from synchronous RTL descriptions," in *NORCHIP Conference*, 2005.

[15] M. Lighthart, K. Frant, R. Smith, A. Taubin, and A. Kondratyev

[16] A. Smirnov, A. Taubin, M. Su, and M. Karpovsky, "An automated fine-grain pipelining using domino style asynchronous library," in *ACSD*, 2005.

[17] M. Ferretti, R. O. Ozdag, and P. A. Beerel, "High performance asynchronous ASIC back-end design flow using single-track full-buffer standard cells," in *Proc. 10th IEEE International Symposium on Asynchronous Circuits and Systems*, 2004.

[18] Synopsys, Inc., "HSIM^{PLUS} © Reference Manual," 2011.

[19] J. Teifel and R. Manohar, "Static tokens: Using dataflow to automate concurrent pipeline synthesis," in *International Symposium on Asynchronous Circuits and Systems*, 2004.

[20] "IEEE standard for Verilog hardware description language," *IEEE Std 1364-2005*, 2006.

[21] A. M. Lines, "Pipelined asynchronous circuits," tech. rep., Caltech, 1998.

[22] C. LaFrieda and R. Manohar, "Reducing power consumption with relaxed quasi delay-insensitive circuits," in *Proc. of 15th IEEE Symposium on Asynchronous Circuits and Systems*, 2009.

[23] R. Karmazin, C. O. Otero, and R. Manohar, "cellTK: Automated layout for asynchronous circuits with nonstandard cells," in *International Symposium on Asynchronous Circuits and Systems*, 2013.

[24] "Nangate 45nm open cell library," in *http://www.nangate.com/*, 2009.

[25] "Overview of ITC-99 benchmarks form Torino, Italy," in *http://www.cad.polito.it/downloads/tools/itc99.html/*, 2010.

[26] "ITC-99 benchmark homepage from University of Texas," in *http://www.cerc.utexas.edu/itc99-benchmarks/bench.html*, 2010.

[27] A. K. Uht, "Uniprocessor performance enhancement through adaptive clock frequency control," in *IEEE Transactions on Computers*, 2005.

Fluid Pipelines: Elastic Circuitry without Throughput Penalty

Rafael Trapani Possignolo
rafaeltp@soe.ucsc.edu

Elnaz Ebrahimi
elnaz@soe.ucsc.edu

Haven Skinner
hskinner@soe.ucsc.edu

Jose Renau
renau@ucsc.edu

University of California, Santa Cruz
1156 High Street
Santa Cruz, CA, USA
http://masc.cse.ucsc.edu

ABSTRACT

In chip design, pipeline depth and cycle time are fixed early in the design process but their impact on physical design can only be assessed when the implementation is mostly done, making it impractical to change such parameters. Elastic Systems are insensitive to latency, and thus enable changes in the pipeline depth late in the design time with low effort. Nevertheless, current Elastic System implementations have significant throughput penalty when stages are added in the presence of pipeline loops. We propose Fluid Pipelines, an evolution that allows pipeline transformations within an Elastic System without throughput penalty. Formally, we introduce "or-causality" in addition to the already existing "and-causality" in Elastic Systems. This gives more flexibility than previously possible but requires the designer to annotate the intended behavior of the circuit. Fluid Pipelines are able to improve the optimal energy-delay (ED) point by shifting both performance (by 176%) and energy (by 5%). Fluid Pipelines also allow for exploration of the Pareto frontier enabling points like delivering 33% better top performance using 83% less energy. Fluid Pipelines open many research opportunities in both EDA and architecture and enable interesting design space exploration. We envision a scenario where automated tools would be able to generate, from the same RTL different pipeline configurations for, *e.g.*, low power, high performance, so forth. In that sense, Fluid Pipelines are able to greatly reduce design effort.

1. INTRODUCTION

In current digital design practices, cycle time and pipeline depth are set early in the design process due to their impact on the other design parameters. Meeting a certain cycle time usually requires multiple time-consuming iterations between design and implementation [11]. Elastic (or latency insensitive) Systems [5, 7, 8, 18] are an alternative to the traditional fixed cycle pipeline paradigm. Elastic systems are based on the assumption that the correctness of the system does not depend on the latency (number of clock cycles) between two subsequent events, but on their order [5, 18]. This allows for the insertion of new stages later in the design time without breaking the circuit correctness [5].

Changing the number of pipeline cycles, also known as Recycling [2, 16], is possible in Elastic Systems but constrained by the presence of sequential loops[1]. This reduces the applicability of recycling, because most complex circuits, such as processors, include sequential loops. Traditional Elastic Systems rely on an automated flow that transform regular synchronous circuitry into elastic. Since the flow does not have knowledge on the intended behavior of the circuit, it has to maintain the completion order of events. This has the side effect of reducing the overall throughput of the circuit [5, 15]. Throughput losses can be mitigated by the use of Early Evaluation [2] but the whole system remains limited by the worst sequential loop, even if such loop is not actually used.

In contrast, Out-of-Order (OoO) execution is omnipresent in modern digital design and is known to improve system throughput. In this paper, we propose Fluid Pipelines, an evolution of current Elastic circuitry, that enable unordered completion order. Since the flow cannot change the behavior of a circuit,Fluid Pipelines rely on designer annotations to the code where ordering can be changed. Fluid Pipelines are a generalization of Elastic Systems, since without user annotations, they behave like Elastic Systems. User defined elasticity has been proposed [3], and is thought to improve design methodologies [18]. We go one step further and evaluate new design paradigms within Elastic Systems, one that allow for OoO behavior. By exposing the Merge and Branch to the designer, the flow becomes aware of the intended behavior, and when the relative completion between operations does not affect design correctness.

Fluid Pipelines are able to re-claim the throughput losses from the automated conversion. The automated flow of Elas-

[1]By sequential loops, we mean cycles in the graph representing the connections between registers, it should not be confused with program loops.

tic Systems transforms a sequential circuit to an elastic one by inserting Fork and Join operators. In short, Fork is used when the output of one stage forks to the inputs of multiple different stages, whereas Join is used when parallel data paths re-unite, and thus the inputs of a single stage come from multiple separate stages. The Join operator requires all the inputs to be valid to proceed. Typical examples are the inputs of an adder unit that need to be present at the same time for the operation to take place. When there is no dependency between the inputs of a block, a Merge operation is said to take place. Merge differs from Join since it is triggered when at least one of the inputs has valid data (and thus has "or-causality"), also, only data from one of the inputs is consumed at each cycle. Its dual, Branch, is an operator that propagates data to only one of multiple output paths, as opposed to sending data to all the output paths as Fork.

This behavior is found in many places in digital system designs. For example, a Floating Point Unit (FPU) has parallel paths for additions, multiplications, divisions, etc. Each path is triggered independently of the others. Another example is a network router, where independent packages come from different inputs, and propagates to a single output (based on the destination and routing policy).

To evaluate the performance of Fluid Pipelines, we propose a new methodology that can evaluate Fluid Pipelines and traditional Elastic Systems. We model Fluid Pipelines and Elastic systems using Coloured Petri Nets (CPN) [14] to determine the overall performance, which considers both throughput and frequency. This is then used to find the optimal pipeline configuration for a given design. This performs faster than RTL simulation for all possible pipeline configurations.

We observed that the presence of loops in the test cases for the traditional elastic approach dramatically deprecated the throughput, and consequently the system performance. Contrarily in Fluid Pipelines, the increase of circuit frequency resulted in a performance increase up to twice the frequency. The Fluid Pipelines area overhead is virtually zero compared to the traditional elastic approach. Our results show improvements of up to 176% in performance, and 5% lower power. With the use of CPN models, it is possible to explore the Pareto frontier and select other interesting design points, depending on the specific application, but also to have a more fair comparison between Elastic Systems.

The contributions of this paper are:

- Fluid Pipelines (Section 4), an Elastic System evolution that improves the Pareto frontier by avoiding typical throughput loss typical in traditional Elastic Systems.
- A new evaluation methodology (Section 5) using Coloured Petri Nets for Elastic Systems and Fluid Pipelines.
- An evaluation (Section 7) of a FPU to quantify the impact of traditional Elastic Systems and Fluid Pipelines.

2. RELATED WORK

Out-of-Order and Speculation have been evaluated in software Dataflow Networks [1]. Dataflow network concepts are used for task scheduling in parallel execution. The proposal relies on speculating what dependencies are real or false dependencies, thus need to trigger re-execution when a misspeculation happens. In our approach, we rely on the designer knowledge of the logic to avoid such scenario.

High Level Synthesis (HLS) [17] is a technique to synthesize the code written in programming languages, as opposed to Hardware Description Languages (HDLs). Using HLS, designers can put effort into functional design, while tools take care of pipelining in a process called scheduling. For instance, Chao et al. [6] propose a scheduling algorithm capable of retiming and reducing the pipeline depth in loops, thus it partially performs recycling. To some extent, HLS attacks the same problems as Fluid Pipelines, but Fluid Pipelines propose a lower level approach, giving the designer a more fine-grained control over the final design. HLS could leverage Fluid Pipelines under the hood to enable recycling in such loops, and in that regard, Fluid Pipelines and HLS can be viewed as orthogonal techniques. In fact, that could improve design time in HLS, because it could enable changes in the pipeline configuration without the need to regenerate RTL.

A formal approach to asynchronous logic has been proposed [10]. This approach expresses logic operators as a function of control signals (equivalent to those used in Elastic circuits). The paper provides good insights on logic optimizations considering 4 logic values (0, 1, NULL, busy), but there is considerable overhead since each logic gate depends on control signals. The formal approach proposed could be adapted to a coarser grain, and thus, could be used in the context of Elastic Systems.

In the context of Elastic Systems, different approaches have been proposed to mitigate the throughput loss due to the presence of sequential cycles. The use of an *Eager Fork* operator [8] lets one of the paths to start executing even if the parallel path still has its stop signal active. Nevertheless, semantics are not changed, and thus Eager Fork has to wait until the second path resets the stop bit. This becomes problematic when only one of the paths is used, and the other path takes a few cycles to reset the stop signal. In such cases, the backpressure will propagate to the stages that precede the Fork. Fluid Pipelines are designed to avoid such scenarios by not waiting for parallel paths when there are no dependencies between them.

Early Evaluation [2] has been proposed in the SELF framework [8]. It is a more sophisticated mechanism that determines which inputs in merging paths are actually needed (for instance, in a mux), and only waits for the inputs that are actually needed. The next token in the remaining inputs will be ignored to maintain correctness. This can be implemented as a counter that keeps the track of how many tokens need to be dropped, or as a back propagation of anti-tokens that annihilate the first token they encounter. Early Evaluation has been shown to improve the throughput in Elastic Systems in the average case. The main issue is that even if a sequential loop is not currently being used, it still limits the overall system throughput.

LI-BDNs [18] are a generalized form of Elastic Systems. They use First In First Out (FIFO) queues as the communication channel between modules. The main advantage is that, the use of FIFOs creates natural clock region boundaries. In LI-BDN systems, a stage-global enable signal is used for all

Clock Cycle	1	2	3	4	5	6	7	8
A	0	4				3		
B	1		2	3				
A+B		1		6				6

Figure 1: Elastic Systems functionality does not depend on the exact cycle events happen, but rather on their order.

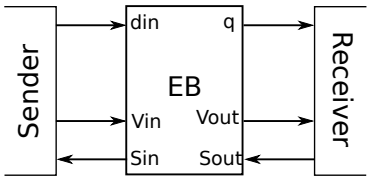


Figure 2: Elastic buffers are the basic construct blocks of Elastic Systems and can be viewed as queues with a limited size.

the state elements (registers). Therefore, a synchronous circuit can be transformed into *Patient Circuits* by "and"-ing enable signals in all the registers of each module with the global module enable. The global module enable signal indicates a *stall* event. The increased buffering capacity due to the presence of FIFOs improves the overall performance, at the cost of more area overhead. Fluid Pipelines performance is better than LI-BDNs and uses less area.

3. BACKGROUND

Elastic System is a system whose functionality only depends on the order of its inputs and not their exact arrival time [4]. An elastic execution example is shown in Figure 1, the arrival of a valid token is represented by a number in a given cell. When a result is produced, the token is consumed and cannot be used anymore. Empty cells in the table denote that no new data arrived in that cycle. Note that the latency between events is arbitrary.

Events or *tokens* are meaningful data flowing through a *channel*. A *channel* is a set of wires (i.e. bus) and its associated control signals: *Valid* (V) and *Stop* (S)[2], which determine three states: *transfer* ($V = 1, S = 0$), *idle* ($V = 0$) and *retry* ($V = 1, S = 1$) [8].

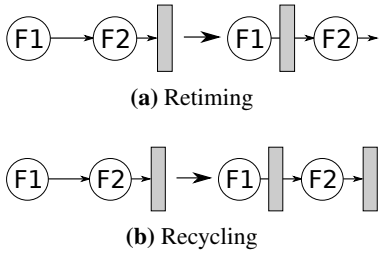
Elastic Buffers (EBs) are storage units that replace registers, they include handshake signals both in the input and output interface. Figure 2 shows the interface of an EB with input and output control signals.

3.1 ReCycling and Retiming in Elastic Systems

To improve the frequency or decrease the area of Elastic Systems, it is possible to move EBs across circuit blocks (Retiming) [2] (Figure 3a), or to insert additional stages in long wires [5] or in between combinational logic (ReCycling) [2] (Figure 3b). Retiming preserves the sequential behavior of the circuit [2] and thus it can be applied to any type of circuit mostly without penalties.

In the case of recycling, Júlvez *et al.* [15] observe that the throughput of a system is limited to the sequential loop with

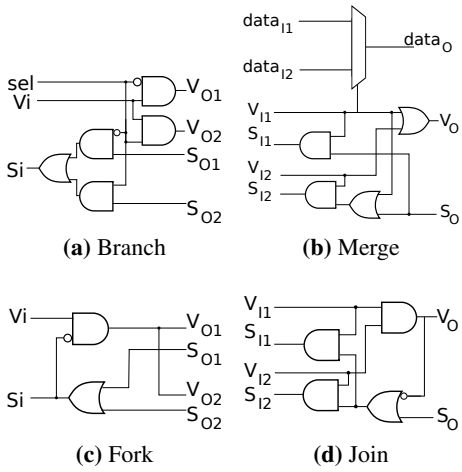
[2]Other equivalent naming conventions have been used, for instance, Elasticity has been expressed in terms of FIFO operation [18].



(a) Retiming

(b) Recycling

Figure 3: Retiming and Recycling are used to improve the circuit frequency, but recycling decrease the throughput of Elastic Systems when applied to sequential loops.



(a) Branch **(b)** Merge

(c) Fork **(d)** Join

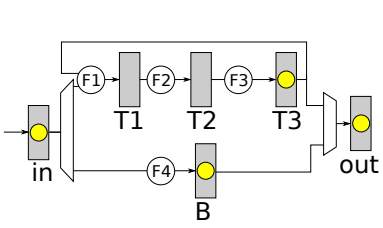
Figure 4: Fluid Pipelines uses different operators to indicate the intended functionality of a circuit and enable better design space exploration. Branch and Merge are used when the relative order of operations can be broken, while Forks and Joins enforce ordering. Note the difference in the handling of "valid" and "stop" signals.

lowest throughput, and that the throughput of a sequential loop can be calculated as the number of tokens in the loop divided by the number of EBs in the loop. The throughput of a cycle can increase with Early Evaluation depending on how often each event occurs [15], but due to back-pressure, there is still a limit on such mitigation. ReCycling is able to reduce cycle time, but may reduce throughput in the case of stage insertion in sequential loops [2, 5, 15].

4. Fluid Pipelines

Fluid Pipelines evolves the traditional Elastic Systems to allow breaking the relative completion order. To implement this behavior, Fluid Pipelines uses four types of operators: *Branch*, *Merge*, *Fork* and *Join* operators (Figure 4) [9]. In Figure 4a, *Selection* (sel) is a data-dependent selection signal that indicates to which output the data will propagate to. The operators can be easily extended to more than 2 inputs/outputs.

Branch is used when the datapath forks into multiple paths, but data should propagate to only one of them, this choice is data dependent and controlled by the selection signal. For instance, an operation in an FPU only needs to propagate to the appropriate functional unit, and the selection signal is encoded by the operation bits. The Merge operates as an arbiter: multiple senders compete for a single output.



(a) Loop Case

ID	Path
I1	Bottom
I2	Top
I3	Bottom
I4	Bottom
I5	Top
I6	Bottom
I7	Bottom

(b) Instructions

Figure 5: Toy case to illustrate the Elastic vs. Fluid approaches. Combinational logic is omitted, and Early Evaluation is assumed for elastic. Dots represent registers with a token.

The sender that wins the arbitration propagates data. In our FPU example, a Merge would be used at the output of the functional units when results from each unit are collected. Another way to think of the Merge is to notice that it fires when at least one of its inputs contains valid data. This is known as *disjoint or-causality* and introduces the *or-firing* rule to the context of Fluid Pipelines. Disjoint or-causality permits low latency arbitration [19]. For simplicity and without loss of generality, the proposed implementation in Figure 4b has fixed priority and could cause starvation, but it can be replaced with any of the existing elaborated arbitration schemes, such as Round-Robin.

Merge and Branch cannot be automatically inserted like Fork and Join, because they alter the relative order between events. As a result, the programmer is responsible for inserting them when needed. For example, in a complex Floating Point Unit, just one Merge and Branch pair is needed after the normalization and denormalization stages to indicate that the floating operations can complete out of order. On the other hand, the Fork and Join operators can be automatically inserted in a similar way as the insertions performed in traditional Elastic Systems. Merge and Branch can be performed with direct Verilog/VHDL instantiation or just code annotations. In this paper, we used direct Verilog annotations, and a more automatic solution is left for our future work.

As conceptual example of the power of Fluid Pipelines, let us analyze the sample execution in the example in Figure 5, where circles represent combinational logic, boxes represent EBs, and the dots inside boxes represent the presence of valid data (tokens). The paths are mutually exclusive (each operation either takes the top or the bottom path), and the mux near the output EB chooses the appropriate path. The instructions can take either the bottom path or the top path in Figure 5b. The execution traces for traditional Elastic Systems and Fluid Pipelines are shown in Table 1.

The execution order of Fluid Pipelines is altered (Table 1), note how in cycle 3, it is possible to move I3 to the bottom path, while the top path is still executing. This re-ordering is a result of the "or-firing" rule. This is fine, because that behavior was specified by the user, and not arbitrarily changed by the tool. In a processor core, the reordering buffer is already doing such function, while in network-on-chips, the re-ordering is usually not performed. Since this requirement is application specific, it is left out of this manuscript. We assume that, if needed, the re-ordering is performed in the design. In the case where order should be maintained, regular Fork and Join operators (defining "and-firing" rules) must be used, which cause the design to behave similar to a regular Elastic System.

Table 1: Sample trace for the toy case, Fluid Pipelines deliver higher throughput then Traditional Elastic.

	Elastic						Fluid					
Cycle	in	T1	T2	T3	B	out	in	T1	T2	T3	B	out
0	I1						I1					
1	I2				I1		I2				I1	
2	I3	I2				I1	I3	I2				I1
3	I3		I2				I4		I2		I3	
4	I3			I2			I5			I2	I4	I3
5	I4				I3	I2	I6	I5				I2
6	I5				I4	I3	I6		I5		I6	I4
7	I6	I5				I4	I7			I5	I7	I6
8	I6		I5									I5
9	I6			I5								I7
10	I7				I6	I5						
11					I7	I6						
12						I7						

4.1 Fluid Pipelines Deadlock Avoidance

In loop structures, deadlocks are a concern. Vijayaraghavan and Arvind [18] show that, in Elastic Systems, deadlocks come from extraneous dependencies, *i.e.*, one output of a module waits for an input that it does not depend upon to fire. Another issue is the creation of a token in the output before the consumption of one in the input. This is specially a problem in Fluid Pipelines since the designer has more freedom than in previous approaches. This is easily avoided by adhering to the following design practices:

- **No extraneous dependencies:** If an output o of a module does not depend on an input i of that module, then, o should be produced regardless of the existence of i. Also, the dependency list of o should be a subset of the inputs of the module.
- **Self cleaning:** A circuit is self cleaning if whenever it produced n tokens in its outputs, it has also consumed n tokens from its inputs.

These directives do not restrict which designs are possible, but rather how to implement each design. To make it clearer, let us consider the example in Figure 6. The synchronous module described in the figure has a pair of inputs (a and b) and outputs (c and d), the value of c depends on the values of a and b, while the value of d depends only on the value of b. Now, assume a designer wants to implement that module as a Fluid Pipelines circuit. There are multiple options for that.

The most straightforward implementation of the block follows the behavior described in Figure 7, which waits until all the inputs have valid data, and until all the outputs can accept new data to perform the operation. This can cause deadlocks depending on the context in which the block is used. For instance, in cases where the output d is connected as a feedback path to a, d will only produce output when both a and b are available. This is a violation to the no extraneous dependencies directive.

A simple solution to this case is the use of a Fork operator. The Fork operator isolates the handshake handling, and thus

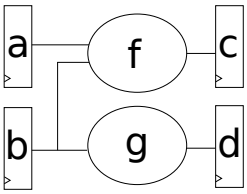


Figure 6: Fluid Pipelines design uses a few design practices to avoid deadlocks. Those are restriction on how to implement a given design and not on which designs can be implemented.

```
always @ (posedge clk)
  if (a.valid && b.valid)
    if(!c.stop && !d.stop)
      c <= f(a,b);
      d <= g(b);
      c.valid <= true;
      d.valid <= true;
      a.stop <= false;
      b.stop <= false;
```

Figure 7: A straightforward implementation of a circuit may be deadlock prone.

```
module fork(in, out1, out2)
  if(in.valid && !out1.stop && !out2.stop)
    out1 <= in
    out2 <= in
    in.stop <= false
    out1.valid <= true
    out2.valid <= true
endmodule

module f_and_g(a, b, c, d)
  fork(b, b1, b2);

  always @ (posedge clk)
    if (a.valid && b1.valid && !c.stop)
      c <= b1;
      c.valid <= true;
      b1.stop <= false;

    if (b2.valid && !d.stop)
      d <= b2;
      d.valid <= true;
      b2.stop <= false;
endmodule
```

Figure 8: Extraneous dependencies can be avoided by using fork to broadcast signals and isolating the false dependencies between stages.

avoids the deadlock situation by avoiding the unnecessary wait on a valid signal in a to propagate d. An implementation using Fork is shown in Figure 8.

The Self-Cleaning property is needed to avoid buffer overflow. Consider a circuit that produces n inputs per token consumed. Now, let the output of this circuit be connected back to its input. For a buffer with size m it is clear that after m/n cycles, the buffer will be full, and a deadlock situation will arise.

5. NEW EVALUATION METHODOLOGY

In order to find the optimal pipeline depth, a designer or an automated tool must estimate the throughput of a given pipeline configuration (*i.e.*, number and position of pipeline stages). To estimate the throughput of Fluid Pipelines designs, we propose the use of Coloured Petri Nets (CPN) [14]. CPNs are used for design specification and evaluation based on events/transitions and data/tokens which is what we need to perform Branch-like operations.

Petri Nets can be defined as a bipartite graph of *places* and *transitions*, connected by *arcs*. Places can contain *tokens* and tokens have data value attached to them. The attached value is the token *colour*. The state of the net (the *marking*) is defined by the number of tokens in each place. The initial marking is changed when transitions *fire*. When a transition fires, tokens are subtracted from its input places and are added to its output places according to *arc expressions*. There is a *capacity* associated with each place that represents the maximum number of tokens in that place, and prevents input transitions from firing (note that this is not part of the original formulation of PNs, but has been proposed as an extension). In CPNs, the tokens are typed (*colour*), and transitions are type-dependent.

DEFINITION 1. *A **Coloured-Petri Net** is a tuple* $CPN = \langle P,T,A,\Sigma,C,G,E,I,Cap\rangle$*:*

- P *is a finite set of* places*.*
- T *is a finite set of* transitions*, such that* $P \cap T = \varnothing$*.*
- $A \subseteq (T \times P) \cup (P \times T)$ *is a set of directed* arcs*. Let* $a.p$ *and* $a.t$ *denote the place and transition connected by a respectively.*
- Σ *is a finite set of non-empty* colour sets*.*
- $C : P \rightarrow \Sigma$ *is a* colour set function *which assigns a colour set to each function.*
- G *is a* guard function *that assigns to each transition* $t \in T$ *a guard function* $G(t) : (\varnothing \cup \Sigma)^{|\bullet t|} \rightarrow \{0,1\}$*, where* $\bullet t = \{p|(p,t) \in A\}$*.*
- E *is an* arc expression function *that assigns to each arc* $a \in A$ *an expression* $E(a)$*, such that the type of* $E(a)$ *should match* $C(a.p)$*.*
- I *is an* initialization function *that assigns to each place* $p \in P$ *an initialization expression* $I(p)$*,* $I(p)$ *must evaluate to* $C(p)$*.*
- $Cap : P \rightarrow \mathbb{I}$ *is a capacity function that attributes to each place a maximum capacity.*

Firing Semantics: Let M, a *marking* function, map each place $p \in P$ into a set of tokens $M(p) \in C(p)$. Let $G(t)(M)$ (resp. $E(a)(M)$) denote the evaluation of $G(t)$ (resp. $E(a)$) with the marking M. A transition t is enabled, and said to *fire* when $G(t)(M) = true$ (*i.e.*, the guard functions are satisfied), and $\forall a \in \{b|b = (p,t), p \in P, b \in A\}, E(a)(M) <= M(a.p)$ (*i.e.*, each input place contains the appropriated tokens), and $\forall p \in t\bullet, M(p) < Cap(p)$, where $t\bullet = \{p|(t,p) \in A\}$ (no output place is "full"). The firing updates the marking function to $M'(p) = (M(p)\ E(p,t) \cup E(t,p) \forall p \in P$.

Timing: In order to evaluate digital circuits, we need to account for timing, which is not included in CPN models. In regular CPNs, only one transaction fires at a given cycle. Without changing the underlying semantics of CPNs,

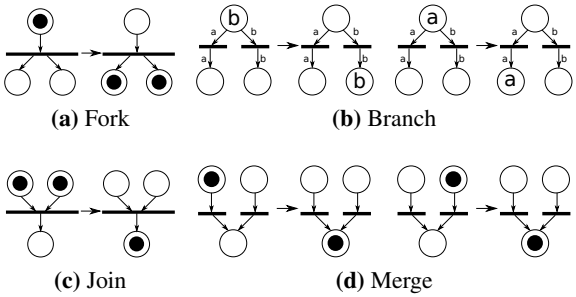


Figure 9: CPN models can be used to estimate the overall throughput of Fluid Pipelines and Elastic Systems.

we modify the model so that *every* transition that is enabled at the beginning of the cycle fires. This is a more accurate description of digital circuits and will help determine the number of clock cycles it takes to execute.

We add one restriction to this formulation. The cardinality of each expression must be 1; this means that for each arc, only one token can be consumed/generated. Also, note that guard functions can only depend on the incoming arcs to a transition. This complies with the constraints defined previously, and thus, avoids deadlocks. The restriction on the cardinality of expressions changes the formalism of CPNs, and a formal analysis of the impact of it is out of the scope of this paper and needs to be further explored in future work.

Figure 9 depicts how the Fluid Pipelines' operators are modeled as CPN transitions. Circles represent places, bars represent transitions, and dots represent tokens in transitions that are not colour dependent while letters represent coloured tokens. In case of Merge operators, the semantic does not define which transition has priority, and thus, conceptually they can occur at the same time, which is compatible with the theoretical formulation of Fluid Pipelines. While places correspond to elastic buffers, transitions do not have a direct translation from the circuit model. However, a mapping can be defined between the guard functions and the handshaking logic.

6. EVALUATION SETUP

To evaluate Fluid Pipelines, we consider a fully compliant IEEE-754 in-house FP Unit, designed both as synchronous (for previous approaches), and annotated with Fluid Pipelines operators. It has a simple structure, but is sufficient to demonstrate how Fluid Pipelines can yield better performance compared with previous Elastic Systems.

A functional block diagram of the FPU unit is presented in Figure 10a. This is a simple structure with multiple parallel paths and simple loops. The CPN model used for the performance evaluation is shown in Figure 10b, considering Fluid Pipelines. In this case, the Merge and Branch operators are used. Note how the division and square root modules use the Branch to choose between the loop when the operation is computing or sending the result to the queue when done. Both division and square root take 64 cycles to complete. For regular elastic, the Fork and Join operators are used instead.

Fluid Pipelines are compared against SELF [2] and LI-BDNs [18]. To implement Elastic Systems, we use an EB implementation with storage capacity of 2. For LI-BDNs, we use queues of size 8. In the SELF implementation adding pipeline stages to all the paths that are parallel to the critical path will yield best performance and that is the performance we have considered in our evaluation.

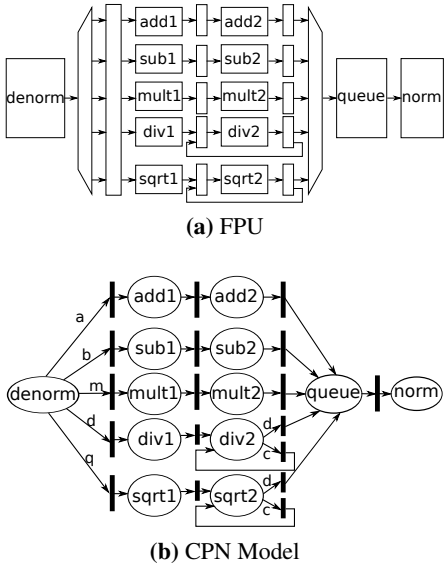


Figure 10: CPN modeling can be used to evaluate system performance.

6.1 ReCycling

Our evaluation considers the addition of extra pipeline stages to each design. Pipeline stages are always added to the blocks with the worst delay. We assumed perfect recycling/retiming (perfect balancing of delays). Although this is usually not possible, this approximation is good enough[3]. It is only necessary to ensure that, after the insertion of a pipeline stage, the two resulting stages have a delay smaller than the second most critical path before insertion. We add 2FO4 delay per added stage to account for the register overhead.

The performance metric used is $throughput \times frequency$ (equivalent to IPS), since ReCycling changes both IPC and timing, thus those are combined. Also, it has been shown that unless power is considered, the ideal pipeline for a design is extremely deep [12, 13]. Thus, we consider energy-delay (ED). We observe that the logic energy consumption (both dynamic and leakage) remains roughly constant. However, the dynamic clock energy consumption increases linearly with both frequency and number of registers, and the leakage clock energy increases linearly with the number of registers.

[3]The requirement is that the delay on each one of stages after the split will be smaller than the delay of the second most critical path. For instance, say the critical path in stage 1 has a delay of 1ns, whereas the critical path in stage 2 has a delay of 0.7ns. We want to add a new register to stage 1 such that the delays of the newly generated stages are less than 0.7ns, but perfect balance between the stages is unnecessary.

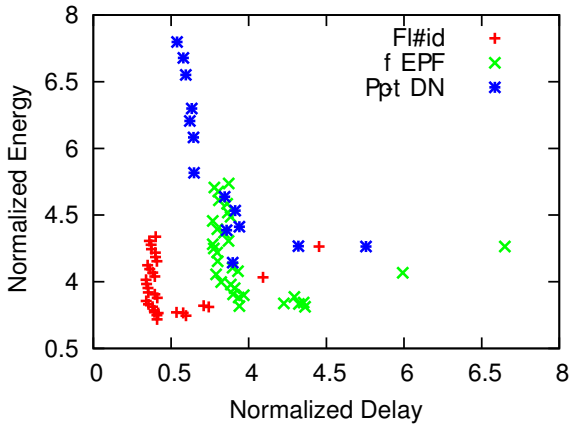


Figure 11: Fluid Pipelines are able to push the Pareto frontier for the FPU by improving both performance and energy.

7. EVALUATION

We start our evaluation by showing the design space exploration of the different approaches. In particular, we show that Fluid Pipelines are able to push the pareto frontier towards better performance and energy efficiency. Then, we proceed to explain the detailed results, such as the maximum frequency, throughput and energy-delay for different pipeline configurations for the FPU.

Fluid Pipelines push the design space towards more energy efficiency and better performance. This is mostly accomplished by avoiding false dependencies between concurrent paths. For most of the design points, Fluid Pipelines were able to deliver both better performance and energy. When comparing SELF with LI-BDNs, the former was able to obtain better performance, but at the cost of energy (and area, which was not evaluated here).

The Pareto frontier (E vs D) is shown in Figure 11). LI-BDNs result in increased energy consumption due to the increased storage, but are able to improve the performance, when compared to SELF. Fluid Pipelines present the best performance and energy out of the three schemes, since it does not requires extra storage. When compared to SELF, Fluid Pipelines were able to improve the best performance by 120%, with 21% less energy, or improve the best energy by 12% with 230% improvement in performance. When compared to LI-BDNs, Fluid Pipelines improved the best performance by 33%, using 83% less energy, or improve the best energy by 38% with 118% better performance.

The results show that Fluid Pipelines designs are both more energy efficient and higher performance than designs possible with current Elastic Systems.

7.1 Detailed Results

The maximum throughput for each of the models is summarized in Table 2. This is calculated by the use of a synthetic workload that only considers the best path (addition, subtraction and multiplier in this case). The initial pipeline depth in the design is 6, thus, there is no data for any configuration with less stages than 6. Fluid Pipelines are able to deliver constant throughput regardless of the number of pipelines. The throughput of SELF decreases when there

Table 2: Maximum FPU throughput for the different evaluated models. Fluid Pipelines deliver constant maximum throughput, regardless of the number of pipeline stages.

Pipeline stages	Fluid Pipelines	SELF	LI-BDN
6	1	1	1
7	1	1	1
8	1	1	1
9	1	0.67	1
10	1	0.50	1
11	1	0.40	0.83
12	1	0 37	0.74
13	1	0.33	0.67

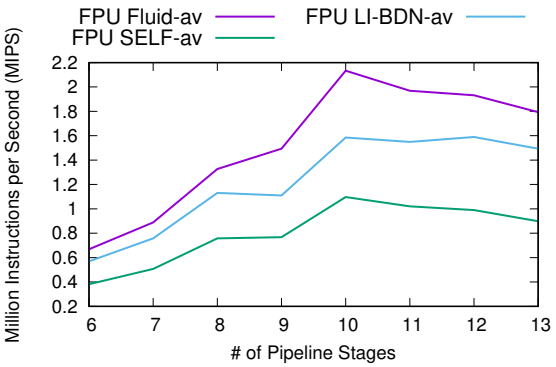


Figure 12: In Fluid Pipelines, circuits can be recycled with higher throughput then possible with Elastic Systems, and thus for better system performance.

is additional pipeline stages in the sequential loops. In the case of LI-BDNs, the extra buffering helps maintaining the throughput even after the insertion of a few stages in the loops, but after a certain number of insertions, there is backpressure due to the dependencies.

Maximum throughput is not a realistic metric. Thus, we calculate the average throughput over a million random instructions, we then report the effective frequency, shown in Figure 12. Note that effective frequency does not necessarily increase with the number of pipeline stages. This is due the fact that despite the frequency gain with the new pipeline stage, the reduced throughput reverts the gains and reduces the overall performance. Since in the average case the loop path is used, there is a reduction in the gap between Fluid Pipelines and the other models. The same fact also causes reduction in the throughput of both SELF and LI-BDN. Despite the reduction in the gap, Fluid Pipelines are still able to deliver a considerably improved performance compared to SELF (120%), and slightly improved performance compared to LI-BDN (40%), but using less resources.

To take into account the extra stages added in the case of SELF, we use energy-delay product (ED), that considers both performance and energy. These numbers are reported in Figure 13. The energy overhead caused by the extra storage in LI-BDNs reverses the advantages when compared to SELF. When comparing Fluid Pipelines with SELF, Fluid Pipelines are able to improve the best ED point by improving performance by 176%, with 5% better energy. Alternatively, Fluid Pipelines are able to deliver 120% better top performance (with 21% less energy). When we compare Fluid Pipelines with LI-BDNs, Fluid Pipelines improve the best

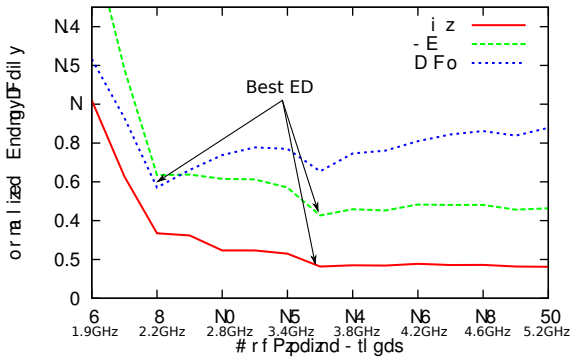


Figure 13: Fluid Pipelines are able to improve the best ED point of the FPU, pushing the depth of the pipeline.

ED point by improving both performance (by 163%) and energy (by 25%). Fluid Pipelines are also able to deliver 33% better top performance (with 83% less energy).

8. CONCLUSION

A new abstraction for Elastic System, Fluid Pipelines, was proposed. By using Fluid Pipelines, the designer has the opportunity to extract out-of-order execution from the circuit, whenever possible, and thus boost the design performance. Fluid Pipelines push the Pareto frontier of designs, by improving both performance and energy. In our experiments, over SELF, Fluid Pipelines improve the optimal energy-delay configuration of a FPU design by improving energy by 5% and performance by 176%. Alternatively, Fluid Pipelines were able to reduce the lowest energy point by 12%, with 120% better performance, or improve the top performance by 33% (but with 83% less energy).

We present a modeling framework for the proposed abstraction, using Petri Nets, which allows us to evaluate the system run-time behavior, and is a powerful tool for early design space exploration of Fluid Pipelines. This framework is used to evaluate Fluid Pipelines against other Elastic System approaches, showing an improvement in the overall throughput of the systems. We argue for the use of this simple tool when evaluating simple event-driven systems.

Fluid Pipelines open many research opportunities in EDA and architecture like automatic repipelining with larger codes than the evaluated FPU. Fluid Pipelines can also benefit from new DSLs for hardware description, and further work to include RTL and gate level evaluation of the proposed model and transformations which in turn leads to a better understanding of the overheads of the new technique, as well as better understanding of the design trade-offs in terms of area and power.

Acknowledgments

We like to thank the reviewers for their feedback on the paper. This work was supported in part by the National Science Foundation under grants CNS-1059442-003, CNS-1318943-001, CCF-1337278, and CCF-1514284. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

9. REFERENCES

[1] D. Baudisch and K. Schneider. Evaluation of speculation in out-of-order execution of synchronous dataflow networks. *Int. J. Parallel Program.*, 43(1):86–129, Feb. 2015.

[2] D. Bufistov, J. Cortadella, M. Galceran-Oms, J. Julvez, and M. Kishinevsky. Retiming and recycling for elastic systems with early evaluation. In *46th Design Automation Conference*, pages 288–291, 2009.

[3] B. Cao, K. Ross, M. Kim, and S. Edwards. Implementing Latency-Insensitive Dataflow Blocks. In *Proceedings of the 13th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign*, MEMOCODE '15, Jul. 2015.

[4] L. P. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli. A Methodology for Correct-by-construction Latency-insensitive Design. In *Computer-Aided Design. Int'l Conf. on*, pages 309 –315, 1999.

[5] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Proceedings of the 37th Design Automation Conference*, pages 361–367, New York, NY, USA, 2000. ACM.

[6] L.-F. Chao, A. LaPaugh, and E.-M. Sha. Rotation scheduling: a loop pipelining algorithm. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(3):229–239, Mar 1997.

[7] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky. Elastic Systems. In *Proceedings of the 8th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign*, MEMOCODE '10, pages 149–158, Jul. 2010.

[8] J. Cortadella, M. Kishinevsky, and B. Grundmann. SELF: Specification and Design of Synchronous Elastic Circuits. In *Proceedings of the ACM/IEEE International Workshop on Timing Issues*, TAU 06, 2006.

[9] G. Dimitrakopoulos, I. Seitanidis, A. Psarras, K. Tsiouris, P. M. Mattheakis, and J. Cortadella. Hardware primitives for the synthesis of multithreaded elastic systems. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2014.

[10] K. Fant and S. Brandt. Null convention logictm: a complete and consistent logic for asynchronous digital circuit synthesis. In *Application Specific Systems, Architectures and Processors, 1996. ASAP 96. Proceedings of International Conference on*, pages 261–273, Aug 1996.

[11] S. . FGPA and S. D. S. Innovations. Altera inc.

[12] A. Hartstein and T. R. Puzak. The optimum pipeline depth for a microprocessor. In *Proc. of the 29th Int'l Symp. on Computer Architecture*, pages 7–13, Washington, DC, 2002. IEEE Computer Society.

[13] M. Hrishikesh, D. Burger, N. P. Jouppi, K. I. Farkas, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays. *Proceedings of the 29th. Int'l Symp. on Computer Architecture*, 2002.

[14] K. Jensen and L. M. Kristensen. *Coloured Petri Nets Modelling and Validation of Concurrent Systems*. Springer-Verlag Berlin Heidelberg, 2009.

[15] J. Julvez, J. Cortadella, and M. Kishinevsky. Performance analysis of concurrent systems with early evaluation. In *Computer-Aided Design. Int'l Conf. on*, pages 448–455, Nov 2006.

[16] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.

[17] M. Oskin, F. Chong, and M. Farrens. HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs. In *International Symposium on Computer Architecture*, pages 71–82, Vancouver, Canada, Jun. 2000.

[18] M. Vijayaraghavan and A. Arvind. Bounded Dataflow Networks and Latency-Insensitive Circuits. In *Proceedings of the 7th IEEE/ACM Int'l Conf. on Formal Methods and Models for Codesign*, pages 171–180, Piscataway, NJ, USA, 2009. IEEE Press.

[19] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with or causality. *Formal Methods in System Design*, 9(3):189–233, 1996.

Amorphous Data-parallelism

Keshav Pingali[1]

[1]UT Austin, USA

Abstract. *Although data-parallelism is ubiquitous in high-performance computing (HPC) algorithms, many algorithms in other areas such as graph analytics, machine learning, and VLSI placement and routing exhibit a more complex kind of parallelism that called "amorphous data-parallelism."*

In this talk, I describe a simple programming model called the operator formulation of algorithms for specifying amorphous data-parallelism, and a system called Galois for exploiting amorphous data-parallelism. Experimental results show that this is a practical approach for exploiting parallelism in complex, irregular applications that are beyond the capabilities of current commercial systems.

Keshav Pingali is a Professor in the Department of Computer Science at the University of Texas at Austin, and he holds the W.A."Tex" Moncrief Chair of Computing in the Institute for Computational Engineering and Sciences (ICES) at UT Austin. Pingali is a Fellow of the IEEE, ACM and AAAS. He was the co-Editor-in-chief of the ACM Transactions on Programming Languages and Systems, and currently serves on the editorial boards of the ACM Transactions on Parallel Computing, the International Journal of Parallel Programming and Distributed Computing. He has also served on the NSF CISE Advisory Committee (2009-2012).

LUT Mapping and Optimization for Majority-Inverter Graphs

Winston Haaswijk*, Mathias Soeken*, Luca Amarù†, Pierre-Emmanuel Gaillardon‡, Giovanni De Micheli*
*Integrated Systems Laboratory, EPFL, Lausanne, VD, Switzerland
†Design Group, Synopsys Inc., Mountain View, CA, USA
‡Laboratory for NanoIntegrated Systems, The University of Utah, Salt Lake City, UT, USA

***Abstract*—A *Majority-Inverter Graph* (MIG) is a directed acyclic graph in which every vertex represents a three-input majority operation and edges may be complemented to indicate operand inversion. MIGs have algebraic and Boolean properties that enable effIcient logic optimization. They have been shown to obtain superior synthesis results as compared to state-of-the-art *And-Inverter Graph* (AIG) based algorithms. In this paper, we extend MIGs to *Functionally Reduced* MIGs (FRMIGs), analogous to the extension of AIGs to *Functionally Reduced* AIGs (FRAIGs). This enables the use of MIGs in a *lossless synthesis* design ow. We present an FRMIG based technology mapper for *lookup tables* (LUTs). Any MIG may be mapped to a k-LUT network. Using *exact synthesis* we may decompose the k-LUT network back into an equivalent MIG. We show how LUT mapping and exact k-LUT decomposition can be used to create an MIG optimization method. Finally, we present the results of applying our new optimization method and LUT mapper to both logic optimization and technology mapping.**

I. Introduction

For several decades, the performance of digital circuits has been largely dependent on the effectiveness of the tools developed by the logic synthesis community. Advancements in logic representation and optimization as well as technology mapping have enabled continued improvement to the capabilities of modern chips.

Within logic representation and optimization there has been a trend from heterogeneous logic representations towards simpler, homogeneous representations. Traditionally, nodes in multi-level logic networks represented complex Boolean functions on varying numbers of inputs [1]–[3]. While this is a rich representation that allows for powerful optimization techniques, homogeneous networks such as *And-Inverter Graphs* (AIGs) have permitted more effIcient implementations. These implementations use less memory and enable better runtimes while maintaining or even improving synthesis results [4], [5]. As such, homogeneous networks appear to be the more scalable approach.

The recently introduced *Majority-Inverter Graphs* (MIGs) are another example of homogeneous networks [6]. A generalization of AIGs, MIGs are directed acyclic graphs consisting of three-input majority nodes with optionally complemented edges. MIGs include AIGs but also have other desirable algebraic and Boolean properties. Algorithms that exploit these properties have been shown to obtain superior results in both logic optimization and technology mapping, especially with respect to depth and delay reduction [6], [7]. Novel work on *exact synthesis* has shown to be another avenue for MIG optimization, enabling MIG size reduction as well as improvements in both depth and area in FPGA technology mapping [8].

Technology mapping is the problem of covering a logic network with a set of primitives. In some literature it is also referred to as *cell-library binding* [3]. Typically, technology mapping occurs after the application of technology independent optimizations, although there are systems that perform both operations simultaneously [9]. The state of the art in technology mapping uses cut enumeration techniques to flnd suitable covers [10]–[12]. We distinguish between two different kinds of technology mapping. In the former, we are given a library of logic primitives such as *standard cells* or *gate arrays*. The available primitives depend on the particular technology. In the latter, we are mapping to lookup table-based FPGAs. Lookup tables are powerful primitives that can implement any function on k variables. Typically, we refer to lookup tables on k variables as k-LUTs. The k-LUT FPGA mapping problem is simpler than the more general case of cell library binding because of the homogeneous nature of the logic primitives. In the remainder, we focus on k-LUT technology mapping, which is applicable to both FPGA technology mapping and resynthesis of general logic networks. We also refer to this process simply as LUT mapping.

Our goal is to improve techniques for MIG logic optimization and LUT mapping. To achieve this goal we present a number of contributions:

1) We show how MIGs can be extended to the *Functionally Reduced* MIGs (FRMIGs). This enables a *lossless synthesis* fiow for MIGs.
2) We present a cut-based LUT mapper for FRMIGs.
3) We present an MIG optimization method that uses the FRMIG mapper and exact k-LUT decomposition.
4) Combining our new MIG optimization method and our FRMIG mapper we show improvements to LUT mapping results.

The results of our experiments on logic optimization and LUT mapping include:

- Improvements to MIG logic optimization. We achieve a reduction in {size, depth} of {10%, 26%}, as compared to previous MIG optimization methods.
- Signiflcant improvements after LUT mapping for several

benchmarks, reducing {size, area} by up to {13%, 56%} as compared to the best known results.
- Signiflcant improvements in depth after LUT mapping for several benchmarks where we do not obtain size reductions. These results allow us to provide an area/depth trade-off.

The remainder of this paper is structured as follows. In Section II we present the background on MIGs, LUT mapping, structural and functional equivalence, and exact synthesis. We show how MIGs can be extended to FRMIGs in Section III and present our FRMIG LUT mapper in Section IV. We describe our new optimization method in Section V. Our MIG optimization and LUT mapping experiments can be found in Section VI.

II. Background

In order to improve on the state of the art in MIG optimization and LUT mapping we extend a number of existing techniques. Here we briefiy present the necessary background in MIG synthesis and optimization, structural and functional equivalence, technology mapping, and exact synthesis.

A. Majority-Inverter Graphs

A Majority-Inverter Graph is a data structure for the representation and optimization of Boolean functions. It is a directed acyclic graph in which every vertex corresponds to a 3-input majority operator. We denote the 3-input majority function of variables a, b, and c as $\langle abc\rangle$, following the convention of [13]. In general the n-input majority function is equal to the value expressed by more than half of its inputs. Therefore, we can express the three-input $\langle abc\rangle$ in disjunctive normal form as:

$$\langle abc\rangle = ab \rightarrow ac \rightarrow bc.$$

Note that by setting any operand to either 0 or 1, the above equation simplifles to a Boolean *and* or *or* operation, respectively. Hence, MIGs are a generalization of *And-Or-Inverter Graphs* (AOIGs) and thus also of AIGs.

More formally, we can deflne an MIG over a set $X = \{x_1, \ldots, x_n\}$ of *primary inputs* as a directed acyclic graph $M = (V, E, Y)$ where

- $V = X \forall N \forall \{1\}$ a flnite set of nodes, where $N = \{o_1, \ldots, o_m\}$ represents the nodes corresponding to the majority operations, and we also include the constant 1 node;
- $E \exists V \times N \times \mathbb{B}$ a flnite multiset of edges, where the flrst element in the tuple is a source node, the second is a target node, and the third is a polarity bit representing whether or not the edge is complemented; and
- $Y \exists V \times \mathbb{B}$ a flnite multiset of outputs.

Each node $n \cup N$ must have three predecessors, reflecting its correspondence to a three-input majority operator. In the remainder of this paper, if $n \cup N$, we write n_1, n_2, and n_3 to denote these inputs. We write p_1, p_2, and p_3 to denote the polarity bits indicating complementation of the inputs.

The functional semantics of an MIG can be described in terms of an interpretation function f that maps MIG nodes to a Boolean function. For convenience we deflne $f^0(v) = v$ and $f^1(v) = \overline{v}$. We deflne f as

$$\begin{aligned} f(1) &= \top \\ f(x) &= x && \text{for } x \cup X \\ f(n) &= \langle f^{p_1}(n_1) f^{p_2}(n_2) f^{p_3}(n_3)\rangle && \text{for } n \cup N \\ f(y) &= f^p(n) && \text{for } y = (n, p) \cup Y \end{aligned}$$

A *sound* and *complete* Boolean algebra based on the 3-input majority operation is presented in [6], and its extension to n-input majority in [14]. There is a direct correspondence between the MIG structure and the axioms of these algebras. This means that we can use the algebraic rules directly to manipulate MIGs in a sound and complete way. This is the basis for the algebraic MIG optimizations shown in [6]. These algebraic optimizations are shown to lead to an average reduction of 22%, 14%, 11% in delay, area, power respectively, as compared to state-of-the-art academic and commercial synthesis tools.

Boolean MIG optimizations are presented in [7]. The majority operator possesses an inherent error-correcting mechanism. This enables us to insert *orthogonal* errors into an MIG structure that enable signiflcant optimization opportunities. Boolean MIG optimization reduces the average delay/area/power by 15.07%, 4.93%, 1.93% respectively, over 27 academic and industrial benchmarks.

This section serves as a brief overview of the MIG structure and capabilities. For more complete descriptions the interested reader is referred to [6], [8], [14].

B. FPGA Technology Mapping

Generally speaking, an FPGA is an integrated circuit that consists of an array of programmable logic blocks and interconnects. Often these logic blocks are lookup tables that can implement arbitrary logic functions. Typically an FPGA also includes a number of non-programmable *hard blocks* that implement specifllc functionality, such as adders, multipliers, embedded memories, and even microprocessors. As we are reaching the limits of modern CMOS scaling, FPGA are seen as an approach to enable more computational capabilities in an energy effllcient way [15], [16]. In order to make use of these advantages we require good logic optimization and technology mapping algorithms for FPGAs.

LUT mapping is the special case in which we cover a logic network with k-bounded lookup tables (k-LUTs). A k-LUT is a lookup table with k inputs; a powerful logic primitive that can represent any function on k variables.

A breakthrough in delay-oriented LUT mapping came with the introduction of the *FlowMap* algorithm [17]. FlowMap was the flrst algorithm to show how k-feasible cuts can be used to obtain a minimum-depth k-LUT cover. Several improvements of FlowMap have since been made. Some of these improvements include generalizing the algorithm to a more general cut enumeration basis, improving the runtime

and memory requirements, as well as improving different aspects of the flnal cover such as area reduction [11], [18]–[21].

More recently, developments in technology mapping have aimed at reducing *structural bias* [10], [22]. Structural bias is a problem of cut-based technology mapping algorithms. It is caused by the original decomposition of the network, which is not unique. One approach to mitigate this problem has been to use the concepts of *choice nodes* and *functional reduction* to merge structurally different equivalent networks into a single representation [9], [23]. *Functionally Reduced AIGs* (FRAIGs) are an example of such a representation [24]. The concept of functional reduction naturally leads to a so-called *lossless synthesis* fiow. In lossless synthesis, multiple different logic networks may be merged into a single graph that can be used for technology mapping. The different networks correspond to points in the optimization space that are no longer lost in a lossless synthesis fiow [12], [25].

The state-of-the-art in LUT mapping algorithms is based on cut enumeration. However, a drawback of cut-based algorithms is that they are susceptible the problem of structural bias [22]. Given a logic network, they flnd a cover based on the feasible cuts in that network. However, the given logic network is typically just one of many possible representations. Other networks (with different structure) may lead to different covers. Hence, the output of the algorithm is inherently biased by the structure of the input network. This is a problem because some covers may be "better" than other in terms of size or depth.

The notion of *lossless synthesis* was proposed in [22] as a way to mitigate the problem of structural bias. In a traditional optimization and mapping fiow, an input network is optimized by applying one or more optimization scripts to it. Afterwards the optimized network is then mapped. This optimization process loses information: it may be that during optimization an intermediate networks is better in some respect than the flnal network. In lossless synthesis, multiple equivalent (intermediate) networks are stored, so that the desirable parts of the networks are never lost.

To efflciently store the different networks generated by lossless synthesis, they may be combined into a single network graph with *choice nodes* [23]. Intuitively, a choice node may be understood as a vertex in a logic network that encodes different implementations of the same function (up to complementation). Equivalent points in different networks can be found through a combination of simulation and combinational equivalence checking (SAT). This notion of flnding equivalent points in logic networks leads to the notion of *functional reduction*. *Functionally Reduced* AIGs (FRAIGs) were flrst introduced in [24]. In FRAIGs, equivalent AND nodes are merged into choice nodes that represent different implementations of the same logic function. The current state of the art in FPGA technology mapping combines cut enumeration with the FRAIGing of differently optimized logic networks [10], [12], [24], [25].

C. Exact Synthesis

In exact synthesis, we are interested in flnding the optimum representation for a Boolean function. Of course, optimality depends on the optimization objective (such as size or depth) as well as the chosen representation form. In this paper we use exact synthesis to refer to the optimum MIG representation.

Exact synthesis can be formulated as a decision problem and may be solved by an SMT solver. SMT solvers are constraint solvers for decision problems that can be formulated in flrst-order logic. Typically they work on formulas within some background theory, such as a theory of bit vectors. To see how we can use an SMT solver in exact synthesis, suppose that the solver has some method of checking if a given function on n variables can be represented by an MIG with k nodes. In order to flnd the smallest possible MIG for that function, we start checking with $k = 0$, and we keep increasing k until we have found a k for which an MIG exists that can represent the function. Similar algorithms can be used to flnd the minimum depth or to optimize other criteria. The checking method can be implemented by formalizing the exact synthesis criteria (e.g. size or depth) as an SMT theory. An in-depth, formal example of how this may be done can be found in [8]. Thus, we can use an SMT solver for exact synthesis.

The computational complexity of exact synthesis as described here prevents it from being used to synthesize arbitrarily large function. SMT is a generalization of SAT and is therefore NP-hard. As such, the time complexity of this approach grows at least exponentially with n. However, for small enough n ($n \wedge 4$) it has been shown to work [8]. In Section V we show how we can use exact synthesis to decompose k-LUT networks (for small enough k) into locally optimal sub-MIGs. This is the basis for our optimization algorithm.

D. Structural and Functional Equivalence

MIGs are not canonical representation forms: a Boolean function may have structurally different but *functionally equivalent* MIG representations. See for example Fig. 1 in which two different MIGs represent the function $f = \overline{abc}$. Other homogeneous networks, such as AIGs, have similar limitations with different structures representing equivalent functions.

Despite the non-canonical nature of MIGs, there are still methods to remove redundancies and simplify MIG structures. One such method is *structural hashing*. With structural hashing, before adding a node to an MIG, we check if a node with the same fan-in already exists. This can be done efflciently with a hash table. Typically there is an order on the fan-ins of the node so that swapping them does not create a different node. Thus, structural hashing merges nodes that are *structurally equivalent*. Fig. 2 shows how structural hashing can be used to reduce the node count and remove redundant nodes.

In order to reduce structural bias we would like to be able to flnd and merge equivalent nodes, even if they are not structurally equivalent. Nodes with different fan-ins may still be *functionally equivalent*. The process of merging functionally

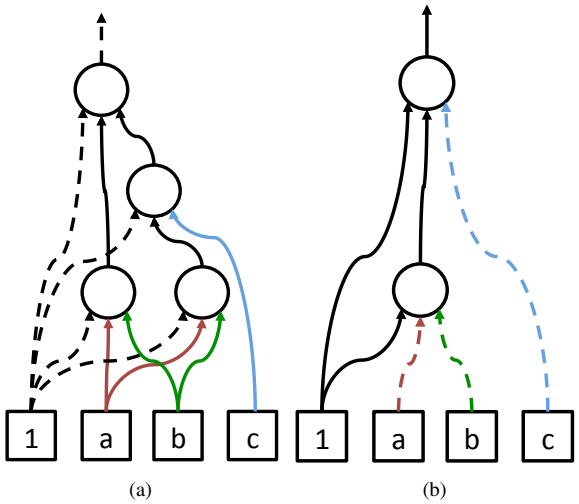


Fig. 1. An example of two structurally different, but functionally equivalent MIGs for the function $f = \overline{abc}$. Complementation is indicated by dashed edges.

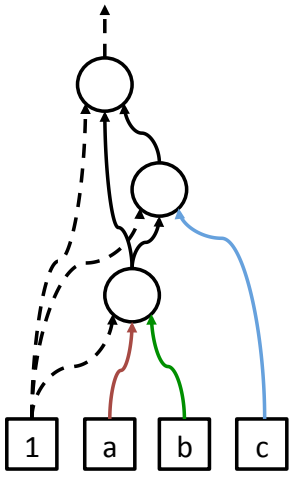


Fig. 2. Structural hashing can be used on the MIG in Fig 1 (a) to remove one redundant node.

equivalent nodes is commonly referred to as functional reduction. Functional reduction is typically achieved through BDD or SAT sweeping [26], but can also be achieved incrementally during construction [24].

Storing functionally equivalent nodes efflciently can be achieved by *choice nodes*. Choice nodes were developed in the 90s as a way to efflciently encode sets of logic networks to be considered in technology mapping [9], [23]. Intuitively, one can think of a choice node as representing an equivalence class of functions (up to complementation). Fig. 1 shows two structurally different MIGs that are equivalent up to complementation. Using only structural hashing we would not be able to flnd this equivalence. However, through a combination of simulation and SAT sweeping we can efflciently flnd these equivalence points. We refer to an MIG in which functionally equivalent nodes have been merged into choice nodes as a *Functionally Reduced MIG* (FRMIG). An FRMIG reduces structural bias by containing different equivalent structures at once.

III. Functionally Reduced MIGs

In Section II-D we reviewed the concepts of structural and functional equivalence. Applying these concepts to MIGs leads to the concept of the *Functionally Reduced* MIG (FRMIG). Here, we deflne FRMIGs as well as a procedure to construct FRMIGs from MIGs.

Given an MIG $M = (V, E, Y)$, we deflne a *functional equivalence* relation on the nodes in V. A relation $\simeq$ is a functional equivalence relation if for all $v_1, v_2 \cup V$

$$v_1 \simeq v_2 \equiv (f(v_1) = f(v_2)) \rightarrow (f(v_1) = \neg f(v_2))$$

Let $V_{\simeq}$ denote the set of equivalence classes of V under $\simeq$. We say that a subset $\mathcal{V}$ is a *set of functional representatives* iff it contains exactly one node v_C from each equivalence class $C \cup V_{\simeq}$. We say that v_C is the *functional representative* of C.

Let $\widehat{M} = (V, E, Y)$ be an MIG, $\simeq$ a functional equivalence relation, and $\mathcal{V}$ a set of functional representatives. Then, $\widehat{M}$ is a *Functionally Reduced* MIG w.r.t. $\simeq$ and $\mathcal{V}$, iff $(v, n, b) \cup E \equiv v \cup \mathcal{V}$. In other words, only equivalence class representatives have outgoing edges. This means that for any $n \cup N$ its predecessors n_1, n_2, n_3 are in $\mathcal{V}$.

We sometimes loosely refer to equivalence class representatives as *choice nodes*. This follows the intuition that each choice node represents a set of alternative implementations for an equivalence class of functions.

Note that according to this deflnition, any MIG is trivially an FRMIG. Suppose we have an arbitrary MIG $\widehat{M}_T = (V, E, Y)$. Let $\simeq_T$ be the relation $n_1 \simeq_T n_2$ iff $n_1 = n_2$. One can easily verify that this is a functional equivalence relation. Let $\mathcal{V}_T = V$. Then $\widehat{M}_T = (V, E, Y)$ is an FRMIG under $\simeq_T$ and $\mathcal{V}_T$. However, this trivial deflnition is not very interesting. In the remainder of this paper we assume that for every FRMIG the functional equivalence relation is deflned as: *equality up to complementation*.

Fig. 3 shows how the functionally equivalent MIGs from Fig. 1 can be merged in the same FRMIG structure. Note how only one of the equivalent nodes is chosen as the equivalence class representative and has an outgoing edge.

We now sketch a procedure for constructing an FRMIG from an MIG. The procedure starts by simulating the MIG on a number of random bit vectors. We then traverse the MIG in topological order, creating new nodes only after structural hashing. When a new node is created, we check its simulation vector to see if there are any potentially equivalent nodes. If an equivalence between two nodes is found, the node which appears earlier in the topological order is chosen to be the equivalence class representative. Note that this means that primary inputs are always chosen to be the representatives of their equivalence classes. One can intuitively think of the

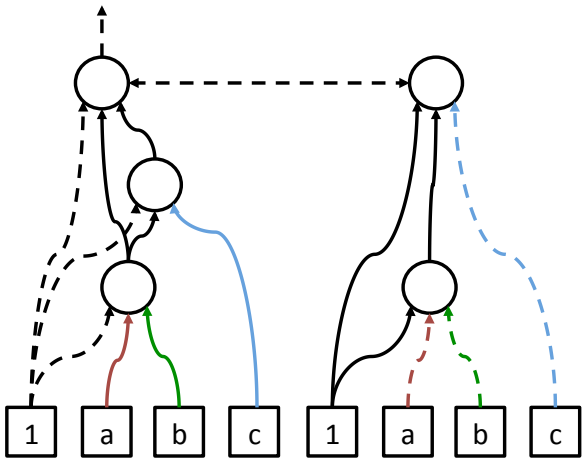


Fig. 3. This figure demonstrates the use of *choice nodes* in FRMIGs. Two equivalent (up to complementation) MIG structures are represented by one equivalence class representative. This representative is implemented as a choice node with one outgoing edge. The horizontal dashed line indicates equivalence between the MIGs. It is dashed because both MIG structures represent the complement of the other.

equivalence class representative as a choice node containing MIG structures that are equivalent up to complementation.

Equivalence checking of nodes is done using the SMT solver Z3 [27]. As the MIG is traversed we incrementally add clauses to the solver that represent the functionality of the nodes, similar to how the Tseitin transformation is used for combinational equivalence checking. Suppose that we have two nodes n_1 and n_2 that have an identical simulation vector. In order to check their equivalence we proceed as follows:

1) Run the solver with the assumption $\neg n_1 \cap n_2$. This checks if there is any interpretation under which n_1 is false and n_2 is true. If so, then the two nodes are clearly not equivalent and we return false. If not, the nodes may be equivalent.
2) We then run the solver with the assumption $n_1 \cap \neg n_2$, checking the converse. Again, if this assumption is SAT, we return false. If this is UNSAT, we have now shown that the nodes are equivalent and we return true.

This procedure can be extended in a simple way to check for equivalence up to complementation, as well as to support an *undefined* result if we want to bound the running time of equivalence checks.

A procedure that is similar to the one above can be used to enable the merging of multiple MIGs into a single FRMIG structure. Merging MIGs together enables a lossless MIG synthesis and optimization flow, in which we can take advantage of different optimization points during technology mapping.

IV. FRMIG LUT Mapping

FRMIGs are an extension of MIGs that reduce structural bias by combining multiple MIG structures. To take advantage of this new representation we present a LUT mapper that maps FRMIGs to k-LUTs. The architecture of this mapper is based on the work in [25]. As LUT mapping is based on cut enumeration, we first show how to enumerate the cuts of an MIG. We then show how cut enumeration for MIGs may be extended to FRMIGs.

Let n be a node in an MIG. A *cut* of n can be defined as a set c of nodes in its transitive fan-in such that every path from a primary input to n passes through a node in c. We say a cut c of n is redundant is there is some $c^- \subset c$ such that $c^-$is also a cut of n. A k-feasible cut of n is an irredundant cut c such that $|c| \wedge k$. When mapping a MIG we are only interested in finding k-feasible cuts, since any larger cuts cannot be covered by k-LUTs. For any node n, we consider the *trivial cut* $\{n\}$ to be a valid k-feasible cut.

Given an MIG $M = (V, E, Y)$ on primary inputs X, we use $\Phi(v)$ to denote the set of k-feasible cuts for $v \cup V$. We may define Φ recursively as:

$$\begin{aligned}
&\Phi(1) = \{\{\}\} \\
&\Phi(x) = \{\{x\}\} && \text{for } x \cup X \\
&\Phi(n) = \{\{n\}\} \forall\ (\Phi(n_1) \otimes \Phi(n_2) \otimes \Phi(n_3)) && \text{for } n \cup N
\end{aligned}$$

where $\otimes$ is an an operation that gives an over-approximation of the k-feasible cuts of a node. It is defined as

$$A \otimes B = \{a \forall\ b \mid a \cup A, b \cup B, |a \forall\ b| \wedge k\}$$

This definition of Φ may lead to the inclusion of some redundant cuts, but these can be easily filtered out during cut enumeration. Details of efficient cut computation are discussed in [25] and [10].

In order to support FRMIGs, we need to extend the above cut enumeration procedure to work with choice nodes. Let $M = (V, E, Y)$ be an FRMIG under $\simeq$ and $\mathcal{V}$. We compute cuts for both equivalence classes in $\mathcal{V}$ and nodes in V as follows. The set of cuts for an equivalence class $C \cup \mathcal{V}$ is

$$\Phi_{\simeq}(C) = \bigcup_{n \in C} \Phi(n)$$

In other words, the set of cuts for an equivalence class is simply the union of the cuts of all the nodes in that class.

We now alter the definition of Φ to

$$\begin{aligned}
&\Phi(1) = \{\{\}\} \\
&\Phi(x) = \{\{x\}\} && \text{for } x \cup X \\
&\Phi(n) = \{\{n\}\} \forall\ (\Phi_{\simeq}(n_1) \otimes \Phi_{\simeq}(n_2) \otimes \Phi_{\simeq}(n_3)) && \text{for } n \cup \mathcal{V} \\
&\Phi(n) = \Phi_{\simeq}(n_1) \otimes \Phi_{\simeq}(n_2) \otimes \Phi_{\simeq}(n_3) && \text{otherwise}
\end{aligned}$$

Note that we include the trivial cut only for equivalence class representatives. As a consequence, for any cut c, $n \cup c \equiv n \cup \mathcal{V}$.

After extending cut enumeration to work with choice nodes, our mapper can essentially be defined as a traditional cut-based FPGA mapper. Suppose we want to map an FRMIG $\widehat{M}$ to k-LUTs. We can do so by traversing $\widehat{M}$ in topological order, enumerating k-feasibly cuts for all $n \cup \mathcal{V}$. We compute a cost function for all cuts we find. The specific cost of a

cut depends on the optimization objective. Typically, in depth-oriented mapping, the cost is a function of the cut depth. After computing the cuts and their costs, a k-LUT cover is easily found by traversing the FRMIG in reverse topological order from outputs to inputs.

V. Exact MIG Optimization

In Section II-C we reviewed the notion of *exact synthesis*. We may use exact synthesis on small functions in order to synthesize MIGs that are optimal with respect to some optimization objective. We show here how this idea may be used in MIG optimization.

We first introduce some notation that will be useful in our discussion. Let $\mathcal{N}$ be a logic network. We use $d(n)$ to indicate the depth of nodes n in $\mathcal{N}$. If n is a primary input node, then $d(n) = 0$. Otherwise, let $\text{inputs}(n)$ denote the set of inputs of n. The depth of n is then $d(n) = \max\{d(i) \mid i \cup \text{inputs}(n)\} + 1$. Let $\text{outputs}(\mathcal{N})$ be the set of outputs of $\mathcal{N}$. We use $d(\mathcal{N}) = \max\{d(o) \mid o \cup \text{outputs}(\mathcal{N})\}$ to denote the depth of the logic network itself.

Let M be a MIG that computes some Boolean function $f(X)$ over $|X| = n$ variables. Suppose we map M to a depth-optimal k-LUT network L_k ($k \geq 3$). Then $d(L_k)$ is a lower bound on $d(M)$, since the k-LUTs can compute any function on k variables. Note that we do not mean that $d(L_k) \wedge d(M^{-})$ for *any* MIG M^{-} that computes $f(X)$. The lower bound is just on the specific structure of M. We call this a *structural lower bound*. To see why L_k gives a structural lower bound, note that any k-feasible cuts that contain multiple levels of majority nodes may be merged into a single k-LUT, thereby decreasing the depth of L_k relative to M. This means that for any depth-optimal cover L_k we have $d(L_k) \wedge d(M)$. As k increases, $d(L_k)$ decreases, since we are able to merge more functionality into the k-LUTs. In other words

$$\lim_{k \to \infty} d(L_k) = 1$$

since, as k approaches ∞, the entire logic network may be merged into a single k-LUT.

In addition to giving a structural lower bound, L_k also removes structural bias. In the extreme case, $k \geq n$ and all structure is removed, because M can be represented by single k-LUT. The LUT can be viewed as a specification of $f(X)$. Different structural representations can be extracted from the network by decomposing the LUT in different ways. Suppose we could use exact synthesis to decompose the LUT into a new MIG M^{-}. We would then have found an optimal MIG representation M^{-} of $f(X)$.

In practice, we cannot map most MIGs into a single k-LUT, because the LUT would require an efficient way of representing $f(X)$. Presumably, the most efficient representation available for $f(X)$ is the MIG itself. Suppose we could somehow map the MIG into a single LUT and efficiently represent $f(X)$. Using exact synthesis to decompose the LUT would still be infeasible, since n could be arbitrarily large. However, for smaller k, this method of mapping MIGs into k-LUTs becomes feasible. If k is "small enough", we can represent the different k-LUT functions, and we can use exact synthesis to decompose them. We do not remove all of the structural bias, but still enough to provide alternative optimization points. This is the basis for our exact k-LUT optimization procedure.

Algorithm 1: An MIG size optimization procedure using LUT mapping and exact synthesis.

```
function size-optimize(M, k) :=
Input  : MIG M
Output: Optimized MIG M⁻
M⁻ ← M;
do
    M ← M⁻;
    M⁻ ← new_mig();
    Perform area-oriented mapping of M into k-LUTs;
    foreach primary input i in M do
        create_input(M⁻, i);
    end
    foreach LUT l in the cover in topological order do
        f ← function computed by l;
        opt_mig ← exact_mig(f);
        create_nodes(M⁻, opt_mig);
    end
while size(M⁻) < size(M);
return M⁻;
```

This optimization method is not limited to reducing MIG depth. Instead of mapping into depth-optimal k-LUT networks, we could for instance focus on finding small k-LUT covers. Decomposing small k-LUT covers into locally minimum-sized MIGs can then be used to decrease MIG size.

The results of the method depend not only on the k-LUT cover, but also on the exact synthesis optimality criteria. For example, we may choose to decompose k-LUTs into locally size-optimal MIGs or into locally depth-optimal MIGs. Different choices lead to different optimization results. In a lossless synthesis flow these different optimization points may then be merged into a single FRMIG structure.

We present a MIG size optimization procedure based on this exact optimization method. It takes an MIG M and proceeds as follows. First, we map M into a k-LUT network while heuristically minimizing area. We then use exact synthesis to decompose the LUTs into minimum-size sub-MIGs. These are connected to produce a functionally equivalent MIG M^{-}. This process is iterated until the area of M^{-} is the same as that of M. The pseudocode for this procedure can be found in Algorithm 1.

Depth optimization techniques for MIGs are well developed. For example, the best known results for depth-optimal LUT mapping on the EPFL benchmark suite[1] were obtained by combining MIG optimization with the iterative application of ABC's technology mapper. MIG size optimization techniques, on the other hand, are less well developed. The current state of the art in MIG size optimization is based on *functional hashing* [8]. However, this approach results in only modest

[1] lsi.epfl.ch/benchmarks

size reductions. One drawback to this approach is that it has only a local view of size reductions. Furthermore, the best size reductions come at the cost of increasing MIG depth.

Our aim is to use MIGs as a general purpose logic synthesis representation. As such, we require methods that work well for both depth and size optimization. In Section VI we show how our new size optimization procedure improves MIG size optimization, thereby extending the range of logic synthesis and optimization objectives to which MIGs can be applied.

VI. Experiments

We show here that our new optimization method enables signiflcant improvements in both logic optimization and LUT mapping.

1) Methodology: We have developed a C++ logic manipulation package that implements the size optimization procedure described in Section V. It uses the methods presented in [8] to perform exact synthesis. The largest k for which exact MIGs were available at the time of these experiments was $k = 4$. Therefore, in these experiments our procedure maps MIGs into 4-LUTs. We run our experiments on the EPFL arithmetic benchmark suite. The choice for this suite was motivated by the fact that it contains relatively large circuits, as compared to other well known benchmarks suites. Note that our procedure is not limited to just arithmetic benchmarks.

2) Results: We flrst compare our new MIG size optimization procedure to the functional hashing approach described in [8] The results of the comparison can be found in Table I. We refer to our new procedure as *LUT-based size optimization*.

LUT-based size optimization reduces both depth and area by about 50% for the adder benchmark as compared to the functional hashing algorithm. On average functional hashing reduces MIG size by 3%, whereas LUT-based size optimization reduces MIG size by an average of 13%. Furthermore, LUT-based size optimization reduces MIG depth by 19% on average. Functional hashing, on the other hand, adds an average of 7% in depth to obtain its area reductions.

We can see that functional hashing has an inversely proportional relation between area and depth: it *increases* depth by 2.33% for every 1% in area reduction. LUT-based size optimization turns this into a directly proportional relation: it *decreases* depth by 1.46% for every 1% reduction in size.

In the next experiment we apply our LUT mapper to the size-optimized MIGs. We compare the results with the state-of-the-art LUT count results for the EPFL arithmetic benchmark suite. These best results were obtained by using ABC's AIG size optimization and LUT mapping algorithms. The results can be found in Table II.

Our procedure performs signiflcantly better on three of the benchmarks. It reduces area/depth by 4.5%/13.3% for the Adder, by 8.7%/15% for the Multiplier, and by 13%/56% for the Square. In other words, it is able to signiflcantly reduce area, while simultaneously reducing depth.

On the Log2 and Max benchmarks, our procedure increases area by 3.3% and 30%, respectively. However, it reduces depth by 7.8% and 41.2%, respectively. These results present an interesting area/depth trade-off.

Our mapper performs worse than ABC in the Sine benchmark, with an increase in area and depth of 4% and 6.4%, respectively. ABC performs signiflcantly better on the Divisor and Square-root benchmarks. We do not know which speciflc techniques were used to generate ABC's best results. As such, it is difflcult to determine what speciflcally allowed it to perform so much better on these particular benchmarks. We do know that, in order to obtain these results, a variety of ABC commands were used. These commands were iterated, always storing the best global result, until no further improvement was found [28]. Our results, on the other hand, were obtained by a single LUT mapping pass.

VII. Conclusions and Future Work

The goal of this paper was to improve techniques for MIG logic optimization and LUT mapping. In order to compete with the state-of-the-art in LUT mapping we have extended MIGs to FRMIGs. This allows us to reduce structural bias and to use MIGs in a lossless synthesis fiow.

Using our FRMIG mapper, we have introduced a general MIG optimization method based on k-LUT mapping and exact k-LUT decomposition. This method can be used in both depth and area optimization. We have shown how we can obtain up to 50% better results in both MIG size and area as compared to previous approaches in MIG size optimization. When compared to functional hashing, our approach reduces area and depth by 10% and 26%, respectively.

Using our new optimization method and LUT mapper we also obtain signiflcantly better results in both LUT count and LUT depth for 3 of the heavily optimized EPFL arithmetic benchmarks, reducing area and depth by up to 13% and 56%, respectively.

In our experiments we were limited to use exact synthesis for functions on up to 4 variables. We expect that increasing the number of variables will signiflcantly improve our optimization method. Preliminary results also show that our optimization procedure has promise in MIG depth optimization and improved depth-optimal LUT mapping.

Acknowledgment

This research was supported by SNSF-200021-146600 and H2020-ERC-2014-ADG 669354 CyberCare.

References

[1] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, 1990.

[2] A. L. Sentovich, E.M. and Singh, K.J. and Lavagno, L. and Moon, C. and Murgai, R. and Saldanha, A. and Savoj, H. and Stephan, P.R. and Brayton, Robert K. and Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. May 1992, 1992.

[3] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[4] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," *43rd ACM/IEEE Design Automation Conference*, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1146909.1147048

TABLE I
COMPARISON OF MIG SIZE OPTIMIZATION METHODS

Benchmark	I/O	Size	Depth	Functional Hashing Size	Functional Hashing Depth	LUT-based Size Optimization Size	LUT-based Size Optimization Depth
Adder	256/129	1020	255	1020	255	513	130
Barrel shifter	135/128	3336	12	3240	15	3336	12
Divisor	128/128	57247	4372	57247	4403	35993	3607
Hypotenuse	256/128	214335	24801	214307	24878	196842	11317
Log2	32/32	32060	444	29795	461	32060	444
Max	512/130	2865	287	2865	337	2479	276
Multiplier	128/128	27062	274	24903	271	22634	165
Sine	24/25	5416	225	5254	230	5416	255
Square-root	128/64	24618	5058	24618	6021	24170	6073
Square	64/128	18484	250	17893	250	17562	130
Average improvement (new/old):				0.97	1.07	**0.87**	**0.81**

TABLE II
COMPARISON OF 6-LUT AREA MINIMIZATION

Benchmark	I/O	Size	Depth	ABC Size	ABC Depth	MIG Mapper Size	MIG Mapper Depth	Size (MIG/ABC)	Depth (MIG/ABC)
Adder	256/129	1020	255	201	73	192	64	0.96	0.87
Barrel shifter	135/128	3336	12	512	4	512	4	1.00	1.00
Divisor	128/128	57247	4372	3813	1542	10328	1915	2.70	1.24
Log2	32/32	32060	444	7344	142	7592	131	1.03	0.92
Max	512/130	2865	287	532	192	692	113	1.30	0.59
Multiplier	128/128	27062	274	5681	120	5192	102	0.91	0.85
Sine	24/25	5416	225	1347	62	1402	66	1.04	1.06
Square-root	128/64	24618	5058	3286	1180	6810	2070	2.07	1.75
Square	64/128	18484	250	3800	116	3309	74	0.87	0.64

[5] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verifıcation Tool," in *Computer Aided Verification*, vol. 6174, 2010, pp. 24–40.

[6] L. Amarú, P.-E. Gaillardon, and G. D. Micheli, "Majority-Inverter Graph : A Novel Data-Structure and Algorithms for Efficient Logic Optimization," *Design Automation Conference*, 2014.

[7] ——, "Boolean Logic Optimization in Majority-Inverter Graphs," in *Proc. of Design Automation Conf. (DAC)*, 2015.

[8] M. Soeken, L. G. Amarù, P.-e. Gaillardon, and G. D. Micheli, "Optimizing Majority-Inverter Graphs With Functional Hashing," in *Design Automation and Test in Europe*, 2016, pp. 1030–1035.

[9] G. Chen and J. Cong, "Simultaneous Logic Decomposition with Technology Mapping in FPGA Designs," in *ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, 2001, pp. 48–55. [Online]. Available: http://portal.acm.org/citation.cfm?doid=360276.360298

[10] S. Chatterjee, "On Algorithms for Technology Mapping," Ph.D. dissertation, University of California at Berkeley, 2007.

[11] A. Mishchenko and R. Brayton, "Combinational and Sequential Mapping with Priority Cuts," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 354–361.

[12] A. Mishchenko, R. Brayton, and S. Jang, "Global Delay Optimization Using Structural Choices," in *FPGA '10*, 2010, pp. 181–184. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1723112.1723144

[13] D. E. Knuth, *The Art of Computer Programming, Volume 4A, Part 1*, 1st ed. Addison-Wesley Professional, 2011.

[14] L. Amarú, P.-E. Gaillardon, A. Chattopadhyay, and G. De Micheli, "A Sound and Complete Axiomatization of Majority-n Logic," 2016.

[15] C. Märtin, "Multicore Processors: Challenges, Opportunities, Emerging Trends," in *Embedded World Conference*, 2014, pp. 25–27.

[16] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, G. Jan, G. Michael, H. Scott, H. Stephen, A. Hormati, J.-y. K. Sitaram, L. James, and L. Eric, "A Reconflgurable Fabric for Accelerating Large-Scale Datacenter Services - Microsoft Research," in *41st Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[17] J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.

[18] ——, "On Area/Depth Trade-Off in LUT-Based FPGA Technology Mapping," in *IEEE Transactions on Very Large Scale Integration Systems*, 1994, pp. 137–148.

[19] J. Cong and Y.-Y. Hwang, "Simultaneous Depth and Area Minimization in LUT-based FPGA Mapping," in *Third International ACM Symposium on Field-Programmable Gate Arrays*, 1995, pp. 68–74.

[20] J. Cong, C. Wu, and Y. Ding, "Cut Ranking and Pruning: Anabling A General And Efflcient FPGA Mapping Solution," in *FPGA '99*, 1999, pp. 29–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=296425

[21] D. Chen and J. Cong, "DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs," in *International Conference on Computer Aided Design*. IEEE, 2004, pp. 752–759.

[22] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing Structural Bias in Technology Mapping," *IEEE/ACM International Conference On Computer Aided Design*, pp. 512–526, 2005.

[23] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic Decomposition During Technology Mapping," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 8, 1997, pp. 813–834.

[24] A. Mishchenko, S. Chatterjee, R. Jiang, and R. Brayton, "FRAIGs : A Unifying Representation for Logic Synthesis and Verifıcation," Tech. Rep., 2005.

[25] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, 2007, pp. 240–253.

[26] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verifıcation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[27] L. de Moura and N. Bjørner, "Z3: An Efflcient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[28] R. K. Brayton and A. Mishchenko, "Private Communication," 2016.

Inversion Minimization in Majority-Inverter Graphs

Eleonora Testa[1], Mathias Soeken[1], Luca Gaetano Amaru[2], Pierre-Emmanuel Gaillardon[3], Giovanni De Micheli[1]
[1] EPFL, Switzerland [2] Synopsys, CA, USA [3] University of Utah, UT, USA
eleonora.testa@epfl.ch

***Abstract*—In this paper, we present new optimization techniques for the recently introduced *Majority-Inverter Graph* (MIG). Our optimizations exploit intrinsic algebraic properties of MIGs and aim at rewriting the complemented edges of the graph without changing its shape. Two exact algorithms are proposed to minimize the number of complemented edges in the graph. The former is a dynamic programming method for trees; the latter finds the exact solution with a minimum number of inversions using *Boolean satisfiability* (SAT). We also describe a heuristic rule based algorithm to minimize complemented edges using local transformations. Experimental results for the exact algorithm to fanout-free regions show an average reduction of 12.8% on the EPFL benchmark suite. Applying the heuristic method on the same instances leads to a total improvement of 60.2%.**

I. Introduction

Nanotechnologies are recently being studied as replacement or enhancement for CMOS. Devices in these nanotechnologies have logic models different from standard transistors and many of them realize majority gates as primitive building blocks. Examples of these nanotechnologies are *Quantum-dot Cellular Automata* (QCA, [1], [2]), *Spin Wave Devices* (SWD, [3], [4]), *Spin-Transfer-Torque* Devices (STT, [5]), *Resistive Random Access Memories* (RRAMs, [6], [7], [8]). To properly assess these post-CMOS technologies, *Electronic Design Automation* (EDA) tools necessitate new logic synthesis techniques and abstractions [9]. Much work concerning majority synthesis has been carried out back in the 1960s [10], [11]. Recently, *Majority-Inverter Graphs* (MIG, [12]) are found to suitably abstract novel majority based nanotechnologies [13], [14], besides being a useful tool to reduce area and delay in standard CMOS circuits. MIGs use the majority-of-three function $\langle xyz \rangle = xy \vee xz \vee yz$ and negation as only logic primitives; negations are simply represented as complemented edges in the graph, similarly as in BDDs.

Inversion minimization plays a predominant role in emerging technologies whose circuits are built using only majorities (MAJ) and inverters (INV). Area and delay costs depends on the number of INVs in the circuit; for instance, a QCA majority gate and inverter are presented in [2]; the majority gate requires five QCA cells, while the inverter gate requires up to 13 QCA cells.

Previous work has considered inversion minimization [15]. In this work, we exploit the intrinsic algebraic properties of MIGs which allow for superior rewriting possibilities as compared to other logic representations such as *And-Inverter Graphs* (AIGs). As an example, negations can be freely propagated through an MIG using self-duality, i.e., $\langle \bar{x}\bar{y}\bar{z} \rangle = \overline{\langle xyz \rangle}$.

In this paper, we present MIG rewriting techniques that target at optimizing inversions within the logic network. The core idea is to minimize the number of complemented edges in the graph without changing its shape. The shape is the way in which all the nodes are connected through edges by disregarding complemented edges. We present two exact algorithms to minimize inversions in the MIG. First, we focus on a dynamic programming algorithm for trees; then we propose a minimization method based on *Boolean Satisfiability* (SAT). These methods show good results, but the former is optimal for fanout-free MIGs, while the latter can be applied to small graphs only. We also describe heuristic approaches to obtain a local minimum in the number of complemented egdes.

Our experimental results show that the tree based exact algorithm can be used to minimize inversions in the MIG by identifying trees within the whole graph. A further optimization is possible by using the heuristic algorithm on the entire MIG. The combination of these methods leads a total reduction of 60.2% on average.

The paper is organized as follows. Section II introduces background on MIG; Section III focuses on the two exact approaches and on the heuristic method used to minimize complemented edges. Section IV shows experimental results and Section V concludes the paper.

II. Background

In this section, we describe MIGs [12], [16], [17], which are logic representation forms based on majority logic. A MIG is a data structure for Boolean function representation and optimization. It is defined as a homogeneous logic network consisting of 3-input majority nodes and regular/complemented edges.

MIGs can efficiently represent Boolean functions thanks to the expressive power of the majority operator. Indeed, a majority operator can be configured to behave as a traditional conjunction (AND) or disjunction (OR) operator. 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 MAJ, traditional *AND/OR/INV Graphs* (AOIGs) are a special case of MIGs and MIGs can be easily derived from AOIGs. An example of MIG representations derived from its optimal AOIG is depicted by Fig. 1a. AND/OR operators are replaced node-wise by MAJ-3 operators with a constant input.

Intuitively, MIGs are at least as compact as AOIGs. However, even smaller MIG representation arises when fully

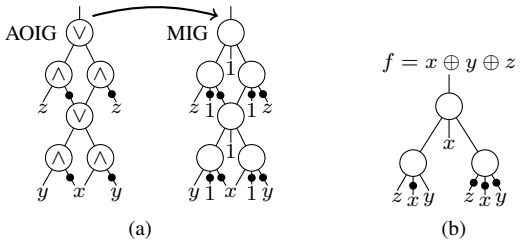


Fig. 1. Example (a) of MIG representations (right) for $f = x \oplus y \oplus z$ derived by transposing its optimal AOIG representations (left). Complement attributes are represented by bubbles on the edges. Example (b) of optimized MIG for f.

exploiting the majority functionality, i.e., with non-constant inputs [12].

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 [16]. The axiomatic system for the MIG Boolean algebra, referred to as Ω, is defined by the five following primitive transformation rules.

$$\Omega \begin{cases} \textbf{Commutativity} - \boldsymbol{\Omega.C} \\ \langle xyz \rangle = \langle yxz \rangle = \langle zyx \rangle \\ \textbf{Majority} - \boldsymbol{\Omega.M} \\ \langle xxy \rangle = x \qquad \langle x\bar{x}y \rangle = y \\ \textbf{Associativity} - \boldsymbol{\Omega.A} \\ \langle xu\langle yuz \rangle \rangle = \langle zu\langle yux \rangle \rangle \\ \textbf{Distributivity} - \boldsymbol{\Omega.D} \\ \langle xy\langle uvz \rangle \rangle = \langle \langle xyu \rangle \langle xyv \rangle z \rangle \\ \textbf{Inverter Propagation} - \boldsymbol{\Omega.I} \\ \overline{\langle xyz \rangle} = \langle \bar{x}\bar{y}\bar{z} \rangle \end{cases} \quad (1)$$

Some of these axioms are inspired from median algebra and others from the properties of the median operator in a distributive lattice. A strong property of MIGs and their algebraic framework is about reachability. It has been proven that, by using a sequence of transformations drawn from Ω, it is possible to traverse the entire MIG representation space [12]. In other words, given any two equivalent MIG representations, it is possible to transform one into the other by just using axioms in Ω. This results is of paramount interest to logic synthesis because it guarantees that the best MIG, for a given target metric, can always be reached. Unfortunately, deriving such an optimal sequence of transformations is an intractable problem. As for traditional logic optimization, heuristic techniques provide here fast solutions with reasonable quality [18]. An efficient depth optimization heuristic has been introduced that iterates local Ω rules over the critical path in order to push up variables with late arrival times.

As previously anticipated, by using the MIG algebraic framework, it is possible to obtain better MIGs for the example in Fig. 1a. Fig. 1b shows the new MIG structure, which is optimized in both depth (number of levels) and size (number of nodes). These MIG can be reached using a sequence of Ω axioms starting from their unoptimized structures. We refer the reader to paper [12] for an in-depth discussion on MIG optimization recipes.

III. Inversion Minimization

This section describes the techniques employed to minimize the number of complemented edges. First, we present two exact algorithms to minimize inversions in the graph. Then, we focus on a heuristic minimization method based on the local manipulation of each node using the axiom $\Omega.I$.

A. Tree based Exact Algorithm

We first present an exact algorithm to minimize complemented edges for the case when the MIG is a tree, i.e., if every node has at most one output. Here, inversion minimization is possible using the axiom $\Omega.I$. The axiom $\overline{\langle xyz \rangle} = \langle \bar{x}\bar{y}\bar{z} \rangle$ states that the complemented edges can be moved without neither changing the number of nodes nor the edges connections in the graph. For each node, the complemented edges configuration includes three input edges and one output edge; the original configuration is the configuration of the node in the MIG, while the changed configuration is the configuration of the node changed according to the axiom $\Omega.I$. The cost of a node is the sum of the number of complemented edges on three children and the cost of each children. Complements on constant inputs are not accounted in the cost, since in physical implementations both constants `0` and `1` are available.

The presented method uses dynamic programming and computes best configurations in a topological order. We notice (i) that the cost of complemented edges depends on the cost of the three children, and (ii) that two optimum configurations for a node have different effects on the parent node only if they show opposite polarities for the node's output.

The algorithm is applied to the nodes going from the inputs (leaves) to the outputs (roots) of the graph. Since we traverse the nodes in topological order, for every node v, we know the minimum complemented edges configuration for each children and the cost of each configuration.

The algorithm considers two cases depending on which level the node belongs:

(i) nodes with all children being primary inputs,
(ii) all other nodes.

Case (i): We save both the original configuration of complemented edges and the one changed according to $\Omega.I$. These two configurations have different output polarities and then they cause a different effect on the parent node. The two configurations and their costs are stored.

Case (ii): At this point, we have computed at most two optimum configurations for each child from which we can compute at most 8 configurations for the node. From these the best configuration for each output polarity is saved. If only one output polarity is covered by all configurations, we apply $\Omega.I$ on the computed configurations and pick the best one from the newly generated ones.

It is worth noticing that for the output nodes only one configuration is saved. The configuration with the smallest cost is saved independently of the output polarity; if two

configurations have the same cost, only one is saved since there is no effect on the MIG.

The algorithm is explained in Alg. 1. An MIG $M = (V, E, Y)$ is a DAG consisting of a finite set of nodes V, a finite multiset of edges E and a finite multiset of outputs Y [19]. The MIG resulting from the algorithm is a new MIG $\hat{M}$ with a different complemented edges configuration. The nodes are evaluated in topological order and depending on their level. For nodes v with all children being primary inputs, two configurations are saved, v and $\hat{v}$. For other nodes, all the combinations of children configurations are considered. W is a finite set including all configurations of node v, while $\hat{W}$ consists of configurations changed according to the axiom. Two configurations are saved, which respectively are the best one in W and $\hat{W}$. If v is an output node, the configuration with the lower cost is add to $\hat{M}$.

Data: MIG $M = (V, E, Y)$
Result: Optimized MIG $\hat{M}$
foreach $v \in$ topsort(V) **do**
  if *all children of* v *are primary inputs* **then**
    set $\hat{v} \leftarrow \Omega.I(v)$;
    set conf$(v) \leftarrow \{v, \hat{v}\}$;
  else
    let v_1, v_2, v_3 be the children of v;
    set $W \leftarrow \{\}$, $\hat{W} \leftarrow \{\}$;
    foreach *combination* v' *of* conf(v_1), conf(v_2), conf(v_3) **do**
      set $W \leftarrow W \cup \{v'\}$;
      set $\hat{W} \leftarrow \hat{W} \cup \{\Omega.I(v')\}$;
    end
    set conf$(v) \leftarrow \{\min W, \min \hat{W}\}$;
    if v *is an output node* **then**
      add the best configuration in conf(v) to $\hat{M}$.
    end
  end
end

Algorithm 1: Complemented edges minimization in a tree.

Example 1: Alg. 1 is applied to the MIG in Fig. 2a. We will use the symbol $\hat{}$ for nodes changed according to axiom $\Omega.I$. First, nodes 1 and 2 are considered since their inputs are primary inputs only. For these nodes, two configurations are saved. The costs of the nodes are evaluated and stored. For instance, for node 1 the cost is equal to 2, while for node $\hat{1}$ the cost is 1, since only one input is complemented. Four configurations are possible for node 3, since two of its children (1 and 2) have two configurations stored. They are shown in Fig. 3a. The costs are given by the sum of cost of node 3 and the costs of its children. Four configurations are possible for node $\hat{3}$ and they are shown in Fig. 3b. All the configurations of node 3 have an opposite polarity with respect to node $\hat{3}$. Two configurations with opposite polarities need to be saved. We save the ones with the smallest cost for node 3 and node $\hat{3}$, which respectively are configuration (i) for node 3 and (ii) for $\hat{3}$. For node 4 and $\hat{4}$, two combinations are present. Since

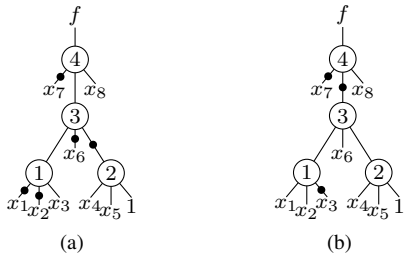


Fig. 2. Example: MIG for function f, before (a) and after (b) minimization of complemented edges using the exact algorithm for trees.

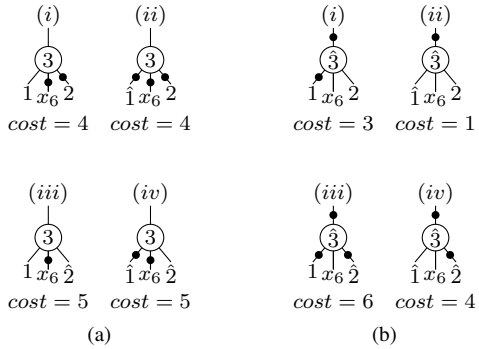


Fig. 3. All combinations for node 3 (a) and node $\hat{3}$ (b).

4 is an output node, only the configuration with the smallest cost is saved.

The final MIG is shown in Fig. 2b. The total amount of complemented edges is reduced from 5 to 3.

B. SAT based Exact Algorithm

Here, we aim at minimizing the number of complemented edges using a SAT based approach. Given an MIG, the Boolean function f that it represents, and the number of complemented edges p, the purpose is to find the minimum number of complemented edges $p_{\min}$ for the given graph. In order to be consistent with the previous algorithm, the shape and the number of nodes of the graph are not changed by this method. It is worth noticing that the SAT based algorithm can be adopted to find an MIG with $p_{\min}$ and a different number of nodes and shape.

Our approach finds optimum solutions by solving decision problems that ask whether there exists an MIG in which the number of complemented edges is $p_{\min}$. The algorithm used and the encoding of the problem are similar to the ones proposed in [19]. It is worth noticing that this SAT problem can also be formulated in different ways and alternative encodings are possible.

In our approach, the instance is a Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ represented by the MIG, which has k nodes and p complemented edges. To find the solution, we start by solving the decision problem for $p_{\min} = p$ and decrease the value until the solution becomes unsatisfiable. According to [19], the number of nodes is equal to k and each node with index $l \in \{1, ..., k\}$ can be represented with 10 variables:

- three inputs $a_{1,l}^{(j)}, a_{2,l}^{(j)}, a_{3,l}^{(j)} \in \mathbb{B}$ of gate l,
- one output $b_l^{(j)} \in \mathbb{B}$ of gate l,
- three select variables $s_{1,l}, s_{2,l}, s_{3,l} \in \mathbb{B}^{\lceil \log_2(n+l) \rceil}$ that encode which are the children of gate l, and
- polarity variables $p_{1,l}, p_{2,l}, p_{3,l} \in \mathbb{B}$ that represented regular/complemented edges of the children.

Variable j ranges from 0 to $2^n - 1$ and each j represents one input assignment in f; hence each node is duplicated 2^n times.

Regarding the problem constraints, the *Majority Functionality* and the *Function Semantics* are ensured by constraints proposed in [19]. Two more constraints are considered for this problem: (i) one on the select variables in order to have the same MIG structure as the original one and (ii) one on the polarities variables for the minimum number of complemented edges. These two constraints are discussed below.

1) Input Connections: For this decision problem, the number of nodes k is given and it is equal to the number of nodes of the original MIG. Since we want to have the same structure for the MIG, the values for the three select variables $s_{1,l}, s_{2,l}, s_{3,l}$ are known and are taken from the MIG. For each node, since all its children belong to a lower level, the values for the three select variables range from 0 to $n + (l - 1)$, where n is the number of inputs. If $s_{c,l}$ is equal to 0, it means that the constant value 0 is child c of node l. If the input of a node is a primary input, then its select variable is lower than or equal to n; while a value between n and $n + (l - 1)$ ensures a connection with an internal node in the lower level.

2) Minimum Complemented Edges: To make sure that the number of complemented edges equal to $p_{\min}$, some clauses are added to the problem. A SAT encoding for cardinality constraints such as $x_1 + \cdots + x_m \leq r$ is proposed in [20]. We add similar clauses to ensure that the sum of all the polarities variables and the polarity of the output is equal to $p_{\min}$. It is worth noticing that also in this algorithm, complemented edges that point to constant 0 are not considered in the sum. The number m of polarity variables is $3k + 1 - z$, where k is the number of nodes of the MIG and z is the number of constant inputs. For this case, $r = p_{\min}$. We consider $(m - r) \cdot r$ new variables q_v^h with $1 \leq v \leq (m - r)$ and $1 \leq h \leq r$, and new clauses for the SAT problem:

$$\bar{q}_v^h \vee q_{v+1}^h, \text{ for } 1 \leq v < (m - r) \text{ and } 1 \leq h \leq r, \quad (2)$$

$$\bar{x}_{v+h} \vee \bar{q}_v^h \vee q_v^{h+1}, \text{ for } 1 \leq v \leq (m - r) \text{ and } 0 \leq h \leq r, \quad (3)$$

where $\bar{q}_v^h$ is not added for $h = 0$ and q_v^{h+1} is omitted when $h = r$.

The decision problem is solved using the SMT solver Z3 [21]. Fig. 4a is given in order to explain the approach used. It shows an MIG in which the initial number of complemented edges is $p = 6$ and the number of nodes is $k = 3$. The select variables are:

$$s_{1,1} = 1, s_{2,1} = 2, s_{3,1} = 3; \quad (4)$$

$$s_{1,2} = 7, s_{2,2} = 5, s_{3,2} = 4; \quad (5)$$

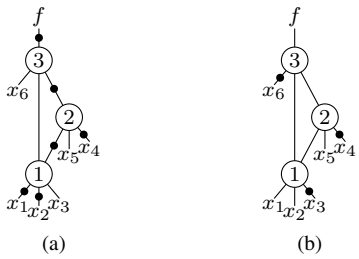


Fig. 4. Example: MIG for function f, before (a) and after (b) minimization of complemented edges using a SAT-solver based approach.

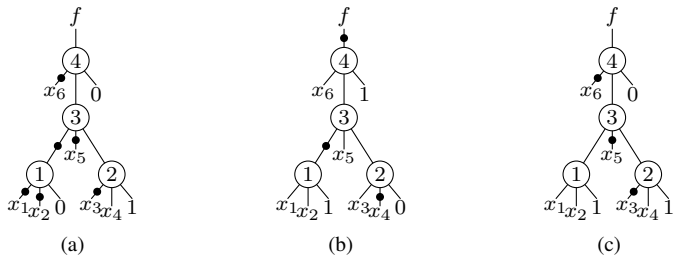


Fig. 5. Another example: MIG for function f (a); minimization of complemented edges using the tree method (b) and a SAT-solver based approach (c).

$$s_{1,3} = 6, s_{2,3} = 7, s_{3,3} = 8. \quad (6)$$

The problem we want to solve is

$$p_{1,1} + p_{2,1} + p_{3,1} + p_{1,2} + p_{2,2} + p_{3,2} + p_{1,3} + p_{2,3} + p_{3,3} + y \leq p,$$

where y is the output polarity according to [19]. To find the minimum number of complemented edges, we decrease p until the problem becomes unsatisfiable and $p_{\min}$ is found. For the example in Fig. 4, the minimum number of complemented edges is 3. The final result is shown in Fig. 4b. The number of nodes and the structure of the MIG are the same as the input graph.

As shown by the previous example, this approach is not limited to fanout-free circuits as the method described in Section III-A. Fig. 5 shows a circuit optimized with both the tree method (Fig. 5b) and the SAT based method (Fig. 5c). The resulting circuit has a different complemented edges configuration, but the minimum number is the same for both methods.

This SAT-oriented approach allows us to find the exact solution in terms of number of complemented egdes; but it does not scale for large functions.

We would like to remark that mapping into SAT is only one possible way of facing the inversions minimization and other alternatives can be found. For instance, the problem can be formulated as an instance of graph rewriting [22]. Each combination of the inverter propagation rule $\Omega.I$ is expressed in terms of a graph rewrite rule and a new graph with a different complemented edges configuration replaces the original MIG. Both heuristic and exact graph rewrite algorithms can be employed to create the new graph and solve the inversion minimization.

C. Heuristic Algorithm by Local Rewriting

Both the exact approaches described in previous section suffer from some limits: the former can be applied to fanout-free graphs only, while the second is perfect for graphs of limited size. Here, we propose a heuristic method to minimize complemented edges. We aim at minimizing the complemented edges in the MIG by recursively applying the inverter propagation axiom $\Omega.I$, given by $\overline{\langle xyz \rangle} = \langle \bar{x}\bar{y}\bar{z} \rangle$.

The main idea to minimize complemented edges is to use $\Omega.I$ to move complemented edges on the inputs to the output. To reduce the number of complemented edges, we apply the transformations rules mentioned below on all the nodes of the MIG. These transformations do not change the depth nor the size of the MIG. Different ways of applying $\Omega.I$ can be adopted depending on whether the node has constant inputs or not. Furthermore, the axiom can be applied considering one node at a time (one-level) or evaluating savings in the number of complemented edges for two levels of the MIG (two-level). A taxonomy of the rules used to decrease complemented edges is proposed in the following; then the procedure used to minimize complemented edges is described.

1) One-level Rules for MAJ: Here, we describe rules that apply to *MAJ* nodes, which are nodes with no constant inputs. The rules used to decrease complemented edges aim at moving complemented edges of the inputs to the output. They can be formalized as:

$$\text{Rules} \begin{cases} \textbf{MAJ\_3} \\ \begin{cases} \langle \bar{x}\bar{y}\bar{z} \rangle = \overline{\langle xyz \rangle} \\ \overline{\langle \bar{x}\bar{y}\bar{z} \rangle} = \langle xyz \rangle \end{cases} \\ \textbf{MAJ\_2} \\ \overline{\langle x\bar{y}\bar{z} \rangle} = \langle \bar{x}yz \rangle \end{cases} \tag{7}$$

The MAJ_3 rules consider nodes in which the three inputs are complemented. Considering each node, these transformations lead to a decrease in the number of complemented edges equal to:

$$3 + (\#\text{CO} - \#\text{NCO}) \tag{8}$$

where $\#$CO is the number of complemented outputs of the node and $\#$NCO are the uncomplemented outputs. Savings for the MAJ_2 rule are equal to:

$$1 + (\#\text{CO} - \#\text{NCO}) \tag{9}$$

For these rules, only one node is considered at a time. We evaluate savings according to the formulas mentioned above; each node is changed using one of the transformation rules if savings larger than 0 can be achieved. An example is given in Fig. 6. Fig. 6a shows the MIG of the full adder composed by the three nodes 1, 2 and 3. It is possible to apply the MAJ_3 rule on node 2 with savings of 3 and the MAJ_2 on node 1 with savings of 2. Fig. 6b illustrates the MIG of the full adder after the one-level transformations applied on the nodes of the first level. In this example, the number of complemented edges is reduced from 7 to 2.

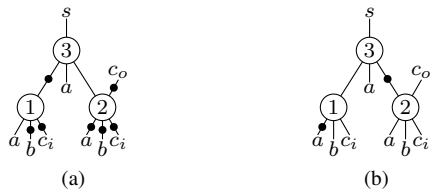


Fig. 6. MIG for a full adder, before (a) and after (b) one-level rules to decrease the number of complemented edges. Bubbles represent complementation of the edges

2) One-level Rules for AND/OR: All the rules mentioned above are applied to nodes in which none of the inputs is set to a constant value. It is worth noticing that they equally work for *AND* and *OR* nodes, which are majority nodes in which one of the input is set to `0` and `1`, respectively. Our MIG data structure only has constant `1`, hence in AND nodes the constant input is $\bar{1} = 0$. The rules used to decrease complemented edges for AND/OR are:

$$\text{AND/OR Rules} \begin{cases} \textbf{AND\_3} \\ \begin{cases} \langle \bar{x}\bar{y}\bar{1} \rangle = \overline{\langle xy1 \rangle} \\ \overline{\langle \bar{x}\bar{y}\bar{1} \rangle} = \langle xy1 \rangle \end{cases} \\ \textbf{AND\_2} \\ \overline{\langle x\bar{y}\bar{1} \rangle} = \langle \bar{x}y1 \rangle \\ \textbf{OR\_2} \\ \overline{\langle \bar{x}\bar{y}1 \rangle} = \langle xy\bar{1} \rangle \end{cases} \tag{10}$$

Note that, as in the previous algorithms, complements on constant inputs are not accounted in the total amount of complemented edges. The savings estimation is adjusted for the AND/OR cases. The AND_3 rule has savings equal to:

$$2 + (\#\text{CO} - \#\text{NCO}) \tag{11}$$

while for the AND_2 rule they are equal to:

$$\#\text{CO} - \#\text{NCO} \tag{12}$$

For OR rule savings are equal to:

$$2 + (\#\text{CO} - \#\text{NCO}) \tag{13}$$

3) Two-level Rules: In two level transformations, we consider one node (main node) and all the nodes on its outputs (parents). By doing this, we account that transforming only the main node may not result in savings greater than 0, but this transformation changes the complemented edges pattern of the parents and consequently may result in a total two-level savings larger than 0 when applying transformations on the parents. We evaluate the total two-level savings as the sum of the savings obtained by changing the main node and the savings achieved on the parents if the main node is changed. If the total two-level savings are positive, first the main node is changed according to the rules (7), then all the one-level rules are applied to the parents. In Fig. 7a, none of the nodes, which are called here node 1, 2 and 3, can be changed by the one-level transformation since there is no benefit in the total number of complemented edges. On the other hand, changing

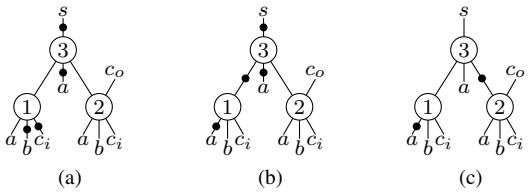


Fig. 7. Two-level transformation applied on the MIG of a full adder.

node 1 according to the inverter propagation axiom generates the possibility of applying one of the one level rules on node 3. If node 1 is changed, its savings are equal to 0, but, thanks to this first transformation, the MAJ_2 rule can be applied to node 3. Fig. 7b shows the full adder after changing node 1; while Fig. 7c shows the final circuit. In this case, the number of complemented edges is not reduced from step a to step b but it is reduced from step b to c, with total savings equal to 2.

4) Minimization Procedure: We minimize the complemented edges using the reduction rules proposed above in the way given in Alg. 2. We describe a possible order of applying the rules, but they can be applied in a different way producing different results. When the number of complemented edges before optimization is equal to the number of complemented edges after the optimization, it means that no further rules can be applied to the MIG and we have reached the minimal number of complemented edges achievable with our procedure. We apply each single rule from node 0 to the last node of the circuit. At the end of the minimization process, the MIG has the same logical depth and the same size of the unoptimized one, since all transformation rules do not change the shape of the graph but aim at changing the number of complemented edges.

```
1 while the number of compl. edges is decreasing do
2 |   AND_3(n) for each node n;
3 |   AND_2(n) for each node n;
4 |   MAJ_3(n) for each node n;
5 |   MAJ_2(n) for each node n;
6 |   Two-level(n) for each node n;
7 end
```

Algorithm 2: Heuristic algorithm to minimize complemented edges.

IV. Experimental Results

In this section, we present results obtained in terms of minimization of complemented edges. First, we test the tree algorithms on the fanout-free regions of the arithmetic circuits of the EPFL benchmarks,[1] then we describe the further improvement obtained on the same instances with the heuristic approach and the minimization rules proposed in Section III-C. Preliminary experimental results are provided for the SAT based approach.

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

TABLE I
Exact tree algorithm on fanout-free regions

Benchmark	$\#N$	#FFR	S_{ffr}	#CE	$\#\text{CE}_{\text{tree}}$	Impr.
adder	2978	1447	2.1	2905	2532	12.8%
divisor	75666	28058	2.7	78302	67622	13.6%
log2	37582	13374	2.8	38585	33877	12.2%
max	7202	1421	5.1	6543	4778	27.0%
multiplier	41885	14886	2.8	39589	37530	5.2%
sin	7890	3002	2.6	7625	6642	12.9%
sqrt	52344	19697	2.7	53327	44788	16.0%
square	19200	11277	1.7	23390	22752	2.7%
Average			2.8			12.8%

$\#N$: number of MIG nodes, #FFR: number of fanout-free regions, S_{ffr}: average size of the fanout-free regions, #CE: number of complemented edges, $\#\text{CE}_{\text{tree}}$: number of complemented edges after minimization with the exact tree algorithm on the fanout-free regions, Impr: improvement with regards to #CE.

A. Tree based Exact Algorithm on Fanout-free Regions

We developed a C program to implement the exact algorithm for trees proposed in III-A. The algorithm has been applied to instances which are optimized using the rewrite recipe described in [12]. The algorithm has been used on the fanout-free regions of the MIGs. It means that we identified trees within the entire graph and we applied the algorithm to subcircuits that do not violate the tree shape. The fanout-free regions are evaluated in topological order and the nodes with more than one output are not changed by the algorithm.

We compare the number of complemented edges; results are shown in Table I.

The size and the depth of the MIG are the same before and after minimization. The improvement in the number of complemented edges is 12.8% on average. Table I lists the number of fanout-free regions and their average size for each instance. As expected, the larger improvement in the number of complemented edges is for the instance in which the fanout-free regions have larger size (max).

Since, on average, the fanout-free regions have small size, an optimization across fanout-free regions boundaries is required.

B. Heuristic Algorithm by Local Rewriting

To overcome limits due to the reduced size of fanout-free regions, we developed a C program to implement the heuristic algorithm proposed in Section III-C. We applied the algorithm to further optimize instances in which the number of complemented edges is already reduced with the method discussed in Section III-A. The code implements all the rules described in Section III-C. The rules are applied as proposed in Alg. 2 on all the nodes of the MIG, from the first one to the last one. A loop iterates the process until the minimal number of complemented edges is reached. The code changes the number of complemented edges of the MIG without transforming the shape of the graph. Results are shown in Table II.

The largest improvement in the number of complemented edges is obtained after the first loop. On average, 3.5 loops are necessary to reach the minimal number of complemented

TABLE II
HEURISTIC ALGORITHM BY LOCAL REWRITING

Benchmark	#CE	$\#CE_{tree}$	$\#CE_{local}$	Impr.	Tot_Impr.
adder	2905	2532	1192	52.9%	59.0%
divisor	78302	67622	30748	54.5%	60.7%
log2	38585	33877	14367	57.6%	62.8%
max	6543	4778	3464	27.5%	47.1%
multiplier	39589	37530	18829	49.8%	52.4%
sin	7625	6642	3017	54.6%	60.4%
sqrt	53327	44788	19889	55.6%	62.7%
square	23390	22752	5404	76.2%	76.9%
Average				53.6%	60.2%

#CE: number of complemented edges, $\#CE_{tree}$: number of complemented edges obtained in IV-A, $\#CE_{local}$: number of complemented edges after minimization with heuristic algorithm by local rewriting, Impr: improvement with regards to $\#CE_{tree}$, Tot_Impr: improvement with regards to #CE.

edges. The algorithm is applied on instances in which the number of complemented edges has been previously decreased with the tree algorithm on fanout-free regions. The further improvement is of 53.6% on average.

The combination of these two methods allows a total improvement in the number of complemented edges equal to 60.2% on average.

C. SAT based Algorithm

So far, we have a preliminary experimental evaluation for the SAT based algorithm. We have run the exact synthesis algorithm on all unique (fully DSD decomposable) 6 variable functions obtained using structural cut enumeration on all instances of the MCNC, ISCAS, and ITC benchmarks as proposed in [23]. That is, first an MIG with an optimum number of nodes is found which is then resynthesized to achieve the optimum number of complemented edges in a second step. 40195 functions are evaluated and a timeout of 1 minute is given for each function. Among these 40195, we encountered 2264 timeouts and in the remaining ones, we achieved a reduction in the number of complemented edges of 30.8% in average, requiring 0.3 seconds. In the best case, the complemented edges are reduced by 83.3% (from 12 to 2) with a runtime equal to 0.7 seconds.

V. CONCLUSION

We described two exact algorithms to minimize the number of complemented edges in a MIG. The tree exact algorithm can be applied to MIGs in which each node has one output only, while the SAT based approach is suitable for MIGs of limited size. We illustrated also a heuristic method to decrease complemented edges.

At the logic level, our results showed that a decrease in the number of complemented edges of 12.8% on average can be achieved when applying the tree exact algorithm on the fanout-free regions of the circuits. Experiments showed that due to the limited size of fanout-free regions, an optimization that considers fanout-free regions boundaries is needed. Experimental results illustrated that a total improvement of 60.2% on average can be reached using a combination of the exact algorithm and the heuristic method.

The preliminary experimental evaluation for the SAT based algorithm shows promising results. Possible directions for future works include (i) an alternative encoding of the SAT based algorithm using a smaller number of variables and (ii) a peephole optimization of small subnetworks in large MIGs using the SAT based method.

ACKNOWLEDGMENT

This research was supported by H2020-ERC-2014-ADG 669354 CyberCare.

REFERENCES

[1] I. Amlani, "Digital logic gate using quantum-dot cellular automata," *Science*, vol. 284, no. 5412, pp. 289–291, 1999.

[2] K. Kong, Y. Shang, and R. Lu, "An optimized majority logic synthesis methodology for quantum-dot cellular automata," *IEEE Trans. On Nanotechnology*, vol. 9, no. 2, pp. 170–183, 2010.

[3] T. Schneider, A. A. Serga, B. Leven, B. Hillebrands, R. L. Stamps, and M. P. Kostylev, "Realization of spin-wave logic gates," *Applied Physics Letters*, vol. 92, no. 2, pp. 1–4, 2008.

[4] O. Zografos, L. Amarú, P.-E. Gaillardon, P. Raghavan, and G. De Micheli, "Majority logic synthesis for spin wave technology," *Conference On Digital System Design*, pp. 691–694, 2014.

[5] Z. Pajouhi, S. Venkataramani, K. Yogendra, A. Raghunathan, and K. Roy, "Exploring spin-transfer-torque devices for logic applications," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. PP, no. 99, p. 1, 2015.

[6] E. Linn, R. Rosezin, C. Kügeler, and R. Waser, "Complementary resistive switches for passive nanocrossbar memories," *Nature materials*, vol. 9, no. 5, pp. 403–6, 2010.

[7] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (plim) computer," in *Design Automation and Test in Europe*, 2016.

[8] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs," *Design Automation and Test in Europe*, 2016.

[9] L. G. Amarú, P. Gaillardon, S. Mitra, and G. D. Micheli, "New logic synthesis as nanotechnology enabler," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2168–2195, 2015.

[10] S. B. Akers, "Synthesis of combinational logic using three-input majority gates," *Annual Symposium on Switching Circuit Theory and Logical Design*, pp. 149–158, 1962.

[11] R. Lindaman, "A theorem for deriving majority-logic networks within an augmented Boolean algebra," *IRE Trans. On Electronic Computers*, vol. 47, pp. 338–342, 1960.

[12] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: a new paradigm for logic optimization," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. PP, no. 99, pp. 1–14, 2015.

[13] L. Amarú, P.-E. Gaillardon, S. Mitra, and G. De Micheli, "New logic synthesis as nanotechnology enabler," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2168–2195, 2015.

[14] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-based synthesis for nanotechnologies," *Asia and South Pacific Design Automation Conference*, pp. 499–502, 2016.

[15] S. Muroga, *Threshold logic and its applications*. NY, New York: John Wiley & Sons Inc., 1971.

[16] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: a novel data-structure and algorithms for efficient logic optimization," *Design Automation Conference*, pp. 194:1 –194:6, 2014.

[17] ——, "Boolean logic optimization in majority-inverter graphs," *Design Automation Conference*, 2015.

[18] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[19] M. Soeken, L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Optimizing majority-inverter graphs with functional hashing," *Design Automation and Test in Europe*, 2016.

[20] C. Sinz, "Towards an optimal CNF encoding of Boolean cardinality constraints," *Conference on Principles and Practice of Constraint Programming*, pp. 827–831, 2005.
[21] N. Bjørner and L. de Moura, "Z3: An efficient SMT solver," *Tools and Algorithms for the Constructions and Analysis of Systems*, pp. 337–340, 2008.
[22] G. Rozenberg, Ed., *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
[23] A. Mishchenko, "Enumeration of irredundant circuit structures," *International Workshop on Logic Synthesis*, 2015.

Versatile SAT-based Remapping for Standard Cells

Alan Mishchenko Robert Brayton

Department of EECS, UC Berkeley

{alanmi, brayton}@berkeley.edu

Thierry Besson Sriram Govindarajan

Harm Arts Paul van Besouw

ICD Synthesis, Mentor Graphics

{Firstname_Lastname}@mentor.com

Abstract

The paper describes a versatile framework for optimizing a mapped netlist according to user-specified requirements to reduce a combination of area, delay, and power. A variety of delay models is supported. The framework is based on a SAT-based engine enumerating gate-level implementations of a node while using the node's full flexibility in the network. The enumeration favors considering low-cost alternatives early in the process. Full flexibility at a node is constructed as a Boolean circuit representing a Boolean relation, which is sampled by recursive cofactoring using different variable orders. Each variable order can result in an 8-input truth table of a function at the node, compatible with the relation. The computation is made efficient by reusing minterm samples of the relations at different nodes.

1. Introduction

In modern design flows a mapped netlist is often optimized using a cost function. For example, it is often desirable to reduce area while preserving timing, or to improve timing at the cost of a reasonable area increase. In both cases, the user provides a cost-function, which is used by the design flow to perform the required optimization.

This paper presents a versatile SAT-based framework that can be integrated into a design flow to enable user-guided remapping of a netlist with a specific goal in mind. The framework is versatile because it can work with practically any cost function. It is SAT-based because it relies on Boolean satisfiability [2] to compute alternative implementations of the target node.

To our knowledge, this is the first SAT-based framework, which efficiently explores the space of feasible standard-cell mappings. Previous work includes a BDD-based framework for standard-cell remapping [3] and a SAT-based framework for LUT-based remapping.

The proposed framework uses a novel SAT-based procedure to enumerate standard-cell implementations of the target node. The procedure performs recursive sampling of a Boolean circuit, representing the Boolean relation of the full flexibility of the target node and candidate divisors while taking don't-cares into account. The procedure exploits structural analysis, guided simulation, counter-examples derived by the SAT solver, and a cache of successful proofs to enable efficient exploration of the search space of alternative implementations.

The rest of the paper is organized as follows. Section 2 contains necessary background. Section 3 gives a top-level view of the remapping framework. Section 4 describes library preprocessing. Section 5 explains the SAT-based Boolean relation solver. Experimental results are given in Section 6. Section 7 concludes the paper.

2. Background

2.1 Boolean function

In this paper, *function* refers to a completely specified Boolean function $f(X)$: $B^n \rightarrow B$, $B = \{0,1\}$. The *support* of function f is the set of variables X, which influence the output value of f. The support size is denoted by $|X|$.

Expressions $\bar{x}$ and x are the *negative literal* and the *positive literal* of variable x, respectively. "Negative" and "positive" are *polarities* of variable x in the literals.

Boolean function $R(X, Y, Z)$ in terms of input variables X, intermediate variables Y, and output variables Z, can be seen as a characteristic function of a Boolean relation relating valuations of inputs X and outputs Z. Thus, valuation $(x_1, x_2, \ldots, x_n; z_1, z_2, \ldots, z_m)$ belongs to the relation if there exists a valuation $(y_1, y_2, \ldots, y_k)$ of intermediate variables such that $R(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_k, z_2, \ldots, z_m) = 1$.

In this paper, we will be using the SAT solver to check whether a given minterm $(x_1, x_2, \ldots, x_n; z_1, z_2, \ldots, z_m)$ belongs to the relation specified as a CNF formula.

2.2 Boolean network

A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to Boolean functions and edges corresponding to wires connecting the nodes.

A node n has zero or more fanins, i.e. nodes driving n, and zero or more fanouts, i.e. nodes driven by n. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes of the network, connecting it to the environment. A transitive fanin (fanout) cone (TFI/TFO) of node n is a subset of the network nodes, reachable through the fanin (fanout) edges of the node. If the network is sequential, it contains registers whose inputs/outputs are treated as additional POs/PIs.

Internal flexibilities of a node in a network arise because of limited controllability and observability. Lack of controllability occurs because some combinations of values are never produced at the fanins of the node. Lack of observability occurs because the node's value does not have impact on the values of the POs under some values of the PIs. Examples can be found in [7].

These internal flexibilities result in *don't-cares* at the node *n*. The complement of the don't-cares is the *care set*.

Given a network with PIs *x* and PO functions $\{z_i(x)\}$, the care set $C_n(x)$ of a node *n* is a Boolean function of the PIs:

$$C_n(x) = \sum_i [z_i(x) \oplus z_i'(x)]$$

where $z_i'(x)$ are the POs in a copy of the network but with node *n* is complemented, as shown in Figure 2 [7].

2.3 Mapped network

A *mapped network* is a Boolean network whose nodes are associated with cells (gates) from the given standard cell library represented in Liberty format [4]. Each cell in the library is characterized by its name, Boolean function, and various parameters. The parameters relevant for this work, are cell area and delay information.

Area is a floating point number associated with each cell. Delay information is given as a set of two-dimensional tables specifying delay and slew at the output of the cell as functions of capacitance and slew at each of the inputs.

The presented remapping engine uses a load-independent abstraction of the delay information, which approximates the delay at the output using the delay at each of the inputs based on the gain-based approach [10]. The timing model can be extended to handle a load-dependent delay model; however, this extension is beyond the scope of this paper.

A typical standard-cell library contains multiple cells, with the same function, that differ only in cell size. In this work it is assumed that each cell has only one size. Removing this restriction is left for future work.

Finally, although a typical library contains sequential cells, this paper is limited to combinational logic mapping.

2.4 Boolean satisfiability

A *satisfiability problem* (SAT) takes a propositional formula representing a Boolean function and decides if the formula is satisfiable or not. The formula is *satisfiable* (SAT) if there is an assignment of variables that evaluates the formula to 1. Otherwise, the formula is *unsatisfiable* (UNSAT). A software program that solves SAT problems is called a *SAT solver*. SAT solvers provide a satisfying assignment when the problem is satisfiable.

Modern SAT solvers can accept assumptions, which are single-literal clauses holding for one call to the SAT solver. The process of determining the satisfiability of a problem under given assumptions is called *incremental SAT solving*.

2.5 Conjunctive Normal Form (CNF)

To represent a propositional formula for the SAT solver, important aspects of the problem are encoded using Boolean *variables*. The presence or absence of a given aspect is represented by a positive or negative *literal* of the variable. A disjunction of literals is called a *clause*. A conjunction of clauses is called a *CNF*. CNFs can be processed efficiently by mainstream CNF-based SAT solvers, such as MiniSAT [2], used in this work.

Deriving a CNF for a subset of nodes of the Boolean network is performed by putting together CNFs obtained by converting each node. A CNF for a node is derived by deriving SOPs of the on-set and off-set of the Boolean function of the node, and converting these SOPs into CNF using the De Morgan rule.

3. Remapping framework

Remapping is done by a software framework designed to perform several interdependent tasks. The computation begins by preprocessing the standard cell library, which involves collecting functions up to 8 inputs realizable using at most two cells from the library.

Next, remapping considers nodes in an order determined by the optimization goals. For example, area optimization is performed by processing nodes in a topological order, while delay optimization is performed by computing the critical paths and optimizing nodes on the paths.

For each target node, the remapping framework performs the following steps, as shown in Figure 1:

- It computes a structural window centered around a target node and containing nodes in the limited TFI/TFO of the target node, along with nodes found on the reconvergent paths.
- It identifies a subset of nodes in the window, which can be used as divisors to express the target node. It orders the candidate divisors in reduced desirability. For example, for delay optimization, the order first lists the nodes having short delay and low criticality.
- It converts the window into CNF and derives a SAT instance representing the complete flexibility of the target node [7]. The SAT variables corresponding to the target node and those of the candidate divisors are identified and stored. These will be used to express assumptions during subsequent SAT solving.
- The SAT instance is used to derive one or more implementations of the target node using candidate divisors, as described in Section 5.
- The implementations are matched by Matcher with the current pre-processed library, timed by Timer using the selected timing model, and compared by Evaluator against the current implementation using the given cost-function, as shown in Figure 1.

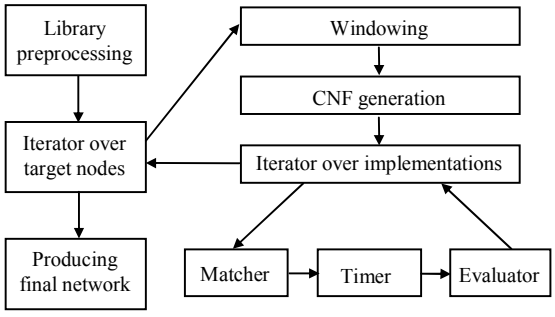


Figure 1. Illustration of a miter used in CDC computation.

The best implementations among those improving the cost functions, are used to replace the current target node. When the optimization is finished or a timeout occurs, the final network is produced and returned to the user.

4. Standard-cell library preprocessing

A typical standard-cell library contains cells with different functionalities and sizes. In the preprocessing phase, representative cells are selected from each functionality class and resource-aware enumeration of cell pairs is done, deriving two-cell super-cells with support up

to 8 inputs [6]. The resulting super-cells are hashed using their truth tables as hash keys while keeping track of their area and pin-to-pin delays. Super-cells that are identical to (or dominated by) other super-cells are discarded.

The preprocessing takes a fraction of a second and is performed only once for a given library even if remapping is performed repeatedly or for several designs.

In principle, the preprocessing can compute super-cells with more than 8 inputs containing more than two cells. However, experiments indicate that this is unlikely to improve the quality of results, while pre-computation time and memory used for storage, could increase dramatically. This might make it necessary to do this offline and then to read in the result from a file.

5. SAT-based Boolean relation solver

5.1 Constructing SAT instance

The SAT instance is constructed by deriving CNF for the Boolean relation representing the care set of the target node in the structural window carved out in the network. The window contains a fixed number of TFI/TFO levels of logic centered at the node, plus all paths originating in the limited TFI and terminating in the limited TFO.

The computation of care set is based on the formulation used for node optimization with don't-cares [7] illustrated in Figure 2. The circuit contains two copies of the node's TFO, one of which has an inverter inserted at the node n output. The comparator XORs are added for the pairs of corresponding outputs and the output of the comparator's OR gate is assumed to be 1, meaning that complementing the node's function makes a difference at the POs, and thus the constructed circuit represents the care set of the node.

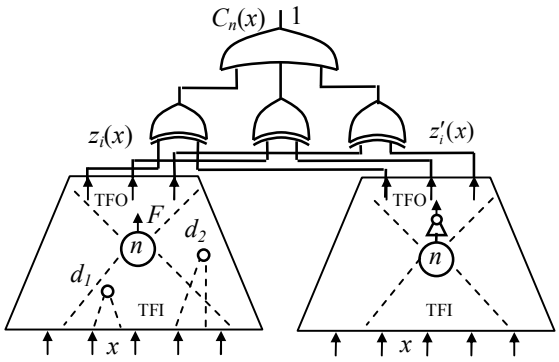


Figure 2. Deriving Boolean relation for computing multiple implementations of the target node in terms of candidate divisors.

The Boolean relation represented by the circuit structure in Figure 2 relates the function of the node n with functions of candidate divisors whose support is a subset of the support of the node's TFI, such as divisors d_1 and d_2, shown at the bottom of Figure 2.

5.2 Deriving a feasible implementation of the node

Figure 3 contains the pseudo-code of procedure SolveBR_rec() used to derive one implementation of the target node that is in agreement with the Boolean relation. Recursive calls to SolveBR_rec() (Line 5) build the resulting implementation as a co-factoring tree whose nodes are multiplexers and leaves are constants or elementary variables. The tree is collapsed on the fly and its function is returned as a truth table. To reduce the clutter in the pseudo-code, it is not shown that the procedure returns "None" if the support has more than 8 variables.

The procedure SolveBR_rec() takes an ordered set of unassigned divisors and a set of divisors currently assigned a value. The assigned divisors are used as assumptions in the SAT calls, which check if the cofactors of Boolean relation w.r.t. the divisor variables are UNSAT. For example, if the following three cofactors are all UNSAT: (1) $F = 0$ & $d_1 = 1$ & $d_2 = 1$, (2) $F = 1$ & $d_1 = 0$, (3) $F = 1$ & $d_2 = 0$, the node can be implemented as $F = \text{AND}(d_1, d_2)$.

Procedure SolveBR_rec() in Figure 3 begins by detecting trivial cases (Lines 1 and 2). In particular, if constraining the node's function F to a constant is UNSAT under the assumptions, the complement constant function is returned. Next (Line 3), the procedure tries to implement the cofactor of a target node as a candidate divisor d. This can be done if the following two conditions are UNSAT: (1) $F = 0$ & $d = 1$ and (2) $F = 1$ & $d = 0$. To prove that the target node can be expressed as an inverter, we show that the following are UNSAT: (1) $F = 0$ & $d = 0$ and (2) $F = 1$ & $d = 1$.

Finally, if the node's function is not a constant and cannot be expressed in terms of one divisor, the next unassigned variable is selected for cofactoring (Line 4). The resulting functions computed for the two cofactors are multiplexed with the given variable and returned (Line 5). Note that forthcoming cofactoring is effected by adding the selected variable or its complement to the set of assigned variables.

Multiple implementations of the target node can be derived using procedure SolveBR(), shown in Figure 3. The implementations differ because different variable orders are used to cofactor the Boolean relation. It is possible that two different variable orders lead to the same implementation, but in practice such situations are rare. Each implementation is checked against the pre-computed super-cell library for a Boolean match and evaluated according to the cost function.

5.3 Using counter-examples to speed up search

The input space of the Boolean relation is the set of all candidate divisors, $\{d_i\}$. The output space is the target node, F. All other variables used to represent the relation in the SAT solver are treated as intermediate variables, according to the discussion in Section 2.1.

Counter-examples produced by the SAT solver are either care set minterms of the function of the target node, according to the Boolean relation. These minterm samples are collected and represented as two *sample matrices*, one for the onset and one for the offset. The rows of the sample matrices represent minterms while the columns represent candidate divisors. The matrices are reused while working on the same node.

While enumerating feasible implementations of the node, we will check satisfiability of many cofactors of the Boolean relation. Recursive cofactoring of the relation is mirrored in our implementation by recursive splitting of the onset/offset sample matrices. This amounts to selecting a subset of samples that agree with the current values assigned to the divisor variables. Only when an onset/offset sample matrix is empty (there are no samples that agree

with the assigned divisors), is the SAT solver run to check if the cofactor is a constant, or if it is satisfiable. In the latter case, the new minterm provided by the counter-example is added to the appropriate sample matrix.

A significant reduction in runtime (2-5x) is obtained due to the use of the sample matrices in this application.

5.4 Reusing counter-examples across windows

In many SAT-based applications, including this one, the runtime is dominated by satisfiable SAT calls. Each SAT call produces a relevant on-set or off-set minterm of the target node in terms of candidate divisors and helps us converge on a feasible implementation of the node.

The number of satisfiable SAT calls can be achieved by seeding the sample matrices described in Section 5.3 with care-set minterms computed using random simulation. However, it was found experimentally that randomly generated minterms are not as useful, for guiding the search, as minterms generated by the SAT solver. Thus, when we move to work on a new target node, instead of simulating the node's window randomly, we re-simulate counter-examples computed by the SAT solver for previous nodes. To this end, when optimization of a node is completed, we store counter-examples generated for this node in an internal data-structure. When a new node is selected, the backed-up counter-examples are retrieved and re-simulated through the new node's window. At the end of the simulation, those counter-examples that do not belong to the care-set of the new node are removed while the remaining ones are used to seed the sample matrices.

5.5 Caching successful proofs

While trying to find a feasible implementation of the target node, the SAT solver is called repeatedly with different sets of assumptions. Each of these calls checks the satisfiability of one cofactor of the Boolean relation. Since different unrelated variable orders are used, often the same cofactors are checked multiple times. To avoid duplicated SAT calls, a cache of *unsatisfiable* calls can be maintained. The unsatisfiable calls are cached by remembering sets of assumptions that were tried and found unsatisfiable.

6. Experimental results

The proposed optimization framework for standard cells was implemented as command *mfs3* in ABC [1]. This command differs from two optimization frameworks developed earlier for LUT-based FPGAs (command *mfs* and its improved version, *mfs2*) [8].

The main difference between optimization for K-LUTs and that for standard cells, is that the former expresses the target node using any K-input function, while the proposed framework uses only functions expressible in terms of standard cells. Expressing the resulting functions using standard cells requires a dedicated SAT-based engine introduced in this paper.

The experimental evaluation reported in this section is meant to illustrate the use of the framework and to show its scalability; it is not meant to be a comprehensive evaluation of remapping with different cost functions. Thus, the experimental results are limited to area-oriented optimization using typical cells from an industrial library where a unit-area model has been imposed. Under these assumptions, only the number of cells is optimized, without considering the actual area and delay listed in the library.

The benchmarks considered are ten combinational logic cones used in earlier publications by the first author [9]. The benchmarks were preprocessed by logic synthesis and mapped into the target library using two area-oriented mappers in ABC (commands *amap* and *&nf –R 1000*). The proposed remapping engine is applied in each case as a post-processing step. Combinational equivalence checking was performed using command *&cec* in ABC.

The results are shown in Table 1. The number of primary inputs and outputs is reported in columns *Inputs* and *Outputs*. The number of cells after mapping is reported in columns *amap* and *&nf*. The number of cells after remapping is reported in columns *+mfs3*. The table shows that the proposed engine has reduced the number of cells by 2.5% after running *amap* and by 0.5% after running *&nf*. It should be noted that these improvements are on top of well-tuned heuristic mappers, which perform area-optimization mode with the same unit-area version of the library. Moreover, calling the mappers repeatedly without synthesis does not produce similar improvements.

In this experiment, command *mfs3* was used with command line options (*mfs3 -ae -I 4 -O 2*) selected to limit the scope of optimization to 4 levels of TFI and 2 levels of TFO. With these options, the total runtime of *mfs3* is close to that of each of the mappers (*amap* and *&nf*) and was about 3 minutes for all the logic cones listed in Table 1.

The transcript of running *mfs3* for one testcase (*ex02*) from Table 1 is shown in Figure 4. Besides numerous internal parameters, the transcript shows the runtime breakdown for individual tasks performed by the optimizer. In particular, the SAT solver performs 5.4 million incremental calls while trying to optimize 34,817 nodes with changes introduced to 8,764 nodes. The SAT solver runtime accounts for the 97% of the total time (11 seconds), meaning that the framework is running with little overhead on top of the SAT solver.

7. Conclusions

The paper describes a novel SAT-based optimization framework applicable to netlists mapped into standard cells. The engine considers one node at a time, and looks for alternative implementations of the node using one or two cells from the library. The implementations are evaluated using a cost-function and the best is accepted, before moving on to the next node. The engine is novel in that it expresses implementations of a node in terms of standard cells from a given library rather than arbitrary functions [8]. Efficient implementation is based on incremental SAT used to sample the Boolean relation of a node's full flexibility represented as a CNF formula.

Future work will focus on customizing the framework to support different cost functions and on developing other flavors of SAT-based mapping. It may be also interesting to optimize the implementation by using cubes rather then minterms in representing the sampling matrix, which may lead to improved runtime and better quality of results.

8. REFERENCES

[1] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[2] N. Een and N. Sörensson, "An extensible SAT-solver", *Proc. SAT'03*, LNCS 2919, pp. 502-518.

[3] V. N. Kravets and P. Kudva, "Implicit enumeration of structural changes in circuit optimization". *Proc. DAC'04*, pp. 438-441.

[4] Liberty Format. https://www.opensourceliberty.org/

[5] A. Malik, R. K. Brayton, A. R. Newton, and A. Singiovanni-Vincentelli, "Two-level minimization of multi-valued functions with large offsets", *IEEE TCAD'91*, Vol. 10(4), pp. 413-424.

[6] A. Mishchenko, X. Wang, and T. Kam, "A new enhanced constructive decomposition and mapping algorithm", *Proc. DAC '03*, pp. 143-148.

[7] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization", *DATE '05*, pp. 418-423.

[8] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang. "Scalable don't-care-based logic optimization and resynthesis", *ACM TRETS*, Vol. 4(4), April 2011, Article 34.

[9] A. Mishchenko, "Enumeration of irredundant circuit structures", *Proc. IWLS'14*.

[10] I. E. Sutherland and R. F. Sproull, "Logical effort: designing for speed on the back of an envelope", *Proc. VLSI'91*, p.1-16.


```
one_boolean_function SolveBR_rec( ordered set of divisors D, assigned divisors A, cnf C ) {
    if ( F == 1 is UNSAT under assumptions in A ) return 0;  // constant 0 Boolean function   (Line 1)
    if ( F == 0 is UNSAT under assumptions in A ) return 1;  // constant 1 Boolean function   (Line 2)
    if ( F == d_i or F == d_i' for some divisor d_i in D under assumptions in A )  return d_i or d_i' ;  (Line 3)
    d = next variable in D;                                                                  (Line 4)
    return d ? SolveBR_rec( D \ d, A ∪ d, C ) : SolveBR_rec( D \ d, A ∪ d_i', C );           (Line 5)
}
set_of_boolean_functions SolveBR( number of functions N, set of divisors D, cnf C ) {
    result = ∅;
    for ( i = 0; i < N; i = i + 1 ) {
        generate a new ordering of divisors in D;
        result = result ∪ SolveBR_rec( D, ∅, C );
    }
    return result;
}
```

Figure 3: Pseudo-code of SAT-based solving of Boolean relation during remapping.

```
abc 01> r ex02.aig;  amap; ps;  mfs3 -aev -I 4 -O 2; ps;  time; echo
ex02 : i/o =25237/18422  lat =    0  nd = 34817  edge = 110193  area =34817.00  delay =308.95  lev = 13
Library processing: Var = 6. Cell = 71. Fun = 38660. Obj = 38660. Ave = 1.00. Skip = 0. Time = 0.07 sec
Remapping parameters: TFO = 2. TFI = 4. FanMax = 10. MffcMin = 1. MffcMax = 3. DecMax = 1. Effort = yes.
Node = 34817. Try = 34817. Change = 8764.   Buf = 44. Inv = 7655. Gate = 1065. AndOr = 0. Effort = 962.
MaxDiv = 111. MaxWin = 136.   AveDiv = 8. AveWin = 11.   Calls = 5442733. (Sat = 2698400. Unsat =
2744333.)
Lib    =    0.07 sec (  0.62 %)
Win    =    0.12 sec (  1.06 %)
Cnf    =    0.10 sec (  0.88 %)
Sat    =   10.98 sec ( 97.08 %)
 Sat   =    8.19 sec ( 72.41 %)
 Unsat =    1.73 sec ( 15.30 %)
Eval   =    0.00 sec (  0.00 %)
Timing =    0.00 sec (  0.00 %)
Other  =    0.04 sec (  0.35 %)
ALL    =   11.31 sec (100.00 %)
Cone sizes:  1=7699  2=5  3=164  4=34  5=861  6=1    Gate sizes:  1=7867  2=897
Reduction:   Nodes    1210 out of  34817 (  3.48 %)   Edges    1702 out of 110193 (  1.54 %)
ex02  : i/o =25237/18422  lat =    0  nd = 33607  edge = 108491  area =33607.00  delay =308.95  lev = 13
```

Figure 4: The transcript produced by running "mfs3" during area-only remapping of a design.

Table 1: Applying area-oriented remapping after area-oriented mapping using two mappers.

Benchmark	Inputs	Outputs	amap	+mfs3	&nf	+mfs3
Ex01	13601	13601	54795	54058	49127	48879
Ex02	25237	18422	34817	33607	26696	26587
Ex03	16300	11243	51169	50435	44783	44408
Ex04	28341	26312	80846	80505	80547	80434
Ex05	26380	22788	82100	81196	71887	71654
Ex06	52711	45549	164256	162319	143673	143206
Ex07	18677	12441	56600	55853	48734	48399
Ex08	15780	12892	78558	72502	53348	52976
Ex09	36087	31407	56949	55190	44856	44764
Ex10	11400	10896	23945	23151	20296	20160
Geomean1			1.000	0.975	0.842	0.838
Geomean2					1.000	0.995

Boosting the Performance of MapReduce Applications via Distributed Accelerators on a Chip-Multiprocessor

Abraham Addisie†, Rawan Abdel-Khalek†, Ritesh Parikh‡, and Valeria Bertacco†

†University of Michigan
{abrahad, rawanak, valeria}@umich.edu

‡Intel Corporation
parikh@intel.com

ABSTRACT

MapReduce is a commonly-used programming model that provides a simple and high-performance implementation of data-intensive applications, by separating the workload into a map stage, where data is organized into a uniform key-value pairs format, and a reduce stage, where these key-value pairs are aggregated to generate the desired outcome. The execution of MapReduce on chip multi-processors (CMP) allows distributing the workload over multiple cores, and thus boosting its performance. However, common CMP-based implementations entail the use and management of complex data structures, which limit the performance benefits enabled by the parallel architecture.

In this work, we propose to equip each core in the design with a hardware accelerator module that frees the core from the frequent memory accesses and the hash function computations required by the MapReduce framework. The distributed accelerators provide a logically unified large memory, where most key-value pairs from all cores can be aggregated and stored. We keep spilling from this unified memory to a minimum by leveraging a novel mechanism that prioritizes space for the most frequent keys. Our experimental evaluation on a 64-core design indicates that our solution provides over a 3 times speedup, averaged over several applications, compared with a best-in-class software implementation, reduces network traffic by at least 50% over prior accelerated solutions, and keeps the area overhead against the baseline CMP below 10%.

1. INTRODUCTION

MapReduce is a programming paradigm that facilitates the parallel processing of large data sets and provides programmers with a simple abstraction to implement a wide range of data-intensive applications. Today, MapReduce frameworks are deployed in a wide range of result- and performance-critical applications, *e.g.*: image processing, web-search engines, genomics, *etc.* In image processing, image recognition is powered by MapReduce applications, with impact in domains ranging from security to medical and beyond. In web-search applications, MapReduce is deployed in a wide variety of algorithms that deliver the search speed and quality that we have come to expect today: finding search keywords in pages, ranking pages when presenting search results, *etc.* Another example application of MapReduce is in genome sequencing: today it is possible to sequence a sample of human genome in just 26 hours [1]. While this result shows great progress in making genome sequencing widely available, the performance of this application is still too poor to, for instance, allow the real-time sequencing of frequently mutating tissue, such as cancer tissue, which in turn would enable more effective treatments. The massively parallel problem underpinning genome sequencing is equivalent to *wordcount*, that is, counting the number of occurrences of each term (bases) in a word bank (DNA sequence), a classic application of MapReduce.

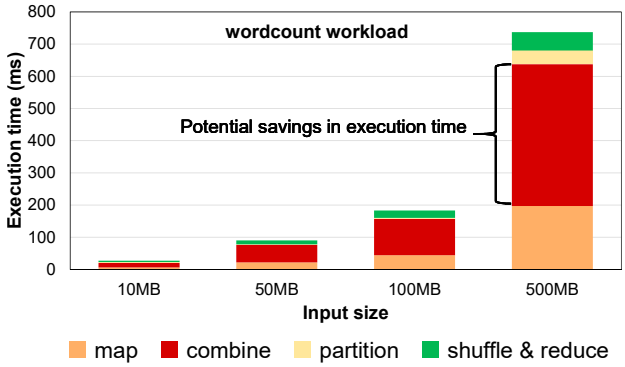


Figure 1: **Motivating study**. The chart plots the execution time's breakdown for *wordcount* with a range of input dataset sizes, running on the Phoenix++ framework [2] on an 8-core Intel Core i7-4790K machine. Each stage was isolated by leveraging the synchronization barrier (see Figure 3) and by executing multiple times with distinct termination points. Note that, while we measure the longest time among all cores, in practice the load is fairly balanced and each core computes approximately for the same amount of time in each stage. As it is evident form the plot, the combine stage drastically dominates the overall execution. By eliminating this stage alone, the overall execution time would be slashed by over 2x.

The goal of our work is to boost the performance of MapReduce applications running on multi-core architectures by leveraging a network of distributed hardware accelerators, thus breaking the performance barriers that make some of the problems above impractical.

MapReduce abstracts a workload through two basic primitives: a map function that parses the input

data and emits it in the form of intermediate key-value (kv) pairs and a reduce function that aggregates each set of intermediate kv-pairs associated with the same key. In a CMP-based framework, input data is allocated equally to each core, which then applies both the ***map*** function on each input, and the ***combine*** stage, aggregating data locally as much as possible. The kv-pairs obtained are then ***partitioned*** among the cores, so that each one is responsible for carrying out the reduce function on some of the keys. The kv-pairs are then ***shuffled*** through the interconnect and ***reduced*** in a distributed fashion.

To illustrate this flow, on the left side of Figure 3, we show how the various stages of MapReduce are sequenced and synchronized among the cores in a state-of-the-art CMP-based framework, as is Phoenix++ [2].

Motivating Study. As a motivating study, we analyzed the execution of MapReduce on a *wordcount* application, over a range of input data sizes. Figure 1 plots the distribution of the execution time of each of the stages listed above for each of the data sets considered. In this study the application was implemented in the Phoenix++ framework [2], a state-of-the-art framework for MapReduce on chip multi-processors. We ran the experiment on an 8-core Intel i7-4790K and we derived the time taken by each stage by completing multiple runs, where we excluded all but one stage (all but two in the case of shuffle & reduce). For instance, to compute the execution time of the combine stage, we first measured the time to reach the synchronization barrier at the end of the combine stage. Then we executed the framework again, but forced each core to execute only the map stage. By difference, we then derived the time spent in the combine stage alone.

The findings of our study suggest that the greatest opportunity for improvement lies in eliminating the combine stage, which accounts for over half of the total execution time. Thus, the goal of our solution is to offload this stage completely onto the hardware accelerators, so that it can be executed concurrently with the map stage running on the cores. We further managed to offload also the partition and reduce stages to the accelerators, bringing further performance benefits to our solution.

Contributions. In our proposed solution, each CMP node is augmented with an accelerator module, which also interfaces with the on-chip interconnect and the memory subsystem, as illustrated in Figure 2. We called our solution MR.NITRO: MapReduce with NITRO speed-boosting, as in when adding nitro-oxide to a vehicle's engine. Similarly to purely software implementations, each core is tasked with running the map function. However, as soon as a kv-pair is generated, it is emitted to the local accelerator to be aggregated and stored. Each accelerator consists of a scratchpad memory to store and manage intermediate kv-pairs, and data access to the scratchpad memory occurs through a hardware-based hash function. Upon receiving a kv-pair from its associated core, the accelerator aggregates it with any previously-stored data that belongs to the same key.

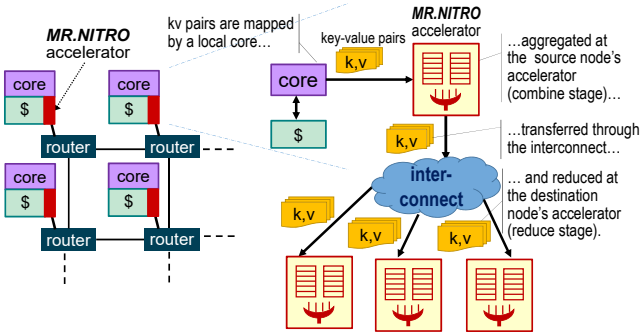


Figure 2: **MR.NITRO deployed in a CMP architecture**. Left side - MR.NITRO adds a local accelerator to each CMP's node to carry out MapReduce tasks. Right side - Each accelerator is responsible for completing the combine stage, aggregating kv-pairs emitted by the local core (source aggregation). Source-aggregated kv-pairs are then transferred among the accelerators through the CMP's interconnect. At the destination node, accelerators execute the reduce stage, so that there is only one kv-pair per key.

When two keys are hashed to the same scratchpad entry and thus collide, we rely on a novel frequency-based replacement policy that prioritizes the storage of frequently occurring keys over others. kv-pairs that fail to gain/maintain space in the scratchpad are first transfered to a local victim scratchpad, and then forwarded to the destination's node accelerator (*i.e.*, the node assigned to reduce all data associated with that unique key) through the CMP's interconnect.

Thus, while the CMP's general-purpose cores are executing the map stage, MR.NITRO's accelerators carry out the combine stage in the background, incrementally aggregating kv-pairs as they are generated. By offloading the combine stage to the hardware accelerator, we avoid slow software computations of hash functions and complex data structures. Note that the scratchpad memories within the accelerators effectively constitute a large unified memory where we can aggregate most kv-pairs. Compared to a single-accelerator solution, MR.NITRO aggregates kv-pairs at the source nodes, thus greatly reducing the number of data transfers across the interconnect.

In summary, MR.NITRO makes the following novel contributions:

- A novel distributed-accelerator solution for MapReduce, providing over a 3 times performance improvement on average, compared to chip multiprocessor frameworks.
- Our distributed accelerators partially reduce intermediate kv-pairs at their source nodes, saving significant interconnect traffic by reducing transfers to the remote accelerators.
- MR.NITRO is transparent to the MapReduce programming interface, thus it preserves its simplicity to the application's developer. Its silicon area overhead consists of the accelerators' silicon footprint, which we estimated to be <10% of the local node's cache area.

2. RELATED WORK

The MapReduce programming model was originally introduced by Google [3] to provide efficient execution of data-intensive applications on a cluster of commodity-machines. Hadoop [4] is also a popular cluster-based open-source framework that implements MapReduce. In addition, [5] uses an MPI-based MapReduce framework, to benefit from the rich MPI collective functions. Our solution can be easily deployed on MapReduce built for MPI and other cluster-based frameworks.

With the adoption of chip-multiprocessor (CMP) architectures, several MapReduce frameworks have been proposed targeting these systems [6, 7, 8, 9, 10, 11, 2]. In particular, Phoenix++ [2] is an optimized implementation of MapReduce for multi-core systems. It provides a simple programming interface for users, while internally managing the execution of the MapReduce tasks. However, for most applications, Phoenix++ relies on the use of complex data structures and software-based hash functions, limiting the performance gains that can be attained on these distributed architectures. MR.NITRO offloads the time-consuming and complex data accesses, as well as the reduce function execution, to special-purpose accelerators. Therefore, we achieve better performance on all types of workloads, and are able to run several MapReduce stages concurrently.

In addition, MapReduce has been increasingly adopted on Graphic Processing Units (GPUs) computing platforms, such as [12, 13]. While GPUs provide massive data-level parallelism, they suffer from several limitations related to memory management. In particular, GPU architectures do not support efficient dynamic memory allocation and the overhead of frequently transferring data between the CPU and the GPU limits the performance gains that can be achieved.

Finally, there are several works on accelerating MapReduce using FPGA platforms. The work in [14] implements the complete MapReduce framework, but it performs poorly because it does not leverage partial reduction of intermediate kv-pairs to achieve better performance. In contrast, MR.NITRO accelerates both the combine and reduce stages using a distributed network of accelerators. [15] offloads the reduce stage to a single, centralized FPGA-based accelerator. This accelerator receives mapped kv-pairs from all cores. Then it performs in-hardware aggregation and stores final results into its internal memory. The main drawback of this solution is that it is not scalable over a large number of cores. It also uses a cuckoo hash function [16] to implement key lookups. The function is applied repetitively until a free entry in the scratchpad (organized as a hashtable) is found. While this approach provides a better usage of the bounded-size scratchpad, it potentially leads to infinite loops of hash function computations[15]. On the other hand, we provide a scalable and distributed accelerator solution, where each CMP core is augmented with an accelerator module. Our accelerators perform aggregation (that is, partial reduction) of kv-pairs at the source node, which is then completed in the reduce stage. Therefore, we achieve performance improvements, while also reducing the number of kv-pairs exchanged among the cores over the network.

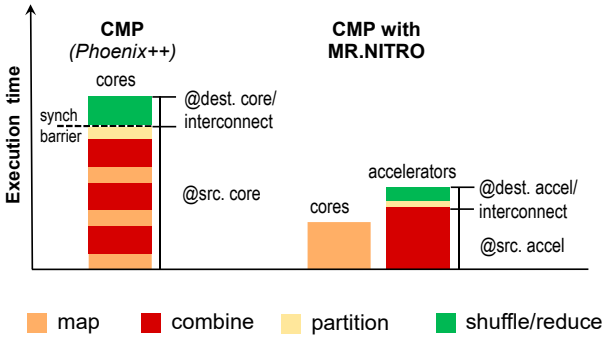


Figure 3: **Execution flow of MapReduce**. Left side - on a typical CMP-based framework, the map and combine stages are carried for each kv-pair by the source core, then keys are partitioned over all destination cores. A barrier synchronizes execution at the end of this stage. Finally, kv-pairs are shuffled through the interconnect and reduced at the destination core. Right-side - When deploying MR.NITRO, cores are only responsible for the map stage. They then transfer kv-pairs to the local accelerator, which combines them, partitions them and transfers them to the destination accelerator for the final reduce stage.

3. MAPREDUCE BACKGROUND

MapReduce provides a simple programming model for data-intensive applications. Users can write complex parallel programs by simply defining a map and a reduce function. Whereas, the remaining aspects of parallel programming, including the data combine and data shuffle stages, are handled by the MapReduce framework. In a typical framework, the application's input data is first partitioned among the cores in the system. Then, each core runs the user-defined map function, which processes the input data and produces a list of intermediate kv-pairs. Once the map stage is complete, the intermediate data is shuffled within the network and the cores assume the role of reducers, such that all kv-pairs with the same key are transferred to the same reducer. In the final stage, each core executes the user-defined reduce function to complete the aggregation of kv-pairs received from the source nodes. MapReduce implementations often introduce a combine stage between the map and shuffle stages to partially reduce the intermediate kv-pairs and thus conserve network bandwidth. The left part of Figure 3 illustrates this sequence of steps on a CMP-based framework with a time diagram: at first each core considers its portion of data, one data input at a time, maps it into a kv-pair (orange) and combines with its current database of kv-pairs (red). Then it partition its aggregated kv-pairs over all the cores based on their key (yellow). At this point the concurrent program is synchronized so that cores switch from operating as source nodes to operating as destination nodes. kv-pairs are then transferred through the interconnect to the destination node and reduced there (green).

wordcount is a classic application example: it computes the frequency of occurrence of each word in a document. It is leveraged in a wide range of practical appli-

```
map(doc):
for each word w in doc
   emitKeyValue(w,1);
```

```
reduce(key,values):
result = 0;
for each v in values
   result += v;
emitFinal(key,result);
```

Figure 4: **The map and reduce functions for *wordcount*.** The map function emits a kv-pair for each word in the document. The reduce function aggregates the word counts and emits the final sum.

cations, such as search engines and social networks (*e.g.* indexing, identifying trending topics), genome sequencing, artificial intelligence, image processing, *etc.* Figure 4 provides the pseudo-code for its map and reduce functions: the map function parses the input document based on the user-defined word delimiter and identifies all words. It then emits each word as part of a kv-pair, where the word is the key and the initial value is 1. The reduce function collects all kv-pairs emitted from the cores and sums up the values for each word, generating a final list of unique words together with the number of times they occurred in the input document.

4. THE ACCELERATOR DESIGN

The focus of this work is to boost the performance of MapReduce applications by leveraging hardware acceleration. In general, CPU cores are most suitable in managing data transfers and organization, while systematic data processing can be completed more efficiently through a dedicated hardware structure, as an accelerator. Consequently, in our solution, the map stage of MapReduce is carried out by CMP's cores, which retrieve the input data from memory, parse it and emit kv-pairs directly to their local accelerator. The accelerator then leverages its scratchpad memory to aggregate the pairs' values. Once all kv-pairs have been received and combined by the source node's accelerator, each source accelerator computes independently the partitioning of keys over the destination accelerators. It then proceeds in transferring all its kv-pairs to the correct destination which, in turn, leverages its scratchpad memory to complete the reduce stage. Note that the same hardware module serves both the roles of source and destination accelerator simultaneously.

The right part of Figure 3 illustrates the flow described by showing where each stage of the computation occurs when deploying our MR.NITRO solution. Note that in our solution no synchronization barrier is necessary, since each accelerator can operate concurrently as a source and destination accelerator. Eliminating the barrier, not only simplifies the execution, but also distributes the interconnect's traffic over longer periods of time, reducing the exposure to congestion delays.

Each MR.NITRO's accelerator comprises a scratchpad memory, organized into two partitions, one to serve kv-pairs incoming from the local processor core (source scratchpad), and one to serve kv-pairs incoming from the interconnect (destination scratchpad). Each scratchpad is completed by a small "victim scratchpad", similar to a victim cache that stores data recently evicted from the main scratchpad. The accelerator also includes dedicated hardware to compute a range of reduce functions, used both to aggregate data in the source scratchpad and in the destination one. Logic to compute hash functions, both for indexing the scratchpads and for partitioning kv-pairs over destination nodes, completes the design.

Note that both source and destination scratchpad memories are of fixed size, as a result it may not be always possible for them to store all the kv-pairs that they receive. When a source scratchpad cannot fit all the kv-pairs, it defers their aggregation to the reduce stage, by transferring them to the destination accelerator. When a destination scratchpad encounters this problem, it transfers its kv-pairs to main memory, and then lets the destination core be in charge of carrying out the last few final steps of the reduce function. Both scratchpads resort first to a small victim scratchpad before folding into transferring out kv-pairs, as we discuss in detail in Section 4.2.

Figure 5 provides a schematic of the architecture described. When an accelerator receives a kv-pair from either its associated core or the network, it first processes the key through its *key hash unit* and through the *partition stage unit*. The purpose of the former is to generate a hash value from the key, and use it to index the scratchpad memory. The latter determines which destination node is responsible for reducing this kv-pair: if the local node is also the destination node (*node_is_dest*), then we send the kv-pair to the destination scratchpad, along with the hash index and an enable signal, otherwise we send it to the source scratchpad. Note that all kv-pairs incoming from the network will be aggregated at the destination scratchpad. In addition, some of the pairs incoming from the local core may also be aggregated at the destination scratchpad, if the local core is both the source and the destination for that pair. Each scratchpad is internally organized as a 2-way cache augmented with a small victim cache. Associated with each scratchpad is an *aggregate unit*, responsible for deploying the specified reduce function to combine two kv-pairs with the same key. Each scratchpad is also equipped with a dedicated unit, called *frequency/collision update unit*, to keep up to date the replacement policy information. Finally note that, when a kv-pair is spilled from a scratchpad, it is transferred out through the network, either to another compute node or to memory. kv-pair may be spilled because it is evicted from the victim scratchpad or because it collided with another entry in the scratchpad and did not qualify to replace it.

Below we discuss in detail the operation of the *key hash* and *partition stage units* in Section 4.1, of the *scratchpad unit* in Section 4.2, and of the *aggregate unit* in Section 4.3.

4.1 Hash Function Units

Our accelerator includes two units computing hash functions: the *key hash unit* and the *partition stage*

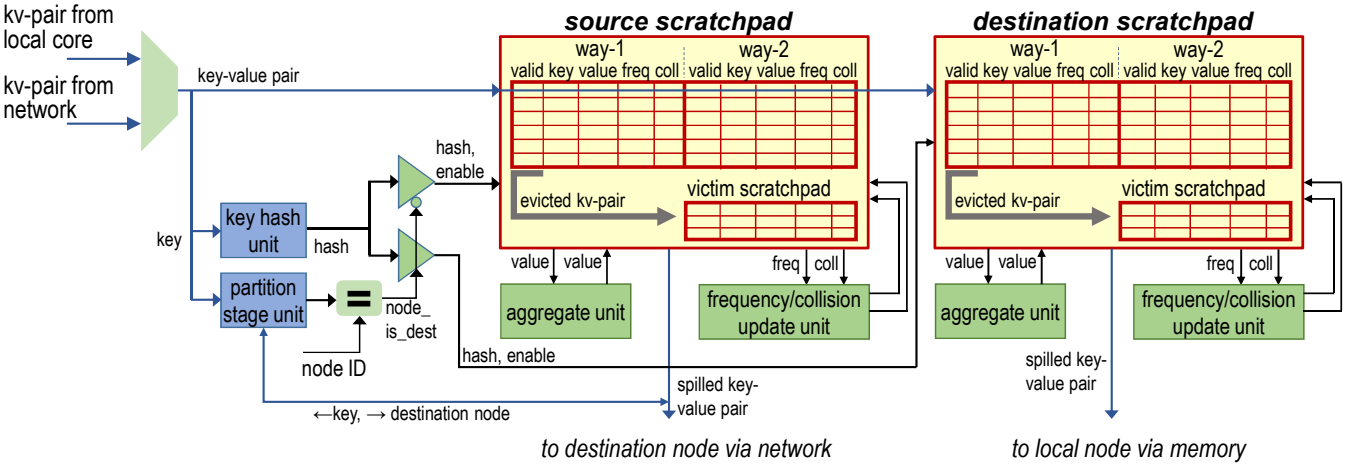


Figure 5: **MR.NITRO's accelerator architecture**. Each accelerator includes two separate *scratchpad memories*, organized as hashtables, to aggregate kv-pairs incoming from the local core or the interconnect. The *scratchpad memories* are organized as 2-way associative caches, augmented with a small *victim scratchpad*. The *aggregate units* are responsible for aggregating values for kv-pairs stored in the scratchpads. The *frequency/collision update units* enforce our novel kv-pair replacement policy. Finally, each accelerator includes a *key hash unit* to compute a hash value for each incoming key, and a *partition stage unit*, responsible for deriving the destination node ID in charge of applying the reduce function to each unique key. Note that kv-pairs evicted from the scratchpad memories are transferred out to either their destination node or memory, so that the destination node can complete the reduce stage.

unit. The *key hash unit* is used to compute the index value to access the scratchpad memory, which is organized as a hashtable. The *partition stage unit* uses a hash function to create a unique mapping from keys to a node ID, so that kv-pairs from each node can be distributed among the destination accelerators for the final reduce stage, and each destination accelerator is responsible for the reduce stage of all the keys hashed to its ID.

For the *partition stage unit*, we use a XOR-rotate hash function. The function reduces the kv-pair's key to a number of bits that is the minimum to represent the number of cores in the system. If the number of cores in the target CMP is not precisely a power-of-2, then we complete this function by adding a final modulo computation step which, at this point, simplifies to a simple subtraction. The specific operation of the XOR-rotate hash function is the same as the one for the key hash unit, which is detailed later in this Section.

We considered two factors in selecting the hash function for the *key hash unit*. The first factor is the rate at which cores emit kv-pairs to their local accelerator. The emission rate sets an upper bound on the acceptable time for the hash function to generate the final index value. A high emission rate of kv-pairs, coupled with a high latency hash function, prevents the accelerator from keeping up with the execution of its local core. In such cases, the core's execution must be stalled or the accelerator must include additional buffering resources. The second factor is the hash function collision rate, which depends on the function itself, the size of the scratchpad memory and the number of unique keys generated by the core. A good hash function must be able to evenly distribute keys over the scratchpad entries to minimize collisions and maximize the use of available storage.

Both the number of unique keys generated and the kv-pair emission rate by a core depend on the specific MapReduce application and the input data set. For example, *histogram* is a common MapReduce workload, which we evaluated in our experiments using a bitmap image input. In this specific context, the task of *histogram* is to compute the frequency of occurrence of each intensity (ranging 0 to 255) of each R, G, and B component (for a total of 768 unique keys) over all pixels of an image. Thus, the number of keys generated by the *histogram* application is fixed and the emission rate is high, since the overhead of processing each key in the map stage is minimal. On the other hand, the *word-count* application, described in Section 3, can generate a wide range of unique keys. It also exhibits a low key emission rate from the map function, due to the high overhead of reading the input data file and parsing it into keys of variable character length.

To gather insights on the quality of a range of hash functions, and their impact on our framework's performance, we analyzed the collision rate for a number of them, using the applications of our evaluation in Section 7.3. It can be noted that, among the workloads considered, XOR-rotate provides the lowest collision rate. In light of this result, but also in consideration of the wide range of MapReduce applications, and their very large data input sets in practical deployments, we determined to equip our *key hash unit* with two distinct hash functions: the first is a XOR-rotate hash function that provides very low computation latency and is best suited for applications that have a high rate of kv-pair generation. The second is a modulo-based hash function that has longer computation latency, but may provide a lower collision rate for some applications and datasets.

The XOR-rotate hash function partitions the key into fixed-length blocks, then rotates each block by a different amount, and finally computes the XOR of all blocks together. The size of each block is derived as

the log2 of the number of entries in the scratchpad. The modulo-based hash function computes the modulo of the key value over a the largest prime value that is smaller than the number of entries in the scratchpad. Note that, as a result, it is possible that a few entries remain unused. For instance, in the experiment of Section 7.3, we used the prime value 251 for the 256-entry scratchpad. The hardware implementation of the modulo function requires one or more subtractors and one or more comparators [17, 18]. We estimated that the latency of this function for a 64-bit key modulo an 8-bit divisor is approximately 30 clock cycles. Note that in a number of MapReduce applications, each kv-pair requires hundreds of clock cycles to be mapped.

4.2 The Scratchpad Memory

The accelerators' scratchpad memory is a fixed-size storage organized as a 2-way associative cache, where the set to be accessed is derived by applying the selected hash function in the *key hash unit*. When a kv-pair accessing the scratchpad "hits", that is, the keys of the scratchpad entry and the kv-pair are a match, we aggregate the two values by sending them to the aggregate unit, and then update the entry in the scratchpad. When a kv-pair "conflicts" in the scratchpad, that is, both stored keys are different, then we leverage our novel replacement solution to determine which kv-pair (the incoming one or one of those already stored) should be removed from the scratchpad and evicted to the victim scratchpad as discussed below. Of course, if there is an empty entry in the scratchpad corresponding to the current hash index, the incoming kv-pair will be stored there. We maintain separate source and destination scratchpads, one to store kv-pairs undergoing source aggregation, the other for kv-pairs in their final reduce stage. We keep them separate because, particularly for applications with a large number of unique keys, some of the keys tend to be "popular", that is, almost all cores emit those keys often. As a result, in a unified-scratchpad design, those same popular keys would dominate the scratchpads of all the accelerators. Then, for accelerators that are responsible for reducing infrequent keys, all the keys mapped to them for the reduce stage would spill into memory because they are not as frequent as the few "popular" keys. As a result, those accelerators would, *de facto*, only complete source aggregation. In contrast, by deploying a design with separate source and destination scratchpads, we can guarantee that at least the most frequent keys for each destination accelerator, will gain an entry in the destination scratchpad and undergo the reduce stage in hardware.

Note that in our design, we considered both a direct-mapped scratchpad organization and a 2-way set associative one. We opted for the latter one based on the benefits on reduced collision rates, as presented in Section 7.3. Each entry in the scratchpads stores the following fields: a valid bit, a key, the corresponding value to that key, frequency and collision numbers. The valid bit is used to track whether the corresponding entry is available or occupied by a kv-pair. Frequency and collision values are used to implement our replacement policy.

4.2.1 kv-pair replacement policy

The goal of our kv-pair replacement policy is that of prioritizing scratchpad storage for frequently-occurring keys. Since the scratchpad memories are of limited size, it is possible for two distinct keys to collide and be allocated to the same scratchpad entry. Therefore, we implemented a replacement policy to determine whether a previously stored key must be evicted and replaced with an incoming kv-pair. Our replacement policy ensures that the frequently occurring keys are aggregated and stored in the accelerator, so as to optimally utilize the available scratchpad memories and maximize the performance of our accelerator design. Upon storing a new entry in the scratchpad, its collision is initialized to 0 and its frequency to 1. Each time a new kv-pair is aggregated to the current entry, the frequency is incremented for the relevant entry (the collision value of the other entry is unmodified), while each time there is a key conflict, that is, the incoming kv-pair has a different key than those stored in the scratchpad set, the collision is incremented for both kv-pairs in the set.

Everytime an incoming kv-pair conflicts in the scratchpad, we analyze the two entries already in the scratchpad set to determine if one should be replaced. If, for either of those entries, the frequency is greater than the collision, then the entry is a frequent one and we simply update its collision value, but do not replace it. If, instead, collisions exceed frequency, than the entry is deemed infrequent, and it is replaced by the incoming kv-pair. It is possible that both entries are deemed infrequent, in which case we evict the entry that has the lowest frequency. Upon replacement, the new entry's frequency and collision numbers are reset.

4.2.2 victim scratchpad

Depending on the sequence of keys accessing the scratchpad memories, it is possible to incur into thrashing, where a small set of keys keeps overwriting each other in the scratchpad. To limit the impact of this issue, we augment each source and destination scratchpad with a small, fully-associative storage called "victim scratchpad": kv-pairs are stored in the victim scratchpad when they don't gain entry or are evicted from the main scratchpad.

An incoming kv-pair is guaranteed to always gain a spot in the victim scratchpad: if it "hits", that is, there is an entry already with that same key, then the frequency of the entry is incremented and the entry must be swapped with an entry in the main scratchpad. If it "conflicts", then the incoming pair gains a spot at the expense of another pair that will be spilled from the accelerator. In the case of a swap, we must identify the least frequent entry among the two in the set corresponding to the kv-pair to be swapped. In the case of a conflict, the kv-pair to be spilled is selected again by considering the victim scratchpad entry with the lowest

frequency count.

All kv-pairs that are either evicted or rejected by the victim scratchpad are transfered to the destination node (from source scratchpads), or to memory (from destination scratchpads). In the latter case, the destination cores must retrieve the data from memory after all accelerators have completed their operation, and complete the reduce function.

4.3 The Aggregate Unit

The accelerator's *aggregate unit* implements the MapReduce reduce function. Note that, different workloads utilize different types of operations to merge the data corresponding to each key. For example, the *wordcount* application requires an addition module to sum up the number of occurrences of each word. Our accelerator design supports several reduce operators, which cover a wide range of common MapReduce applications. In particular, we support addition, computing the maximum value, the minimum value or the average. The average operation is implemented by separating it into two addition operations: the first maintains the sum of values and the second tracks the total number of values aggregated.

Moreover, note that we require that the reduce operator be both commutative and associative, since the accelerators, due to their internal architecture, process kv-pairs independently from each other, and thus no ordering can be enforced on the kv-pair processing in the combine stage. Even if the set of operator options is constrained, most practical applications satisfy our requirements and employ straightforward and common operators, as the ones we provide. Finally, while currently MR.NITRO provides a fixed set of options for the reduce function, it would be conceivable to deploy a small block of reconfigurable logic to provide flexibility on this aspect.

5. SYSTEM INTEGRATION

Figure 6 shows how MR.NITRO's accelerators interface to the rest of the system. In our solution, each accelerator is tightly coupled with its local core. At the beginning of the execution, each core sends a start command to its accelerator and, based on the type of MapReduce application, it configures the accelerator to use one of the two hash functions we provide in the *key hash unit* (this setup could be easily extended to encompass a broad set of hash functions). The core also sends the initial address of a memory region that the accelerator shall use to store the reduced kv-pairs. The core can then begin to transmit mapped kv-pairs to its accelerator. It then waits for a completion signal from the accelerator, in the form of an interrupt (IPC), upon which it can retrieve the final, reduced kv-pairs from the shared memory region.

Note that the memory region that the core shares with the accelerator is also used as temporary storage for kv-pairs spilled from the destination scratchpad. When this situation arises, the processor core is responsible for completing the reduce function, by aggregating the results from the accelerator with the spilled kv-pairs.

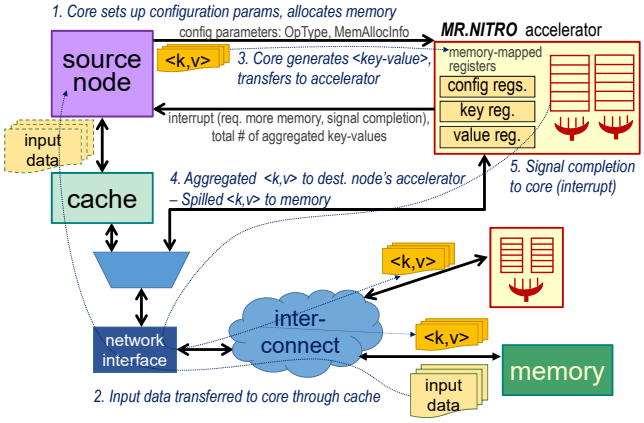


Figure 6: **System Integration**. Each core initiate the MapReduce application by communicating to its accelerator the configuration parameters. It then loads the input data from memory and transmits mapped kv-pairs to the accelerator. Concurrently, the accelerator locally aggregates the data. kv-pairs are transferred to the destination accelerators during the shuffling stage, and to memory if they spill from the destination scratchpad. Upon completion, each accelerator communicates to the core (via an interrupt) that the reduced kv-pairs can be retrieved from the shared memory.

If the accelerator requires additional temporary storage, it requests it to the processor core via an interrupt. All communication from a core to its local accelerator is carried out through store instructions to a set of memory-mapped registers within the accelerators, while accelerators communicate to the core via interrupts. Each accelerator is also directly connected to the network interface to send/receive kv-pairs to/from the other accelerators and the memory.

5.1 Memory Semantics

All communication between cores and accelerators is managed either through memory-mapped registers, or through read/write to a shared portion of memory. The memory-mapped registers are used for buffering incoming kv-pairs, either from a local core, or through the interconnect. The shared storage space is allocated by a core, then written by the accelerator. The core reads out the results only after the accelerator has signaled that the MapReduce execution has completed. Note that the accelerators do not have access to any cache, and write their results directly to memory. Thus cores and accelerators never access the shared memory at the same time, and we do not need to consider coherence aspects in the operation of MR.NITRO. Finally, scratchpad storage is for exclusive access by the accelerator, and it is not shared with any core or other accelerators.

Consequently, the operation of MR.NITRO is transparent to the coherence protocol in operation on the CMP baseline system and MR.NITRO can be deployed along with any coherence protocol.

In accessing the shared storage setup by the local core, each accelerator uses the same virtual memory space as the corresponding core, thus it translates mem-

ory addresses using the same page table as the one of the process initiating the MapReduce application. Once virtual addresses are translated into physical addresses, the specific memory access occurs in the same fashion as for the core, by leveraging the local network interface to reach the required memory node. If a core is allowed to switch contexts while the MapReduce application is in execution, then MR.NITRO requires to additionally store a copy of the page table base register of the MapReduce process in the accelerator's control registers, so that it can access it even if the process is swapped out.

5.2 Software Framework Deployment

MR.NITRO requires minimal modifications to a baseline CMP-based software MapReduce framework, as we discuss below. Note that, thanks to the clear-cut programming model embraced by MapReduce, no modification is required of the MapReduce application. Indeed, an application's developer can deploy seamlessly the same application in a CMP-based framework or in a framework augmented with MR.NITRO's accelerators.

The modifications required for a CMP-based MapReduce framework to leverage our accelerator-based solutions are: 1) the 'emit' function is modified to store the kv-pair into the memory-mapped registers of the local accelerator, instead of the internal data structures; 2) the combine, partition, shuffle stages are eliminated, and the reduce stage becomes an interrupt-serving routine, which gathers the final reduced results, and possibly completes the reduce function (see Section 5). Most other MapReduce services are unmodified: task scheduling, fault tolerance, *etc.*

Finally, note that the MR.NITRO framework requires that the reduce function be pre-parsed to extract the specific aggregate operator needed by the application.

6. LIMITATIONS AND DISCUSSIONS

Reduce function. MR.NITRO provides a number of options for the reduce function, which are executed during both the combine and the reduce stage. As mentioned in Section 4.3, MR.NITRO can only apply reduce functions that are both commutative and associative. This limitation is due to the distributed nature of our accelerator computations, which apply the specified operator to each key, independently of all other keys in the system.

In practice, many MapReduce applications use reduce functions that satisfy our requirements (average does not, but it can be decomposed into two sums, which can then be computed by MR.NITRO). A few exceptions, however, do exist: for instance, searching does not use any reduce operation. In a search application, the purpose of the reduce stage is simply to output the kv-pairs received from the map stage. While MR.NITRO is capable of executing the *null* reduce function, it would not provide any significant performance benefit over a baseline CMP. Another example, is the median function, which is neither commutative nor associative. The reduce function for sorting involves a complex string matching algorithm, which is challenging to implement efficiently in hardware. Thus, all these reduce functions are either not viable for a MR.NITRO solution, or do not benefit from it.

Mapping scratchpads into the local cache. It could be argued that, since there is a local cache in close proximity to each accelerator, we could simply reserve a portion of the cache for scratchpad storage, thus reducing the silicon footprint of the accelerators. We did consider this option while designing MR.NITRO, but we eventually decided against because 1) in mapping the scratchpad to the cache we would have to follow its replacement policies, instead of our own specialized policy that targets specifically the needs of MapReduce's kv-pairs. Moreover, 2) in our current design, a core is free to execute any process while MR.NITRO's accelerators carry out the combine, partition and reduce stages. However, if the accelerators used the cache for the combine stage, access to the cache by the local core would be severely impaired during this time (unless the cache is designed to have separate r/w ports), with a marked negative impact on its ability to carry out almost any computation.

7. EXPERIMENTAL EVALUATION

To evaluate MR.NITRO, we developed two experimental setups: one for a baseline CMP architecture running the Phoenix++ [2] framework, and one for a CMP architecture augmented with MR.NITRO's hardware accelerators. To cope with the unreasonable amount of time and resources required to simulate big data workloads on cycle-accurate simulators, we developed a specialized experimental setup for our architectures. We modeled the baseline CMP solution as a 64-node CMP in a mesh topology, with 64 cores and 4 memory nodes at the corners of the mesh. We created this design in the Gem5 [19]/Garnet [20] simulation infrastructure. We ported the Phoenix++ framework [2] to Gem5 using "m5threads" and carried out the simulations using the "syscall" mode.

For our proposed solution, we still leveraged the Gem5-Garnet infrastructure. In addition, we modeled MR.NITRO's accelerators separately using Python. Our accelerator model is capable of providing a cycle-accurate simulation, tracking the state of the scratchpad memories, and the times of spilling events. It is used to simulate both the combine/partition stages and the reduce stage. Finally, we used BookSim [21], a cycle-accurate network simulator to simulate the shuffle stage, matching the network configuration that we specified for Garnet. We chose BookSim for MR.NITRO's model because of its ease of simulating in standalone mode. Details of the architecture for both experimental setups are reported in Table 1.

Note that while we simulated all components of the MR.NITRO setup in detail to gather an accurate evaluation, we found in hindsight that MR.NITRO's execution is dominated almost exclusively by the time to complete the map stage in the processor's cores. This observation could also have easily been inferred from the

System configuration	Network configuration	Accelerator configuration
64 cores, 1GHz	topology: 8*8 mesh	maximum key size: 64 bits
L1 D cache size: 16KB	# input buffers per input port: 8	maximum value size: 32 bits
L1 I cache size: 16KB	# virtual channels: 4	hash functions: XOR-rot and modulo
L1 D and I cache latency: 1 ns	routing algorithm: XY	scratchpads are 2-way set associative
L2 cache size: 128KB		# entries per scratchpad: 256
L2 cache latency: 12ns		scratchpad size = 18KB
memory type: ddr3_1600		# entries per victim scratchpad: 8
number of memory controllers: 4		freq/coll counters: 8 bits, saturating

Table 1: **Characteristics of our architecture setup.** The first two columns specify the architecture of our CMP baseline design. The last column reports the architecture of MR.NITRO accelerators, which augment the baseline design.

diagram in Figure 3. Indeed, we purposely designed our accelerators to keep up with the cores' kv-pair emission rates, thus eliminating any additional execution time for the combine stage. Furthermore, because of the aggressive source aggregation that we carry out at the source nodes, the traffic crossing the interconnect is minimal and unlikely to cause any congestion.

Our MR.NITRO setup carries out each stage of MapReduce separately: first we simulate the map stage of each individual core using Gem5. We modified the applications by annotating the key and value variables as "volatile", so to bypass Gem5's cache architecture and be sure to transfer them directly to memory. We track these transfer events in simulation and generate a timed trace of kv-pairs emissions from the cores. Then we invoke our accelerator models to carry out the combine stage: we simulate all updates to the scratchpads and track all collisions and the times of kv-pair spilling events to the interconnect. We apply this simulation individually to each accelerator. We then simulate the interconnect exchanges during the combine (because of spilling) and the shuffle stages in BookSim [21]. Finally, our accelerator model is used again to simulate the reduce stage and determine if there is any spilling to memory. In case of kv-pair spilling to memory, the last stage of aggregation is handled by the cores. Consequently, we determine the amount of time it takes to aggregate the spilled kv-pairs with those produced by the accelerators by simulating in Gem5 one more time.

7.1 Workload Characteristics

Table [2] provides information on all the workloads we considered in our evaluation. Some are gathered from the Phoenix++ framework: *wordcount*, *histogram*, and *linear regression*. Others were developed by us: *page view count*, *min-max*, and *average*. We developed *page view count* using the APIs provided by the Phoenix++ framework. In our implementation, instead of using arbitrarily long URLs as keys, we associate a 64-bit "web page id" with each web log entry, which then replaces the original key. For *wordcount* and *histogram*, we used the data sets provided with Phoenix++. For *wordcount*, we only considered words that are <= 8 characters, to limit the size of our keys to 64 bits. In practice we found that the most recurring words in our data sets had fewer than 8 characters. For the other workloads, we used randomly generated data sets.

name	description	#keys
hist	*histogram*: computes a histogram. Input data: an RGB bit image	768
lr	*linear regression*: linear data fitting. Input data: bidimensional points	5
wc	*wordcount*: counts the frequency of words. Input data: multiple documents.	68152-257225
pvc	*page view count*: counts the frequency of web page views. Input data: random page list.	10529
minmax	*min/max*: finds min/max. Input data: temperatures for all locations in a region	28514
avg	*average*: finds average. Input data: temperatures for all locations in a region	28514

Table 2: **Experimental workloads.** We selected these workloads because they represent a wide range of MapReduce applications.

7.2 Performance Evaluation

Figure 7 reports the speedup of MR.NITRO against the baseline CMP running Phoenix++. We report both the speedup that we could achieve with an infinitely large scratchpad, and that of the actual 256-entries scratchpad of our model. While the actual speedup varies with the application, ranging from 150% to 550%, the average speedup over all applications we studied is 320%.
histogram and *linear regression*: In the baseline solution without accelerators, these two applications do not use hash function computations because the number of their unique keys is small and known *a priori*, so a fixed-size array can carry out the combine stage. This optimization improves the performance of the baseline execution, reducing our room for speedup. The speedup of *linear regression* is relatively better because this application is less memory-intensive than *histogram*.
page view count: the speedup of this application is limited by the execution time of the map stage, which requires relatively more parsing than other applications.
Ideal speedup. A number of applications have speedups that are fairly unaffected by the size of the scratchpads. However, a few, namely *wordcount*, *min/max* and *average* suffer from their limited size of the scratchpads, which leads to a lot of kv-pair spilling into the interconnect, thus overloading the reduce stage with additional aggregation and leveraging the slow processor cores for some of the reduce stage computation. This aspect is most pronounced in *wordcount*, which has the largest

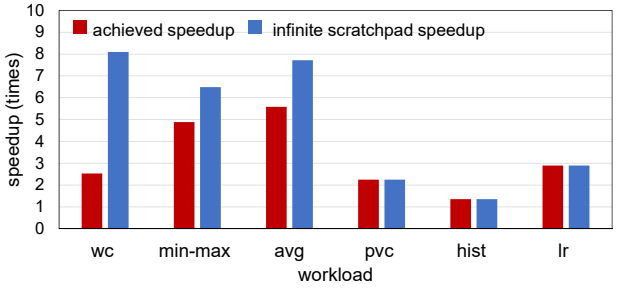


Figure 7: **MR.NITRO performance speedup for a range of workloads.** The chart plots the speedups over a baseline CMP execution of MapReduce, for our 256-entry scratchpad design and for an unbounded-size scratchpad design.

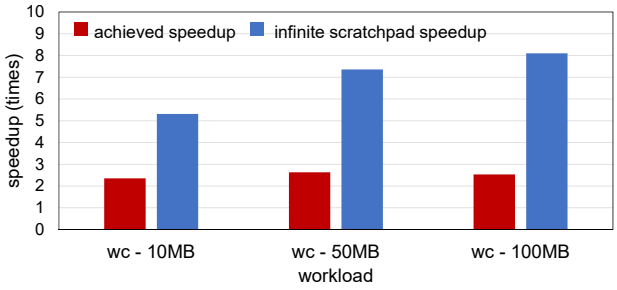


Figure 8: **Speedup analysis over increasing data set sizes.** As data set size increases, a CMP system faces more delays in accessing data and manipulates larger data structures. In contrast, MR.NITRO performance remains steady over larger data sets, leading to a relative improvement.

number of unique keys among all of our applications, sometimes by several orders of magnitude.

In order to determine the effects of input data size on speedup, we considered the execution of one application (*wordcount*) over several data sets of increasing size. Figure 8 plots our findings, suggesting that the benefits brought upon by MR.NITRO persist over varying input data set sizes. Indeed, if the sizes of the scratchpads were unbounded, MR.NITRO's performance benefits would increase as the number of unique keys in the application increases. Intuitively this trend occurs, because the only factor limiting MR.NITRO's performance is the ability to fit kv-pairs in the scratchpads, while the baseline CMP solution suffers from increased memory/cache accesses. Moreover, the longer the time spent on the combine stage by the baseline application, the larger the potential for gains left to MR.NITRO.

As a last performance experiment, we evaluated the scalability of MR.NITRO to a varying number of cores in the underlying CMP. One of the main goals of our effort was to provide a solution that scales nicely with the nodes in the CMP, so that as larger and more complex systems become commercially available, MR.NITRO can continue to rip a growing performance boost. The findings of this experiment are reported in Figure 9, indicating that, barring limitations in the scratchpad size, the performance speedup of MR.NITRO over a baseline system is monotonically increasing after we move past small-scale CMPs.

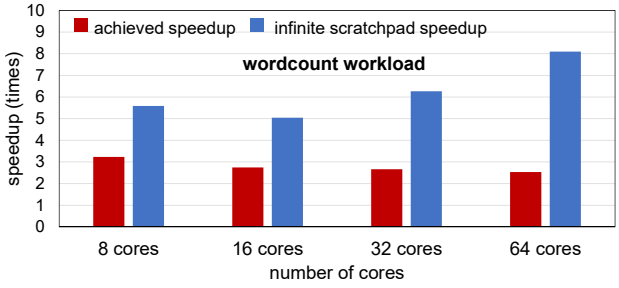


Figure 9: **CMP size analysis.** The plot reports performance speedup between baseline and MR.NITRO systems with increasing cores. As the size of the system grows, more accelerators contribute to a steady improvement in relative performance, barring limitations in scratchpad size.

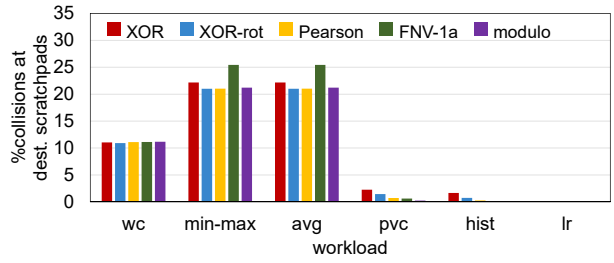


Figure 10: **Impact of hash functions on collision rates.** The functions studied were considered for the *key hash unit*. Note how XOR-rotate ranks with the best performers even in light of its simple implementation.

7.3 Design Parameters and Discussion

In this Section we evaluated a number of design parameters to determine the best setup for our accelerators and the impact of our design choices on MR.NITRO performance. We studied the quality of a number of hash functions for the *key hash unit*, the size and geometry of the scratchpad memory and the size of the victim scratchpad. Note that in most studies we focused on the collision rates at the destination scratchpad, since the performance impact of spilling in this storage is more pronounced.

Hash function. We evaluated a range of hash functions, from very simple in terms of hardware implementation, such as XOR and XOR-rotate, to more involved, such as modulo, and plotted the collision rate at the destination scratchpads. The other hash functions we considered are Pearson and FNV-1a: both have fast software implementations on resource-constrained processors and thus are good hash functions for CMP-based MapReduce frameworks. The results of our experiment, plotted in Figure 10, indicate that XOR-rotate performs best, or on par with the best hash functions for *word-count*, *min-max* and *average*. Modulo hash is best for *page view count* and *histogram*, although the collision rate in these applications is smaller. In light of this finding, in our implementation we provide the option of selecting between two distinct hash functions: XOR-rotate and modulo, so to allow a user to choose the most suitable function for the specific application.

Scratchpad size. Figure 11 plots the rate of collisions

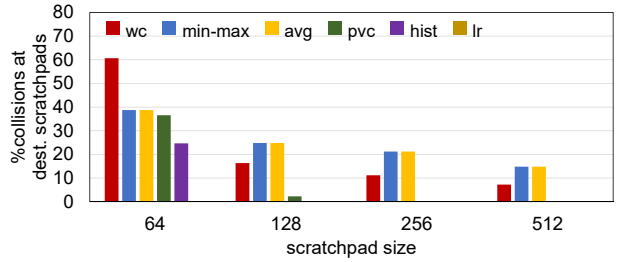


Figure 11: **Impact of scratchpad size on collision rates.** Workload with a larger set of unique keys are more sensitive to the scratchpad size.

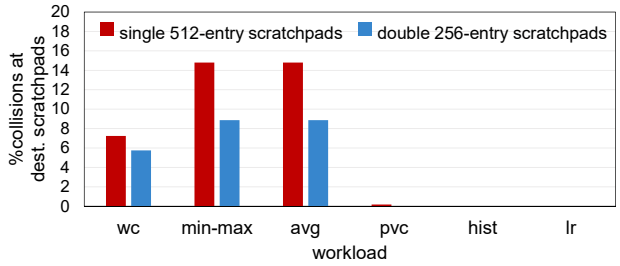


Figure 12: **Impact of scratchpad geometry.** The plot quantifies the advantage of organizing the accelerator storage into separate source and destination scratchpads, as discussed in Section 4.2.

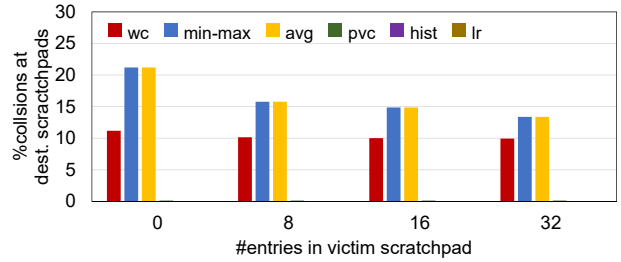


Figure 13: **Impact of victim scratchpad size on collision rates.** The victim scratchpad provides diminishing collision rate benefits as it grows larger.

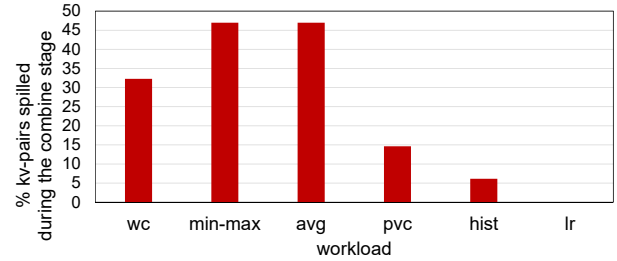


Figure 14: **kv-pairs spilled to the network during the combine stage.** The plot report the fraction of kv-pairs spilled over all the kv-pairs emitted by the cores during the map stage.

on the scratchpad over increasing sizes of scratchpad storage for the workloads in our setup. Note how the collision rate decreases and even disappears as the size of the scratchpad increases. By comparison with Figure 7, we note that, as soon as the scratchpad size matches the needs of key-diversity of an application, the corresponding collision rate becomes negligible. Note also how the collision rate has a steep decrease for *wordcount* from 64 to 128 entries, but further increases provide diminishing returns.

Scratchpad geometry. In Figure 12 we quantified the benefits of separate source and destination scratchpads over a double-sized integrated scratchpad. As discussed in Section 4.2 , the separate scratchpad architecture guarantees that at least some keys are reduced in each accelerator.

Victim scratchpad size. Figure 13 reports our study on effective sizes for the victim scratchpad, which is organized in a fully-associative fashion. The few applications that benefit from it seem to find improvements only up to a certain size. In our design we settled for a victim scratchpad size of 8 entries.

In MR.NITRO, kv-pairs are exchanged through the interconnect during the shuffle stage and for spilling from the source accelerator during the combine stage.

Figure 14 reports the fraction of kv-pairs traversing the network, due to spilling, over the total number of kv-pairs emitted by the processor cores to the accelerators. The additional traffic of transferring the 256 scratchpad entries during the shuffle stage is negligible. Note that, by comparison, the single accelerator solution in [15] must transfer all of the kv-pairs generated by the processor cores. Consequently, the plot indicates that MR.NITRO eliminates over 50% of the MapReduce network traffic, and often well beyond that.

7.4 Area Overhead

The largest components of our accelerators are the two scratchpads. We modeled those and the cores's caches in Cacti [22] and found that the area overhead of the accelerators' storage accounts only for 9.2% of the total storage. The silicon footprint of the frequency/collision unit is minimal, including only an incrementer and comparator. The aggregate unit is a simple addition/comparison unit. Finally the modulo function in the *hash key unit* is heavily pipelined, so it only includes a subtractor and a comparator.

8. CONCLUSIONS

MR.NITRO is a novel distributed hardware accelerator solution, capable of offloading the compute-intensive portion of MapReduce-based applications from the cores of a CMP to their local accelerators. The system is highly scalable with the number of cores. We found experimentally that our solution provides over a 3 times speedup on average over a pure CMP-based solution, while also eliminating more than 50% of the network traffic over a single-accelerator solution. We estimated the silicon footprint of MR.NITRO to account for less than 10% of the local cache storage.

9. REFERENCES

[1] N. A. Miller, E. G. Farrow, M. Gibson, L. K. Willig, G. Twist, B. Yoo, T. Marrs, S. Corder, L. Krivohlavek, A. Walter, J. E. Petrikin, C. J. Saunders, I. Thiffault, S. E. Soden, L. D. Smith, D. L. Dinwiddie, S. Herd, J. A. Cakici, S. Catreux, M. Ruehle, and S. F. Kingsmore, "A 26-hour

system of highly sensitive whole genome sequencing for emergency management of genetic diseases," *Genome Medicine*, 2015.

[2] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: Modular MapReduce for shared-memory systems," in *Proceedings of the Second International Workshop on MapReduce and Its Applications*, 2011.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. of MSST*, 2010.

[5] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for large-scale graph algorithms," *Parallel Comput.*, vol. 37, no. 9, 2011.

[6] M. Lu *et al*, "Optimizing the MapReduce framework on Intel Xeon Phi coprocessor," in *Big Data, 2013 IEEE International Conference on*, 2013.

[7] A. Tripathy, A. Patra, S. Mohan, and R. Mahapatra, "Distributed collaborative filtering on a single chip cloud computer*," in *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pp. 140–145, IEEE, 2013.

[8] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos, "CellMR: A framework for supporting MapReduce on asymmetric cell-based clusters," in *Proc. IPDPS*, 2009.

[9] K. Duraisamy, R. G. Kim, W. Choi, G. Liu, P. P. Pande, R. Marculescu, and D. Marculescu, "Energy efficient mapreduce with vfi-enabled multicore platforms," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 6, 2015.

[10] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 13–24, Ieee, 2007.

[11] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pp. 198–207, 2009.

[12] W. Fang, B. He, Q. Luo, and N. Govindaraju, "Mars: Accelerating MapReduce with graphics processors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 4, 2011.

[13] R. Zheng, K. Liu, H. Jin, Q. Zhang, and X. Feng, "Accelerate MapReduce on GPUs with multi-level reduction," in *Proceedings of the 5th Asia-Pacific Symposium on Internetware*, 2013.

[14] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "FPMR: MapReduce framework on FPGA," in *Proc. of FPGA*, 2010.

[15] C. Kachris, G. Sirakoulis, and D. Soudris, "A reconfigurable MapReduce accelerator for multi-core all-programmable SoCs," in *System-on-Chip (SoC), 2014 International Symposium on*, 2014.

[16] R. Pagh and F. F. Rodler, *Cuckoo hashing*. Springer, 2001.

[17] J. T. Butler and T. Sasao, "Fast hardware computation of x mod z," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 294–297, IEEE, 2011.

[18] A. Amaricai and O. Boncalo, "Srt radix-2 dividers with (5,4) redundant representation of partial remainder," in *NORCHIP, 2013*, pp. 1–5, 2013.

[19] N. Binkert *et al*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

[20] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 33–42, IEEE, 2009.

[21] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[22] G. Reinman and N. P. Jouppi, "Cacti 2.0: An integrated cache timing and power model," *Western Research Lab Research Report*, vol. 7, 2000.

Approximate Identification of Sink Strongly-Connected Components for the Generation of Close-to-Functional Broadside Tests

Irith Pomeranz
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907, U.S.A.
E-mail: pomeranz@ecn.purdue.edu

***Abstract*—The generation of functional scan-based tests requires information about the states that the circuit can visit during functional operation. In the existing procedures, the initial state of the circuit for functional operation is assumed to be known, and additional states are obtained by applying primary input sequences starting from this state. For the case where information about the initial state is not available, this paper observes that functional operation eventually occurs in strongly-connected components of the state diagram that are sinks (i.e., they have no outgoing edges). The paper describes an approximate procedure that identifies sink strongly-connected components by using logic simulation of an all-unspecified primary input sequence starting from random states. Because of the approximation, they are referred to as x-sinks. After replacing the all-unspecified primary input sequence by a fully-specified one, the procedure obtains scan-in states for broadside tests. The tests are referred to as sink-broadside tests. Experimental results demonstrate the similarities between sink-broadside tests and functional broadside tests.**

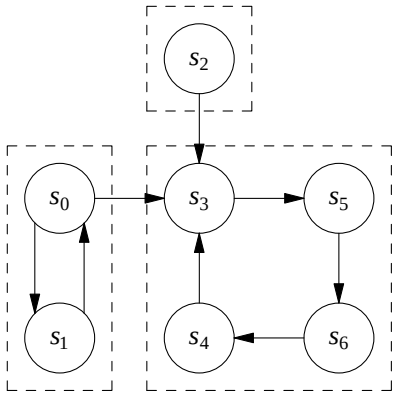


Fig. 1. Example 1 of a state diagram

I. Introduction

Functional and close-to-functional scan-based tests are used for avoiding overtesting of delay faults [1]-[13]. Overtesting may occur when a scan-based test creates non-functional operation conditions. The test may then activate slow logic, or create excessive switching activity that cannot occur during functional operation. In both cases a good circuit may fail the test. For simplicity of discussion it is typically assumed that the primary input vectors are not constrained during functional operation. Under this assumption, the extent to which a test creates functional operation conditions is determined by its scan-in state. This assumption is also made in this paper. Scan-based tests that satisfy functional constraints on primary input vectors are considered in [12].

Assuming primary input vectors that are unconstrained during functional operation, the scan-in state of a functional scan-based test is required to be a reachable state, which is a state that the circuit can visit during functional operation. In [3] and [7] it is assumed that functional operation starts with the application of a synchronizing sequence that takes the circuit into its initial state, s_{init}. In [10], hardware reset is assumed to exist that takes the circuit into its initial state, s_{init}, before functional operation starts. In both cases, a state s_i is reachable if there exists a primary input sequence that takes the circuit from s_{init} to s_i.

The existence of a synchronizing sequence into the state s_{init} has certain implications on the structure of the state diagram of the circuit [14]. Figure 1 illustrates this structure. Figure 1 shows the state diagram of a circuit with seven states. The states are partitioned into three strongly-connected components (within a strongly-connected component there is a path from every state to every other state). For every circuit state s_i, the synchronizing sequence defines a path from s_i to s_{init}. Therefore, s_{init} must be one of the states in the strongly-connected component that consists of s_3, s_4, s_5 and s_6. This strongly-connected component is referred to as S_2. The circuit has two additional strongly-connected components, S_0 that consists of s_0 and s_1, and S_1 that consists of s_2. The graph of strongly-connected components for this circuit is shown in Figure 2(a).

In general, it is possible to show that s_{init} is contained in a strongly-connected component that is a sink (it has no outgoing edges in the graph of strongly-connected components). This points to the importance of sink strongly-connected components in the discussion of functional operation conditions.

With hardware reset, s_{init} can be any one of the states of

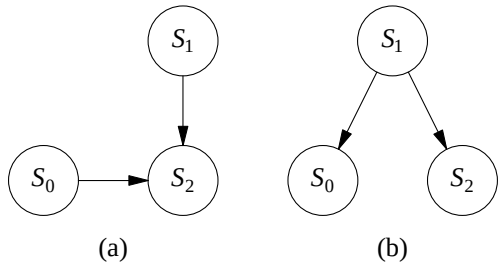


Fig. 2. Graphs of strongly-connected components

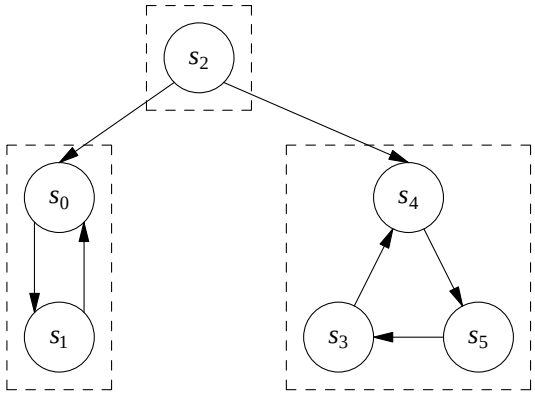


Fig. 3. Example 2 of a state diagram

the circuit. In Figure 1, if s_0 is the reset state, the circuit can visit any one of its states during functional operation, excluding only s_2. After reset, the circuit may visit s_0 and s_1 indefinitely. However, once it enters the sink strongly-connected component S_2, it will only visit its states s_3, s_4, s_5 and s_6.

Figure 3 shows a state diagram with the graph of strongly-connected components shown in Figure 2(b). For this state diagram, a synchronizing sequence does not exist. This can be seen from the fact that there in no state with a path from every state. If the circuit has reset to state s_2, the circuit can enter any one of the two sink strongly-connected components. Once it enters a sink, it does not leave it until it is reset again.

In both examples, functional operation eventually occurs in sink strongly-connected components. Therefore, if information about the way the circuit is initialized is not available, and the existing test generation procedures for functional scan-based tests are not applicable, the requirement to maintain functional operation conditions during test application can be replaced with the requirement to operate in sink strongly-connected components. To address this case, this paper describes an approach that attempts to identify sink strongly-connected components, and use states from such components as scan-in states of broadside tests for transition faults. The tests are referred to as sink-broadside tests to indicate that they attempt to use scan-in states from sink strongly-connected components. To measure the ability of the test generation procedure described in this paper to maintain functional operation conditions, the sink-broadside tests are compared with functional broadside tests that are computed under the assumption that reset is used for initializing the circuit into the all-zero state.

To maintain a manageable computational complexity, it is necessary to use approximations for identifying sink strongly-connected components. The procedure described in this paper is based on the use of the all-unspecified primary input sequence to capture all the possible primary input sequences of the circuit, including ones that take the circuit into sink strongly-connected components. Because of the approximation, the subsets of states that the procedure identifies are referred to as x-sinks. With this approach, all the computations of sink-broadside tests require only logic simulation of the circuit.

The paper is organized as follows. Section II describes the identification of x-sinks. Section III describes the generation of sink-broadside tests. Section IV analyzes the limitations of sink-broadside tests as they are computed in this paper. Section V presents experimental results.

II. Approximate Procedure for Identifying Sink Strongly-Connected Components

This section describes an approximate procedure that attempts to identify sink strongly-connected components. The procedure is based on the following observations.

Let $s_{i,0}$ be a random state in a strongly-connected component $S_{i,0}$. There exists a primary input sequence V that takes the circuit from $s_{i,0}$ to a sink strongly-connected component. The existence of V can be seen as follows. If $S_{i,0}$ is a sink strongly-connected component, then V is empty. Otherwise, there is a sequence $V_{0,1}$ that takes the circuit from $s_{i,0}$ to a state $s_{i,1}$ in a different strongly-connected component, $S_{i,1}$. If $S_{i,1}$ is not a sink strongly-connected component, then there is a sequence $V_{1,2}$ that takes the circuit to a state in a different strongly-connected component, $S_{i,2}$. The graph of strongly-connected components does not have cycles. This is illustrated by Figure 2 and can be proved formally. Therefore, this process must end in a sink strongly-connected component, $S_{i,k}$. The sequence $V = V_{0,1}V_{1,2}...V_{k-1,k}$ takes the circuit from $s_{i,0}$ into this sink.

This process of constructing a primary input sequence V is not feasible for circuits with large numbers of state variables since the graph of strongly-connected components cannot be computed for such a circuit. The computational complexity is addressed in this paper by using the all-unspecified primary input sequence.

Starting from $s_{i,0}$, application of the all-unspecified primary input sequence captures all the possible states that the circuit can enter at every clock cycle starting from $s_{i,0}$. Using the logic values $\{0, 1, x\}$, let $V_X = v_x v_x ... v_x$ be the all-unspecified primary input sequence, where $v_x = xx...x$ is the all-unspecified primary input vector. Starting from $s_{i,0}$ and applying v_x to the primary inputs, let the next-state computed by logic simulation be the partially-specified state $s_{i,1}$. In general, for $u = 1$, 2, ..., suppose that application of v_x to

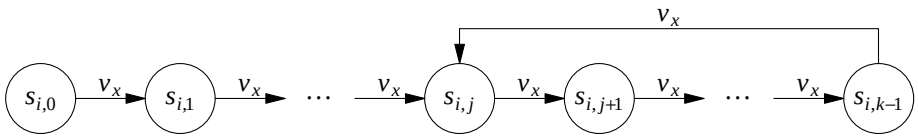


Fig. 4. Identifying an x-sink

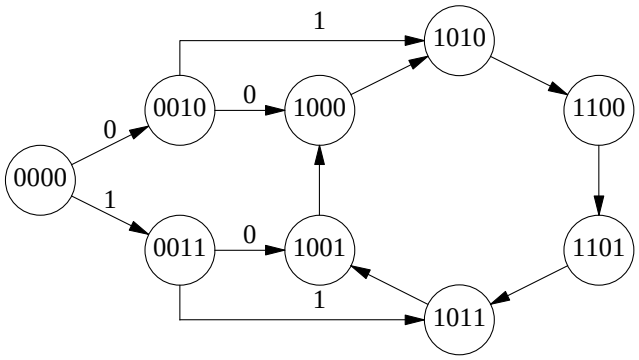


Fig. 5. Example of Compression

the primary inputs when the circuit is in state $s_{i,u}$ results in the next-state $s_{i,u+1}$. Thus, application of V_X starting from state $s_{i,0}$ takes the circuit through the sequence of states $s_{i,0}s_{i,1}s_{i,2}$....

Suppose that, for some $j < k$, $s_{i,j} = s_{i,k}$. This implies that V_X takes the circuit from $s_{i,0}$ into a cycle that consists of $s_{i,j}$, $s_{i,j+1}$, ..., $s_{i,k-1}$. This situation is illustrated by Figure 4. Because the cycle occurs under the all-unspecified primary input sequence V_X, there is no primary input vector that takes the circuit outside of the cycle. This is consistent with the structure of a sink strongly-connected component. Because the use of logic simulation of the all-unspecified primary input sequence introduces inaccuracies, the cycle is referred to as an x-sink.

The procedure for identifying x-sinks is given next as Procedure 1. In Procedure 1, the length of V_X is limited to a constant denoted by L_X. If a cycle is not found within this length, the procedure terminates without finding an x-sink.

Procedure 1: Identifying an x-sink

1) Assign a random state to $s_{i,0}$.
2) For $k = 1, 2, ..., L_X$:
   a) Find the next-state $s_{i,k}$ of $s_{i,k-1}$ under v_x.
   b) If there exists a $0 \leq j < k$ such that $s_{i,j} = s_{i,k}$, return $s_{i,0}$ and j.
3) Stop without finding an x-sink.

Procedure 1 requires only logic simulation of a sequence of length L_X, and comparisons of at most $(L_X + 1)L_X/2$ pairs of states.

Experimental results show that Procedure 1 finds cycles that are short relative to the numbers of states of benchmark circuits. This can be explained by the compression effect of the all-unspecified primary input sequence. This compression effect is illustrated by the state diagram shown in Figure 5, with the state assignment shown in the figure. The circuit has a single primary input. Unlabeled edges represent state-transitions that occur under both primary input vectors 0 and 1.

TABLE I
EXAMPLE CYCLE

$k=0$	$k=1$	$k=2$	$k=3$	$k=4$
0000	0010	1000	1010	1100
	0011	1010	1100	1101
		1001	1000	1010
		1011	1001	1000
0000	001x	10xx	1xxx	1xxx

Suppose that Procedure 1 selects to start from state 0000 of the state diagram in Figure 5. Table I shows the fully-specified states that the circuit visits under the all-unspecified primary input sequence, and their representation as partially-specified states. Procedure 1 identifies an x-sink after computing $s_{i,4}$ and finding that $s_{i,3} = s_{i,4} = 1xxx$. The sink-strongly connected component consists of six states, 1000, 1001, 1010, 1011, 1100 and 1101, on a cycle of length six. The use of the all-unspecified primary input sequence compresses this cycle into a self-loop.

III. GENERATING SINK-BROADSIDE TESTS

This section describes the generation of sink-broadside tests. The test set that the procedure generates is denoted by T_{sink}. The set of target faults is denoted by F.

The procedure iterates through a process where it first calls Procedure 1 to identify an x-sink starting from a random state. If Procedure 1 identifies an x-sink, and returns $s_{i,0}$ and j, the test generation procedure uses $s_{i,0}$ and j to generate sink-broadside tests. Tests that detect faults from F are added to T_{sink}, and the detected faults are removed from F. The procedure terminates after a constant number of consecutive iterations that do not increase the fault coverage. The constant is denoted by N_{TG}.

With $s_{i,0}$ and j, the generation of sink-broadside tests proceeds as follows. The variables $s_{i,0}$ and j indicate that the application of a primary input sequence of length j starting from $s_{i,0}$ takes the circuit into a state $s_{i,j}$ that starts a cycle. Using V_X, the state $s_{i,j}$ is incompletely-specified (in many cases, the all-unspecified state is obtained). To replace $s_{i,j}$ with a fully-specified state that is more likely to allow faults to be detected, the procedure performs logic simulation of the circuit starting from $s_{i,0}$ under a fully-specified primary input sequence of length j. Let the state that the circuit enters at

clock cycle j be $p_{i,j}$. The state $p_{i,j}$ is covered by $s_{i,j}$, and it is fully-specified.

To obtain additional fully-specified states that belong to the same x-sink as $p_{i,j}$, the procedure performs logic simulation starting from $p_{i,j}$ under a fully-specified primary input sequence of length L_{TG}, where L_{TG} is a constant. The states that the circuit visits are included in a set P_i. The states in P_i are used as scan-in states of sink-broadside tests, as follows.

Every state $p \in P_i$ is considered N_{PI} times, for a constant N_{PI}. When p is considered, the procedure defines a sink-broadside test $< p, v_0, v_1 >$, with scan-in state equal to p, and fully-specified primary input vectors v_0 and v_1 (under a broadside test, the primary input vectors v_0 and v_1 are applied in two consecutive functional clock cycles after p is scanned in). The procedure simulates the set of target faults F under the test $< p, v_0, v_1 >$. If any faults are detected, they are removed from F, and the test is added to T_{sink}.

The test generation procedure is summarized next as Procedure 2. Procedure 2 applies forward-looking reverse order fault simulation in order to remove from T_{sink} tests that are not necessary for achieving the fault coverage. Before applying this process the procedure orders the tests by increasing switching activity. This gives a precedence to the removal of a test with a higher switching activity.

The switching activity is the percentage of lines that make a signal-transition during the second cycle of a test. With functional broadside tests, the switching activity does not exceed the switching activity that is possible during functional operation. Excessive switching activity is one of the reasons for overtesting of delay faults according to [2] and [8]. This makes the switching activity an important parameter in the discussion of functional operation conditions.

Procedure 2: Generation of sink-broadside tests

1) Assign $T_{sink} = \emptyset$ and let F be the set of target faults. Assign $n_{tg} = 0$.
2) Call Procedure 1. If Procedure 1 finds an x-sink, and returns $s_{i,0}$ and j:
   a) Perform logic simulation of the circuit starting from $s_{i,0}$ under a fully-specified primary input sequence of length j. Let the final state be $p_{i,j}$.
   b) Perform logic simulation of the circuit starting from $p_{i,j}$ under a fully-specified primary input sequence of length L_{TG}. Include the states that the circuit visits in P_i.
   c) For every state $p \in P_i$, repeat N_{PI} times:
      i) Define a sink-broadside test $< p, v_0, v_1 >$.
      ii) Perform fault simulation of F under $< p, v_0, v_1 >$.
      iii) If any faults are detected, remove them from F, and add $< p, v_0, v_1 >$ to T_{sink}.
3) If no new faults were detected in Step 2, assign $n_{tg} = n_{tg} + 1$. Else:
   a) Assign $n_{tg} = 0$.
   b) Perform forward-looking reverse order fault simulation to remove unnecessary tests from T_{sink}.
4) If $n_{tg} < N_{TG}$, go to Step 2.

All the fully-specified primary input vectors that Procedure 2 uses are determined randomly. The random selection is biased to avoid an effect called repeated synchronization [15]. This implies that certain primary input values are preferred with a high probability if their complements cause one or more state variables to be synchronized. Preferred primary input values are determined using logic simulation.

The computational effort of Procedure 2 is determined by the number of tests for which it performs fault simulation. In every iteration where Procedure 1 finds an x-sink, Procedure 2 simulates at most $(L_{TG} + 1)N_{PI}$ tests. If the procedure detects one fault every N_{TG} iterations, it performs $N_{TG}|F|$ iterations before it terminates. This yields a total of at most $(L_{TG} + 1)N_{PI}N_{TG}|F|$ simulated tests. Fault simulation is performed with fault dropping.

IV. Limitations of Sink-Broadside Tests

Beyond the limitations of logic simulation with unspecified values, the procedure for generating sink-broadside tests has the following limitations.

The first limitation is that a fault may be detectable by a functional broadside test with a reset state s_{init}, but not by a sink-broadside test. This can be seen from the state diagram in Figure 3. Suppose that $s_{init} = s_2$. Let f be a fault that can only be detected by a broadside test with scan-in state s_2. Such a functional broadside test exists for f when the reset state is s_2. With sink-broadside tests, even if Procedure 1 selects s_2 as $s_{i,0}$, it will only find a cycle after applying at least two all-unspecified primary input vectors. This will prevent s_2 from being used as a scan-in state. Therefore, f is not detectable by a sink-broadside test.

The second limitation is that Procedure 2 may use as a scan-in state a state that is not included in a sink strongly-connected component. This can be seen from the state diagram in Figure 1. Suppose that the procedure selects $s_{i,0} = s_0$ randomly, and applies the all-unspecified primary input sequence starting from this state. Because of the outgoing edges from s_0 to s_1 and s_3, the state $s_{i,1}$ obtained after the application of v_x in state $s_{i,0}$ covers at least these two states. A cycle will be obtained after at least one additional clock cycle, with $j \geq 1$. However, when Procedure 2 applies a fully-specified primary input sequence of length j starting from $s_{i,0} = s_0$, the sequence may take the circuit between states s_0 and s_1, and not enter the sink strongly-connected component that consists of s_3, s_4, s_5 and s_6.

In addition, the compression effect that helps the procedure find an x-sink may also cause the path into the x-sink to be too short.

It is interesting to note that a similar problem occurs with the test generation procedures described in [1], [11] and [13]. These procedures are based on the expectation that clocking the circuit in functional mode eventually brings it into a state that it can visit during functional operation. With a cycle such as the one including s_0 and s_1 in Figure 1, this may not be the case after any number of functional clock cycles.

TABLE II
COMPARISON OF TEST SETS (ISCAS-89 AND ITC-99)

circuit	sv	type	iter	tests	f.c.	undet	swa
s641	19	arbit	-	41	96.09	-	71.72
s641	19	func	-	76	81.02	-	47.81
s641	19	sink	194	91	93.83	0.00	47.19
s1423	74	arbit	-	71	88.55	-	65.50
s1423	74	func	-	124	83.27	-	54.53
s1423	74	sink	1158	141	87.49	0.00	55.80
s5378	179	arbit	-	179	92.06	-	66.50
s5378	179	func	-	206	73.50	-	43.83
s5378	179	sink	529	200	72.15	1.91	44.02
s9234	228	arbit	-	359	83.71	-	49.31
s9234	228	func	-	108	18.72	-	21.32
s9234	228	sink	1183	247	43.18	1.65	27.05
s13207	669	arbit	-	360	81.53	-	51.51
s13207	669	func	-	133	29.27	-	16.99
s13207	669	sink	799	186	31.63	4.88	15.61
s15850	597	arbit	-	318	71.65	-	39.11
s15850	597	func	-	87	19.68	-	14.54
s15850	597	sink	440	92	22.00	2.06	17.90
s35932	1728	arbit	-	34	87.21	-	64.28
s35932	1728	func	-	145	87.21	-	54.73
s35932	1728	sink	120	143	87.21	0.00	55.72
b04	66	arbit	-	52	90.11	-	61.03
b04	66	func	-	92	84.54	-	50.09
b04	66	sink	2	93	84.54	0.00	47.90
b05	34	arbit	-	107	79.60	-	49.60
b05	34	func	-	61	41.13	-	37.16
b05	34	sink	1197	93	69.76	0.00	43.08
b07	51	arbit	-	69	79.70	-	58.99
b07	51	func	-	44	54.96	-	42.77
b07	51	sink	498	76	71.13	0.00	42.56
b14	247	arbit	-	242	79.85	-	51.57
b14	247	func	-	524	76.33	-	40.76
b14	247	sink	2130	518	79.49	0.00	39.49
b20	494	arbit	-	307	83.04	-	48.83
b20	494	func	-	983	79.40	-	36.14
b20	494	sink	2972	900	82.26	0.03	40.63

TABLE III
COMPARISON OF TEST SETS (IWLS-05)

circuit	sv	type	iter	tests	f.c.	undet	swa
aes_core	530	arbit	-	309	99.94	-	38.05
aes_core	530	func	-	927	99.94	-	34.26
aes_core	530	sink	29	928	99.94	0.00	34.71
des_area	128	arbit	-	130	100.00	-	43.15
des_area	128	func	-	310	100.00	-	40.92
des_area	128	sink	1	315	100.00	0.00	41.91
i2c	128	arbit	-	81	84.10	-	53.80
i2c	128	func	-	218	78.18	-	35.38
i2c	128	sink	1186	155	78.41	1.14	39.67
pci_spoci_ctrl	60	arbit	-	51	31.52	-	20.59
pci_spoci_ctrl	60	func	-	0	0.00	-	-
pci_spoci_ctrl	60	sink	325	24	11.25	0.00	11.98
sasc	117	arbit	-	45	94.68	-	44.45
sasc	117	func	-	116	78.00	-	27.61
sasc	117	sink	1966	125	88.41	0.46	27.74
simple_spi	131	arbit	-	96	93.72	-	35.92
simple_spi	131	func	-	171	75.26	-	28.32
simple_spi	131	sink	2045	179	80.63	0.76	28.85
spi	229	arbit	-	719	92.40	-	33.52
spi	229	func	-	1241	91.29	-	26.79
spi	229	sink	750	843	84.62	6.91	27.01
systemcaes	670	arbit	-	195	94.27	-	48.52
systemcaes	670	func	-	450	82.12	-	25.46
systemcaes	670	sink	844	485	86.53	0.00	31.73
systemcdes	190	arbit	-	91	99.68	-	56.21
systemcdes	190	func	-	209	99.66	-	46.66
systemcdes	190	sink	1	195	99.66	0.00	49.78
tv80	359	arbit	-	496	78.78	-	28.87
tv80	359	func	-	787	56.97	-	24.84
tv80	359	sink	3228	884	65.27	1.22	27.27
usb_phy	98	arbit	-	55	93.01	-	59.55
usb_phy	98	func	-	110	59.97	-	30.11
usb_phy	98	sink	1044	131	78.58	0.99	32.18
wb_dma	523	arbit	-	162	95.04	-	54.08
wb_dma	523	func	-	294	73.37	-	25.37
wb_dma	523	sink	2793	257	79.22	0.85	29.49

An advantage of sink-broadside tests is that the search for a cycle under an all-unspecified primary input sequence determines the number of functional clock cycles for which a circuit should be clocked in order to bring it into a state that is likely to be included in a sink strongly-connected component. The results presented in the next section demonstrate that this number can vary significantly with the circuit. It also varies with the random state selected by Procedure 1.

V. EXPERIMENTAL RESULTS

This section presents the results of Procedure 2 for transition faults in benchmark circuits.

Procedures 1 and 2 are applied with the following parameter values. Procedure 1 uses an all-unspecified primary input sequence of length $L_X = 1024$ to identify cycles. Procedure 2 performs $N_{TG} = 256$ iterations without increasing the fault coverage before it terminates. It uses a sequence of length $L_{TG} = 128$ to identify scan-in states. It considers $N_{PI} = 128$ tests for every scan-in state. This implies that $129 \cdot 128 = 16512$ tests are simulated with fault dropping in every iteration.

The test set T_{sink} that Procedure 2 produces is compared with two broadside test sets. The test set T_{arbit} is generated without considering functional operation conditions. The test set T_{func} consists of functional broadside tests that are generated under the assumption that functional operation starts by resetting the circuit to the all-zero state.

Tables II and III contain three rows for every circuit. The first row describes the arbitrary broadside test set T_{arbit}. The second row describes the functional broadside test set T_{func}. The third row describes the sink-broadside test set T_{sink}. For every test set, Tables II and III show the following information.

Column sv shows the number of state variables. Column $type$ shows the test set type. For sink-broadside tests, column $iter$ shows the number of iterations of Procedure 2 until the final fault coverage is achieved.

Column $tests$ shows the number of tests in the test set. Column $f.c.$ shows its transition fault coverage. For sink-broadside tests, column $undet$ shows the percentage of faults that are detected by T_{func}, and not detected by T_{sink}. The possibility that faults, which are detected by T_{func}, will not be detected by T_{sink}, was discussed in Section IV. Column swa shows the maximum switching activity of the test set.

Table IV shows additional parameters of sink-broadside tests. The parameters are computed over all the iterations of Procedure 2 until it terminates. Column $iter$ shows the number of iterations of Procedure 2. Column $sinks$ shows the number

TABLE IV
PARAMETERS OF SINK-BROADSIDE TESTS

circuit	sv	iter	sinks	spec	len	cyc
s641	19	450	450	4	4	1
s1423	74	1414	1414	0	7	1
s5378	179	785	785	0	25	1
s9234	228	1439	554	172	507	512
s13207	669	1055	1055	373	39	104
s15850	597	696	161	331	41	4
s35932	1728	120	120	0	33	1
b04	66	258	258	0	7	1
b05	34	1453	1453	2	9	1
b07	51	754	754	0	11	1
b14	247	2386	2386	30	4	1
b20	494	3228	3228	60	4	1
aes_core	530	29	29	0	5	1
des_area	128	1	1	0	1	1
i2c	128	1442	1442	0	9	1
pci_spoci_ctrl	60	581	580	60	558	336
sasc	117	2222	2222	1	13	1
simple_spi	131	2301	2301	1	9	1
spi	229	1006	354	1	6	1
systemcaes	670	1100	1100	0	79	1
systemcdes	190	257	257	0	19	1
tv80	359	3484	3484	0	6	1
usb_phy	98	1300	1300	0	16	1
wb_dma	523	3049	3049	27	9	1

of x-sinks that Procedure 2 identifies (the x-sinks may not be different). The number of x-sinks is smaller than the number of iterations if there are iterations where Procedure 1 does not find a cycle, and therefore, does not identify an x-sink.

Column $spec$ shows the maximum number of specified values in a state $s_{i,j}$ that starts a cycle. Column len shows the maximum length of an all-unspecified primary input sequence that is required for entering the first state of a cycle. This is the maximum value of j for which $s_{i,j}$ starts a cycle. Column cyc shows the maximum length of a cycle. If a cycle is identified because $s_{i,j} = s_{i,k}$, its length is $k - j$.

The following points can be seen from Tables II, III and IV. The fault coverage of T_{sink} is typically higher than that of T_{func}. Thus, the first limitation discussed in Section IV does not have a significant effect on the fault coverage. Nevertheless, there are circuits where a small percentage of the faults that are detected by T_{func} are not detected by T_{sink}.

The fault coverage of T_{sink} is typically lower than that of T_{arbit} because of the constraints on the states that are allowed as scan-in states in order to avoid overtesting.

The maximum switching activity of T_{sink} is similar to that of T_{func}, and it is typically significantly lower than that of T_{arbit}. This is another indication of the ability of the tests in T_{sink} to avoid overtesting of transition faults.

The number of iterations of Procedure 2 varies with the circuit. For most of the circuits, every iteration identifies an x-sink. This implies that $L_X = 1024$ is sufficient.

Considering the maximum number of specified values in the first state of a cycle, $s_{i,j}$, there are several circuits where this number is non-zero. When the number is zero, the cycle is a self-loop and the length of the cycle is one. Longer cycles are obtained for several circuits.

The maximum length of the sequence leading to the first state of a cycle also depends on the circuit. Nevertheless, the compression effect discussed in Section II ensures that cycles can be obtained for all the circuits.

VI. CONCLUDING REMARKS

This paper observed that functional operation of a circuit eventually occurs in sink strongly-connected components of the state diagram. To identify sink strongly-connected components with a manageable computational complexity, the paper used logic simulation of an all-unspecified primary input sequence. A cycle was taken as an indication that the circuit entered a sink strongly-connected component. Because of the approximation, the cycle was referred to as an x-sink. The procedure used a fully-specified primary input sequence to identify a fully-specified state from which scan-in states were obtained by applying a second fully-specified primary input sequence. The scan-in states were used for generating broadside tests that were referred to as sink-broadside tests. Experimental results demonstrated the similarities between sink-broadside tests and functional broadside tests.

REFERENCES

[1] J. Rearick, "Too Much Delay Fault Coverage is a Bad Thing", in Proc. Intl. Test Conf., 2001, pp. 624-633.

[2] J. Saxena, K. M. Butler, V. B. Jayaram, S. Kundu, N. V. Arvind, P. Sreeprakash and M. Hachinger, "A Case Study of IR-Drop in Structured At-Speed Testing", in Proc. Intl. Test Conf., 2003, pp. 1098-1104.

[3] I. Pomeranz, "On the Generation of Scan-Based Test Sets with Reachable States for Testing under Functional Operation Conditions", in Proc. Design Autom. Conf., 2004, pp. 928-933.

[4] Y.-C. Lin, F. Lu, K. Yang and K.-T. Cheng, "Constraint Extraction for Pseudo-Functional Scan-Based Delay Testing", in Proc. Asia and South Pacific Design Autom. Conf., 2005, pp. 166-171.

[5] Z. Zhang, S. M. Reddy and I. Pomeranz, "On Generating Pseudo-Functional Delay Fault Tests for Scan Designs", in Proc. Intl. Symp. on Defect and Fault Tolerance in VLSI Systems, 2005, pp. 398-405.

[6] I. Polian and F. Fujiwara, "Functional Constraints vs. Test Compression in Scan-Based Delay Testing", in Proc. Design, Autom. and Test in Europe Conf., 2006, pp. 1-6.

[7] I. Pomeranz and S. M. Reddy, "Generation of Functional Broadside Tests for Transition Faults", IEEE Trans. on Computer-Aided Design, Oct. 2006, pp. 2207-2218.

[8] S. Sde-Paz and E. Salomon, "Frequency and Power Correlation between At-Speed Scan and Functional Tests", in Proc. Intl. Test Conf., 2008, Paper 13.3, pp. 1-9.

[9] I. Pomeranz and S. M. Reddy, "Definition and Generation of Partially-Functional Broadside Tests", IET Computers & Digital Techniques, Jan. 2009, pp. 1-13.

[10] I. Pomeranz and S. M. Reddy, "On Reset Based Functional Broadside Tests", in Proc. Design Autom. and Test in Europe Conf., 2010, pp. 1438-1443.

[11] E. K. Moghaddam, J. Rajski, S. M. Reddy and M. Kassab, "At-Speed Scan Test with Low Switching Activity", in Proc. VLSI Test Symp., 2010, pp. 177-182.

[12] I. Pomeranz, "Generation of Functional Broadside Tests for Logic Blocks with Constrained Primary Input Sequences", IEEE Trans. on Computer-Aided Design, March 2013, pp. 442-452.

[13] T. Zhang and D. M. H. Walker, "Power Supply Noise Control in Pseudo Functional Test", in Proc. VLSI Test Symp., 2013, pp. 1-6.

[14] I. Pomeranz and S. M. Reddy, "On Synchronizable Circuits and their Synchronizing Sequences", IEEE Trans. on Computer-Aided Design, Sep. 2000, pp. 1086-1092.

[15] I. Pomeranz and S. M. Reddy, "Primary Input Vectors to Avoid in Random Test Sequences for Synchronous Sequential Circuits", IEEE Trans. on Computer-Aided Design, Jan. 2008, pp. 193-197.

Design Automation Challenges in Neuromorphic Systems

Rajit Manohar[1]

[1]Cornell University, USA

Abstract. *Neuromorphic systems are an emerging class of programmable substrates inspired by biological neural networks. These systems are typically implemented using a combination of analog and asynchronous digital components, and recent work has shown how they can be used for low power implementation of neural-network based machine learning algorithms. This talk provides an abbreviated history of the field, and describes our recent work with IBM on an all-digital approach to neuromorphic system engineering that culminated in TrueNorth–a low-power single chip million neuron system. We discuss some of the automation challenges that we faced in the design of such systems, as well as in improving the usability of neuromorphic systems.*

Rajit Manohar is Professor of Electrical and Computer Engineering and a Weiss Presidential Fellow at Cornell. He received his B.S. (1994), M.S. (1995), and Ph.D. (1998) from Caltech. He has been on the Cornell faculty since 1998 and the Cornell Tech faculty since 2012, where his group conducts research on self-timed systems. He is the recipient of an NSF CAREER award, nine best paper awards, seven teaching awards, and was named to MIT technology review's top 35 young innovators under 35 for contributions to low power microprocessor design.

A Branch-and-Bound-Based Minterm Assignment Algorithm for Synthesizing Stochastic Circuit

Xuesong Peng and Weikang Qian
University of Michigan-Shanghai Jiao Tong University Joint Institute
Shanghai Jiao Tong University, Shanghai, China
Email: {sayson, qianwk}@sjtu.edu.cn

***Abstract*—Stochastic computing (SC) is an unconventional paradigm which performs computation on stochastic bit streams using ordinary digital circuits. In SC, a real number is encoded by a stochastic bit stream, where the value is the probability of ones in the stream. SC can perform complex arithmetic computation with simple circuits. It also has strong tolerance to bit flip errors. In a prior work, researchers have proposed a general design of stochastic circuit and a synthesis method. However, the synthesis method cannot produce a circuit with minimal area. In this paper, we propose an improved synthesis method that applies a branch-and-bound algorithm to search for the optimal minterm assignment. We also introduce a few techniques to speed up the synthesis procedure with only small quality loss. Experimental results showed that the new synthesis method produces smaller circuits.**

Keywords—stochastic computing; stochastic circuit synthesis; minterm assignment

I. INTRODUCTION

Stochastic computing (SC) is an alternative to the conventional computing paradigm based on binary radix encoding. In SC, digital circuits are still used to perform computation. However, their inputs are stochastic bit streams [1]. Each stochastic bit stream encodes a value equal to the probability of a 1 in the stream. For example, the stream *A* shown in Fig. 1 encodes the value 0.75.

One major advantage of SC is that it allows complex arithmetic computation to be realized by a very simple circuit. Fig. 1 shows that arithmetic multiplication can be realized by an AND gate, since for an AND gate, the probability of obtaining a 1 in the output bit stream is equal to the product of the probabilities of obtaining a 1 in the input bit streams.

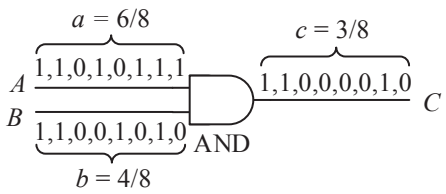


Fig. 1. An AND gate performs multiplication on real values encoded by stochastic bit streams.

Since all the bits in the stream have equal weight and a long bit stream is usually used to encode a value, a single bit flip occurring anywhere in the bit stream only causes very small change to the encoded value. Therefore, SC is highly tolerant to bit flip errors [2].

Given its advantages of low hardware cost and strong error tolerance, SC has been used in a number of applications, including image processing [3], decoding of modern error-correcting codes [4], and artificial neural networks [5].

In early days, various elementary computing units in SC were proposed, such as multiplier, scaled adder, divider, and squaring unit [6]. These units were designed manually and can only perform a limited types of computations.

In order to apply SC to a broad range of target computations, several methods to synthesize stochastic circuits have been proposed recently. The works [2,7,8] focused on synthesizing reconfigurable stochastic circuits. In [2], the authors proposed a method based on Bernstein polynomial expansion to synthesize combinational logic-based stochastic circuits. In [7] and [8], the authors studied the form of the computation realized by SC using sequential circuits and proposed methods to synthesize such designs. The works [9−11] focused on synthesizing fixed stochastic circuits, which take less area than reconfigurable ones. In [9], the authors demonstrated a fundamental relation between stochastic circuits and spectral transform. Based on this, they proposed a general approach to synthesize stochastic circuits. In [10], the authors found that different Boolean functions could compute the same arithmetic function in SC and proposed the concept of stochastic equivalence class. They proposed a method to search for the optimal Boolean function within an equivalence class. However, their method can only be applied to synthesize multi-linear polynomials. In [11], the authors introduced a general combinational circuit for SC and analyzed its computation. They further proposed a method to synthesize low-cost fixed stochastic circuit to realize a general polynomial.

The study in [11] reveals that in SC, there are a large number of different Boolean functions that realize the same target arithmetic function. Of course, the circuits for different Boolean functions have different costs. In previous work [11], a greedy method was used to find a circuit with low area cost. However, given the extremely large search space, the greedy strategy, although very fast, may not give a minimal solution. In this work, we address this problem by applying a branch-and-bound-based algorithm to extensively search for a Boolean function that will lead to a circuit with low cost. Our approach constructs a function by iteratively adding cubes into the on-set of the Boolean function. The optimal set of cubes to be added is determined through the search process. To improve the runtime, we also introduce a few speed-up techniques.

In summary, the main contributions of our work are as follows.

- We introduce a new method that iteratively selects cubes to form a Boolean function that realizes the target computation in SC.
- We develop a branch-and-bound algorithm to search for the optimal set of cubes to be added.
- We propose several speed-up techniques which prune unpromising branches and significantly improve the runtime of the algorithm.

The rest of the paper is organized as follows. In Section II, we give the background on the general design proposed in [11] and illustrate the previous synthesis method. We also present the logic synthesis problem for stochastic computing. In Section III, we present the new algorithm. In Section IV, we discuss several speed-up techniques. In Section V, we show the experimental results. Finally, we conclude the paper in Section VI.

II. Background on Synthesizing Stochastic Circuits

In this section, we give the background on the general form of the stochastic circuit proposed in [11] and discuss the previous method to synthesize a target function. In what follows, when we say the probability of a signal, we mean the probability of the signal to be a one.

A. The General Form and Its Computation

The general form of a stochastic circuit is shown in Fig. 2. The circuit is a combinational circuit. It computes an arithmetic function $f(x_1, \dots, x_n)$, which is encoded by the output bit stream. It has n inputs $X_1, \dots, X_n$, which are supplied with variable probabilities $x_1, \dots, x_n$, respectively. In order to offer freedom for realizing different functions, the circuit has m extra inputs $Y_1, \dots, Y_m$, each supplied with a constant probability of 0.5. They can be easily obtained by a linear feedback shift register (LFSR). The value of m affects the quantization error and is chosen according to the accuracy requirement. The large the value m is, the smaller the quantization error.

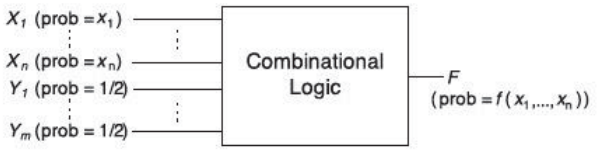


Fig. 2. General form of a stochastic circuit [11].

The study in [11] shows that the general design computes a type of function in the form

$$f(x_1, \dots, x_n) = \sum_{\substack{(a_1,\dots,a_n)\\ \in\{0,1\}^n}} \frac{g(a_1, \dots, a_n)}{2^m} \prod_{j=1}^{n} x_j^{a_j} (1 - x_j)^{1-a_j}, \quad (1)$$

where $0 \le g(a_1, \dots, a_n) \le 2^m$ is an integer. If the combinational circuit realizes a Boolean function $B(X_1, \dots, X_n, Y_1, \dots, Y_m)$, then the value $g(a_1, \dots, a_n)$ is equal to the number of vectors $(b_1, \dots, b_m) \in \{0,1\}^m$ such that $B(a_1, \dots, a_n, b_1, \dots, b_m) = 1$.

Example 1

Suppose the Boolean function of the combinational circuit in Fig. 2 is $B(X_1, X_2, Y_1, Y_2) = X_1Y_1 + X_2Y_2$. Then $B(1,1, Y_1, Y_2) = Y_1 + Y_2$. Since there are three vectors $(b_1, b_2) \in \{0,1\}^2$ making $B(1,1, b_1, b_2)$=1, the value $g(1,1) = 3$. Similarly, we can derive $g(0,0) = 0$, $g(0,1) = 2$, and $g(1,0) = 2$. Since $m = 2$, according to Eq. (1), the output function is

$$f(x_1, x_2) = \frac{1}{2}(1 - x_1)x_2 + \frac{1}{2}x_1(1 - x_2) + \frac{3}{4}x_1x_2. \quad (2)$$ □

The function of the form shown in Eq. (1) is called a *binary combination polynomial (BCP)* [11]. If we expand a BCP, we can obtain a *multi-linear polynomial (MLP)* of the following form

$$f(x_1, \dots, x_n) = \sum_{(a_1,\dots,a_n)\in\{0,1\}^n} \frac{c(a_1, \dots, a_n)}{2^m} \prod_{j=1}^{n} x_j^{a_j},$$

where $c(a_1, \dots, a_n)$'s are integers. The degree of each variable in an MLP is at most 1. For example, expanding Eq. (2), we can obtain an MLP

$$f(x_1, x_2) = \frac{1}{2}x_1 + \frac{1}{2}x_2 - \frac{1}{4}x_1x_2.$$

B. Synthesis of General Function

Given a target function, a procedure was proposed in [11] to synthesize a stochastic circuit of the general form to realize that function. We use an example to illustrate the procedure. Since the computation realized by a general-form stochastic circuit is a polynomial, the target function will be first approximated as a polynomial.

Now suppose the polynomial is $f = \frac{1}{4}x_1^2 + \frac{1}{2}x_2$. Next, it will be transformed into an MLP. This is achieved by introducing two new variables $x_{1,1}$ and $x_{1,2}$ with their values both set as x_1. The MLP obtained is

$$f = \frac{1}{4}x_{1,1}x_{1,2} + \frac{1}{2}x_2.$$

The next step is to map the MLP into a BCP. By a procedure shown in [11], the result is

$$f = \frac{1}{2}(1 - x_{1,1})(1 - x_{1,2})x_2 + \frac{1}{2}(1 - x_{1,1})x_{1,2}x_2 + \frac{1}{2}x_{1,1}(1 - x_{1,2})x_2 + \frac{1}{4}x_{1,1}x_{1,2}(1 - x_2) + \frac{3}{4}x_{1,1}x_{1,2}x_2. \quad (3)$$

Assume that the number of Y-variables is $m = 2$ and the Boolean function is $B(X_{1,1}, X_{1,2}, X_2, Y_1, Y_2)$. Comparing Eq. (3) with Eq. (1), we can obtain that the Boolean function should satisfy that

$$g(0,0,0) = 0, g(0,0,1) = 2, g(0,1,0) = 0, g(0,1,1) = 2,$$
$$g(1,0,0) = 0, g(1,0,1) = 2, g(1,1,0) = 1, g(1,1,1) = 3.$$

However, since $x_{1,1} = x_{1,2} = x_1$, the terms $(1 - x_{1,1})x_{1,2}x_2$ and $x_{1,1}(1 - x_{1,2})x_2$ are the same. Also, the terms $(1 - x_{1,1})x_{1,2}(1 - x_2)$ and $x_{1,1}(1 - x_{1,2})(1 - x_2)$ are the same. Therefore, the requirement for the Boolean function can be relaxed as follows

$$g(0,0,0) = 0, g(0,0,1) = 2, g(0,1,0) + g(1,0,0) = 0,$$
$$g(0,1,1) + g(1,0,1) = 4, g(1,1,0) = 1, g(1,1,1) = 3. \quad (4)$$

In the general case, suppose the target polynomial has k variables $x_1, \dots, x_k$ and the degree of x_i is d_i, for $i = 1, \dots, k$. Define $n = \sum_{i=1}^{k} d_i$. To transform the original target into an MLP, we will introduce n new variables $x_{1,1}$, ..., x_{1,d_1}, ..., $x_{i,1}$, ..., x_{i,d_i}, ..., $x_{k,1}$, ..., x_{k,d_k}, with the values of $x_{i,1}, \dots, x_{i,d_i}$ all set to x_i. The BCP has 2^n product terms of the form

$$\prod_{i=1}^{k} \prod_{j=1}^{d_i} x_{i,j}^{a_{i,j}} \left(1 - x_{i,j}\right)^{1-a_{i,j}},$$

where $\left(a_{1,1}, \dots, a_{1,d_1}, \dots, a_{k,1}, \dots, a_{k,d_k}\right) \in \{0,1\}^n$. Each product term has a one-to-one correspondence to a vector $\left(a_{1,1}, \dots, a_{1,d_1}, \dots, a_{k,1}, \dots, a_{k,d_k}\right) \in \{0,1\}^n$. We call the vector *the characteristic vector* of the product term. We partition the set $\{0,1\}^n$ into $\prod_{i=1}^{k}(1 + d_i)$ *equivalence classes* $I(s_1, \dots, s_k)$, $0 \le s_1 \le d_1, \dots, 0 \le s_k \le d_k$, where

$$I(s_1, \dots, s_k) = \left\{\left(a_{1,1}, \dots, a_{k,d_k}\right) \in \{0,1\}^n : \sum_{j=1}^{d_i} a_{i,j} = s_i, \text{for all } i = 1, \dots, k\right\}.$$

Under the condition that for all $1 \le i \le k$, $x_{i,1} = \cdots = x_{i,d_i} = x_i$, two product terms are the same if and only if their characteristic vectors belong to the same equivalent class. Therefore, to realize the target polynomial, we only require that the sum of the g values over all the vectors in an equivalence class is equal to a specific constant. Mathematically, the requirement is that for all $0 \le s_1 \le d_1, \dots, 0 \le s_k \le d_k$

$$\sum_{\substack{\left(a_{1,1}, \dots, a_{k,d_k}\right) \\ \in I(s_1, \dots, s_k)}} g\left(a_{1,1}, \dots, a_{k,d_k}\right) = G(s_1, \dots, s_k), \qquad (5)$$

where $0 \le G(s_1, \dots, s_k) \le 2^m \prod_{i=1}^{k} \binom{d_i}{s_i}$ is a constant that can be derived by adding up the corresponding g values of an initial BCP transformed from the original target function.

The example shown before corresponds to a situation in which $k = 2, d_1 = 2$, and $d_2 = 1$. Then we have six equivalence classes

$I(0,0) = \{(0,0,0)\}, I(0,1) = \{(0,0,1)\},$
$I(1,0) = \{(0,1,0), (1,0,0)\}, I(1,1) = \{(0,1,1), (1,0,1)\},$
$I(2,0) = \{(1,1,0)\}, I(2,1) = \{(1,1,1)\}.$

Given the above equivalence classes, the requirement on the g values specified by Eq. (5) is same as Eq. (4) we derived before.

C. The Circuit Synthesis Problem

Eq. (5) shows a requirement on the Boolean function to realize the target polynomial. However, there are a large number of Boolean functions that can satisfy the requirement. In order to synthesize an optimal circuit, we need to find an optimal Boolean function that satisfies the requirement. For simplicity, we focus on two-level circuit in this work and we use the literal number of the sum-of-product (SOP) form as the cost measure. The optimization problem is stated as follows.

> Given an integer m and $\prod_{i=1}^{k}(1 + d_i)$ integers $G(0, \dots, 0)$, ... , $G(d_1, \dots, d_k)$ such that $0 \le G(s_1, \dots, s_k) \le 2^m \prod_{i=1}^{k} \binom{d_i}{s_i}$ for any $0 \le s_1 \le d_1, \dots, 0 \le s_k \le d_k$, determine an optimal Boolean function such that its g values satisfy Eq. (5).

The above problem has flexibility in determining the final Boolean function. However, it is different from the traditional logic minimization with don't cares or Boolean relation minimization problem [12]. The problem we consider here has a constraint on the number of input vectors belonging to a subset that make the function evaluate to 1. Thus, the determination of the output for an input vector will reduce the output choices of the other input vectors belonging to the same subset. In contrast, logic minimization with don't cares or Boolean relation minimization does not have that constraint. The determination of the output of an input vector does not reduce the output choices for the other input vectors. Therefore, solving the above problem requires a new method.

Suppose the Boolean function is $B(X_{1,1}, \dots, X_{1,d_1}, \dots, X_{k,1}, \dots, X_{k,d_k}, Y_1, \dots, Y_m)$. We represent it using a matrix, where the columns represent the X-variables and the rows represent the Y-variables. Both the columns and the rows are arranged in Gray code order. An example is shown in Fig. 3 for a case where $k = 1$, $d_1 = 3$, and $m = 2$.

$Y \backslash X$	000	001	011	010	110	111	101	100
00	1	1	1	1	1	1		
01	1	1	1	1	1	1		
11		1	1					
10		1	1					

Fig. 3. The matrix representation of the Boolean function $B(X_1, X_2, X_3, Y_1, Y_2) = \overline{X_1}\overline{Y_1} + X_2\overline{Y_1} + \overline{X_1}X_3$.

Using that matrix representation, the number $g\left(a_{1,1}, \dots, a_{k,d_k}\right)$ is equal to the number of ones in the column $a_{1,1} \dots a_{k,d_k}$. Then the optimization problem is to distribute $G(s_1, \dots, s_n)$ ones to columns corresponding to the vectors in the class $I(s_1, \dots, s_n)$ to achieve an optimal Boolean function. A method was proposed in the previous work [11] to find a good solution. It applies a greedy strategy to distribute the ones. Assume $l = \lfloor G(s_1, \dots, s_n)/2^m \rfloor$. Then the method sets the g values of the first l vectors in the class $I(s_1, \dots, s_n)$ as 2^m, the g value of the $(l+1)$-th vector as $(G(s_1, \dots, s_n) - 2^m l)$, and the g values of the remaining vectors as 0. The following example illustrates how the previous method works.

Example 2

Consider a case where $k = 1$, $d_1 = 3$, and $m = 2$. There are four equivalence classes for this case:

$I(0) = \{(0,0,0)\}, I(1) = \{(0,0,1), (0,1,0), (1,0,0)\},$
$I(2) = \{(0,1,1), (1,0,1), (1,1,0)\}, I(3) = \{(1,1,1)\}.$

Assume the sums of g values over all the vectors in each equivalence class are $G(0) = 2$, $G(1) = 6$, $G(2) = 6$, and

$G(3) = 2$. For equivalence classes $I(0)$ and $I(3)$, each of them covers one column. We set $g(0,0,0) = 2$ and $g(1,1,1) = 2$. For equivalence classes $I(1)$ and $I(2)$, each of them covers three columns. Since $\lfloor G(1)/2^m \rfloor = 1$, we assign $g(0,0,1) = 4$, $g(0,1,0) = 2$, and $g(1,0,0) = 0$. Similarly, for class $I(2)$, we assign $g(0,1,1) = 4$, $g(1,1,0) = 2$, and $g(1,0,1) = 0$. The final assignment of the ones is shown in Fig. 3. The Boolean function is $B = \overline{X_1}\overline{Y_1} + X_2\overline{Y_1} + \overline{X_1}X_3$, which has 6 literals. □

However, the previous method may not give an optimal solution. For the case shown in Example 2, a better assignment is shown in Fig. 4, which gives a function $B = \overline{Y_1}$. In this work, we explore a better solution to the optimization problem.

$Y\backslash X$	000	001	011	010	110	111	101	100
00	1	1	1	1	1	1	1	1
01	1	1	1	1	1	1	1	1
11								
10								

Fig. 4. The matrix representation of the Boolean function $B(X_1, X_2, X_3, Y_1, Y_2) = \overline{Y_1}$.

III. The Proposed Algorithm

In this section, we present the new algorithm. For simplicity, we focus on univariate polynomials, i.e., $k = 1$. Our work can be extended to handle multivariate polynomials. The only difference is that there are more equivalence classes for multivariate cases. For univariate case, we have $n = d_1$ and we assume the n X inputs are $X_1, \dots, X_n$.

The basic approach we use to construct an optimal solution is to add cubes one by one into the on-set of the Boolean function. Although the previous work also uses this strategy, it only adds cubes which cover minterms in the same equivalence class. In contrast, our method also adds cubes across different equivalence classes.

A. Preliminaries

Before presenting the details, we first introduce a few notations and definitions. We use $M(a_1, \dots, a_n, b_1, \dots, b_m)$ to denote the minterm corresponding to an input vector $(a_1, \dots a_n, b_1, \dots, b_m) \in \{0,1\}^{n+m}$. We say a minterm $M(a_1, \dots, a_n, b_1, \dots, b_m)$ is in an equivalence class $I(i)$ $(0 \le i \le n)$ if $(a_1, \dots, a_n) \in I(i)$.

We use a vector $(v_0, \dots, v_n)$ to represent numbers of unassigned minterms for $(n+1)$ equivalent classes. We call such a vector *problem vector*. Initially, the problem vector is equal to $(G(0), \dots, G(n))$, given by the problem specification. With cubes added into the on-set, the entries in the problem vector will be reduced. Eventually, when all the minterms have been decided, the problem vector will become a zero vector.

We can also represent a cube by a vector of length $(n+1)$. It is formed by the numbers of minterms of the cube in each equivalence class. We call such a vector *cube vector*. In order to distinguish it from the problem vector, we represent the cube vector using square brackets. For example, assume that $n = 2$ and $m = 1$. The cube X_1 contains four minterms $X_1X_2\overline{Y_1}$, $X_1\overline{X_2}\overline{Y_1}$, $X_1X_2Y_1$ and $X_1\overline{X_2}Y_1$, as shown in Fig. 5(a). The minterms $X_1\overline{X_2}\overline{Y_1}$ and $X_1\overline{X_2}Y_1$ are in the equivalence class $I(1)$ and the minterms $X_1X_2\overline{Y_1}$ and $X_1X_2Y_1$ are in the equivalence class $I(2)$. There are no minterms of the cube X_1 in the equivalence class $I(0)$. Therefore, the vector of the cube X_1 is $[0,2,2]$. Note that although each cube has a unique cube vector, a cube vector may correspond to a number of different cubes. For example, the cube X_2 has the same cube vector as the cube X_1, as shown in Fig. 5.

$Y_1\backslash X_1X_2$	00	01	11	10
0			1	1
1			1	1

(a) Cube X_1

$Y_1\backslash X_1X_2$	00	01	11	10
0		1	1	
1		1	1	

(b) Cube X_2

Fig. 5. Two different cubes of the same cube vector $[0, 2, 2]$.

Our approach splits the problem vector into a set of cube vectors. In order to manipulate on the vector, it is important to know the valid form of a cube vector. We have the following claim on this.

Theorem 1

A cube vector is of the form $\left[0, \dots, 0, 2^l\binom{r}{0}, 2^l\binom{r}{1}, \dots, 2^l\binom{r}{r}, 0, \dots, 0\right]$, where $0 \le r \le n$ and $0 \le l \le m$ are the numbers of the missing X-variables and missing Y-variables in the cube, respectively. The cube vector has t zeros at the beginning and $(n-t-r)$ zeros at the end, where $0 \le t \le n-r$ is equal to the number of uncomplemented X-variables in the cube and $(n-t-r)$ is equal to the number of complemented X-variables in the cube. □

Proof: Consider the matrix representation of the cube. Since there are l missing Y-variables in the cube, the cube covers 2^l rows and all the covered rows have the same pattern. Note that each covered row is also a cube, which contains all the m Y-variables. Therefore, we only need to show that for such a cube, its cube vector is of the form $\left[0, \dots, 0, \binom{r}{0}, \binom{r}{1}, \dots, \binom{r}{r}, 0, \dots, 0\right]$.

We consider the X-variables of the cube. Suppose that there are t uncomplemented X-variables and r missing X-variables in the cube. Then, the cube has $(n-t-r)$ complemented X-variables. The cube covers 2^r minterms, among which $\binom{r}{i}$ minterms are in the equivalence class $I(t+i)$, for $i = 0, \dots, r$. For any $0 \le j < t$ or $t + r < j \le n$, there are no minterms of the cube in the equivalence class $I(j)$. Therefore, the cube vector is of the form $\left[0, \dots, 0, \binom{r}{0}, \binom{r}{1}, \dots, \binom{r}{r}, 0, \dots, 0\right]$, in which there are t zeros at the beginning and $(n-t-r)$ zeros at the end. □

Example 3

Assume that $n = 3$ and $m = 2$. Then, the cube X_1Y_1 contains 8 minterms $X_1\overline{X_2}\,\overline{X_3}\,Y_1\overline{Y_2}$, $X_1\overline{X_2}\,\overline{X_3}\,Y_1Y_2$, $X_1\overline{X_2}\,X_3\,Y_1\overline{Y_2}$, $X_1\overline{X_2}\,X_3\,Y_1Y_2$, $X_1X_2\,\overline{X_3}\,Y_1\overline{Y_2}$, $X_1X_2\,\overline{X_3}\,Y_1Y_2$, $X_1X_2X_3Y_1\overline{Y_2}$, $X_1X_2X_3Y_1Y_2$. Its cube vector is $[0,2,4,2] =$

$\left[0,2\binom{2}{0},2\binom{2}{1},2\binom{2}{2}\right]$. For this cube vector, $l=1$ is equal to the number of missing Y-variables and $r=2$ is equal to the number of missing X-variables. The number of zeros at the beginning is 1, which is equal to the number of uncomplemented X-variables in the cube. The number of zeros at the end is 0, which is equal to the number of complemented X-variables in the cube. □

B. The Basic Idea

As mentioned at the beginning of this section, our approach iteratively adds cubes into the on-set of the Boolean function. Each time a cube is added, some entries in the problem vector will be reduced. When the problem vector becomes zero, the Boolean function is constructed.

Generally, a cube added later may intersect with a cube added previously. However, in our approach, we restrict that a cube added later should be disjoint to any cubes added before. For simplicity, we call this restriction *disjointness constraint*. Although this restriction may cause some quality loss, it has two benefits. First, it makes the counting of minterms easy, because we do not need to consider the overlapped minterms. With a cube satisfying the disjointness constraint added, the problem vector can be easily updated by subtracting the cube vector from the original problem vector. Second, the constraint eliminates many redundant cases. For example, adding two non-disjoint cubes X_1 and X_2 is equivalent to adding two disjoint cubes X_1 and $\overline{X_1}X_2$. Note that although the Boolean function is constructed by adding disjoint cubes, the final Boolean function will be further simplified by the two-level logic optimization tool ESPRESSO [13]. Thus, the final result is a set of non-disjoint cubes corresponding to a minimum SOP expression.

In each iteration, when picking a cube, we also require that each entry in the cube vector of the cube is no larger than the corresponding entry in the current problem vector. For simplicity, we call this constraint *capacity constraint*. If a cube satisfies both the disjointness constraint and the capacity constraint, we say the cube is *valid*.

In each iteration, we apply a greedy strategy in choosing the cube to be added: we choose the largest cube among all valid cubes. The reasons for this are 1) in two-level logic synthesis, larger cubes have fewer literals and 2) with the largest cubes added, the problem vector is reduced most. The details of how we choose the largest valid cube will be discussed in Section III-C. The procedure of choosing the largest valid cube involves obtaining a cube corresponding to the cube vector, which will be discussed in Section III-D. Since at each iteration, there may exist more than one largest valid cube for the current problem setup, we actually apply a branch-and-bound algorithm to find the optimal solution, which will be discussed in Section III-E.

C. Selecting the Largest Valid Cube

Suppose that at the beginning of one iteration, the problem vector is $(v_0,\dots v_n)$. Let s be the sum of all the entries in the problem vector, i.e., $s=\sum_{i=0}^{n} v_i$. Assume $q=\lfloor \log_2 s \rfloor$. Since the largest valid cube satisfies the capacity constraint, it contains at most 2^q minterms. Our method to find the largest valid cube first checks whether there exists a valid cube with 2^q minterm.

According to Theorem 1, the cube vector should be in the form of $\left[0,\dots,0,2^l\binom{r}{0},2^l\binom{r}{1},\dots,2^l\binom{r}{r},0,\dots,0\right]$, where $0\le r\le n$ and $0\le l\le m$. Furthermore, since the cube contains 2^q minterms, we require that $l+r=q$. We will examine all cube vectors that satisfy the above two requirements and keep those which also satisfy the capacity constraint. Then, for each kept cube vector, we will check whether it has a corresponding cube that satisfies the disjointness constraint. The details of how to check the existence of such a cube will be discussed in Section III-D. If such a cube exists, it is a largest valid cube.

Example 4

Suppose $n=2$, $m=2$, and we are given an initial problem vector of $(2,5,2)$. The sum of all the entries in the problem vector is 9. Thus, the largest valid cube has at most 8 minterms. We first check whether there exists any valid cube with 8 minterms. This type of cubes should be in the form of $\left[0,\dots,0,2^l\binom{r}{0},2^l\binom{r}{1},\dots,2^l\binom{r}{r},0,\dots,0\right]$ with $0\le r\le 2$, $0\le l\le 2$, and $l+r=3$. Given the constraint, we have either $l=2$ and $r=1$, or $l=1$ and $r=2$. Thus, the possible cube vectors are $[0,4,4]$, $[4,4,0]$, and $[2,4,2]$. Among these three cube vectors, only the cube vector $[2,4,2]$ satisfies the capacity constraint. Then, we will further check whether it has a corresponding cube satisfying the disjointness constraint. Since no cubes have been added yet, we can find a valid cube for the cube vector $[2,4,2]$, for example, the cube Y_1. This cube is one largest valid cube. □

In some situations, there may not exist a valid cube with 2^q minterms because either the capacity constraint or the disjointness constraint is violated. The following is an example.

Example 5

Suppose $n=2$, $m=3$, and we are given an initial problem vector of $(1,3,7)$. The sum of all the entries in the problem vector is 11. Thus, the largest valid cube has at most 8 minterms. The possible cube vectors of 8 minterms are $[0,0,8]$, $[0,8,0]$, $[8,0,0]$, $[0,4,4]$, $[4,4,0]$, and $[2,4,2]$. However, none of these cube vectors satisfy the capacity constraint. Therefore, we cannot find a valid cube with 8 minterms. □

If there exists no valid cube with 2^q minterms, then we will reduce the minterm number by half and check whether there exists a valid cube with 2^{q-1} minterms. This procedure will be repeated until we are able to find a valid cube with 2^i minterms for some $0\le i\le q$. Then, that cube is the largest valid cube. Since in the worst case, we can always find a minterm that is valid, the procedure guarantees to terminate at some point.

However, in general cases, the largest valid cube is not unique. This is due to the existence of more than one largest cube vector that satisfies the capacity constraint and the existence of more than one cube for a cube vector.

Example 6

Suppose $n=2$, $m=3$, and we are given an initial problem vector of $(4,8,3)$. The largest possible cube has 8 minterms. Among all cube vectors of 8 minterms, three satisfy the capacity

constraint: [0,8,0], [4,4,0], and [2,4,2]. Furthermore, there exists more than one cube that satisfies the disjointness constraint for each of the three cube vectors. For example, for the cube vector [0,8,0], it corresponds to cubes $X_1\overline{X_2}$ and $\overline{X_1}X_2$, which satisfy the disjoint constraint. Therefore, there exist more than one largest valid cubes for this case. □

When there are multiple choices of the largest valid cubes, we want to evaluate all of them and choose the best one. For this purpose, we apply a branch-and-bound algorithm to find an optimal Boolean function. The details of it will be discussed in Section III-E.

D. Obtainting Cubes for a Cube Vector

In this section, we discuss one important procedure in selecting the largest valid cube: obtaining cubes for a given cube vector that satisfies the disjointness constraint. Since a cube is composed of X-variables and Y-variables, the procedure is divided into two parts: determining the X-variables and determining the Y-variables.

The X-variables are determined based on the form of the cube vector. As shown in Theorem 1, if the vector is of the form $\left[0,\dots,0,2^l\binom{r}{0},2^l\binom{r}{1},\dots,2^l\binom{r}{r},0,\dots,0\right]$ where there are t zeros at the beginning and $(n-t-r)$ zeros at the end, then the set of X-variables is composed of t uncomplemented X-variables and $(n-t-r)$ complemented X-variables. For example, if $n=3$ and the cube vector is of the form [0,4,4,0], then the possible X-variable cubes are $X_1\overline{X_2}$, $X_1\overline{X_3}$, $X_2\overline{X_1}$, $X_2\overline{X_3}$, $X_3\overline{X_1}$, and $X_3\overline{X_2}$.

Next, for each set of possible X-variables, we will further determine all sets of Y-variables so that the cube formed by these X-variables and Y-variables satisfies the disjointness constraint. According to Theorem 1, the set of Y-variables we need to pick consists of $(m-l)$ Y-variables. To obtain all valid sets of Y-variables, we can simply enumerate all cubes consisting of $(m-l)$ Y-variables and keep those when combined with the X-variable cube do not overlap with the current Boolean function. However, we could find a large number of valid Y-variable cubes, which increases the number of largest valid cubes. In order to reduce the choices, in our implementation, we enumerate all cubes with $(m-l)$ Y-variables in the Gray code order and keep the first valid Y-variable cube for each set of possible X-variables.

E. Branch-and-Bound Algorithm

As we mentioned before, in each iteration, there may exist more than one largest valid cube. If this happens, it is hard to decide which one will minimize the literal number of the final Boolean function. Therefore, we apply a branch-and-bound algorithm to evaluate all possible cube choices. An example of the search tree is shown in Fig. 6. Each leaf of the search tree corresponds to a final solution, represented by a set of cubes. Each internal node stores a partial solution composed of a set of cubes added and the remaining problem vector. The root is the initial problem vector. At each internal node, the multiple choices of the largest valid cubes for the current problem vector lead to multiple branches from the node.

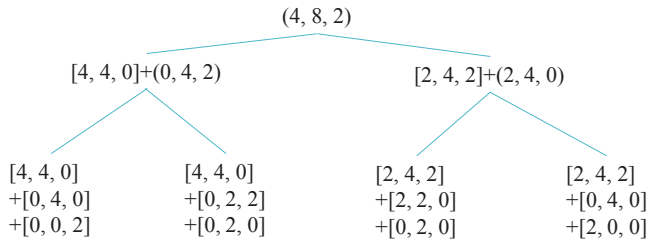


Fig. 6. An illustration of the solution tree for the problem with the problem vector $(4,8,2)$ and $m=2$. Note that for simplicity, we use a cube vector to represent a cube and we only show a partial tree.

Algorithm 1. Branch-and-bound algorithm to find optimal function.

1. **inputs**: problem vector $v=(G_0,\dots,G_n)$ and an integer m
2. **outputs**: the set of cubes of the final Boolean function B
3. initialize a node N: $N.vector \Leftarrow v$; $N.cube\_set \Leftarrow \emptyset$;
4. initialize the optimal literal number $n_o \Leftarrow \infty$;
5. initialize the optimal cube set $S_o \Leftarrow \emptyset$;
6. push the node N into an empty stack Stk;
7. **while** Stk is not empty **do**
8. pop a node N out of Stk;
9. find a list L of largest valid cubes for $N.vector$, $N.cube\_set$, and m;
10. **for** each cube C in the list L **do**
11. **if** $lit\_count(N.cube\_set \cup C) < n_o$ **then**
12. $N_{new}.vector \Leftarrow N.vector - vector(C)$;
13. $N_{new}.cube\_set \Leftarrow N.cube\_set \cup C$;
14. **if** $N_{new}.vector = 0$ **then** // reach a leaf
15. $n_o \Leftarrow lit\_count(N_{new}.cube\_set)$;
16. $S_o \Leftarrow N_{new}.cube\_set$;
17. **else**
18. push the node N_{new} into Stk;
19. **end if**
20. **end if**
21. **end for**
22. **end while**
23. **return** S_o;

In order to apply a brand-and-bound algorithm, we need a lower bound on the candidate solutions from a branch. We choose the lower bound as the minimum literal number for the set of cubes that forms a partial solution at a branch. For example, for the branch $[2,4,2]+(2,4,0)$ shown in Fig. 6, its lower bound is the minimum literal number for the cube with the cube vector [2,4,2]. Strictly speaking, the minimum literal number for the set of chosen cubes at a branch may not be the lower bound for that branch, because with more cubes determined later, it is possible to reduce the literal count due to cube expansion and redundant cube removal. However, since the cubes selected later are no larger than any of the cubes already chosen, it is more likely that with more cubes selected, the literal count will increase. Thus, we use the proposed method to obtain the lower bound. A branch will be pruned if the lower bound for the branch is larger than or equal to the minimum literal count for the best solution obtained so far. In practice, the exact minimum literal number for a set of cubes is computationally expensive to obtain. Instead, we call the powerful two-level logic optimization tool ESPRESSO [13] to estimate the minimum value. Algorithm 1 summarizes the

proposed branch-and-bound algorithm to find an optimal solution. Note that we explore the solution tree using the depth-first traversal.

IV. Speed-up Techniques

Although the branch-and-bound algorithm deletes some unpromising branches, there are still too many branches to process as the degree of the polynomial increases, which increases the runtime considerably. However, there are numerous branches unnecessary to process, either because they are unpromising or because they produce the same results. In this section, we present several techniques to speed up the algorithm with only small quality loss.

A. Removing Branches with Duplicated Cube Sets

For a node in the search tree, even though the sum of all entries in its problem vector is in the interval $[2^q, 2^{q+1} - 1]$, the size of the largest valid cube may not be 2^q. Example 5 shows such a case. If this happens, we may add in sequence multiple cubes of the same size of 2^u, where $u < q$ is an integer. In the original branch-and-bound algorithm, the order that these cubes are added can produce different branches. Nevertheless, in most cases, different orders will finally lead to the same results.

Example 7

Suppose $n = 2$, $m = 3$, and the initial problem vector is $(1,6,2)$. We cannot extract a valid cube of size 8 from the initial problem vector. As a result, the largest valid cube is of size 4. Its cube vector is either $[1,2,1]$ or $[0,4,0]$. With the original algorithm, if the first cube selected is of the cube vector $[1,2,1]$, then the second cube selected will be of the cube vector $[0,4,0]$. On the other hand, if the first cube selected is of the cube vector $[0,4,0]$, then the second cube selected will be of the cube vector $[1,2,1]$. These two branches from the root node will produce the same results. □

Those branches with the same set of cubes as a branch explored before are unnecessary to be explored again. To remove them, we keep track of the sets of cube vectors we have already examined. If the set of the cube vectors at the current branch has been examined before, the branch will be pruned.

B. Bounding by the Optimal Cost at Each Level

In the original algorithm, a branch is pruned only when its lower bound exceeds the value of the optimal full solution known so far. In practice, given that each time we always add a largest valid cube, it is very likely that for any level i in the search tree, the cost of the partial solution at level i in a branch that will be pruned later is larger than the cost of the optimal partial solution at level i. In other words, only those branches with costs close to the optimal partial solution at each level are promising in leading to the optimal full solution. Therefore, we propose another speed-up technique which prunes branches based on the cost of the optimal partial solution at each level. With this technique, we can find and prune many unpromising branches earlier. However, the proposed method is just a heuristic. In order to reduce the quality loss caused by applying this heuristic, we choose the bound at each level as the cost of the optimal partial solution at the current level multiplied by a constant $m_l > 1$. We will only delete those branches whose costs exceed the bound. In real implementation, since we traverse the solution tree in a depth-first way, the optimal partial solution is obtained among all the explored nodes at the current level.

C. Limiting Update Count and Explored Node Number

The previous two speed-up techniques focus on eliminating unpromising branches. However, for some extreme cases, the numbers of nodes explored could still be very large. In order to further reduce the runtime for these extreme cases, we impose limits on the update count and the number of explored nodes.

Our algorithm will update the optimal solution if the current solution is no worse than the optimal one recorded. As a result, each update will either improve the result or leave it unchanged. Experimental results showed that with more updates, the improvement will gradually reduce. Therefore, we consider the solution to be optimal enough after a specific number of updates. Thus, we set a limit on the update number and terminate the algorithm once the limit is reached. From our experimental results, we set this limit as 3. The quality loss is negligible.

Even though limiting the updating number can further improve the runtime for some extreme cases, there are still some cases for which a large number of nodes are explored between two consecutive updates. In our experiment, there is a recorded case for which after the second update, the algorithm processed 16463 other nodes to reach the third update. It took about 57 minutes to explore these nodes, but no improvement was made for the third update. Therefore, we also set a limit on the number of explored nodes. The algorithm records the number of nodes explored. Once the initial solution has been found, the number of nodes explored will be compared against the limit and the algorithm will terminate once the limit is reached. In our experiment, the limit is often set from 15 to 30 for $3 \leq n \leq 7$ and $3 \leq m \leq 7$, or larger if needed. With a larger limit, we can achieve a better solution.

V. Experiment Results

In this section, we show the experimental results of the proposed algorithm. All the experiments were conducted on a desktop with 3.20GHz Intel(R) Core(TM) i5-4570 CPU and 16.0 GB RAM. ESPRESSO is used to evaluate the literal count [13].

We applied the proposed branch-and-bound algorithm with the speed-up technique to univariate polynomials with $3 \leq n \leq 7$ and $3 \leq m \leq 7$. For each pair of n and m, we generated 50 random cases and obtained the average result. Table I shows for each pair of n and m, the average percentage of literal count reduction by the proposed algorithm over the method in [11], the percentage of improved or unchanged cases among all 50 cases, and the average runtime in seconds of the proposed algorithm. The literal reduction percentage, the percentage of improved and unchanged cases, and the runtime are shown in the first row, the second row, and the third row of each cell. For example, for $n = 4$ and $m = 4$, the proposed algorithm saves 13% literal count on average. 100% of the 50 cases have their literal counts reduced or unchanged. The average runtime is 1.42s.

TABLE I. THE AVERAGE PERCENTAGE OF LITERAL COUNT REDUCTION BY THE PROPOSED ALGORITHM OVER THE PREVIOUS METHOD [11] (IN THE FIRST ROW OF EACH CELL), THE PRECENTAGE OF IMPROVED AND UNCHANGED CASES (IN THE SECOND ROW OF EACH CELL), AND THE AVERAGE RUNTIME OF THE PROPOSED ALGORITHM (IN THE THIRD ROW OF EACH CELL) FOR DIFFERENT PAIRS OF n AND m.

$m\backslash n$	3	4	5	6	7
3	6% 100% 0.44s	12% 90% 0.60s	19% 94% 1.60s	18% 88% 2.56s	26% 100% 5.20s
4	3% 98% 0.60s	13% 100% 1.42s	16% 88% 2.46s	22% 92% 4.20s	29% 94% 7.48s
5	2% 92% 0.88s	13% 100% 1.14s	18% 92% 2.92s	18% 84% 8.74s	26% 88% 17.1s
6	2% 92% 1.34s	12% 96% 3.90s	15% 86% 7.04s	18% 88% 16.6s	23% 90% 42.6s
7	2% 88% 2.46s	10% 94% 8.54s	14% 86% 16.6s	17% 90% 39.2s	22% 90% 116s

It can be seen that in the average sense, the proposed algorithm reduces the literal count compared to the previous method. When n is small, the literal count reduction is small because the previous greedy method is able to find a good solution among limited choices. However, as n increases, more percentage of literals is saved. For $n = 7$, the literal saving reaches up to 29%. For each pair of n and m, at least 84% of cases have their literal counts improved or unchanged. For some pairs of n and m, all 50 cases have their literal counts improved or unchanged. With the increase of n and m, the runtime also increases, which is due to the growth of the search space. Notice that the runtime of the previous method is negligible compared to ours, due to its greedy nature. However, the runtime of our algorithm is still affordable. In situations where better circuit quality is pursued, our method gives a better solution under a reasonable amount of runtime.

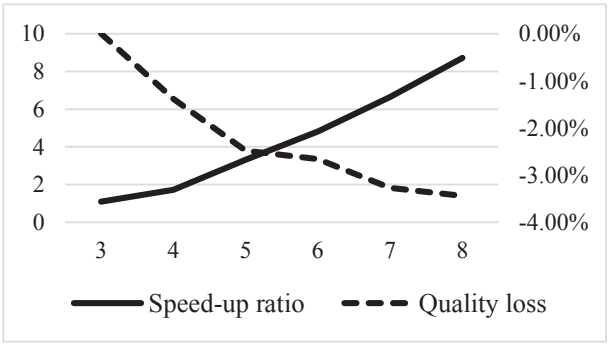


Fig. 7. Comparison between the branch-and-bound algorithm without acceleration and the accelrated algorithm for $n = 3$ and $3 \leq m \leq 8$.

We also compared the proposed accelerated algorithm to the branch-and-bound algorithm without using the speed-up techniques. Due to the inefficiency of the algorithm without acceleration, the comparison was only done for polynomials of degree $n = 3$ and $3 \leq m \leq 8$. Fig. 7 plots the speed-up ratio (shown in solid line, y-axis on the left) and the quality loss (shown in dashed line, y-axis on the right) of the accelerated algorithm for different m values. For the quality loss, the more negative the value is, the more loss the accelerated algorithm has. We can see from Fig. 7 that as the problem instance grows, more runtime can be saved through the speed-up techniques. However, the quality loss also increases. Nevertheless, the quality loss is small. Indeed, in terms of the absolute value, the average quality loss is smaller than one literal. Thus, the speed-up techniques have a negligible impact on the quality.

VI. CONCLUSION

In this work, we proposed a search-based method for synthesizing stochastic circuits. The synthesis problem we considered here is different from the traditional logic synthesis problem in that there exist many different Boolean functions to realize a target computation. We proposed a branch-and-bound algorithm to systematically explore the solution space. A final solution is obtained by adding a series of cubes to the on-set of the Boolean function. We also provided several speed-up techniques. The experimental results showed that our algorithm produces smaller circuits than a previous greedy approach, especially when the target polynomial has a high degree.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China (NSFC) under Grant No. 61204042 and 61472243.

REFERENCES

[1] B. Gaines, "Stochastic computing systems," in *Advances in Information Systems Science*. Plenum, 1969, vol. 2, ch. 2, pp. 37–172.

[2] W. Qian, X. Li, M. Riedel, K. Bazargan, and D. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 93–105, 2011.

[3] A. Alaghi, C. Li, and J. Hayes, "Stochastic circuits for real-time imageprocessing applications," in *DAC*, 2013, pp. 1–6.

[4] S. Tehrani, S. Mannor, and W. Gross, "Fully parallel stochastic LDPC decoders," *IEEE Transactions on Signal Processing*, vol. 56, no. 11, pp. 5692–5703, 2008.

[5] B. Brown and H. Card, "Stochastic neural computation II: Soft competitive learning," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 906–920, 2001.

[6] B. Brown and H. Card, "Stochastic neural computation I: Computational elements," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 891–905, 2001.

[7] P. Li, W. Qian, M. Riedel, K. Bazargan, and D. Lilja, "The synthesis of linear finite state machine-based stochastic computational elements," in *ASPDAC*, 2012, pp. 757–762.

[8] P. Li, D. Lilja, W. Qian, K. Bazargan, and M. Riedel, "The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic," in *ICCAD*, 2012, pp. 480–487.

[9] A. Alaghi and J. Hayes, "A spectral transform approach to stochastic circuits," in *ICCD*, 2012, pp. 315–321.

[10] T.-H. Chen and J. Hayes, "Equivalence among stochastic logic circuits and its application," in *DAC*, 2015, pp. 131-136.

[11] Z. Zhao and W. Qian, "A general design of stochastic circuit and its synthesis," in *DATE*, 2015, pp. 1467-1472.

[12] D. Baneres, J. Cortadella, and M. Kishinevsky, "A recursive paradigm to solve Boolean relations," in *DAC*, 2004, pp. 416-421.

[13] R. Rudell, "Multiple-valued logic minimization for PLA synthesis," *Technical Report, University of. California, Electronics Research Laboratory, Berkeley,* 1986.

A Deterministic Approach to Stochastic Computation

Devon Jenson and Marc Riedel
jens1172@umn.edu, mriedel@umn.edu
Department of Electrical Engineering, University of Minnesota

***Abstract*— Stochastic logic performs computation on data represented by random bit streams. The representation allows complex arithmetic to be performed with very simple logic, but it suffers from high latency and poor precision. Furthermore, the results are always somewhat inaccurate due to random fluctuations. The random or pseudorandom sources required to generate the representation are costly, consuming a majority of the circuit area (and diminishing the overall gains in area). In this paper, we show that randomness is *not* a requirement for this computational paradigm. If properly structured, the same arithmetical constructs can operate on *deterministic* bit streams, with the data represented uniformly by the fraction of 1's versus 0's. This paper presents three approaches for the computation: relatively prime stream lengths, rotation, and clock division. The three methods are evaluated on a collection of arithmetical functions. Unlike stochastic methods, all three of our deterministic methods produce completely accurate results. The cost of generating the deterministic streams is a small fraction of the cost of generating streams from random/pseudorandom sources. Most importantly, the latency is reduced by a factor of $\frac{1}{2^n}$, where n is the equivalent number of bits of precision.**

I. Introduction

In the paradigm of stochastic computation, digital logic is used to perform computation on random bit streams, where numbers are represented by the probability of observing a one [1], [2], [3], [4], [5]. The benefit of such a stochastic representation is that complex operations can be performed with very simple logic. For instance, multiplication can be performed with a single AND gate and scaled addition can be performed with a single multiplexer unit. One obvious drawback is that the computation has very high latency, due to the length of the bit streams. Another is that the computation suffers from errors due to random fluctuations and correlations between the streams. These effects worsen as the circuit depth and the number of inputs increase [5]. A certain degree of accuracy can be maintained by re-randomizing bit streams, but this is an additional expense [6]. While the logic to perform the computation is simple, generating random or pseudorandom bit streams is costly. Indeed, in prior work, pseudorandom constructs such as linear feedback shift registers (LFSRs) accounted for as much as 90% of the area of stochastic circuit designs [3], [4]. This significantly diminishes the area benefits.

This paper suggests that randomness is *not* a requirement for the paradigm. We show that the same computation can be performed on deterministically generated bit streams. The results are completely accurate, with no random fluctuations. Without the requirement of randomness, bit streams can be generated inexpensively. Most importantly, with our approach, the latency is reduced by a factor of approximately $\frac{1}{2^n}$, where n is the equivalent number of bits of precision. (For example, for the equivalent of 10 bits of precision, the bit stream length is reduced from 2^{20} to only 2^{10}.) As with stochastic computation, all bits in our deterministic streams are weighted equally. Accordingly, our deterministic circuits display the same high degree of tolerance to soft errors.

This paper is structured as follows: Section II presents background information on stochastic computing. Section III gives an intuitive view of why stochastic computation works. Section IV shows how computation can be performed on deterministic bit streams in a manner analogous to computation on stochastic bit streams. Section V presents three deterministic methods and describes their circuit implementations. Section VI compares the hardware complexity and latency of stochastic and deterministic methods.

II. Background on Stochastic Logic

In a paradigm first advocated by Gaines, logical computation is performed on stochastic bit streams [1]. There are two possible coding formats: a unipolar format and a bipolar format [1]. These two formats are conceptually similar and can coexist in a single system. In the unipolar coding format, a real number x in the unit interval (i.e., $0 \leq x \leq 1$) corresponds to a bit stream $X(t)$ of length L, where $t = 1, 2, ..., L$. The probability that each bit in the stream is one is $P(X = 1) = x$. For example, the value $x = 0.3$ could be represented by a random stream of bits such as 0100010100, where 30% of the bits are one and the remainder are zero. In the bipolar coding format, the range of a real number x is extended to $-1 \leq x \leq 1$. The probability that each bit in the stream is one is $P(X = 1) = \frac{x+1}{2}$. An advantage of the bipolar format is that it can deal with negative numbers directly. However, given the same bit stream length, L, the precision of the unipolar format is twice that of the bipolar format. For what follows, unless stated otherwise, our examples will use the unipolar format.

The synthesis strategy with stochastic logic is to cast logical computations as arithmetic operations in terms of probabilities. Two simple arithmetic operations – multiplication and scaled addition – are illustrated in Figure 1.

- **Multiplication**. Consider a two-input AND gate, shown in Figure 1(a). Suppose that its inputs are two independent bit streams X_1 and X_2. Its output is a bit stream Y, where

$$\begin{aligned} y = P(Y = 1) &= P(X_1 = 1 \text{ and } X_2 = 1) \\ &= P(X_1 = 1)P(X_2 = 1) = x_1 x_2. \end{aligned}$$

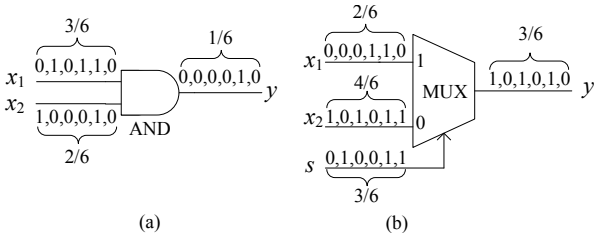


Fig. 1: Stochastic implementation of arithmetic operations: (a) Multiplication; (b) Scaled addition.

Thus, the AND gate computes the product of the two input probability values.

- **Scaled Addition**. Consider a two-input multiplexer, shown in Figure 1(b). Suppose that its inputs are two independent stochastic bit streams X_1 and X_2 and its selecting input is a stochastic bit stream S. Its output is a bit stream Y, where

$$\begin{aligned} y &= P(Y=1) \\ &= P(S=1)P(X_1=1)+P(S=0)P(X_2=1) \\ &= sx_1+(1-s)x_2. \end{aligned}$$

(Note that throughout the paper, multiplication and addition represent *arithmetic* operations, not Boolean AND and OR.) Thus, the multiplexer computes the scaled addition of the two input probability values.

III. Intuitive View of Stochastic Computation

Before presenting our methods, we present two intuitive explanations of why stochastic computation works: computing on averages and discrete convolution.

A. Taking A Look At the Average

Stochastic computation is framed as computation on *probabilities*. This is, of course, an abstraction of what is happening at the bit level. Computation is happening in a statistical sense on the average number of ones and zeros. Because the probability represented by a bit stream is equivalent to its expected value, we can instead view bit streams by the number of ones and zeros we would *expect* to see on average. For example, if we say that bit stream A represents a probability $p_A = 2/3$, this is equivalent to saying that we expect to see two ones for every three bits. In general, the number formats (unipolar, bipolar, etc.) are all defined in terms of the average number of ones and zeros. For example, the probability p of unipolar and bipolar bit streams are given by,

$$p_{\text{uni}} = \frac{N_1}{N_1+N_0} \qquad p_{\text{bi}} = \frac{N_1-N_0}{N_1+N_0}, \tag{1}$$

where N_1 and N_0 are the average number of ones and zeros. Therefore, by showing how each logic gate manipulates the average number of ones and zeros, the operation of the logic gate can be expressed independently of any particular number format.

Using two independent bit streams in a unipolar format, an AND gate multiplies their probabilities. Labeling the input bit streams as A and B, the probability of the output bit stream C is given by,

$$p_C = p_A p_B = \frac{N_{C1}}{N_{C1}+N_{C0}} = \frac{N_{A1}}{N_{A1}+N_{A0}}\frac{N_{B1}}{N_{B1}+N_{B0}} \tag{2}$$

where N_{C1} and N_{C0} represent the average number of ones and zeros in bit stream C. By multiplying out the right side of the Equation 2 and organizing the terms,

$$p_C = \frac{N_{A1}N_{B1}}{N_{A1}N_{B1}+(N_{A1}N_{B0}+N_{A0}N_{B1}+N_{A0}N_{B0})} \tag{3}$$

it can be seen that the fraction has the same form as the unipolar probability of Equation 1. Therefore, the average number of ones and zeros in bit stream C can be written in terms of the average number of ones and zeros in bit streams A and B,

$$\begin{aligned} N_{C1} &= N_{A1}N_{B1} \\ N_{C0} &= N_{A1}N_{B0}+N_{A0}N_{B1}+N_{A0}N_{B0} \end{aligned} \tag{4}$$

Denote the average number of ones and zeros in a bit stream X as the uniform number $N_{X1}\{1\}+N_{X0}\{0\}$. Distributing the AND operation (denoted by $\wedge$) gives the same result, as Equation 4:

$$\begin{aligned} &N_{C1}\{1\}+N_{C0}\{0\} = \\ &(N_{A1}\{1\}+N_{A0}\{0\})\wedge(N_{B1}\{1\}+N_{B0}\{0\}) = \\ &N_{A1}N_{B1}\{1\}+(N_{A1}N_{B0}+N_{A0}N_{B1}+N_{A0}N_{B0})\{0\} \end{aligned} \tag{5}$$

This shows that, by representing probabilities with independent random bit streams, an AND gate operates on average *proportions* of ones and zeros. In general, for any arbitrary logic gate with independent random bit streams A and B as inputs, the proportion of bits at the output is given by,

$$\begin{aligned} &N_{C1}\{1\}+N_{C0}\{0\} = \\ &(N_{A1}\{1\}+N_{A0}\{0\})\square(N_{B1}\{1\}+N_{B0}\{0\}) \end{aligned} \tag{6}$$

where the $\square$ symbol is replaced with any Boolean operator. This demonstrates that independent random bit streams passively maintain the property that the average bits of bit stream A are operated on with the average bits of bit stream B. Independence guarantees that each outcome of a bit stream (one or zero) will "see" the average number of ones and zeros of another bit stream. (Of course, if the bit streams are correlated, the output does not simply depend on the proportions of the bit streams in a straightforward way.) We conclude this section with two examples demonstrating the application of Equation 6.

Example 1 Assume we have two independent bit streams A and B with unipolar probabilities $p_A = 1/3$ and $p_B = 2/3$. This means *on average* we will observe a single one for every three bits of A and two ones every three bits of B.

If these bit streams are used as inputs to an AND gate, the average output and probability are given by,

$$N_{C1}\{1\}+N_{C0}\{0\}=$$
$$(1\{1\}+2\{0\})\wedge(2\{1\}+1\{0\})=$$
$$2\{1\wedge 1\}+1\{1\wedge 0\}+4\{0\wedge 1\}+2\{0\wedge 0\}=$$
$$2\{1\}+7\{0\}$$
$$\Rightarrow p_C=\frac{2}{2+7}=\frac{2}{9}$$

Example 2 Assume we have two independent bit streams A and B with bipolar probabilities $p_A = 4/6$ and $p_B = -3/5$. This means *on average* we will observe five ones for every six bits of A and a single one for every five bits of B. If these bit streams are used as inputs to an XNOR gate, the average output and probability are given by,

$$N_{C1}\{1\}+N_{C0}\{0\}=$$
$$(5\{1\}+1\{0\})\equiv(1\{1\}+4\{0\})=$$
$$5\{1\equiv 1\}+20\{1\equiv 0\}+1\{0\equiv 1\}+4\{0\equiv 0\}=$$
$$9\{1\}+21\{0\}$$
$$\Rightarrow p_C=\frac{9-21}{30}=-\frac{12}{30}$$

We can see from Examples 1 and 2 that we can find the output of a stochastic logic gate by taking an *average view* of the random bit streams and applying Equation 6.

B. Insight: Convolution

In basic terms, convolution consists of three operations: slide, multiply, and sum. For bit streams X and Y, each with L bits, the discrete convolution operation is

$$\sum_{i=1}^{L}\sum_{j=1}^{L}X_iY_j \tag{7}$$

The previous sections showed an AND gate multiplies proportions if each bit of one bit stream "sees" every bit of the other bit stream. Intuitively, this is equivalent to sliding one operand past the other.

Example 3 By sliding the following five-bit operands past each other,

$$\begin{array}{r}11100\\ 01100\longrightarrow\end{array}$$

each bit of the top operand sees two ones and three zeros and each bit of the bottom operand sees three ones and two zeros. In this way, a stochastic representation maintains the sliding of average bit streams.

A significant attribute of the stochastic representation is that it is a uniform encoding. Uniform numbers have the interesting property that the order of elements does not matter (i.e., the values are not weighted). This means partial products can be summed by simple concatenation. The following example demonstrates how this contrasts with binary multiplication.

Example 4 To multiply binary numbers, we perform bitwise multiplication and sum the weighted partial products. It takes two operations, bitwise multiply and sum, to go from binary inputs to a binary output. In contrast, to multiply uniform numbers the partial products simply need to be concatenated. By performing bitwise multiplications sequentially in time, concatenation is performed passively.

$$\begin{array}{r}10\\ \times 10\\ \hline 00\\ 10\end{array}\ \textit{Binary}\ \Big\}\xrightarrow{\text{sum}}100 \qquad \begin{array}{r}10\\ \times 10\\ \hline 00\\ 10\end{array}\ \textit{Uniform}\ \Big\}\xrightarrow{\text{concatenate}}1000$$

When using a uniform encoding, we do not need to sum the output of a logic gate in a particular order to get back the same representation as the inputs. We have "proportions in", "proportions out". In contrast, a weighted encoding requires additional circuitry to add the partial products in the correct manner.

This is why the arithmetic logic of a stochastic representation is so simple, the slide and sum operations of convolution are passively provided by the representation. Convolution of proportions only requires logic operations that result in bitwise (or element-wise) multiplication of the particular number format. Looking back at Example 2, we can think of the bipolar format as containing positive entities ('1's) and negative entities ('0's). Multiplication of entities that can be both positive and negative is defined by the following truth tables:

a	b	$a\times b$
-	-	+
-	+	-
+	-	-
+	+	+

a	b	$a\equiv b$
0	0	1
0	1	0
1	0	0
1	1	1

where the truth table on the right is identical to the truth table implemented by an XNOR gate. Therefore, by using the logic gate that multiplies the format of the entities, the average bit streams are convolved. This is why, in particular, multiplication and scaled addition are extremely simple operations with stochastic logic.

These insights lead us to ask: if the process can be described as multiplying every bit of one proportion by every bit of another proportion, or equivalently, by sliding and multiplying deterministic numbers, is randomness actually a requirement? Can the cost and latency be reduced if one approaches the problem deterministically?

IV. Deterministic Interpretation

A. A Link Between Representations

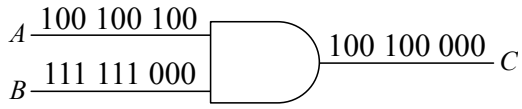
Equation 6 gives us a link between independent stochastic bit streams and deterministic bit streams. We can substitute independent stochastic bit streams for deterministic bit streams if Equation 6 holds, that is, if we maintain the property that proportion A sees every bit of proportion B.

Example 5 Two registers contain deterministic unipolar proportions $p_A = 1/3$ and $p_B = 2/3$. How can we generate bit streams such that a single AND gate performs multiplication?

From Equation 6 we know each bit of p_A must be ANDed with each bit of p_B. Therefore, each bit stream should be a

redundant encoding that maintains Equation 6. One method, shown in a later section, is to clock divide one proportion while the other repeats:

$$p_A = 1/3 = 100 \rightarrow 100\ 100\ 100$$
$$p_B = 2/3 = 110 \rightarrow 111\ 111\ 000$$

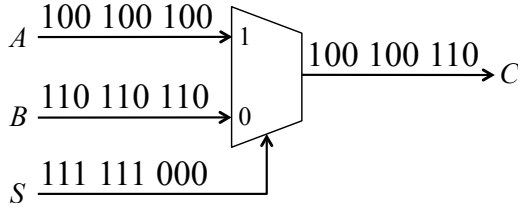


$$p_C = p_A p_B = 2/9$$

Example 6 Three registers contain deterministic unipolar proportions $p_A = 1/3$, $p_B = 2/3$, and $p_S = 2/3$. How can we generate bit streams such that a two-input multiplexer performs scaled addition?

A multiplexer performs the logical operation $(S \wedge A) \vee (\neg S \wedge B)$, where $\wedge$ is AND, $\vee$ is OR, and $\neg$ is NOT. It can be constructed using two AND gates, an inverter, and an OR gate. Because the circuit simply selects the output of either AND gate, bit streams A and B do not need to be independent from each other. Only bit stream S is required to be independent from A and B. Clock dividing S while A and B repeat performs scaled addition:

$$p_A = 1/3 = 100 \rightarrow 100\ 100\ 100$$
$$p_B = 2/3 = 110 \rightarrow 110\ 110\ 110$$
$$p_S = 2/3 = 110 \rightarrow 111\ 111\ 000$$



$$p_C = p_S p_A + (1 - p_S) p_B = 2/9 + 2/9 = 4/9$$

In these examples, Equation 6 is maintained on deterministic bit streams. (For convenience, we will use "independent" to describe both random and deterministic bit streams that obey Equation 6.)

B. Comparison of the Representations

A stochastic representation passively maintains the property that each bit of one proportion sees every bit of the other proportion, but this property occurs *on average*, meaning the bit streams have to be much longer than the resolution they represent due to random fluctuations. Equation 8 defines the bit stream length N required to estimate the average proportion within an error margin ε [7].

$$N > \frac{p(1-p)}{\varepsilon^2} \tag{8}$$

To represent a value within a binary resolution $1/2^n$, the error margin ε must equal $1/2^{n+1}$. Therefore, the bit stream must be greater than 2^{2n} uniform bits long, as the $p(1-p)$ term is at most equal to 2^{-2} [7]. This means the length of a stochastic bit stream increases *exponentially* with the desired resolution. This results in enormously long bit streams. For example, if we want to find the proportion of a random bit stream with 10-bit resolution ($1/2^{10}$), we'll have to observe at least 2^{20} bits. This is over a thousand times longer than the bit stream required by a deterministic uniform representation.

The computations also suffer from some level of correlation between bit streams. This can cause the results to bias away from the correct answer. For these reasons, stochastic logic has only been used to perform approximate computations.

Another related issue is that the LFSRs must be at least as long as the desired resolution in order to produce bit streams that are sufficiently random. A "Randomizer Unit", described in [4], uses a comparator and LFSR to convert a binary encoded number into a random bit stream. Each independent random bit stream requires its own generator. Therefore, circuits requiring i independent inputs with n-bit resolution need i LFSRs with length L approximately equal to $2n$. This results in the LFSRs dominating a majority of the circuit area.

By using deterministic bit streams, we avoid all problems associated with randomness while retaining all the computational benefits associated with a stochastic representation. For instance, the deterministic representation retains all the fault-tolerance properties attributed to a stochastic representation because it also uses a uniform encoding. To represent a value with resolution $1/2^n$ in a deterministic representation, the bit stream must be 2^n bits long. The computations are also completely accurate; they do not suffer from correlation.

To utilize a deterministic representation, bit stream generators must explicitly maintain Equation 6. The next section discusses three methods for generating independent deterministic bit streams and gives their circuit implementations. Without the requirement of randomness, the hardware cost of the bit stream generators is small.

V. Deterministic Methods

Each method is implemented using a bit stream generator formed by a group or interconnection of converter modules, as shown in Figure 2. Each converter module uses the general circuit topology of Figure 3. The modules are similar to the "Randomizer Unit"; the difference is that the LFSR is replaced by a deterministic number source. The generator takes in operands and generates bit streams such that:

$$G(C_0, C_1, ..., C_{i-1}) \rightarrow$$
$$C_0\{1\} + (2^{n_0} - C_0)\{0\} \ \square\ C_1\{1\} + (2^{n_1} - C_1)\{0\} \ \square \tag{9}$$
$$... \ \square\ C_{i-1}\{1\} + (2^{n_{i-1}} - C_{i-1})\{0\}$$

where i is total number of converter modules that make up the generator, n_i is the binary resolution of the ith individual module, C_i is an operand defining the proportion (or encoded value) of the bit stream, and each $\square$ can be any arbitrary logical operator.

Conceptually, we can view the converter module circuit of Figure 3 as selecting bits from a collection or array a (see Figure 4). The binary constant C determines the contents of the array: any elements less than index C contain a one,

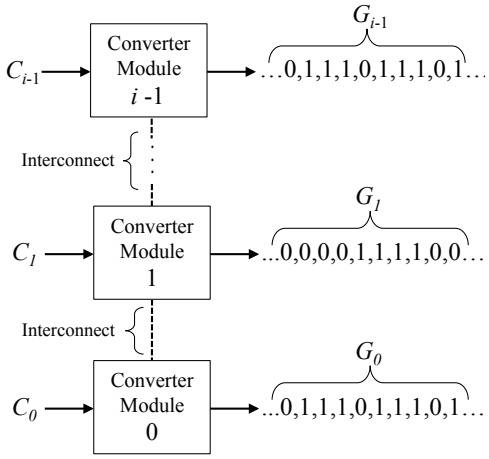


Fig. 2: Deterministic bit stream generator

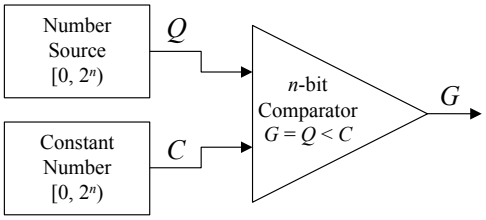


Fig. 3: Converter module

otherwise they contain a zero. In this way, C defines a uniform proportion of ones and zeros: $C \rightarrow C\{1\} + (2^n - C)\{0\}$. Using the array analogy, we can represent the proportion as a sequence $a_0, a_1, \ldots a_{2^n-1}$. The number Q, determined by the deterministic number source, points to different indices of the array. Each clock cycle, the element that Q points to is chosen as the next output. To ensure the generated bit stream has the same proportion of ones and zeros (i.e., represents the number C), Q must point to each index an equal number of times (within some period). In other words, the input C defines the proportion or collection of bits from which the bit stream is *uniformly* generated. The sequence generated by the number source is used to maintain the independence between bit streams. This requires the converter modules to have certain properties *relative* to one another. Depending on the method used, these properties manifest into either interconnections between modules or as differences in certain parameters.

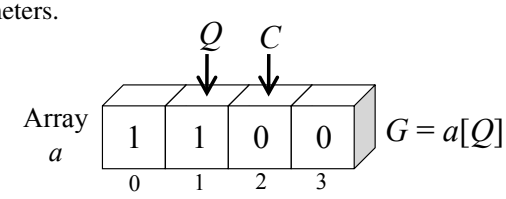


Fig. 4: Analogy of the circuit operation of Figure 3

The methods maintain independence by using relatively prime bit lengths, rotation, or clock division. For each method, the hardware complexity of the circuit implementation is given. The computation time of each method is the same.

A. Relatively Prime Bit Lengths

The relatively prime method maintains independence by using proportions that have relatively prime lengths (i.e., the ranges [0, R_i) between converter modules are relatively prime). Figure 5 demonstrates the method with two bit streams A and B, one with operand length four and the other with operand length three. The bit streams are shown in array notation to show the position of each bit in time.

$$\begin{array}{c} a_0\, a_1\, a_2\, a_3\, a_0\, a_1\, a_2\, a_3\, a_0\, a_1\, a_2\, a_3 \\ b_0\, b_1\, b_2\, b_0\, b_1\, b_2\, b_0\, b_1\, b_2\, b_0\, b_1\, b_2 \end{array}$$

Fig. 5: Two bit streams generated by the relatively prime method

Independence between bit streams is maintained because the remainder, or overlap between proportions, always results in a new rotation (or initial phase) of a proportion. Intuitively, this occurs because the bit lengths share no common factors. This results in every bit of each operand seeing every bit of the other operand. For example, a_0 sees b_0, b_1, and b_2; b_0 sees a_0, a_3, a_2, and a_1; and so on. Using two bit streams with relatively prime bit lengths j and k, the output of a logic gate repeats with period jk. This means with multi-level circuits the output of the logic gates will also be relatively prime. Figure 6 demonstrates this with a two level circuit.

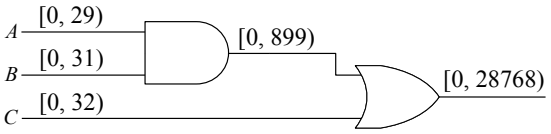


Fig. 6: Arbitrary multi-level circuit with bit streams generated by the relatively prime method

Therefore, by using relatively prime bit lengths up front, we can guarantee that Equation 6 is maintained for subsequent levels. This allows for the same arithmetic logic as a stochastic representation.

A circuit implementation of the relatively prime method is shown in Figure 7. Each converter module uses a counter as a number source for iterating through each bit of the proportion. The state of the counter Q_i is compared with the proportion constant C_i. The relatively prime counter ranges R_i between modules maintain independence; there are no interconnections between modules. In terms of general circuit components, the circuit uses i counters and i comparators, where i is the number of generated independent bit streams. Assuming the max range is a binary resolution 2^n and all modules are close to this value (i.e., 256, 255, 253, 251...), the circuit contains approximately i n-bit counters and i n-bit comparators.

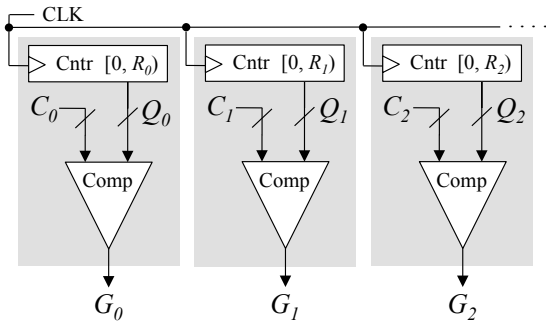


Fig. 7: Circuit implementation of the relatively prime method

A limitation of this method is that it requires the inputs to have relatively prime lengths. In this paper we focus on a digital representation of data, but the relatively prime method may also work well with an analog interpretation of the bit streams, where the value is encoded as the fraction of time the signal is high and the independence property is maintained by using relatively prime frequencies. An analog implementation can also benefit from a simplified clock distribution circuit [8].

B. Rotation

In contrast to the relatively prime method, the rotation method allows proportions of arbitrary length to be used. Instead of relying on relatively prime bit lengths, the proportions are explicitly rotated. This requires the sequence generated by the number source to change after it iterates through its entire range. For example, a simple way to generate a bit stream where the proportion rotates in time is to inhibit or stall a counter every 2^n clock cycles (where n is the length of the counter). Figure 8 demonstrates this method with two bit streams, both with proportions of length four.

$$\begin{array}{cccccccccccccccc} a_0 & a_1 & a_2 & a_3 & a_0 & a_1 & a_2 & a_3 & a_0 & a_1 & a_2 & a_3 & a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 & b_3 & b_0 & b_1 & b_2 & b_2 & b_3 & b_0 & b_1 & b_1 & b_2 & b_3 & b_0 \end{array}$$

Fig. 8: Two bit streams generated by the rotation method

By rotating bit stream B's proportion, it is straightforward to see that each bit of one bit stream sees the other bit stream's proportion. Assuming all proportions have the same length, we can extend the two bit stream example to work with multiple bit streams by inhibiting counters at powers of the operand length. This allows the operands to rotate relative to longer bit streams. For example, consider the circuit in Figure 9. Bit stream A does not rotate, bit stream B rotates every 2^n clock cycles, and bit stream C rotates every 2^{2n} clock cycles. The resultant bit stream AB of the AND gate *repeats* every 2^{2n} clock cycles and bit stream C *rotates* every 2^{2n} bits. Therefore bit stream C rotates relative to the bit stream AB, maintaining the rotation property for multi-level circuits.

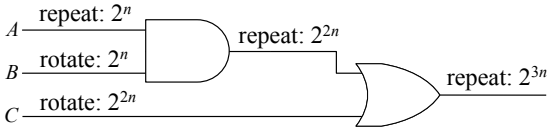


Fig. 9: Arbitrary multi-level circuit with bit streams generated by the rotation method

A circuit implementation follows from the previous example. We can generate any number of independent bit streams as long as the counter of every ith converter module is inhibited every 2^{ni} clock cycles. This can be managed by adding additional counters between each module. These counters control the phase of each converter module and maintain the property that each converter module rotates relative to the other modules. Using n-bit binary counters and comparators, the circuit requires i n-bit comparators and $2i-1$ n-bit counters.

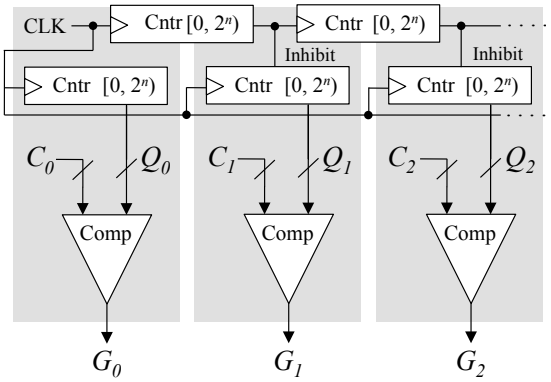


Fig. 10: Circuit implementation of the rotation method

The advantage of using rotation as a method for generating independent bit streams is that we can use operands with the same resolution, but this requires more basic components than the relatively prime method.

C. Clock Division

The clock division method works by clock dividing operands. Similar to the rotation method, it also allows proportions to have arbitrary lengths. This method was first seen in Example 5. Figure 11 demonstrates this method with two bit streams, both with proportions of length four. Bit stream B is clock divided by the length of bit stream A's proportion.

$$\begin{array}{cccccccccccccccc} a_0 & a_1 & a_2 & a_3 & a_0 & a_1 & a_2 & a_3 & a_0 & a_1 & a_2 & a_3 & a_0 & a_1 & a_2 & a_3 \\ b_0 & b_0 & b_0 & b_0 & b_1 & b_1 & b_1 & b_1 & b_2 & b_2 & b_2 & b_2 & b_3 & b_3 & b_3 & b_3 \end{array}$$

Fig. 11: Two bit streams generated by the clock division method

Assuming all operands have the same length, we can generate an arbitrary number of independent bit streams as long as the counter of every ith converter module increments every 2^{ni} clock cycles. This can be implemented in circuit form by simply chaining the converter module counters together, as shown in Figure 12. Using n-bit binary counters and comparators, the circuit requires i n-bit comparators and i n-bit counters. This means the clock division method allows operands of the same length to be used with approximately the same hardware complexity as the relatively prime method.

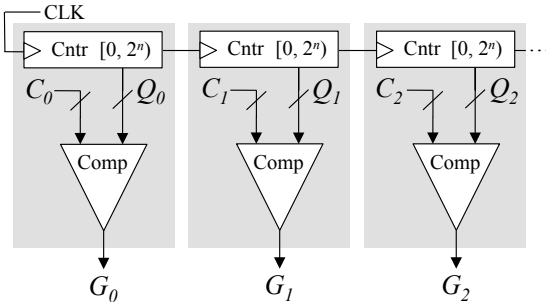


Fig. 12: Circuit implementation of the clock division method

VI. Experiments

In this section we compare the hardware complexity and latency of the deterministic methods with conventional

stochastic methods. We also compare implementations of Bernstein polynomials, a general method for synthesizing arbitrary functions, for both precise and approximate computations.

A. Generator Comparison

Perfectly precise computations require the output resolution to be at least equal to the product of the independent input resolutions. This is demonstrated in Equation 6, where to precisely compute the output of a logic gate given two proportions, each bit of one proportion must be operated on with every bit of the other proportion. For example, with proportions of size n and m, the precise output contains nm bits.

Assuming each independent input i has the same resolution $1/2^{n_{in}}$, the output resolution is given by $1/2^{n_{out}} = 1/2^{n_{in}i}$. As discussed in Section IV, a stochastic representation requires bit streams that are 2^{2n} bits long to represent a value with $1/2^n$ precision. Also, to ensure the generated bit streams are sufficiently random and independent, each LFSR must have at least as many states as the required output bit stream. Therefore, to compute with perfect precision each LFSR must have at least length $2n_{in}i$. In this way, the precision of the computation is determined by LFSR length.

With the deterministic methods, the resolution n of each input i is determined by the length of its converter module counter. The output resolution is simply the product of the counter ranges. For example, with the clock division method, each converter module counter is connected in series. With i inputs each with resolution n, the series connection forms a large counter with 2^{ni} states. This shows that output resolution is not determined by the length of each individual number source, but by their concatenation. This allows for a large reduction in circuit area compared to stochastic methods.

To compare the area of the circuits in terms of gates, we assume three gates for every cell of a comparator and six gates for each flip-flop of a counter or LFSR (this is similar to the hardware complexity used in [9] in terms of fanin-two NAND gates). For i inputs with n-bit binary resolution, the gate count for each basic component is given by:

Component	Gate Count
Comparator	$3n$
Counter	$6n$
LFSR	$12ni$

Using the basic component totals for each deterministic method from Section V and the fact that each "Randomizer unit" needs one comparator and one LFSR per input, the total gate count and bit stream length for precise computations in terms of independent inputs i with resolution n is given by Table I.

TABLE I

Representation	Method	Gate Count	Latency
Stochastic	Randomizer	$12ni^2+3ni$	2^{2ni}
Deterministic	Rel. Prime	$9ni$	2^{ni}
	Rotation	$15ni-6n$	
	Clock Div.	$9ni$	

The equations of Table I show that the deterministic methods use less area and compute to the same precision in exponentially less time. In addition, because the computations do not suffer from correlation, they are completely accurate. Table II compares the gate count and latency of the conventional stochastic method with the clock division method using numerical values of i and n.

TABLE II

i	n	Randomizer		Clock Div.		$\frac{determ.product}{stoch.product}$
		Gates	Latency	Gates	Latency	
2	4-bit	216	2^{16}	72	2^{8}	1.30×10^{-3}
	8-bit	432	2^{32}	144	2^{16}	5.09×10^{-6}
3	4-bit	468	2^{24}	108	2^{12}	5.63×10^{-5}
	8-bit	936	2^{48}	216	2^{24}	1.38×10^{-8}
4	4-bit	816	2^{32}	144	2^{16}	2.69×10^{-6}
	8-bit	1632	2^{64}	288	2^{32}	4.11×10^{-11}
5	4-bit	1260	2^{40}	180	2^{20}	1.36×10^{-7}
	8-bit	2520	2^{80}	360	2^{40}	1.30×10^{-13}

For the given number of inputs i and resolution n, the clock divide method has a 66-85% reduction in area and an exponential decrease in computation time. The deterministic area-delay product is orders of magnitude smaller than the stochastic area-delay product.

B. Bernstein Polynomial Implementation

In this subsection we compare the implementation of Bernstein polynomials using stochastic and deterministic methods. A Bernstein polynomial can be used to synthesize power-form polynomial functions or approximate non-polynomial functions that map values from the unit interval to values in the unit interval, examples include $g(x) = 6x^3 - 8x^2 + 3x$ and $f(x) = x^{0.45}$. This arithmetic circuit can be implemented using an adder block and multiplexer block, as shown in Figure 13. Bit streams z_i form the coefficients of the Bernstein polynomial and bit streams x_i form the input x. Additional details can be found in [10].

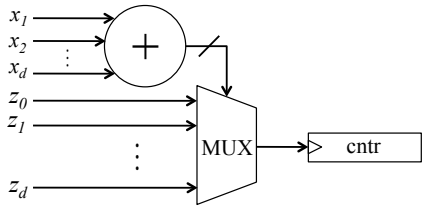


Fig. 13: Circuit implementation of Bernstein polynomial with output counter

For both representations, the adder block is formed by a d-bit adder made up of $2d$ gates and a $(d+1)$-channel multiplexer made with $3d$ gates. Each method needs the same number of bit streams and therefore requires $2d+1$ comparators. Each z bit stream must be independent from the input x bit streams, but they do not have to be independent from each other. This is because only one z bit stream is selected at a time. Therefore, Equation 6 does not have to be maintained between the coefficient bit streams and the same number source can be used for all z bit streams. Both methods require $d+1$ number sources.

Using the above basic component counts and including the output counter, Table III characterizes the hardware complex-

ity and latency for implementing a Bernstein polynomial of degree d with n-bit binary input resolution.

TABLE III

Bernstein Polynomial (degree d, n-bit input resolution)		
Generator	Gate Count	Latency
Randomizer	$12nd^2+42nd+27n+5d$	$2^{2n(d+1)}$
Rel. Prime	$18nd+15n+5d$	$2^{n(d+1)}$
Rotation	$24nd+15n+5d$	
Clock Div.	$18nd+15n+5d$	

Table IV compares the clock division method to the stochastic method with numerical values of degree d and binary input resolution n.

TABLE IV

d	n	Randomizer		Clock Div.		$\frac{determ.product}{stoch.product}$
		Gates	Latency	Gates	Latency	
2	4-bit	646	2^{24}	214	2^{12}	8.09×10^{-5}
3	4-bit	1059	2^{32}	291	2^{16}	4.19×10^{-6}
4	4-bit	1568	2^{40}	368	2^{20}	2.24×10^{-7}
5	4-bit	2173	2^{48}	445	2^{24}	1.22×10^{-8}

With 4-bit binary inputs, the clock divide method provides 66-79% reduction in circuit area over the given range of polynomial degrees. Again, the latency of the deterministic representation is exponentially less than the stochastic representation. The area-delay product of the clock divide method for computing Bernstein polynomials is orders of magnitude less than the stochastic method.

If a lower output resolution is desired to reduce the bit stream lengths, the resolution of the inputs can be relaxed:

$$n_{in} = \lceil \frac{n_{out}}{i} \rceil = \lceil \frac{n_{out}}{d+1} \rceil \tag{10}$$

In general, stochastic computation uses imprecise output resolutions to avoid long delays and reduce the size of the LFSRs. By keeping the output resolution fixed, the LFSR lengths are linearly proportional to d. Using an n_{out}-bit output resolution and relaxing the inputs, the hardware complexity for the stochastic method with a constant output resolution is given by $12n_{out}d+24n_{out}+6\lceil\frac{n_{out}}{d+1}\rceil d+3\lceil\frac{n_{out}}{d+1}\rceil+5d$. Table V compares the methods with constant output resolution (where the inputs of the deterministic methods are relaxed according to Equation 10).

TABLE V

d	n_{out}	Randomizer		Clock Div.		$\frac{determ.product}{stoch.product}$
		Gates	Latency	Gates	Latency	
3	8-bit	537	2^{16}	153	2^{8}	1.11×10^{-3}
4	10-bit	794	2^{20}	194	2^{10}	2.39×10^{-4}
5	12-bit	1099	2^{24}	235	2^{12}	5.22×10^{-5}
6	14-bit	1452	2^{28}	276	2^{14}	1.16×10^{-5}
7	16-bit	1853	2^{32}	317	2^{16}	2.61×10^{-6}

The results of Table V show the clock divide method provides a 71 to 82% reduction in area for practical implementations of a Bernstein polynomial with constant output resolution.

VII. Conclusion

There has been widespread interest in the idea of stochastic logic in recent years. We point to [5] for a survey of work in the area. While numerous papers have advocated the advantages, the narrative has never been compelling. Yes, the paradigm permits complex arithmetic operations to be performed with remarkably simple logic, but the logic to generate pseudorandom bit streams is costly, essentially offsetting the benefit. The long latency, poor precision, and random fluctuations are near disastrous for most applications.

While it is easy conceptually to understand how stochastic computation works, randomness is costly. This paper argues that randomness is not necessary. Instead of relying upon statistical sampling to operate on bit streams, we can explicitly "convolve" them: we slide one operand past the other, performing bitwise operations. We argue that the logic to perform this convolution is less costly than that to generate pseudorandom bit streams. More importantly, we can use much shorter bit streams to achieve the same accuracy as with statistical sampling through randomness. Indeed, the results of our computation are predictable and completely accurate for all input values.

Of course, compared to a binary radix representation, our deterministic representation is still not very compact. With M bits, a binary radix representation can represent 2^M distinct numbers. To represent real numbers with a resolution of 2^{-M}, i.e., numbers of the form $\frac{a}{2^M}$ for integers a between 0 and 2^M, we require a stream of 2^M bits. In contrast, a stochastic representation requires 2^{2M} bits to achieve the same precision!

We conclude that there is no clear reason to compute on stochastic bit streams. Even when randomness is free, say harvested from thermal noise or some other physical source, stochastic computing entails very high latency. In contrast, computation on deterministic uniform bit streams is less costly, has much lower latency, and is completely accurate.

References

[1] B. R. Gaines, *Stochastic Computing Systems*, ser. Advances in Information Systems Science. Springer US, 1969.

[2] B. D. Brown and H. C. Card, "Stochastic neural computation i: Computational elements," *IEEE Transactions On Computers, vol. 50, no. 9, pages 891-905*, 2001.

[3] W. Qian and M. D. Riedel, "Synthesizing logical computation on stochastic bit streams," *Proceedings of the Design Automation Conference*, 2009.

[4] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *IEEE Transactions on Computers*, vol. 60, pp. 93–105, 2011.

[5] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM Transaction on Embedded Computing*, vol. 12, 2013.

[6] S. S. Tehrani, A. Naderi, G.-A. Kamendje, S. Hemati, S. Mannor, and W. J. Gross, "Majority-based tracking forecast memories for stochastic ldpc decoding," *IEEE Transactions on Signal Processing*, vol. 58, pp. 4883–4896, 2010.

[7] W. Qian, "Digital yet deliberately random: Synthesizing logical computation on stochastic bit streams," Ph.D. dissertation, University of Minnesota, 2011.

[8] M. H. Najafi, D. J. Lilja, M. D. Riedel, and K. Bazargan, "Polysynchronous stochastic circuits," *IEEE/ACM Asia and South Pacific Design Automation Conference*, 2016.

[9] P. Li, W. Qian, and D. J. Lilja, "A stochastic reconfigurable architecture for fault-tolerant computation with sequential logic," *IEEE 30th International Conference on Computer Design (ICCD)*, 2012.

[10] W. Qian and M. D. Riedel, "The synthesis of robust polynomial arithmetic with stochastic logic," *ACM Design Automation Conference*, 2008.

Decomposition of Index Generation Functions Using a Monte Carlo Method

Tsutomu Sasao
Dept. of Computer Science
Meiji University
Kawasaki 214-8571, Japan

Jon T. Butler
Dept. Electrical and Computer Engr.
Naval Postgraduate School
Monterey, CA, 93943-5121, U.S.A

***Abstract*—This paper considers functional decompositions of index generation functions. A Monte Carlo method to predict column multiplicities of the decomposition charts is presented. With this, we can efficiently find a circuit structure to implement the function. Experimental results confirm the theory.**

I. Introduction

One of the important tasks in logic synthesis is to find a circuit structure that is suitable for implementation. **Functional decomposition** [1], [3] is a technique to decompose a circuit into two parts with a lower cost than the original circuit. Various techniques to find functional decompositions have been presented. They are routinely used in logic synthesis. Many circuits that are used in computers have some structure, and they are not random [2]. On the other hand, almost all randomly generated switching functions have no effective decompositions.

Recently, we are working on index generation functions [7], [8]: $f : \{0,1\}^n \rightarrow \{0,1,2,\ldots,k\}$, where $k \ll 2^n$. They are used for access control lists, routers, and virus scanning for the internet, etc.. Index generation functions found in practical applications have properties similar to those of randomly generated index generation functions.

In this paper, we show that index generation functions often have effective decompositions. To show this, we use a Monte Carlo method to predict the column multiplicity of the decomposition charts for random index generation functions.

An index generation function can be efficiently implemented by an **IGU** (Index Generation Unit), which is programmable [7]. Currently, it is implemented by a combination of field programmable gate array (FPGA) and memories [5]. However, a single-chip custom LSI can also be used to implement an index generation function.

Suppose that LSIs for IGUs with n inputs and weight k are available. For a function with larger k, we can partition the set of vectors into several sets, and implement each by an independent IGU. The outputs of the IGUs can be combined by an OR gate to produce the final output [9]. This is a **parallel decomposition**. For a function with larger n, we can partition the set of input variables into two sets X_1 and X_2, and implement the function by a **serial decomposition** (Fig. 2.2).

In our applications, when we prepare the programmable IGU chip, we only know the values of n and k, but do not know the detail of the functions. This situation is well-known by FPGA companies. Functions to be implemented are different for different users. FPGA companies have to predict the number of LUTs in advance.

The rest of the paper is organized as follows: Section II defines the notation. Section III introduces the properties of index generation functions. Section IV shows a Monte Carlo method to derive column multiplicity of decomposition charts for random index generation functions. Section V shows a procedure for computing the column multiplicity of decompositions. Section VI shows the experimental results. Section VII shows a method to assess the programmable architecture for index generation functions. Section VIII concludes the paper.

II. Definitions

In this part, we introduce basic concepts.

Definition 2.1: A mapping $f : B^n \rightarrow \{0,1,\ldots,k\}$, where $B = \{0,1\}$ is a **binary-input multi-valued function**.

Definition 2.2: [1] Let $f(X)$ be a function, and (X_1, X_2) be a partition of the input variables, where $X_1 = (x_1, x_2, \ldots, x_s)$ and $X_2 = (x_{s+1}, x_{s+2}, \ldots, x_n)$. The **decomposition chart** for f is a two-dimensional matrix with 2^s columns and 2^{n-s} rows, where each column and row is labeled by a unique binary assignment of values to the variables. Each assignment maps under f to $\{0,1,\ldots,k\}$. The function represented by a column is a **column function** and is dependent on X_2. Variables in X_1 are **bound variables**, while variables in X_2 are **free variables**. In the decomposition chart, the **column multiplicity**, denoted by μ, is the number of different column functions.

Example 2.1: Fig. 2.1 shows a decomposition chart of a 4-variable switching function. $X_1 = (x_1, x_2)$ denotes the bound variables, and $X_2 = (x_3, x_4)$ denotes the free variables. Since all the column patterns are different and there are four of them, the column multiplicity is $\mu = 4$. ■

Theorem 2.1: [3] For a given function f, let X_1 be the bound variables, let X_2 be the free variables, and let μ be the column multiplicity of the decomposition chart. Then, the function f can be represented as $f(X_1, X_2) = g(h(X_1), X_2)$, and is realized with the network shown in Fig. 2.2. The number of signal lines connecting blocks H and G is $r = \lceil \log_2 \mu \rceil$, where H and G realize h and g, respectively.

		0	0	1	1	x_1
		0	1	0	1	x_2
0	0	0	0	0	1	
0	1	1	1	0	0	
1	0	0	1	0	0	
1	1	0	0	0	0	
x_3	x_4					

Fig. 2.1. Decomposition chart of an logic function.

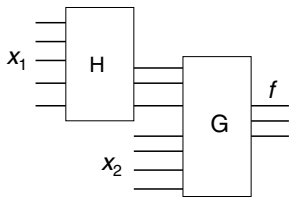


Fig. 2.2. Realization of a logic function by decomposition.

The logic functions for H and G can be realized by memories, and the complexities for G and H can be measured by the number of bits in the memories. The signal lines connecting H and G are called **rails**. When the number of rails r is smaller than the number of input variables in X_1, we can often reduce the total amount of memory to realize the logic in Fig. 2.2 [7].

III. Index Generation Functions and Their Properties

Definition 3.1: Consider a set of k different binary vectors of n bits. These vectors are **registered vectors**. For each registered vector, assign a unique integer from 1 to k. A **registered vector table** shows, for each registered vector, its **index**. An **index generation function** f produces the corresponding index if the input matches a registered vector, and produces 0 otherwise. k is the **weight** of the index generation function. An index generation function represents a mapping: $f : B^n \rightarrow \{0, 1, 2, \ldots, k\}$, where $B = \{0, 1\}$.

Example 3.1: Table 3.1 shows a registered vector table with $k = 4$ vectors. The corresponding index generation function is shown in Table 3.2. In this case, the output is is represented by 3 bits. So, it shows a mapping $B^4 \rightarrow \{0, 1, 2, 3, 4\}$. Note that the index values from Table 3.1 are shown in binary in bold. Also note that registered vectors missing in Table 3.1 are shown in Table 3.2 mapped to 000. ■

Typically, k is much smaller than 2^n, the total number of input combinations.

Example 3.2: Consider the decomposition chart in Fig. 3.1. It shows an index generation function $f(X)$ with weight 7. $X_1 = (x_1, x_2, x_3, x_4)$ denotes the bound variables, and $X_2 = (x_5)$ denotes the free variable. Note that the column multiplicity of this decomposition chart is 7. ■

TABLE 3.1
Registered vector table.

Vector				Index
x_1	x_2	x_3	x_4	
1	0	0	1	1
1	1	1	1	2
0	1	0	1	3
1	1	0	0	4

TABLE 3.2
Index generation function.

Input				Output		
x_1	x_2	x_3	x_4	y_1	y_2	y_3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	1	**0**	**1**	**1**
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	0	1	**0**	**0**	**1**
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	**1**	**0**	**0**
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	**0**	**1**	**0**

Lemma 3.1: Let $\mu(f(X_1, X_2))$ be the column multiplicity of a decomposition chart of an index generation function f, let k be the weight of f, and let s be the number of variables in X_1. Then,

$$\mu(f(X_1, X_2)) \leq \min\{2^s, k + 1\}.$$

(Proof) Since the number of non-zero outputs is k, the column multiplicity never exceeds $k + 1$. □

Lemma 3.2: Let f be an index generation function with weight k. Then, there exists a functional decomposition $f(X_1, X_2) = g(h(X_1), X_2)$, where g and h are index generation functions, the weight of g is k, and the weight of h is at most k.

(Proof) Consider a decomposition chart, in which X_1 denotes the bound variables, and X_2 denotes the free variables. Let $X_1 = (x_1, x_2, \ldots, x_s)$, where $s \geq \lceil \log_2(k + 1) \rceil$. Let h be a function where the variables are X_1, and the output values are defined as follows: Consider the decomposition chart, where assignments of values to X_1 label columns (i.e., bound variables). For the assignments to X_1 corresponding to columns with only zero elements, $h = 0$. For other inputs, the outputs are distinct integers from 1 to w_h, where w_h denotes the number of columns that have non-zero element(s). Since $w_h \leq k$, the weight of h is at most k, and the number of output values of h is at most $k + 1$. On the other hand, the function g is obtained from f by reducing some columns that have all zero output in the decomposition chart. Thus, the number of non-zero outputs in g is equal to the number of non-zero outputs in f. Thus, g is also an index generation function with weight k. □

Example 3.3: Consider the decomposition chart in Fig. 3.1. Let the function $f(X)$ be decomposed as $f(X_1, X_2) = g(h(X_1), X_2)$, where $X_1 = (x_1, x_2, x_3, x_4)$, and $X_2 = (x_5)$. Table 3.3 shows the function h. It is a 4-variable 3-output index generation logic function with weight 6. The decomposition chart for the function g is shown in Fig. 3.2. As shown in this example, the functions obtained by decomposing the index generation function f are also index generation functions, and the weights of f and g are both 7. ■

	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_1
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_2
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	x_3
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	x_4
0	0	1	0	0	3	0	4	5	0	0	0	0	0	0	0	0	
1	0	0	2	0	0	0	0	6	0	0	0	0	7	0	0	0	
x_5																	

Fig. 3.1. Decomposition chart for f.

TABLE 3.3
TRUTH TABLE FOR h.

x_1	x_2	x_3	x_4	y_1	y_2	y_3
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	0	0
0	1	0	0	0	1	1
0	1	0	1	0	0	0
0	1	1	0	1	0	0
0	1	1	1	1	0	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

	0	0	0	0	1	1	1	1	y_1
	0	0	1	1	0	0	1	1	y_2
	0	1	0	1	0	1	0	1	y_3
0	0	1	0	3	4	5	0	0	
1	0	0	2	0	0	6	7	0	
x_5									

Fig. 3.2. Decomposition chart for g.

IV. BALLS INTO BINS MODEL

In this part, we show a method to predict the column multiplicity of a functional decomposition using a balls into bins model.

In Definition 2.2, we specified that the first s variables $x_1, x_2, \ldots, x_s$ are in X_1, and the remaining $n-s$ variables are in X_2. However, in many cases, we can select any s variables for X_1. In this case, the problem of functional decomposition is to partition the variables into two sets, so that the number of rails r between two blocks is minimized. To do this, we want to reduce μ so that it is equal to or less than the smallest power of 2.

Given a function table, we have to search $\binom{n}{s}$ combinations, where n is the total number of variables, and s is the number of the bound variables. From Lemma 3.1, we have an upper bound on μ:

$$\mu(f(X_1, X_2)) \leq \min\{2^s, k+1\}.$$

When the functions are uniformly random, we can predict a lower bound on μ.

The column multiplicity μ of an index generation function with weight k can be predicted by the **balls into bins model** as follows: Assume that there are 2^s distinct bins, and k distinct balls are randomly thrown into these bins. If all the balls fall into the same bin, then $\mu = 1 + 1 = 2$. If all the balls fall into different bins, then $\mu = k + 1$. However, in most cases, $2 < \mu \leq k + 1$. The number of non-empty bins plus one is equal to the column multiplicity μ. Note that "+1" corresponds to the empty bin.

It is also represented by the **integer set model**: Assume that a set of k integers of s bits are randomly generated. Then, $\mu-1$ is equal to the number of different integers in the set.

The column multiplicity can be efficiently predicted by a Monte Carlo simulation using the integer model as follows:

Example 4.1: Consider the registered vector table shown in Table 4.1. It is a 20-variable random index generation function with weight $k = 20$. We need to find an effective decomposition that reduces the implementation cost. The number of decompositions is equal to the number of ways to partition the set of variables into two non-empty sets, that is $2^n - 2$. If we know the expected column multiplicity, then we can predict how likely exhaustive search can find a good decomposition. Suppose that the number of bound variables is $s = 8$. Then, the number of decompositions to check is $\binom{20}{8} = 125970$. By exhaustive search, we find the minimum column multiplicity to be $\mu = 15 + 1 = 16$, where the bound variables are $X_1 = (x_1, x_2, x_5, x_7, x_{10}, x_{13}, x_{14}, x_{17})$. In this case, the number of rails is reduced to from five to four.

Next, we predict the column multiplicity for random index generation functions of $n = 20$ variables and weight $k = 20$ by the Monte Carlo approach. The number of different n variable index generation functions with weight k is $P(2^n, k) = \frac{2^n!}{(2^n-k)!}$. For $n = 20$ and $k = 20$, this is about 2.6×10^{120}, which is too large to search exhaustively.

Instead of selecting $s = 8$ variables out of $n = 20$ in a particular function, we randomly generate many sets of $k = 20$ integers of $s = 8$ bits, and obtain the values of distinct numbers among the different integers for each set. Table 4.2 shows the result of this Monte Carlo simulation. In this case, we generated 10^6 samples.

Table 4.2 shows that, in most cases, the column multiplicity is 21 or 20. However, the column multiplicity can be reduced to $\mu = 14$ for one case, $\mu = 15$ for 20 cases, and $\mu = 16$ for 297 cases. This implies that the given function has a non-zero probability with a decomposition where $\mu \leq 16$. However, the probability of a decomposition with $\mu \leq 13$ is quite low, since a sample set of size 10^6 failed to produce a single decomposition with $\mu \leq 13$. ∎

We assume that the functions that appear in the search of

TABLE 4.1
REGISTERED VECTOR TABLE FOR AN 20-VARIABLE INDEX GENERATION FUNCTION.

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}	x_{20}	f
0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0	1
0	0	0	1	1	0	0	0	1	0	0	1	1	0	1	0	1	1	0	0	2
1	0	1	0	1	1	0	1	0	0	0	1	1	0	0	0	0	0	1	1	3
0	0	1	1	0	1	0	1	1	0	0	1	1	0	1	0	1	1	1	0	4
1	0	0	0	0	1	0	1	0	1	0	0	1	1	0	1	1	1	1	1	5
1	1	0	0	1	0	0	1	0	0	1	0	1	0	1	1	0	0	1	1	6
1	0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	0	1	7
1	0	0	0	1	0	1	1	0	0	1	1	0	0	1	0	1	1	1	1	8
0	1	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0	1	0	1	9
0	1	0	1	1	0	0	1	0	1	1	1	0	1	0	0	1	0	0	0	10
1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	1	0	0	11
0	1	0	0	1	0	0	1	1	1	0	1	1	0	1	0	1	1	0	1	12
0	0	0	0	0	0	1	0	1	1	1	1	0	0	1	0	1	0	1	0	13
0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	1	1	1	1	0	14
0	1	1	1	1	0	1	1	1	1	0	0	1	1	1	1	1	1	1	1	15
1	0	1	1	1	0	1	0	1	0	0	0	0	0	1	1	1	0	1	0	16
0	0	1	1	0	0	0	0	1	0	1	1	1	0	1	1	1	0	0	0	17
1	0	0	1	1	0	0	0	0	0	1	0	1	0	1	1	0	0	1	1	18
0	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	0	0	0	19
1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	1	1	1	20

TABLE 4.2
COLUMN MULTIPLICITIES OF DECOMPOSITION CHARTS FOR n=20 VARIABLE FUNCTIONS ($s = 8$).

Rails r	Column Multiplicity μ	Occurrence
4	$13+1$	1
4	$14+1$	20
4	$15+1$	297
5	$16+1$	3229
5	$17+1$	25749
5	$18+1$	129422
5	$19+1$	374505
5	$20+1$	466777

the decompositions, and the random set of integers used in the Monte Carlo simulation have similar statistical properties. That is, the probability distribution functions for the functional decompositions are similar to that of random integer sets.

V. PROCEDURE TO COMPUTE THE COLUMN MULTIPLICITY

To validate the use of a Monte Carlo technique in estimating the column multiplicity, we compare the results obtained by the Monte Carlo technique with an exhaustive enumeration in which we enumerate all binary arrays according to the column multiplicity. Every binary array is enumerated exactly once, and, as such, it can be considered a proxy for a 'perfect' Monte Carlo simulation. We use the balls into bins model.

Lemma 5.1: Assume that there are t non-distinct bins and k distinct balls. The number of different ways to put k distinct balls into t non-distinct bins is

$$S(k,t) = \begin{cases} 1, & \text{if } t = 1 \text{ or } t = k \\ S(k-1,t-1) + tS(k-1,t), & \text{otherwise.} \end{cases}$$

(Proof) The proof will be done by a mathematical induction. $S(k,t)$ can be calculated for three cases:

When $t = 1$**:** All the balls are in one bin. So, there is only one way.

When $t = k$**:** All the balls are in k bins. So, there is only one way.

Otherwise: Assume that $k-1$ balls are already in the bins, and the k-th ball will be put into one of the bins. In this case, there exist two cases:

1) The $k-1$ balls are in $t-1$ bins, but one bin is empty. In this case, the number of ways to put the first $k-1$ balls into $t-1$ bins is $S(k-1,t-1)$, by the hypothesis. The k-th ball is put into the empty bin.

2) The $k-1$ balls are in t bins. No bin is empty. In this case, the number of ways to put the first $k-1$ balls into t bins is $S(k-1,t)$, by the hypothesis. Also, there are t ways to put the k-th ball into one of t bins. So, the total number of ways is $tS(k-1,t)$. From these, we have the lemma. □

TABLE 5.1
VALUES FOR $S(k,t)$

	t								
k	1	2	3	4	5	6	7	8	9
1	1								
2	1	1							
3	1	3	1						
4	1	7	6	1					
5	1	15	25	10	1				
6	1	31	90	65	15	1			
7	1	63	301	350	140	21	1		
8	1	127	966	1701	1050	266	28	1	
9	1	255	3025	7770	6951	2646	462	36	1
10	1	511	9330	34105	42525	22827	5880	750	45

Note that $S(k,t)$ is the **Stirling number of the second kind** [4].

Example 5.1: Table 5.1 shows the values of $S(k,t)$ for $k \leq 10$ and $t \leq 9$.

Theorem 5.1: Consider the binary arrays that have s columns and k rows. The number of arrays with t distinct rows is

$$c_{k,t} = P(2^s, t)S(k,t),$$

where $S(k,t)$ is the Stirling number of the second kind, and $P(n,r) = \frac{n!}{(n-r)!}$.

(Proof) The number of ways to permute t distinct patterns out of 2^s distinct patterns is $P(2^s, t)$. □

To validate the Monte Carlo technique, we ran the simulation for $s = 3$ and $k = 8$ for a total of $(2^s)^k = 2^{24} = 16777216$ samples, which is the number of 3×8 binary arrays. We also used the exhaustive method described above. Table 5.2 compares the results. This shows that there is a close correlation between the Monte Carlo simulation and exhaustive enumeration. For small k, we can pre-compute the

TABLE 5.2
COMPARISON OF THE MONTE CARLO TECHNIQUE WITH EXHAUSTIVE ENUMERATION ($s = 3, k = 8$).

# of Distinct Rows	Monte Carlo	Exhaustive Enumeration
1	4	8
2	7162	7112
3	326013	324576
4	2857425	2857680
5	7053129	7056000
6	5362312	5362560
7	1130883	1128960
8	40288	40320
Total	16777216	16777216

table of the Stirling numbers, and store in the hard disk to compute $c_{k,t}$. However, for large k, the table becomes too large. Thus, we use the Monte Carlo approach for large k.

VI. EXPERIMENTAL RESULTS

A. Decompositions of 20-Variable Functions

We assume that the distribution of the column multiplicities during decompositions is similar to the random integer sets used in the Monte Carlo simulation.

To confirm this, we generated 10 sample index generation functions of $n = 20$ and $k = 20$. Then, for each function, we counted the column multiplicities for all the decompositions, where the number of bound variables is $s = 8$. Note that the number of ways to select 8 variables out of 20 variables is $\binom{20}{8} = 125970$.

Table 6.1 summarizes the distributions of column multiplicities. The first column shows the number of rails $r = \lceil \log_2 \mu \rceil$. The second column shows the column multiplicity μ. The third to 12th columns show the distributions for f_1 to f_{10}. For example, in f_1, the minimum column multiplicity is $\mu = 15+1$, and 20 decompositions produce this μ. For f_5, the minimum multiplicity is $\mu = 13 + 1$, and 10 decompositions produce this μ. For f_9, the minimum multiplicity is $\mu = 14 + 1$, and 18 decompositions produce this μ. For the other 8 functions, the minimum multiplicities are $\mu = 15 + 1$. The column headed with *SUM* denotes the sum of 10 sample functions. The rightmost column, headed with *Monte* denotes the result of Monte Carlo simulation. The number of sample sets generated for the simulation is $10 \times \binom{20}{8} = 1259700$, which is equal to the total number of decompositions for 10 sample functions.

Table 6.1 shows that, for all 10 sample functions, the column multiplicities can be reduced to $\mu = 15+1$ or less. This means that the number of rails r can be reduced from five to four. The Monte Carlo simulation shows that, for $374 + 25 + 2 = 401$ combinations out of 1259700, the column multiplicities are reduced to $\mu = 15 + 1$ or less. This shows that there is an incentive to find a decomposition with small column multiplicity, but it may be hard to find.

B. Decomposition of a 64-Variable Function

In the previous experiment, the number of variables was only 20, and the number of the decompositions was $\binom{20}{8} = 125970$.

TABLE 6.2
DECOMPOSITION OF A 64-VARIABLE INDEX GENERATION FUNCTION WITH $k = 20$ ($s = 8$).

r	μ	$Occurrence$	$Ratio$	$Monte$
4	$12 + 1$	20	$4.51858E - 09$	
4	$13 + 1$	971	$2.19377E - 07$	1
4	$14 + 1$	33301	$7.52367E - 06$	20
4	$15 + 1$	759722	0.000171643	297
5	$16 + 1$	11265390	0.002545181	3229
5	$17 + 1$	103232971	0.023323343	25749
5	$18 + 1$	562995509	0.127197125	129422
5	$19 + 1$	1675277777	0.378494169	374505
5	$20 + 1$	2072599707	0.468260796	466777

In this part, we used a sample function with $n = 64$ and $k = 20$. Note that, the number of decompositions is now $\binom{64}{8} = 4426165368$. Table 6.2 shows the results. The first column shows the number of rails r; the second column shows the column multiplicities μ; the third column shows the number of decompositions that produced the corresponding μ. The fourth column shows: $Ratio = \frac{Occurrence}{4426165368}$. The rightmost column headed by $Monte$ shows the number of occurrences in the Monte Carlo simulation. In the Monte Carlo simulation, the number of possible ways to generate sets of 20 integers of 8 bits is $(2^8)^{20} = 2^{160} \simeq 1.46 \times 10^{48}$. In this experiment, we generated 10^6 sample sets.

In this example, the Monte Carlo method provides a good approximation when μ is large, but not so good when μ is small. Note that the CPU time for the Monte Carlo simulation was less than 200 milliseconds, while that for the exhaustive decomposition search was 22 minutes.

With the Monte Carlo simulation, we can see that the probability of finding a decomposition with $\mu = 16$ (i.e., $r = 4$ rails) is quite high, while that with $\mu \leq 8$ (i.e., $r = 3$ rails) is almost zero. Thus, once we find a decomposition with $\mu \leq 16$, we can stop the search; it is not likely that we will find a decomposition with a smaller r.

VII. A METHOD TO ASSESS PROGRAMMABLE ARCHITECTURE

Problem 1: Design a programmable architecture for an index generation function with $n = 500$ and $k = 100$ using a pair IGUs.

(Solution) In a decomposition, the number of the rails is at most $q = \lceil \log_2(k + 1) \rceil = 7$. A Monte Carlo simulation with $s = 11$ and $k = 100$ shows that the minimum column multiplicity of the decompositions among 10^6 samples is $\mu = 87$. Since, the number of rails is $r = \lceil \log_2 \mu \rceil = 7$, the function can be realized by a cascade as shown in Fig. 7.1. IGU1 has 255 inputs and 7 outputs, while IGU2 has $7 + 245 = 252$ inputs and 7 outputs.

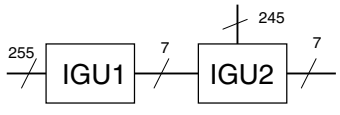


Fig. 7.1. Architecture using two IGUs.

TABLE 6.1
DECOMPOSITIONS OF 20-VARIABLE INDEX GENERATION FUNCTIONS WITH $k = 20$, $(s = 8)$.

r	μ	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	SUM	$Monte$
4	$13+1$	0	0	0	0	10	0	0	0	0	0	10	2
4	$14+1$	0	0	0	0	181	0	0	0	18	0	199	25
4	$15+1$	20	18	27	7	1338	5	24	3	138	9	1589	374
5	$16+1$	457	571	353	164	5478	144	366	122	1237	172	9064	4101
5	$17+1$	4673	4775	3346	1669	14934	1761	2645	1683	6413	2242	44141	32505
5	$18+1$	22084	20615	18172	10841	28903	11668	14770	13178	23163	13887	177281	163015
5	$19+1$	48736	49356	49694	41775	40076	42141	46962	47846	49122	46922	462630	471686
5	$20+1$	50000	50635	54378	71514	35050	70251	61203	63138	45879	62738	564786	587992

Problem 2: Design a programmable architecture for an index generation function with $n = 20$ and $k = 68$ using a pair of LUTs.

(Solution) In a decomposition, the number of rails is at most $q = \lceil \log_2(k+1) \rceil = 7$, by Lemma 3.1. Let the number of bound variables be $s = 10$. A Monte Carlo simulation with $s = 10$ and $k = 68$ shows that the minimum column multiplicity of the decompositions among 10^6 samples is $\mu = 57$. Thus, the number of rails is reduced to $\lceil \log_2 \mu \rceil = 6$. The function can be realized by a cascade as shown in Fig. 7.2. LUT1 has 10 inputs and 6 outputs, while LUT2 has $6 + 10 = 16$ inputs and 7 outputs. Since LUT2 has 16 inputs, and is much larger than LUT1, the circuit is not so efficient.

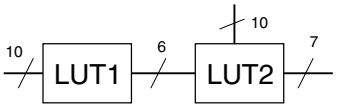


Fig. 7.2. Architecture using two LUTs.

So, we increase the number of inputs to LUT1 to $s = 12$. A Monte Carlo simulation with $s = 12$ and $k = 68$ shows that the minimum column multiplicity of the decompositions among 10^6 samples is $\mu = 62$. Thus, the number of rails is still $\lceil \log_2 \mu \rceil = 6$. The function can be realized by a cascade as shown in Fig. 7.3, which is more efficient than Fig. 7.2.

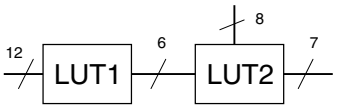


Fig. 7.3. Architecture using two LUTs.

Next, we further increase the number of inputs to LUT1 to $s = 14$. A Monte Carlo simulation with $s = 14$ and $k = 68$ shows that the minimum column multiplicity of the decompositions among 10^6 samples is $\mu = 65$. Thus, the number of rails is increased to $\lceil \log_2 \mu \rceil = 7$, as shown in Fig. 7.4, which is less efficient than Fig. 7.3.

VIII. CONCLUSION AND COMMENTS

In this paper, we present a Monte Carlo method to predict the column multiplicity of the decomposition charts for random index generation functions. We also show a procedure to compute the multiplicities of decomposition charts. Comparison with the exact enumerations shows that the Monte Carlo method produces good approximations to exact enumeration.

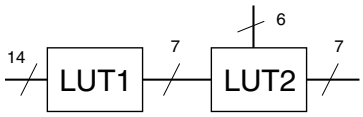


Fig. 7.4. Architecture using two LUTs.

When we design a programmable architecture for index generation functions, in many cases, we know only the numbers of inputs and registered vectors, but not the detail of the functions. In such cases, the method to assess the programmable architecture presented in this paper is quite useful.

ACKNOWLEDGMENTS

This research is partly supported by the Japan Society for the Promotion of Science (JSPS) Grant in Aid for Scientific Research. Discussion with Mr. Kyu Matsuura was useful to improve Section V. Also, the reviewers' comments improved the presentation of the paper.

REFERENCES

[1] R. L. Ashenhurst, "The decomposition of switching functions," *Inter. Symp. on the Theory of Switching*, pp. 74-116, April 1957.
[2] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.
[3] H. A. Curtis, *A New Approach to the Design of Switching Circuits*, D. Van Nostrand Co., Princeton, NJ, 1962.
[4] C. L. Liu, *Introduction to Combinatorial Mathematics*, McGraw-Hill, 1968.
[5] H. Nakahara, T. Sasao, M. Matsuura, H. Iwamoto, and Y. Terao, "A memory-based IPv6 lookup architecture using parallel index generation units," *IEICE Trans. Inf. and Syst.* Vol. E98-D, No. 2, pp. 262-271, Feb., 2015.
[6] T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.
[7] T. Sasao, *Memory-Based Logic Synthesis*, Springer, 2011.
[8] T. Sasao, "Index generation functions: Tutorial,"*Journal of Multiple-Valued Logic and Soft Computing*, Vol. 23, No. 3-4, pp. 235-263, 2014.
[9] T. Sasao, "A realization of index generation functions using multiple IGUs," *Inter. Symp. on Multiple-Valued Logic*, (ISMVL-2016), Sapporo, Japan, May 17-19, 2016.

Author Index