View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

An Efficient Manipulation Package for Biconditional Binary Decision Diagrams

Luca Amarú, Pierre-Emmanuel Gaillardon, Giovanni De Micheli
Integrated Systems Laboratory (LSI), EPFL, Switzerland

***Abstract*— *Biconditional Binary Decision Diagrams* (BBDDs) are a novel class of binary decision diagrams where the branching condition, and its associated logic expansion, is *biconditional* on two variables. Reduced and ordered BBDDs are remarkably compact and unique for a given Boolean function. In order to exploit BBDDs in *Electronic Design Automation* (EDA) applications, efficient manipulation algorithms must be developed and integrated in a software package. In this paper, we present the theory for efficient BBDD manipulation and its practical software implementation. The key features of the proposed approach are (i) *strong canonical form* pre-conditioning of stored BBDD nodes, (ii) recursive formulation of Boolean operations in terms of *biconditional* expansions, (iii) performance-oriented memory management and (iv) dedicated BBDD re-ordering techniques. Experimental results show that the developed BBDD package achieves an average node count reduction of 19.48% and a speed-up factor of 1.63x with respect to a state-of-art decision diagram manipulation package. Employed in the synthesis of datapath circuits, the BBDD manipulation package is capable to advantageously restructure arithmetic operations producing 11.02% smaller and 32.29% faster circuits as compared to a commercial synthesis flow.**

I. Introduction

Efficient Boolean function representation and manipulation is essential in *Electronic Design Automation* (EDA). *Binary Decision Diagrams* (BDDs) [1]–[3] are a well established data structure for this purpose. BDDs are employed in many EDA applications, such as digital circuits synthesis [4], verification [5], testing [6], simulation [7], and others. Original BDDs are driven by the Shannon's expansion to decompose a Boolean function until the constant logic values are encountered. Reduced and ordered BDDs [3] are unique for a given variable order, i.e., canonical, thus enabling efficient logic manipulation [8], [9]. Unfortunately, there exist Boolean functions for which BDDs are not compact or even too large to be handled [10]. In order to improve the compactness of BDDs, and therefore the efficiency of their applications, canonical extensions of BDDs are proposed in literature, e.g., [11]–[14]. We refer the reader to [15] for a more complete list of BDD extensions.

In this work, we focus on *Biconditional Binary Decision Diagrams* (BBDDs), a promising class of canonical binary decision diagrams recently introduced in [14]. While original BDDs are based on the single-variable Shannon's expansion, BBDDs employ a two-variable *biconditional* expansion [14], making the branching condition at each decision node dependent on two variables per time. Such feature improves the expressive power of the binary decision diagram. Moreover, BBDDs represent also the natural and native design abstraction for emerging technologies where the circuit primitive is a comparator, rather than a switch [16]–[18]. The work in [14] first introduces the concept of BBDDs but does not provide a detailed discussion about their automated manipulation.

In this paper, we present the theory for efficient BBDD manipulation and its practical software implementation. We do not address here theoretical issues related to BBDDs, such as asymptotic size bounds or other properties, but we focus on the design of a new BBDD package to support EDA applications. The main attributes enabling efficient BBDD manipulation are (i) *strong canonical form* pre-conditioning of stored BBDD nodes, (ii) recursive formulation of Boolean operations in terms of *biconditional* expansions, (iii) performance-oriented memory management and (iv) dedicated BBDD re-ordering techniques. The developed BBDD package is open-source and available online [19]. Experimental results over the MCNC benchmark suite show that the BBDD package is 1.63x faster than a state-of-art decision diagram package [20] and produces 19.48% smaller logic representations. In order to showcase the interest of a BBDD package for EDA, we also propose a new synthesis approach for datapath circuits. Standard synthesis techniques face challenges to satisfactory handle datapaths, that are intensive in arithmetic operations, such as comparators and voters. Instead, BBDDs are remarkably compact for arithmetic operations as the comparator function is inherently embedded in a BBDD node functionality [14]. Motivated by this consideration, we employ the BBDD package as front-end to a commercial synthesis tool to structure arithmetic operations in datapaths. Synthesis results show that datapaths pre-structured by the BBDD package are 11.02% smaller and 32.29% faster than their counterparts directly designed by a commercial synthesis flow.

The remainder of this paper is organized as follows. Section II provides a background on traditional binary decision diagrams and existing software packages. In Section III, BBDDs are introduced with their basic definitions for reference. Section IV presents the BBDD manipulation theory and discusses the software package results. Section V presents the application of the BBDD package as front-end for datapath design and discusses the synthesis results. The paper is concluded in Section VI.

II. Background and Motivation

This section provides a background on traditional binary decision diagrams and related logic manipulation packages.

A. Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are data structures representing Boolean functions. The general concept of BDDs is first introduced by Lee [1] and Akers [2], but later popularized by Bryant in [3], where it is shown that, under ordering and reduction rules, BDDs are a canonical representation form.

We refer the reader to [15] for a review about terminology and fundamentals of BDDs. The common adoption of BDDs in EDA [4]–[7] is driven by efficient BDD manipulation algorithms [8], [9] and related software implementations [20]. A BDD manipulation package can be used as a general-purpose software for manipulating Boolean functions, task at the core of EDA. We discuss hereafter the main features of a state-of-art BDD package.

B. State-of-art BDD Package Implementation

Contemporary BDD packages, such as CUDD [20], are designed to be memory-efficient and to have fast runtime. Memory efficiency is achieved by the use of hash-tables and caches. A global hash-table, commonly called the *unique table*, stores the nodes of the BDDs in a *strong canonical form*, which is a form of data pre-conditioning to reduce the complexity of equivalence test [8]. A global cache, commonly called the *computed table*, is used to temporarily store performed Boolean operations in case of later use. Fast BDD manipulation is achieved by a recursive formulation of Boolean operations between existing decision diagrams [8]. Additionally, variable re-ordering algorithms [9] are commonly used to minimize the number of stored nodes and maximize the package performance.

Many BDD extensions improve the expressive power of binary decision diagrams. However, only a limited number of them are supported by an efficient manipulation theory and even fewer of them by a practical software implementation.

In this work, we present the manipulation theory and efficient software implementation for a promising class of decision diagrams [14], based on a novel equality/inequality switching paradigm.

III. Biconditional BDD Overview

In this section, we review *Biconditional Binary Decision Diagrams* (BBDDs) [14], a canonical class of binary decision diagrams where the branching condition, and its associated logic expansion, acts as a two-variable comparator function.

A. BBDD Fundamentals

A BBDD is a *Direct Acyclic Graph* (DAG) representing a Boolean function $f(v, w, .., z)$. A BBDD is uniquely identified by its *root*, the set of *internal nodes*, the set of *edges* and the *1/0-sink nodes*. Each internal node (Fig. 1) in a BBDD

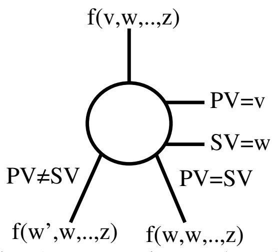

Fig. 1: BBDD internal node.

is labeled by two Boolean variables: v, the *Primary Variable* (PV), and w, the *Secondary Variable* (SV), and has two out-edges labeled $PV \neq SV$ and $PV = SV$. Each internal node represents the biconditional expansion with respect to v, w:

$$f(v, w, .., z) = (v \oplus w) \cdot f(w', w, .., z) + (\overline{v \oplus w}) \cdot f(w, w, .., z) \tag{1}$$

where the symbol $\oplus$ represents the XOR operator and w' is the complement of w.

The $PV \neq SV$ and $PV = SV$ edges connect to $f(w', w, .., z)$ and $f(w, w, .., z)$ functions, respectively. For the sake of simplicity, we refer hereafter to $f(w', w, .., z)$ and $f(w, w, .., z)$ as to $f_{v \neq w}$ and $f_{v=w}$, respectively. Functions of a single variable v cannot be directly decomposed by the biconditional expansion in Eq. 1. In such a condition, v is assigned to the PV and a fictitious variable $w = 1$ is introduced and assigned to the SV, collapsing the biconditional expansion into a Shannon's expansion. With this boundary condition, any Boolean function can be fully decomposed and represented with a BBDD.

B. BBDD Ordering

Constraining the order of variable appearance in every root to sink path is one of the requirements to achieve canonicity in BBDDs. For this purpose, the *Chain Variable Order* (CVO), introduced in [14], imposes a variable order on all root to sink paths for PVs and a rule for the adjacent SVs. Given a Boolean function f and an input variable order $\pi = (\pi_0, \pi_1, .., \pi_{n-1})$, the CVO assigns PVs and SVs by levels as:

$$\begin{cases} PV_i = \pi_i \\ SV_i = \pi_{i+1} \end{cases} \text{with } i = 0, 1, .., n-2; \begin{cases} PV_{n-1} = \pi_{n-1} \\ SV_{n-1} = 1 \end{cases} \tag{2}$$

CVO Example: From $\pi = (a, b, c)$, the corresponding CVO is obtained following Eq. 2. The consecutive ordering by couples (PV_i, SV_i) is thus $((a, b), (b, c), (c, 1))$.

Note that the size of an ordered BBDD depends on the particular variable order π assigned through the CVO. Dedicated re-ordering techniques are presented in Section IV-A4 to handle this issue.

C. BBDD Reduction

In order to improve the representation efficiency, BBDDs can be reduced according to a set of rules [14]. A BBDD is said reduced when it respects the following rules:

R1) It contains no two nodes, root of isomorphic subgraphs.
R2) It contains no nodes with identical children.
R3) It contains no empty levels.
R4) Subgraphs representing single variable functions degenerates into a single node with SV=1, i.e., a BDD node.

Note that rules **R1-2** are the straightforward extension of BDD [3] reduction rules. Then, rules **R3-4** derive from the enhanced expressive power of the *biconditional* expansion [14].

D. Reduced and Ordered BBDD Canonicity

Reduced and ordered BBDDs are canonical, i.e., unique for a given input variable order $\pi = (\pi_0, \pi_1, .., \pi_{n-1})$ and corresponding CVO. Canonicity is preserved if a complement attribute is enabled at $PV \neq SV$ edges and only the 1 sink node is permitted. Formal proofs about BBDD canonicity are given in [14]. Unless specified otherwise, we refer hereafter to BBDDs as to ordered and reduced BBDDs.

BBDDs improve the expressive power of binary decision diagrams while remaining a canonical representation form. In order to fully harness such opportunity in EDA, a BBDD package is presented in the following section.

IV. BBDD Manipulation Package

This section presents an efficient manipulation theory for BBDDs and its practical software implementation [19]. Experimental results for the developed BBDD package are given and compared to a state-of-art BDD package.

A. Efficient Manipulation of BBDDs

Nowadays, one fundamental reason to keep decision diagrams small is not just to successfully fit them into the memory, that in a modern server could store up to 1 billion nodes, but more to maximize their manipulation performance. Following this trend, we design the BBDD manipulation algorithms and data structures aiming to minimize the runtime while keeping under control the memory footprint. The key concepts unlocking such target are (i) *unique* table to store BBDD nodes in a *strong canonical form*, (ii) recursive formulation of Boolean operations in terms of *biconditional* expansions with relative *computed* table, (iii) memory management to speed up computation and (iv) *chain* variable re-ordering to minimize the BBDD size. We discuss in details each point hereafter.

1) Unique Table: BBDD nodes must be stored in an efficient form, allowing fast lookup and insertion. Thanks to canonicity, BBDD nodes are uniquely labeled by a tuple {CVO-level, $\neq$-child, $\neq$-attribute, $=$-child}. A *unique* table maps each tuple {CVO-level, $\neq$-child, $\neq$-attribute, $=$-child} to its corresponding BBDD node via a hash-function. Hence, each BBDD node has a distinct entry in the *unique* table pointed by its hash-function, enabling a *strong canonical form* representation for BBDDs.

Exploiting the *strong canonical form*, equivalence test between two BBDD nodes corresponds to a simple pointer comparison. Thus, lookup and insertion operations in the *unique* table are efficient. Before a new node is added to the BBDD, a lookup checks if its corresponding tuple {CVO-level, $\neq$-child, $\neq$-attribute, $=$-child} already exists in the *unique* table and, if so, its pointed node is returned. Otherwise, a new entry for the node is created in the *unique* table.

2) Boolean Operations between BBDDs: The capability to apply Boolean operations between two BBDDs is essential to represent and manipulate large functions of interest in EDA. Consequently, an efficient algorithm to compute $f \otimes g$, where $\otimes$ is any Boolean function of two operands and $\{f, g\}$ are two existing BBDDs, is mandatory in the manipulation package. A recursive formulation of $f \otimes g$, in terms of *biconditional* expansions, allows us to take advantage of the information stored in the existing BBDDs and hence reduce the computation complexity. Algorithm 1 shows the outline of the recursive implementation for $f \otimes g$. The input of the algorithm are the BBDDs for $\{f, g\}$ and the two-operand Boolean function $\otimes$ that has to be computed between them. If f and g are identical, or one of them is the sink 1 node, the operation $f \otimes g$ reaches a terminal condition. In this case, the result is retrieved from a pre-defined list of trivial operations and returned immediately (Alg.1α). When a terminal condition is not encountered, the presence of $\{f, g, \otimes\}$ is first checked in a *computed* table, where previously performed operations are stored in case of later use. In the case of positive outcome, the result is retrieved from the *computed* table and returned immediately (Alg.1β). Otherwise, $f \otimes g$ has to be explicitly computed

Algorithm 1 $f \otimes g$

```
INPUT: BBDDs for $\{f,g\}$ and Boolean operation $\otimes$.
OUTPUT: BBDD for $f \otimes g$, edge attribute ($Attr$) for $f \otimes g$.
  if (terminal case)$||(f == g)$ then
    $\{R, Attr\}$ = identical_terminal($\{f, g, \otimes\}$);        $\}\ \alpha$
    return $\{R, Attr\}$;
  else if computed table has entry $\{f, g, \otimes\}$ then
    $\{R, Attr\}$ = lookup computed table($\{f, g, \otimes\}$);      $\}\ \beta$
    return $\{R, Attr\}$;
  else
    $i = max_{level}\{f, g\}$;
    $\{v, w\} = \{PV, SV\}@(level = i)$;
    if $(|supp(f)| == 1)||(|supp(g)| == 1)$ then
      chain-transform($f, g$);
    end if
    $\{E, E \to Attr\} = f_{v=w} \otimes g_{v=w}$;
    $\otimes_D = update_{op}(\otimes, f_{v\neq w} \to Attr, g_{v\neq w} \to Attr)$;
    $\{D, D \to Attr\} = f_{v\neq w} \otimes_D g_{v\neq w}$;
    if reduction rule R4 applies then                                  $\}\ \gamma$
      $R$ =BDD-node $@(level = i)$;
    else if $\{E, E \to Attr\} == \{D, D \to Attr\}$ then
      $R = E$;
    else
      $D \to Attr = update_{attr}(E \to Attr, D \to Attr)$;
      $R = lookup\_insert(i, D, D \to Attr, E)$;
    end if
    insert computed table ($\{f, g, \otimes\}, R, E \to Attr$);
    return $\{R, E \to Attr\}$;
  end if
```

(Alg.1γ). The top level in the CVO for $f \otimes g$ is determined as $i = max_{level}\{f, g\}$ with its $\{PV_i = v, SV_i = w\}$. The root node for $f \otimes g$ is placed at such level i and its children computed recursively. Before proceeding in this way, we need to ensure that the two-variable *biconditional* expansion is well defined for both f and g, particularly if they are single variable functions. To address this case, single variable functions are prolonged down to $min_{level}\{f, g\}$ through a chain of consecutive BBDD nodes. This temporarily, and locally, may violate reduction rule **R4** to guarantee consistent $\neq$- and $=$-edges. However, rule **R4** is enforced before the end of the algorithm. Provided such handling strategy, the following recursive formulation, in terms of *biconditional* expansions, is key to efficiently compute the children for $f \otimes g$:

$$f \otimes g = (v \oplus w)(f_{v\neq w} \otimes g_{v\neq w}) + (v \overline{\oplus} w)(f_{v=w} \otimes g_{v=w}) \tag{3}$$

The term $(f_{v\neq w} \otimes g_{v\neq w})$ represents the $\neq$-child for the root of $f \otimes g$ while the term $(f_{v=w} \otimes g_{v=w})$ represents the $=$-child. In $(f_{v\neq w} \otimes g_{v\neq w})$, the Boolean operation $\otimes$ needs to be updated according to the regular/complemented attributes appearing in the edges connecting to $f_{v\neq w}$ and $g_{v\neq w}$. After the recursive calls for $(f_{v=w} \otimes g_{v=w})$ and $(f_{v\neq w} \otimes g_{v\neq w})$ return their results, reduction rule **R4** is applied. Finally, the tuple {top-level, $\neq$-child, $\neq$-attribute, $=$-child} is found or added in the *unique* table and its result updated in the *computed* table.

Observe that the maximum number of recursions in Eq. 3 is determined by all possible combination of nodes between

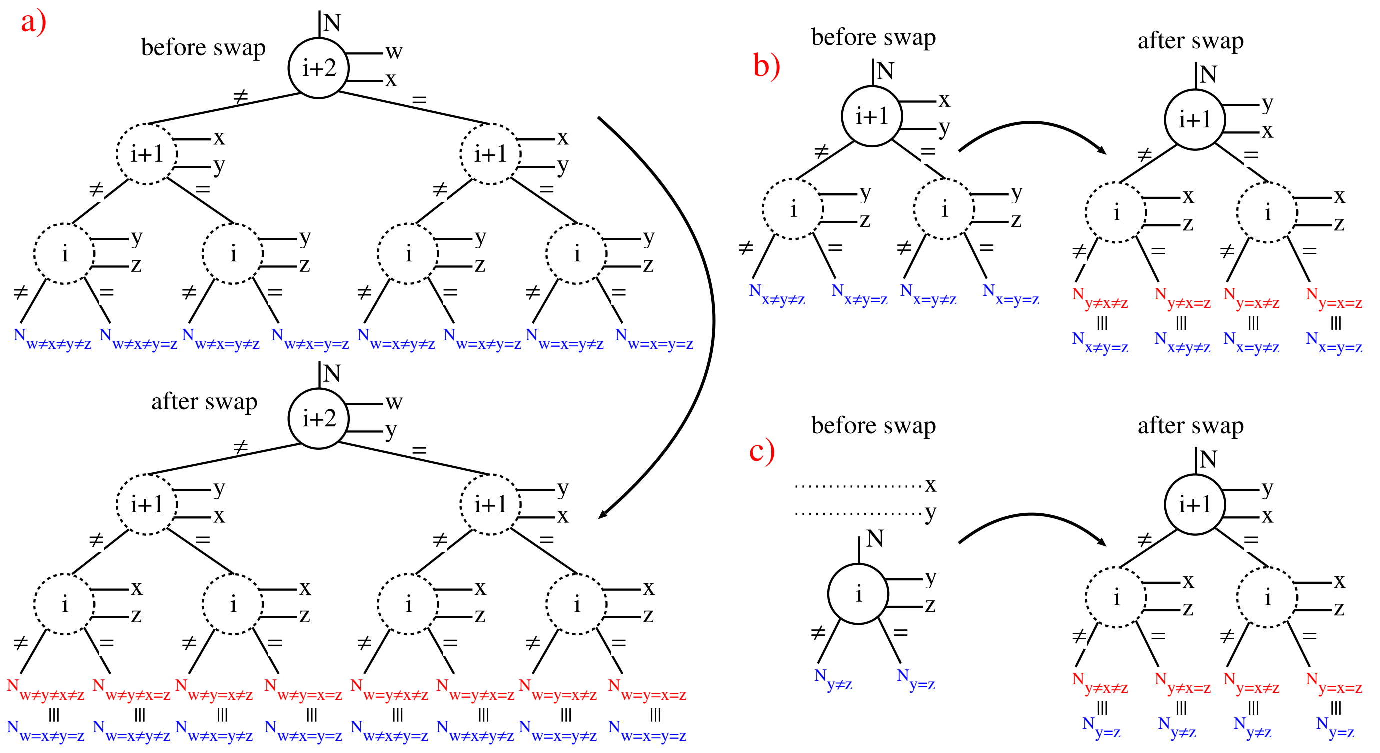

Fig. 2: Variable swap $i \rightleftharpoons i+1$ involving the CVO levels ($PV_{i+2} = w$, $SV_{i+2} = x$), ($PV_{i+1} = x$, $SV_{i+1} = y$) and ($PV_i = y$, $SV_i = z$). Effect on nodes at level $i+2$ (a) $i+1$ (b) and i (c).

the BBDDs for f and g. Assuming constant time lookup in the *unique* and *computed* tables, it follows that the time complexity for Algorithm 1 is $O(|f| \cdot |g|)$, where $|f|$ and $|g|$ are the number of nodes of the BBDDs of f and g, respectively.

3) Memory Management: The software implementation of data-structures for *unique* and *computed* tables is essential to control the memory footprint but also to speed-up computation. In traditional logic manipulation packages [20], the *unique* and *computed* tables are implemented by a hash-table and a cache, respectively. We follow this approach in the BBDD package, but we add some specific additional technique to further speed-up computation. Informally, we minimize the access time to stored nodes and operations by dynamically changing the data-structure size and hashing function, on the basis of a {size×access-time} quality metric.

The core hashing-function for all BBDD tables is the Cantor pairing function between two integer numbers [21]:

$$C(i, j) = 0.5 \cdot (i + j) \cdot (i + j + 1) + i \qquad (4)$$

which is a bijection from $\mathbb{N}_0 \times \mathbb{N}_0$ to $\mathbb{N}_0$ and hence a *perfect hashing function* [21]. In order to fit the memory capacity of computers, modulo operations are applied after the Cantor pairing function allowing collisions to occur. To limit the frequency of collisions, a first modulo operation is performed with a large prime number m, e.g., $m = 15485863$, for statistical reasons. Then, a final modulo operation resizes the result to the current size of the table.

Hashing functions for *unique* and *computed* tables are obtaining by nested Cantor pairings between the tuple elements with successive modulo operations.

Collisions are handled in the *unique* table by a linked list for each hash-value, while, in the *computed* table, the cache-like approach overwrites an entry when collision occurs.

Keeping low the frequency of collisions in the *unique* and *computed* table is of paramount importance to the BBDD package performance. Traditional garbage collection and dynamic table resizing [20] are used to address this task. When the benefit deriving by these techniques is limited or not satisfactory, the hash-function is automatically modified to re-arrange the elements in the table. Standard modifications of the hash-function consist of nested Cantor pairings re-ordering and re-sizing of the prime number m.

4) Chain Variable Re-ordering: The chain variable order for a BBDD influences the representation size and therefore its manipulation complexity. Automated chain variable re-ordering assists the BBDD package to boost the performance and reduce the memory requirements. Among traditional variable re-ordering techniques for binary decision diagrams, we extend Rudell's sifting algorithm [9]. Thanks to its general formulation, the sifting algorithm [9] can be directly extended to BBDDs, but a new swap theory handling the chain variable order is needed.

Variable swap in the CVO exchanges the PVs of two adjacent levels i and $i+1$ and updates the neighbor SVs accordingly. The effect on the original variable order π, from which the CVO is derived as per Eq. 2, is a direct swap of variables π_i and π_{i+1}. Note that all the nodes/functions concerned during a CVO swap are overwritten (hence maintaining the same pointer) with the new tuple generated at the end of the operation. In this way, the effect of the CVO swap remains local, as the edges of the above portion of the BBDD still point

TABLE I: Experimental Results for the BBDD Manipulation Package

Benchmarks	Inputs	Outputs	BBDD Package (this work)			BDD Package (CUDD)		
			Node Count	Build time (s)	Sift time (s)	Node Count	Build time (s)	Sift time (s)
C1355	41	32	54225	0.23	0.11	74056	0.06	0.59
C1908	33	25	14918	0.06	0.23	17980	0.09	0.34
C499	41	32	135784	1.56	3.21	160691	3.04	4.28
seq	41	35	4554	0.07	0.33	5607	0.14	0.44
my_adder	33	17	166	0.13	0.15	1006	0.15	0.14
frg1	28	3	284	<0.01	<0.01	296	<0.01	<0.01
misex3	14	14	745	0.02	<0.01	885	0.03	0.02
misex1	8	7	51	<0.01	<0.01	68	<0.01	<0.01
comp	32	3	97	<0.01	<0.01	330	0.23	0.67
count	35	16	328	<0.01	<0.01	342	<0.01	0.01
cordic	23	2	54	<0.01	<0.01	80	<0.01	0.01
alu4	14	8	1076	<0.01	<0.01	897	<0.01	<0.01
C17	5	2	15	<0.01	<0.01	13	<0.01	<0.01
9symml	9	1	19	<0.01	<0.01	25	<0.01	<0.01
z4ml	7	4	21	<0.01	<0.01	37	<0.01	<0.01
decod	5	16	46	<0.01	<0.01	96	<0.01	<0.01
parity	16	1	9	<0.01	<0.01	17	<0.01	<0.01
Average	22.64	12.82	1.24e04	0.12	0.24	1.54e04	0.22	0.37

to the same logical function.

A variable swap $i \rightleftharpoons i+1$ involves three CVO levels ($PV_{i+2} = w$, $SV_{i+2} = x$), ($PV_{i+1} = x$, $SV_{i+1} = y$) and ($PV_i = y$, $SV_i = z$). The level $i+2$ must be considered as it contains in SV the variable x, which is the PV swapped at level $i+1$. If no level $i+2$ exists ($i+1$ is the top level) the related operations are simply skipped. In the most general case, each node at level $i+2$, $i+1$ and i has 8, 4 and 2 possible children on the portion of BBDD below level i. Some of them may be identical, following to reduction rules **R1-4**, or complemented, deriving by the $\neq$-edges attributes in their path. Fig. 2 depicts the different cases for a general node N located at level $i+2$, $i+1$ or i, with all their possible children. After the swap $i \rightleftharpoons i+1$, the order of comparisons $w \star x \star y \star z$ is changed to $w \star y \star x \star z$ and the children of N must be rearranged consequently ($\star \in \{=, \neq\}$). Using the *transitive property of equality and congruence* in the binary domain, it is possible to remap $w \star x \star y \star z$ into $w \star y \star x \star z$ as:

$$\begin{array}{l} \star \in \{=,\neq\}, \quad \bar{\star} : \{=,\neq\} \rightarrow \{\neq,=\} \\ (w \star_{i+2} x = y \star_i z) \rightarrow (w \star_{i+2} y = x \star_i z) \\ (w \star_{i+2} x \neq y \star_i z) \rightarrow (w \bar{\star}_{i+2} y \neq x \bar{\star}_i z) \end{array} \tag{5}$$

Following remapping rules in Eq. 5, the children for N can be repositioned coherently with the variable swap. In Fig.2, the actual children rearrangement after variable swap is shown. In a bottom-up approach, it is possible to assemble back the swapped levels, while intrinsically respecting reduction rules **R1-4**, thanks to the *unique* table *strong canonical form.*

Based on this CVO swap theory, the BBDD re-ordering (sifting) algorithm is derived from [9] and works as follows. Let n be the number of variables in the initial order π. Each variable π_i is considered in succession and the influence of the other variables is locally neglected. Swap operations are performed to move π_i in all n potential positions in the CVO. The best BBDD size encountered is remembered and its π_i position in the CVO is restored at the end of the variable processing. This procedure is repeated for all variables. It follows that BBDD sifting requires $O(n^2)$ swap operations.

B. BBDD Results

We have developed a BBDD package in C language implementing the techniques described in this work. The package is available online at [19] and consists of about 4k lines of code. For the sake of comparison, we considered the latest release of CUDD [20], the state-of-art manipulation package for BDDs. BDDs [20] and BBDDs [19] packages run on a Xeon X5650 24-GB RAM machine. The benchmarks considered are taken from the traditional MCNC suite.

CUDD [20] receives as input a BLIF format description of a combinational logic network. BDDs are first built using the initial order provided in the BLIF file and later sifted [9].

The BBDD package [19] receives as input a Verilog description of a combinational logic network, flattened onto primitive Boolean operations (XOR, AND, OR, INV, BUF). Then, it provides as output a Verilog description for the built BBDD together with its log information. As for the BDD-based counterpart, BBDDs are first built using the initial order provided in the Verilog file and later sifted.

Table I shows experimental results for the BBDD package [19] and CUDD package [20]. On average, BBDDs built and sifted by the package developed in this work are 19.48% smaller, in terms of node count, with respect to BDDs built and sifted by CUDD [20]. This is thanks to the expressive power of BBDDs, where the branching decision at each node is *biconditional* on two variables per time, rather than only one as in standard BDDs. Moreover, the building and sifting time are faster for BBDDs than for BDDs, achieving an overall speed-up factor of 1.63x as compared to CUDD [20]. The reason for such speed-up is twofold. On the one hand, BBDDs have fewer nodes than BDDs so the number of operations required is directly reduced. On the other hand, the BBDD-package [19] has a memory management strategy expressly targeting low collision frequency, further boosting the package performance.

V. Case Study: Datapath Synthesis Front-End

This section showcases the interest of the BBDD package in the automated design of datapath circuits. A BBDD-based

TABLE II: Experimental Results for the BBDD-based Datapath Synthesis

Benchmarks	Inputs	Outputs	BBDD Package + Commercial Synthesis Flow			Commercial Synthesis Flow		
			Area (μm^2)	Delay (ns)	Gate Count	Area (μm^2)	Delay (ns)	Gate Count
Adder 32	64	33	41.01	2.17	186	45.98	3.42	216
Adder 64	128	65	83.05	4.46	380	93.02	7.01	440
Equality 32	64	1	17.78	0.11	63	18.27	0.18	72
Equality 64	128	1	35.57	0.13	119	36.18	0.20	136
Magnitude 32	64	1	13.65	0.82	41	21.77	1.16	186
Magnitude 64	128	1	29.44	1.64	102	44.17	2.30	378
Barrel 32	39	32	71.68	0.50	545	76.44	0.50	569
Barrel 64	70	64	165.42	0.58	1255	178.50	0.60	1320
Average	85.62	24.75	57.20	1.30	336.38	64.29	1.92	414.62

synthesis methodology is proposed and compared to a commercial synthesis flow, for a 22nm CMOS technology.

A. Re-writing Datapaths with BBDDs

Datapath circuits are essential components in today's integrated circuits. Standard synthesis techniques face challenges to optimize datapaths, as they are dominated by arithmetic operations which are not natively supported by conventional logic representation forms. Differently, BBDD nodes inherently act as two-variable comparators, a basis function for arithmetic operations. This feature enables datapaths to be efficiently represented with BBDDs. Nevertheless, the BBDD representation efficiency is not just limited to datapaths. Motivated by this consideration, we employ the BBDD package as front-end to a commercial synthesis tool to pre-structure datapaths and facilitate their synthesis.

B. Synthesis Results

Adder, comparator and shifter datapaths are considered, in operand 32 and 64 bit-widths, as synthesis benchmarks. They are written in Verilog language. A standard cell library consisting of MAJ-3, XOR-2, XNOR-2, NAND-2, NOR-2 and INV logic gates is characterized in CMOS 22 nm technology [22] and used to support the synthesis flow. When employed stand-alone, the commercial synthesis tool identifies arithmetic building blocks in datapaths and optimizes their corresponding implementation. We use the BBDD package to rewrite datapaths prior to the synthesis with the commercial tool. In this approach, such synthesis tool may be precluded to identify arithmetic operations but forced to ordinarily synthesize the restructured circuit, making the BBDD representation the main responsible for the quality of the designed datapath.

Table II shows synthesis results. On average, datapath designed by BBDD re-writing followed by the standard synthesis tool are 11.02% smaller and 32.29% faster as compared to the same circuits designed by the standard synthesis tool alone. BBDD representation is capable to evidence advantageous logic structures in datapath, not apparent with traditional synthesis techniques.

VI. Conclusions

We presented the theory, and software implementation, for efficient manipulation of *Biconditional Binary Decision Diagrams* (BBDDs), a novel class of decision diagrams based on an equality/inequality switching paradigm. As compared to a state-of-art decision diagram manipulation package, the developed BBDD software achieves an average node count reduction of 19.48% and a speed-up factor of 1.63x, measured over a standard set of logic benchmarks. The key factors enabling such improvements are (i) *strong canonical form* preconditioning of stored BBDD nodes, (ii) recursive formulation of Boolean operations in terms of *biconditional* expansions, (iii) performance-oriented memory management and (iv) dedicated BBDD re-ordering techniques. Employed in the synthesis of datapaths, the BBDD manipulation package is capable to advantageously restructure arithmetic operations producing 11.02% smaller and 32.29% faster circuits, as compared to a commercial synthesis flow.

Acknowledgements

This research was supported by ERC-2009-AdG-246810.

References

[1] C.Y. Lee, *Representation of Switching Circuits by Binary-Decision Programs*, Bell Systems Technical Journal, 1959.

[2] S.B. Akers, *Binary Decision Diagrams*, IEEE Trans. Comp., C-27(6):509-516, June 1978.

[3] R.E. Bryant, *Graph-based algorithms for Boolean function manipulation*, IEEE Transactions on Computers, C-35: 677-691, 1986.

[4] C. Yang and M. Ciesielski, *BDS: A BDD-Based Logic Optimization System*, IEEE Trans. CAD, vol. 21, pp. 866-876, July 2002.

[5] S. Malik *et al.*, *Logic verification using binary decision diagrams in a logic synthesis environment*, Proc. ICCAD, 1988.

[6] M.S. Abadir *et al.*, *Functional test generation for digital circuits using binary decision diagrams*, IEEE Trans. Comput., C35 (1986), pp.375-379.

[7] C. Scholl, R. Drechsler, B. Becker, *Functional simulation using binary decision diagrams*, Proc. ICCAD, 1997.

[8] K.S. Brace, R.L. Rudell, R.E. Bryant, *Efficient implementation of a BDD package*, Proc. DAC, 1990.

[9] R. Rudell, *Dynamic variable ordering for ordered binary decision diagrams*, Proc. ICCAD, 1993.

[10] R.E. Bryant, *On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication*, IEEE Trans. on Comp., vol. 40, no. 2, p. 205, February 1991.

[11] S. Minato, *Zero-suppressed BDDs for set manipulation in combinatorial problems*, Proc. DAC, pp. 272-277, 1993

[12] R. Drechsler *et al.*, *Ordered Kronecker functional decision diagrams: a data structure for representation and manipulation of Boolean functions*, IEEE Trans. CAD, Vol. 17, Issue 10, pp. 965-973, Oct. 1998.

[13] E.I. Goldberg, Y. Kukimoto, R.K. Brayton, *Canonical TBDD's and Their Application to Combinational Verification*, Proc. IWLS, 1997.

[14] L. Amaru, P.-E. Gaillardon, G. De Micheli, *Biconditional BDD: A Novel Canonical Representation Form Targeting the Synthesis of XOR-rich Circuits*, Proc. DATE 2013.

[15] R. Drechsler, B. Becker, *Binary Decision Diagrams: Theory and Implementation*, Kluwer Academic Publisher, 1998.

[16] M. De Marchi *et al.*, *Polarity control in Double-Gate, Gate-All-Around Vertically Stacked Silicon Nanowire FETs*, Proc. IEDM 2012.

[17] Y. Lin *et al.*, *High-Performance Carbon Nanotube Field-Effect Transistor with Tunable Polarities*, IEEE Trans. Nanotech., 4(5): 481-489, 2005.

[18] D. Lee *et al.*, *Combinational Logic Design Using Six-Terminal NEM Relays*, IEEE Trans. CAD, Vol. 32, Issue: 5, pp-653-666, May 2013.

[19] *BBDD package available at*: *http://lsi.epfl.ch/page-102566-en.html.*

[20] *CUDD: CU Decision Diagram Package Release 2.5.0*

[21] P. Tarau, *Pairing Functions, Boolean Evaluation and Binary Decision Diagrams*, Proc. CICLOPS, 2008

[22] Predictive Technology Model (PTM), http://ptm.asu.edu/