BDS-MAJ: A BDD-based Logic Synthesis Tool Exploiting Majority Logic Decomposition

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

Abstract—**Despite the impressive advance of logic synthesis during the past decades, a general methodology capable of efficiently synthesizing both control and datapath logic is still missing. Indeed, while synthesis techniques for random control logic (AND/OR-intensive) are well established, no dominant method for automated synthesis of datapath logic (XOR/MAJ-intensive) has yet emerged. Recently, *Binary Decision Diagrams* (BDDs) have been adopted to create an optimization system, named BDS, that supports integrated synthesis of both AND/OR- and XOR-intensive functions through functional logic decomposition on the BDD structure. However, it does not support direct decomposition and manipulation of majority logic which, instead, is widely used in datapath circuits. In this paper, we present the first BDD-based majority logic decomposition method and a logic decomposition system, BDS-MAJ, that enables efficient logic synthesis for both random control and datapath circuits. Experimental results show that logic synthesis based on BDS-MAJ produces CMOS circuits having on average 28.8% and 26.4% less area and, at the same time, 12.8% and 20.9% smaller delay with respect to academic ABC and BDS synthesis tools. Compared to commercial Synopsys *Design Compiler* synthesis tool, BDS-MAJ reduces on average the circuit area by 6.0% and decreases the delay by 7.8%.**

Categories and Subject Descriptors
B.6.3 [Design Aids]: Automatic Synthesis, Optimization

General Terms
Algorithms, Design, Performance, Theory.

Keywords
Majority Logic, Decomposition, BDD, Logic Synthesis.

I. Introduction

Virtually all digital integrated circuits are realized using logic synthesis techniques [1]. Whereas most circuits contain datapath and control functions, current logic synthesis tools are better at synthesizing control logic as compared to datapaths, especially when arithmetic functions are involved. Indeed, original logic synthesis techniques [2]–[5], which are the basis for current commercial tools, exploited algorithms using AND/OR representations, while arithmetic/datapath circuits are rich in XOR and MAJORITY functions. A major aim for today's synthesis tools is to handle properly both random control and datapath logic to fully and efficiently automatize ASIC designs.

A step toward this direction is enabled by the use of *Binary Decision Diagrams* (BDDs) [6]–[8] as logic representation structures, because they are typically compact for a wide class of functions, including AND/OR- and XOR/MAJ-intensive functions. To exploit this opportunity, BDDs are considered in [9]–[14]. In these works, BDDs support automated synthesis through efficient logic decomposition. In particular, a BDD-based decomposition theory is proposed in [10] to support various logic structures, *e.g.*, AND, OR, XOR and MUX. Based on this theory, a practical synthesis tool named BDS is described in [10] and refined in [11]. BDS advantageously synthesizes both AND/OR- and XOR-intensive functions thanks to an unified methodology. However, BDS still does not manipulate majority logic losing further optimization opportunities in datapath circuits.

In this paper, we aim to extend the capability of current logic synthesis methods by focusing on majority decomposition which is useful for both for datapath (XOR/MAJ-intensive) and control (AND/OR-intensive) logic. To this end, we present the first majority logic decomposition method based on BDD. We integrate the proposed majority decomposition technique in the current state-of-art BDD-based decomposition tool, BDS-PGA [11], in order to create a new complete decomposition tool, BDS-MAJ. Large MCNC and custom datapath benchmarks are used to evaluate BDS-MAJ. Thanks to its runtime efficient algorithms, BDS-MAJ decomposes the largest benchmarks in few seconds. BDS-MAJ produces decomposed networks having on average 29.1% fewer nodes as compared to BDS-PGA. To exploit the new decomposition feature, we employ BDS-MAJ in a standard *optimization-mapping* synthesis flow. Experimental results over MCNC and custom datapath benchmarks show that BDS-MAJ outperforms academic ABC and BDS synthesis tools by 28.8% and 26.4% in less area, respectively, and by 12.8% and 20.9% in smaller delay, respectively. When compared to commercial Synopsys *Design Compiler*, BDS-MAJ produces, on average, circuit with 6.0% less area and 7.8% smaller delay.

The remainder of this paper is organized as follows. Section II provides a background on BDD-based logic decomposition. In Section III, the new majority logic decomposition method is presented. Then, in Section IV, the implementation of the BDS-MAJ decomposition system is discussed. Experimental results for BDS-MAJ are presented and compared with state-of-art commercial and academic synthesis tools in Section V. We conclude the paper in Section VI.

II. Background and Motivation

This section presents relevant background about *Binary Decision Diagrams* (BDDs) and related logic decomposition. Notations and definitions for BDDs are also introduced.

A. Binary Decision Diagrams

BDDs are logic representation structures that were first introduced by Lee [6] and Akers [7]. The notions of ordering and reduction of BDDs were introduced by Bryant in [8], where it was shown that, with these restrictions, BDDs are a canonical logic representation form. Canonical reduced and ordered BDDs are often compact and easy to manipulate, and are therefore widely used in EDA and other fields. We assume that the reader is familiar with basic concepts of Boolean algebra and BDDs (for a review see [1], [10]). We review hereafter only the basic notation used in the rest of the paper.

B. Notation

A BDD is a *Direct Acyclic Graph* (DAG) representing a Boolean function. A BDD is uniquely identified by its *root*, the set of *internal nodes*, the set of *edges* and the *1/0-sink nodes*.

Each internal node in a BDD is labeled by a Boolean variable v and has two out-edges labeled 0 and 1. Each internal node represents the Shannon's expansion with respect to its variable v and the 1- and 0-edges connect to positive and negative Shannon's cofactors, respectively.

Edges are characterized by a regular/complemented attribute: complemented edges indicate to invert the function pointed by that edge.

We refer hereafter to BDDs as to *canonical reduced and ordered* BDDs [8], that are BDDs where (i) each input variable is encountered at most once in each root to sink path and in the same order on all such paths, (ii) each internal node represent a distinct logic function and (iii) only 0-edges can be complemented.

C. BDD-based Logic Decomposition

BDDs are exploited to achieve efficient logic function decomposition [9]–[14] thanks to the notable characteristics of the BDD structure. Special classes of nodes, defined as dominators, are used in [10], [11], [14] to guide the decomposition process directly on the BDD structure. Dominator nodes allow us to uniquely identify substructures in the BDD that are corresponding to specific decomposition types. In [14], 0- and 1-dominators are introduced to support disjoint AND/OR decompositions. Similarly, in [10], x-dominators are defined to support disjoint XNOR decomposition. Generalized 0-, 1- and x-dominators are also introduced in [10] to achieve general non-disjoint decompositions. The decomposition system BDS [10] is based on dominator nodes driven decomposition. BDS exhibits better decomposition results for XOR/XNOR-intensive circuits as compared to traditional AND/OR optimization techniques while maintaining good results quality also for random control logic [10]. A successive version of BDS has been proposed in [11], named BDS- PGA, incorporating further decomposition schemes that generate area-minimal logic networks. Despite BDS and its evolutions are efficient to decompose a great variety of logic functions, they are still missing the opportunity to identify majority decomposition structures, that are widely used in datapath circuits as well as in some random control applications. Note that early attempts to achieve majority logic decomposition are already reported in the 60's [15], but, due to their intractable complexity, failed to gain momentum later in automated logic synthesis. We address, in this paper, the unique opportunity led by efficient majority logic decomposition employed in a contemporary synthesis flow.

III. Majority Logic Decomposition

In this section, we present our novel majority logic decomposition theory. First, the existence of majority decompositions for general logic functions is studied. Then, the search for majority dominator nodes on BDDs is introduced, and, subsequently, the actual majority decomposition construction, optimization and selection phases are described. Finally, the computational complexity of the proposed decomposition method is evaluated.

A. Majority Decomposition

The aim of majority logic decomposition is to express a Boolean function F in the form $Maj(F_a, F_b, F_c)$.

Theorem 3.1: (Existence of majority decomposition) Given a Boolean function F, there always exists a decomposition for F in the form $F = Maj(F_a, F_b, F_c)$.

Proof: (*By construction*) We consider, in the context of the proof, to operate at truth table level to build functions F_a, F_b and F_c. The truth table for the original function F has $2^{|supp(F)|}$ rows, each one indicating a distinct input combination and the corresponding logic value assumed by F. Now, we add in the truth table for F, other additional three columns indicating the logic values assumed by F_a, F_b and F_c for the same input combinations considered in F. For each of these input combinations (row of the truth table) we impose that two functions over three (the choice of the couple is free for each row) among F_a, F_b and F_c must assume the same value of F while the remaining one is free to be set to any logic value. In this way, $Maj(F_a, F_b, F_c) = F$ is respected for each row, and therefore for all the possible input combinations. ■

Thus, as a consequence of the symmetry in the majority operator, there exist many possible majority decomposition structures for a function F. In order to reduce the search space for majority decompositios, it is useful to first determine one of the three functions and then apply majority construction methods for the remaining two. The choice of the first function, F_a, is a crucial point that determines the quality of the resulting majority decomposition. BDDs offer the opportunity to efficiently identify candidates for F_a through the use of a special class of nodes, *majority dominators*, defined in the next subsection.

Our majority decomposition method is presented in Algorithm 1. First, candidates for the function F_a are searched on the BDD $(\boldsymbol{\alpha})$

Algorithm 1 Majority decomposition method

```
INPUT: Boolean function $F$
OUTPUT: $F = Maj(F_a, F_b, F_c)$
FUNCTION: MajDecomp($F$)
  Build a BDD for $F$
  $m$-dom-list ←Search for non-trivial $m$-dominators              $(\boldsymbol{\alpha})$
  for all nodes $v$ in $m$-dom-list do
    $F_a$ is the function rooted at node $v$                          $(\boldsymbol{\alpha})$
    $F_b = ITE(F_a \oplus F, F, F_{F_a})$                               $(\boldsymbol{\beta})$
    $F_c = ITE(F_a \oplus F, F, F_{F_a'})$                              $(\boldsymbol{\beta})$
    while (improvement)&&(iterations<limit) do
      for all couples $(X, Y)$ among $F_a$,$F_b$ and $F_c$ do
        $F_x = X \oplus Y$                                             $(\boldsymbol{\gamma})$
        XOR-decompose $F_x$ in $M$ and $K$ (balance)                  $(\boldsymbol{\gamma})$
        $X_{opt} = ITE(F_x, K, X)$                                     $(\boldsymbol{\gamma})$
        $Y_{opt} = ITE(F_x, M, Y)$                                     $(\boldsymbol{\gamma})$
      end for
      evaluate improvement
      increase iteration count
    end while
    if isbest(current decomposition)==1 then
      best decomposition ← current decomposition                    $(\boldsymbol{\omega})$
    else
      keep previous best decomposition                              $(\boldsymbol{\omega})$
    end if
  end for
```

using majority dominators. Then, an initial solution for the majority decomposition $F = Maj(F_a, F_b, F_c)$ is determined $(\boldsymbol{\beta})$. Successively, the initial solution is optimized via an iterative procedure $(\boldsymbol{\gamma})$. Finally, the best decomposition over all the F_a candidates is selected $(\boldsymbol{\omega})$. The four major phases of Algorithm 1 are detailed in the following subsections.

B. Majority Dominator on a BDD

BDDs allow us to identify advantageous logic decompositions. Dominator nodes have been reported in [10] to support AND, OR, XOR, XNOR and MUX decompositions. Similarly to this approach, we search for nodes whose characteristics lead to an efficient majority decomposition, $(\boldsymbol{\alpha})$-phase in Algorithm 1. We call such nodes m-dominators. A m-dominator node is the root for the candidate function F_a. Following Theorem 3.1, every node in the BDD can be a valid m-dominator but some of them lead to a non-advantageous decomposition. For this reason, we introduce some characteristics to select *non-trivial* m-dominators leading to potential advantageous majority decompositions. We refer to a *non-trivial* m-dominator as to an internal BDD node that:

(i) it is not a simple x-, 0- or 1- dominator.

(ii) has more than one non complemented 0-incoming edges and 1-incoming edges,

Condition (i) avoids simple x-, 0- and 1- dominators [10] since they uniquely indicate XNOR, OR and AND disjoint decompositions. The intuition behind condition (ii) is that the function F_a in $Maj(F_a, F_b, F_c)$ must be reached for all the input combinations corresponding to $Maj(F_a, 0, 1)$ and $Maj(F_a, 1, 0)$ and therefore is most likely a highly connected node (high fan-in) in the BDD. Fig. 1 depicts the BDD for a simple $F = ab + bc + ac$ and highlights its non-trivial m-dominator.

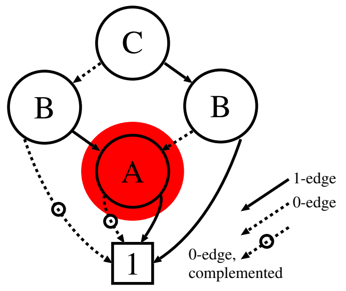

Fig. 1: BDD for the function $ab + bc + ac$, *non-trivial* m-dominator highlighted in red.

C. Majority Decomposition Construction

Once a candidate F_a is identified, the majority decomposition is constructed computing two F_b and F_c functions, $(\boldsymbol{\beta})$-phase in Algorithm 1. These two functions, F_b and F_c, must respect $F = Maj(F_a, F_b, F_c)$ for all the possible input combinations. In particular, for the input combinations such that F and F_a have different values, the functions F_b and F_c must assume the same logic value as F to guarantee $Maj(F_a, F_b, F_c) = F$. On the other hand, for the remaining input combinations, F and F_a have the same value and only one function between F_b and F_c must assume the same value of F to guarantee $Maj(F_a, F_b, F_c) = F$, the remaining one is free to assume any logic value. This concept is formalized in the following theorem.

Theorem 3.2: (Majority logic construction given F_a) Given a function F and a candidate function F_a, a majority decomposition $F = Maj(F_a, F_b, F_c)$ is valid <u>*if and only if*</u>:

$$\begin{cases} F_b = ITE(F_a \oplus F, F, H) \\ F_c = ITE(F_a \oplus F, F, W) \end{cases} \tag{1}$$

where ITE is the *if-then-else* logic operator and H, W are logic functions satisfying the equation:

$$(H \overline{\oplus} F) + (W \overline{\oplus} F) = 1 \tag{2}$$

Proof: Let S be the set of all the possible input combinations for F. Let $S_{(F \neq F_a)}$ be the subset of S such that $F_a \neq F$ (i.e., $F_a = \overline{F}$). Let $S_{(F=F_a)}$ be the subset of S such that $F_a = F$. Note that $S_{(F=F_a)} \cap S_{(F \neq F_a)} = \emptyset$ and $S_{(F=F_a)} \cup S_{(F \neq F_a)} = S$. We need to prove the theorem valid for all the input combinations (S). To this end, we divide the rest of the proof in two parts, first for the input combinations in $S_{(F \neq F_a)}$ and successively for $S_{(F=F_a)}$.

(i) Input combinations from $S_{(F \neq F_a)}$. For these inputs, the ITE operators in Equation 1 always return the *then* part, since $F \oplus F_a$ is always true when $F \neq F_a$. Consequently, $F_b = F_c = F$ and $Maj(F_a, F_b, F_c) = Maj(\overline{F}, F, F) = F$ which is the only valid majority decomposition for $S_{(F \neq F_a)}$ input set. Indeed, note that if F_b or F_c assume any other value than F, the decomposition $Maj(F_a, F_b, F_c)$ will be equal to $\overline{F}$ and not anymore to F.

(ii) Input combinations from $S_{(F=F_a)}$. For these inputs, the ITE operators in Equation 1 always return the *else* part. Therefore, $F_b = H$, $F_c = W$ and $Maj(F_a, F_b, F_c) = Maj(F, H, W)$. Since Equation 2 imposes that, for every input combinations, at least one function between H and W is equal to F, the majority decomposition always have at least two terms equal to F, which is a sufficient condition to say $Maj(F, H, W) = F$. To show that this is the only valid decomposition, consider to use the complement of Equation 2. In this case, both H and W are not equal to F but to its complement $\overline{F}$, therefore the decomposition $Maj(F, H, W) = Maj(F, \overline{F}, \overline{F})$ will be equal to $\overline{F}$ and not anymore to F. ■

Following Theorem 3.2, the choice of H and W functions respecting Equation 2 is the only freedom left to design a majority decomposition for F, given F_a.

Consequently, the choice of H and W is a key point to obtain minimum-sized F_b and F_c functions. A trivial solution is $H = F$ and $W = "Don't\ Care"$. Obviously, this is an inefficient solution since F_b reduces to the original F itself.

Unfortunately, exact-methods to find the best H and W optimizing a given metric and respecting Equation 2 are intractable. For this reason, we propose to use as *initial seed* the two following functions:

$$\begin{cases} H = F_{F_a} \\ W = F_{F'_a} \end{cases} \tag{3}$$

where the expression X_Y stands for the generalized cofactor of function X w.r.t. function Y. The generalized cofactor can be efficiently computed using BDD algorithms such as *restrict* [17] and *constraint* [18]. We prove that the proposed *initial seed* lead to a valid majority decomposition in the following theorem.

Theorem 3.3: (H and W initial seed) The H and W functions of Equation 3 respect the condition in Equation 2.

Proof: $(H \overline{\oplus} F) + (W \overline{\oplus} F)$ condition from Equation 2 reduces to $(F_{F_a} \overline{\oplus} F) + (F_{F'_a} \overline{\oplus} F)$. Expanding F into $F_{F_a} F_a + F_{F'_a} F'_a$ inside the formula, we get $F_{F_a} F_{F'_a} + F'_{F_a} + F'_{F'_a} + F_{F_a} F'_a + F_{F'_a} F_a$ that can be further simplified in $F_{F_a} F_{F'_a} + (F_{F_a} F_{F'_a})' + F_{F_a} F'_a + F_{F'_a} F_a$ which is indeed a tautology. Equation 2 is therefore respected. ■

Using the initial seeds for H and W functions, a *non-trivial* majority decomposition can be constructed starting from the original function F and the candidate F_a function.

Example (Majority decomposition construction): $F = ab+bc+ac$. F_a is a as highlighted by Fig. 1. $H = F_{F_a} = b + c$. $W = F_{F'_a} = bc$. Applying the ITE operator we get $F_b = b + c$ and $F_c = bc$. $Maj(F_a, F_b, F_c) = Maj(a, b + c, bc) = ab + bc + ac$ is valid.

As evidenced by the previous example, the H and W formula in Equation 3 may not highlight the most convenient F_b and F_c functions. In order to further optimize the couple of functions (F_b, F_c), but also (F_a, F_c) and (F_a, F_b), we propose a cyclic optimization procedure.

D. Majority Decomposition Optimization

Given a majority decomposition $F = Maj(F_a, F_b, F_c)$, it is possible to minimize F_a, F_b and F_c, while maintaining the decomposition validity. This is done during $(\boldsymbol{\gamma})$-phase in Algorithm 1, exploiting the majority operator functionality. Indeed, for each possible input combination, if a pair of functions (X,Y) among (F_a, F_b and F_c) assume the same value, this is the output value of the majority operator, while if the logic values of (X,Y) are opposite, the majority operator is uniquely determined by the remaining function. In the latter case, the actual values of (X,Y) are not important, it is only needed that X≠Y. This opens up the possibility to restructure and balance the pair of functions (X,Y) in order to reduce the complexity of the current majority decomposition. The majority balancing concept is illustrated by Fig. 2 and formalized in the following theorem.

Theorem 3.4: (Majority decomposition balancing) Given a majority decomposition $F = Maj(F_a, F_b, F_c)$, any pair of functions from (F_a, F_b and F_c), say for example (F_b, F_c), can be restructured as:

$$\begin{cases} F_{b-res} = ITE(F_b \oplus F_c, K, F_b) \\ F_{c-res} = ITE(F_b \oplus F_c, M, F_c) \end{cases} \tag{4}$$

where K and M are logic functions satisfying the equation:

$$(M \oplus K) = (F_b \oplus F_c) \tag{5}$$

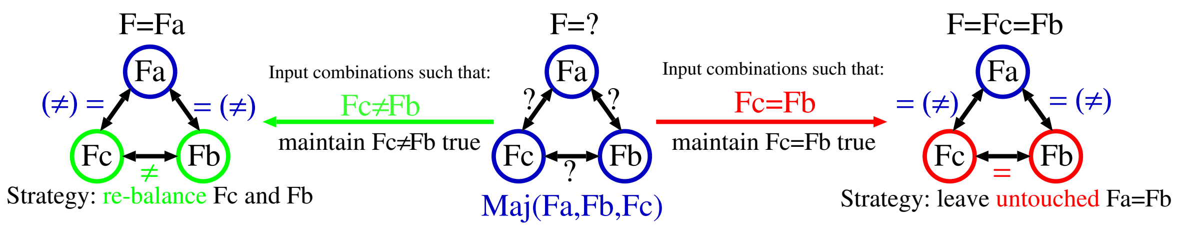
F=Fa
Fa
(≠) =
= (≠)
Fc
Fb
≠
Strategy: re-balance Fc and Fb
Input combinations such that:
Fc≠Fb
maintain Fc≠Fb true
F=?
Fa
?
?
Fc
Fb
?
Maj(Fa,Fb,Fc)
Input combinations such that:
Fc=Fb
maintain Fc=Fb true
F=Fc=Fb
Fa
= (≠)
= (≠)
Fc
Fb
=
Strategy: leave untouched Fa=Fb

Fig. 2: Majority decomposition balancing. For the input combinations such that $F_b = F_c$, the original function becomes $F = F_b = F_c$ and the values of F_b, F_c cannot be touched, without involving F_a. Instead, for the input combinations such that $F_b \neq F_c$, the original function becomes $F = F_a$ and the actual values of F_b, F_c do not matter provided that $F_b \neq F_c$ remains valid. It is then possible to restructure F_b,F_c.

Proof: Similarly to the proof of Theorem 3.2, we define S as the set of all the possible input combinations for F, $S_{(F_b=F_c)}$ the subset of S such that $F_b = F_c$ and $S_{(F_b \neq F_c)}$ the subset of S such that $F_b \neq F_c$. We divide the proof in two parts, first for the input combinations in $S_{(F_b=F_c)}$ and successively for $S_{(F_b \neq F_c)}$.

(i) Input combinations from $S_{(F_b=F_c)}$. For these inputs, the ITE operators in Equation 4 always return the *else* part. Consequently, $F_{b-res} = F_b$ and $F_{c-res} = F_c$ and $Maj(F_a, F_b, F_c)$ remains valid.

(ii) Input combinations from $S_{(F_b \neq F_c)}$. For these inputs, the ITE operators in Equation 4 always return the *then* part. Consequently, $F_{b-res} = K$ and $F_{c-res} = M$ and $Maj(F_a, F_b, F_c) = Maj(F_a, K, M)$. Note that for the considered inputs in $S_{(F_b \neq F_c)}$, $Maj(F_a, F_b, F_c) = F_a$ since F_b and F_c assumes opposite logic values. Indeed, $Maj(F_a, K, M) = Maj(F_a, K, \overline{K})$ due to Equation 5 evaluated for the same inputs in $S_{(F_b \neq F_c)}$. Finally, $Maj(F_a, K, M) = Maj(F_a, K, \overline{K}) = F_a$ remains a valid majority decomposition. ■

The effectiveness of the decomposition balancing operation depends on the way K and M functions are chosen. BDD-based XOR decomposition methods in [10] offer an efficient opportunity to compute balanced M and K functions starting from $(F_b \oplus F_c)$, and therefore respecting Equation 5. In Algorithm 1, we employ such core techniques to obtain M and K functions. Then, the decomposition balancing/optimization operation is iterated over all the possible functions pairs and till there is a complexity reduction, or the maximum number of iterations is reached.

We present the majority optimization method applied to the previous example.

Example (Majority decomposition balancing): $F = ab + bc + ac$. $F_a = a$, $F_b = b + c$ and $F_c = bc$ from the previous example. $(F_b \oplus F_c) = ((b + c) \oplus (bc)) = (b \oplus c)$. The XOR-decomposition of $(b \oplus c)$ leads to $K = b$, $M = c$. The ITE operators of Equation 4 finally achieve $F_b = b$ and $F_c = c$. No more optimization is needed: $F_a = a$, $F_b = b$ and $F_c = c$, $Maj(a, b, c) = ab + bc + ac$.

E. Majority Decomposition Selection

Algorithm 1 produces a number of different majority decompositions $F = Maj(F_a, F_b, F_c)$ equal to the number of *non-trivial* m-dominators, corresponding to F_a candidates, in the BDD for F. In order to evaluate the best majority decomposition among the others, $(\boldsymbol{\omega})$-phase in Algorithm 1, some metric is needed. We use as a first metric the size ($|F|$) of the decomposed functions: a majority decomposition 1 is superior to another majority decomposition 2 if $(|F_{a1}| + |F_{b1}| + |F_{c1}|) < (|F_{a2}| + |F_{b2}| + |F_{c2}|)$. However, this condition alone does not permit to evaluate the specific balance between each decomposed function. We employ as additional condition taking in account this property $(k|F_{a1}| \leq |F_{a2}|)\&(k|F_{b1}| \leq |F_{b2}|)\&(k|F_{c1}| \leq |F_{c2}|)$, where $\&$ stands for the logical AND operator and k is a sizing factor determined heuristically.

F. Majority Decomposition Computational Complexity

In order to estimate the computational complexity of the proposed decomposition method, we denote by N the number of nodes in the BDD for the original function F. We consider in this paper a fully BDD-based implementation of the majority decomposition method. Thanks to the efficiency of BDDs manipulation algorithms and to the BDDs representation canonicity [8], the ITE and 2-operands Boolean operators are always guaranteed to produce minimized BDDs, for a given variable order, despite having redundancy in the inputs. The most expensive operation in Algorithm 1 is the ITE operator that, as employed there, has a computational complexity of $O(N^3)$ [19]. Instead, any Boolean operator of 2 arguments [8] has here a computational complexity of $O(N^2)$. The cyclic majority optimization loop is limited to take a prefixed maximum number of iterations. The number of *non-trivial* m-dominators is in general $O(N)$ but can be adjusted on the fly specifying tighter selection constraints about the fan-in of m-dominators. The overall majority decomposition algorithm has $O(N^4)$ computational complexity. Note that, in practice, the runtime is much less than $O(N^4)$. Indeed, (i) the ITE and *2-operand* Boolean operators have a typical runtime performance close to the size of the resulting function ($|F|$) [19] and (ii) the number of *non-trivial* m-dominators can be made remarkably small with tight selection constraints.

IV. BDS-MAJ System Implementation

This section introduces a complete logic optimization system, BDS-MAJ. BDS-MAJ integrates the majority decomposition method proposed in Section III with the *BDD Decomposition System* (BDS) presented in [10], [11]. The synthesis flow for BDS-MAJ is shown in Fig. 3. The *network partitioning* and *factoring trees optimization* phases are maintained the same as in BDS. We refer the reader to [10] for a detailed description of these phases. The BDD-based decomposition engine from [10] is here adapted to support MAJ decomposition in addition to XOR, AND, OR and MUX decompositions that are currently supported in BDS [10], [11]. A concise description of the network partitioning phase is provided in the next subsection. Then, details for the BDD-decomposition engine are given. Finally, factoring trees optimization is briefly reviewed.

A. Network Partitioning

Since the manipulation of a global BDD may be impractical for large logic circuits [20], in BDS [10] a preprocessing of the input Boolean network is proposed. It consists of a partial collapsing of the input circuit into a set of supernodes. Each of these super nodes is then efficiently represented as a local BDD. The actual partial collapsing method is implemented in [10] using an evolution of the *eliminate* procedure described in [21]. In BDS-MAJ, we maintain untouched the network partitioning phase from BDS [10].

B. BDD Decomposition Engine

The BDD-decomposition engine in [10] takes in input each BDD produced by the network partitioning phase. As a first step, it performs variable reordering to compact the size of the input BDD. Then,

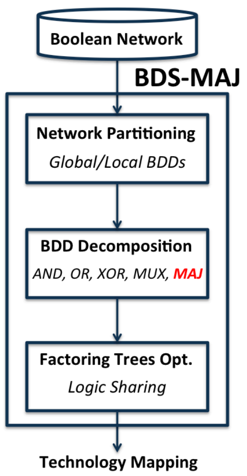

Fig. 3: BDS-MAJ synthesis flow and its main phases.

it starts a search for efficient BDD decompositions. Dominator nodes are used to guide the decomposition process. Simple dominator nodes (0-, 1- and x- dominators) are first considered since they indicate advantageous disjoint decompositions. If no simple dominator is found, the search continues for general dominators that enable non-disjoint decompositions. As a last resort, if no dominator nodes are found, the BDD is decomposed by cofactoring with respect to the top variable (MUX).

We embed our majority decomposition method on the top of the dominator nodes search. Even though that the proposed method obtains general non-disjoint decomposition, a complex radix-3 decomposition (MAJ) is potentially much more advantageous than the traditional radix-2 decompositions (XOR, AND, OR).

In BDS-MAJ, if no m-dominators are found or if the obtained majority decomposition is considered not advantageous for the global decomposition, the standard dominator nodes search in [10] is continued. In order to evaluate if the majority decomposition is useful, we use a similar metric to the one defined in Section III-E where the right hand of the equations is substituted with the size of the original BDD to be decomposed. We distinguish this metric as *global* majority selection while we refer to the one described in Section III-E as *local* majority selection. For the *global* majority selection, the sizing factor k is set to 1.6 by extensive simulations. In a similar way, for the *local* majority selection, the sizing factor k is set to 1.5. Note that the number of iterations in the majority decomposition cyclic optimization is set to 5.

C. Factoring Trees Optimization

In BDS [10], the result of the decomposition is stored in a *factoring tree*. Logic sharing between *factoring trees* is applied in order to further optimize the synthesis result. The bottom up construction of factoring trees (corresponding to the top-down decomposition of BDDs) enables efficient on-line logic sharing detection during the decomposition process. Moreover, the canonicity of BDDs simplifies the actual sharing detection task. In BDS-MAJ, the *factoring trees* optimization procedure of [10] is maintained.

V. Experimental Results

In this section, we evaluate the advantage of the proposed majority logic decomposition method at both logic optimization and logic synthesis levels. First, BDS-MAJ is employed to decompose large logic functions, comprising random control and datapath logic, taken both from the MCNC suite and *ad hoc* large HDL descriptions. Then, the decomposed circuits are mapped onto a simple standard cell library, characterized for CMOS 22nm technology node [22]. Logic optimization and synthesis based on BDS-MAJ are compared to academic BDS-PGA [11], ABC [16], and commercial Synopsys *Design Compiler* (DC) tools, fed with the same standard cell library.

A. Logic Optimization

We present here experimental methods and results for logic optimization performed by BDS-MAJ.

1) Methods: BDS-MAJ is compared to BDS-PGA [11] in terms of node count in the decomposed network. The benchmarks are taken both from the MCNC suite and custom HDL descriptions. HDL descriptions are converted in blif format using a HDL-to-blif translator. Default execution options are used for BDS-PGA and maintained in BDS-MAJ. Local and global majority selection sizing factors are kept the same as in Section IV.

2) Results: Table I summarizes experimental results for logic circuit decomposition. The average node count of BDS-MAJ is 29.1% smaller than BDS-PGA, highlighting the superior decomposition power enabled by the use of majority logic. Majority logic nodes account for 9.8% of the total node count in BDS-MAJ. This result evidences that even a small fraction of majority nodes advantageously restructures the logic function producing consistently more compact logic circuits compared to ordinary techniques. The runtime of BDS-MAJ is almost the same as the one of BDS-PGA, only a slight 4.6% average runtime increase is reported.

B. Logic Synthesis

Experimental methods and results for BDS-MAJ based logic synthesis are presented hereafter.

1) Methods: We evaluate the advantage of BDS-MAJ employed in a traditional *optimization-mapping* synthesis flow. To this end, a standard cell library consisting of MAJ-3, XOR-2, XNOR-2, NAND-2, NOR-2 and INV logic gates is characterized for CMOS 22nm technology [22]. Technology mapping after BDS-MAJ logic optimization is performed in two steps. First, MAJ, XOR and XNOR nodes are directly assigned to logic cells in order to preserve such highlighted functions, otherwise potentially hidden by standard technology mappers. Then, the rest of the logic circuit is mapped using ABC [16] mapper. BDS-MAJ synthesis flow is compared to academic ABC, BDS and commercial Synopsys *Design Compiler* (DC) synthesis tools. Defaults and options for ABC, BDS and DC flows are:

- ABC: ABC *resyn2* optimization script and ABC mapper.
- BDS: BDS logic optimization and ABC mapper.
- DC: Synopsys Design Compiler *compile -area effort high*.

2) Results: Table II summarizes experimental results for logic synthesis using BDS-MAJ. The average area of circuits synthesized by BDS-MAJ is 26.4% and 28.8% smaller than BDS and ABC, respectively. The average delay is 20.9% and 12.8% smaller than BDS and ABC, respectively. Considering the commercial DC flow, BDS-MAJ produces logic circuits that have on average 6.0% less area and 7.8% smaller delay.

3) Discussion: BDS-MAJ based logic synthesis exhibits promising results. The advantage enabled by the majority decomposition method leads to faster and smaller circuits compared to state-of-art synthesis tools. Indeed, the majority decomposition is a radix-3 decomposition that naturally leads to more compact circuits compared to traditional radix-2 decomposition structures. The efficient runtime of the decomposition techniques employed in BDS-MAJ highlight the interest of its use in logic synthesis for real-life applications. On a standard workstation (2.2 GHz Intel dual-core processors and 4 GB of RAM), BDS-MAJ took, on average, only 1.4 ms per gate count of the final circuit, to run the optimization procedure.

VI. Conclusions

We presented in this paper the first BDD-based majority logic decomposition technique enabling unprecedented logic synthesis opportunities for both datapath and random control logic. We integrated the proposed majority decomposition method with the state-of-art BDD-based decomposition engine, BDS-PGA, to form a complete

TABLE I: Decomposition Results: BDS-MAJ $vs.$ BDS-PGA

	BDS-MAJ							BDS-PGA						
	Node number						Seconds	Node number						Seconds
Benchmarks	AND	OR	XOR	XNOR	MAJ	Total	Runtime	AND	OR	XOR	XNOR	MAJ	Total	Runtime
MCNC Benchmarks														
alu2	45	99	4	10	13	171	0.9	71	129	7	13	0	220	0.4
C6288	369	378	66	320	139	1272	0.6	711	764	65	355	0	1895	0.6
C1355	14	44	14	80	31	183	0.1	46	26	46	66	0	184	0.3
dalu	126	408	80	21	133	768	1.4	463	895	25	62	0	1445	2.3
apex6	253	289	9	10	16	577	0.4	243	437	7	7	0	694	0.3
vda	65	203	0	0	22	290	0.2	24	392	0	0	0	416	0.3
f51m	18	24	1	10	4	57	0.1	26	41	1	7	0	75	0.1
misex3	337	704	0	1	21	1063	1.0	377	860	2	2	0	1241	0.9
seq	331	1175	0	0	55	1561	6.7	1159	1471	1	2	0	2633	5.6
bigkey	400	1494	64	87	194	2239	2.8	1058	1834	4	31	0	2927	4.0
HDL Benchmarks														
SQRT 32 bit	162	289	60	158	142	811	0.5	254	471	74	132	0	931	0.4
Wallace 16 bit	208	189	178	302	158	1035	0.6	491	785	169	259	0	1704	0.4
CLA 64 bit	179	208	41	53	167	648	0.1	320	481	35	47	0	883	0.2
Rev (1/X) 19 bit	1223	2109	401	1265	599	5597	13.4	2263	4199	383	1121	0	7966	11.2
Div 18 bit	705	1598	255	422	188	3168	7.1	1290	2918	136	308	0	4652	6.4
MAC 16 bit	322	487	177	541	160	1687	0.5	532	891	187	365	0	1975	1.4
4-Op ADD 16 bit	30	32	10	86	52	210	0.1	87	89	9	85	0	270	0.1
Average	281.6	572.5	80.0	198.0	123.0	**1255.1**	**2.1**	553.8	981.3	67.7	168.3	0.0	1771.2	2.0

TABLE II: Logic Synthesis, CMOS 22nm Technology Node

	BDS-MAJ			BDS-PGA			ABC			Design Compiler		
Benchmark	A. (μm^2)	G.C.	D. (ns)	A. (μm^2)	G.C.	D. (ns)	A. (μm^2)	G.C.	D. (ns)	A. (μm^2)	G.C.	D. (ns)
MCNC Benchmarks												
alu2	34.16	238	0.34	40.81	295	0.40	66.50	503	0.41	50.54	373	0.57
C6288	348.78	1422	0.98	360.78	1441	1.11	355.18	1350	1.08	355.11	1453	1.26
C1355	55.23	188	0.30	56.42	200	0.33	60.69	213	0.29	55.44	190	0.31
dalu	111.30	825	0.40	244.09	1731	0.47	171.36	1292	0.44	103.74	743	0.41
apex6	94.85	811	0.25	106.40	813	0.30	100.73	733	0.26	96.04	745	0.31
vda	71.26	567	0.24	114.24	893	0.20	133.56	1035	0.20	70.98	564	0.25
f51m	13.23	78	0.15	13.86	88	0.19	26.18	199	0.17	17.85	135	0.22
misex3	186.90	1440	0.30	236.25	1825	0.28	225.12	1753	0.28	185.01	1424	0.36
seq	266.35	2086	0.33	541.17	4167	0.27	488.32	3678	0.26	304.15	2325	0.30
bigkey	428.29	3512	0.24	528.22	4121	0.30	713.79	5692	0.22	434.49	3526	0.22
HDL Benchmarks												
SQRT 32 bit	205.22	920	3.22	236.81	1029	4.17	226.31	1058	3.66	211.40	990	3.44
Wallace 16 bit	291.89	1455	0.65	385.49	1995	0.88	413.56	2118	0.77	319.41	1541	0.69
CLA 64 bit	145.32	1455	0.65	170.17	1160	1.08	181.44	1126	0.76	161.07	1114	0.67
Rev (1/X) 19 bit	1044.26	5339	3.09	1506.96	7425	4.56	1545.67	8175	4.26	1160.60	5432	3.14
Div 18 bit	702.03	4255	8.54	957.53	6403	10.24	931.35	6302	9.52	734.02	4948	9.22
MAC 16 bit	365.22	1492	0.67	449.33	2150	0.95	491.12	2560	0.72	383.67	1431	0.70
4-Op ADD 16 bit	59.93	171	0.40	65.17	221	0.51	86.18	391	0.50	63.63	201	0.44
Average	**260.25**	**1510.41**	**1.22**	353.75	2115.12	1.54	365.71	2245.76	1.40	276.89	1596.18	1.32

logic decomposition tool, BDS-MAJ. BDS-MAJ produces decomposed circuits having 29.1% less nodes on average compared to BDS-PGA. This advantage traduces to smaller and faster logic circuits when BDS-MAJ is employed in a traditional *optimization-mapping* synthesis flow. Experimental results show that BDS-MAJ produces on average CMOS circuits having 28.8% and 26.4% smaller area and, at the same time, 12.8% and 20.9% smaller delay with respect to academic ABC and BDS synthesis tools. Compared to commercial Synopsys *Design Compiler*, BDS-MAJ produces on average circuits with 6.0% less area and 7.8% smaller delay.

Acknowledgements

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

References

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, 1994.

[2] R.L. Rudell, A. Sangiovanni-Vincentelli, *Multiple-valued minimization for PLA optimization*, IEEE Trans. CAD, Vol. 6, Iss. 5, pp. 727-750, 1987.

[3] R.K. Brayton, *et al.*, *MIS: A Multiple-Level Logic Optimization System*, IEEE Trans. CAD, vol. 6, pp. 1062-1081, Nov.1987.

[4] E. Sentovich, *et al.*, *SIS: A System for Sequential Circuit Synthesis*, ERL, Dept. EECS, Univ. California, Berkeley, UCB/ERL M92/41, 1992.

[5] R.K. Brayton, C. Mc Mullen, *The Decomposition and Factorization of boolean expressions*, Proc. ISCAS 1982.

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

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

[8] R.E. Bryant, *Graph-based algorithms for Boolean function manipulation*, IEEE Trans. Comput., C-35: 677-691, 1986.

[9] V. Bertacco, M. Damiani, *The Disjunctive Decomposition of Logic Functions*, Proc. ICCAD, 1997

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

[11] N. Vemuri, P. Kalla and R. Tessier, *BDD-based Logic Synthesis for LUT-based FPGAs*, ACM Trans. TODAES, Vol.7, pp. 501-525, Oct. 2002.

[12] T. Bengtsson, A. Martinelli, E. Dubrova *A BDD-Based Fast Heuristic Algorithm for Disjoint Decomposition*, Proc. ASP-DAC 2003.

[13] S. Plaza, V. Bertacco, *STACCATO: Disjoint Support Decompositions from BDDs through Symbolic Kernels*, Proc. ASP-DAC 2005.

[14] K. Karplus, *Using if-then-else DAGs for multi-level logic minimization*, Univ. California, Santa Cruz, UCSC-CRL-88-29, 1988.

[15] Y. Tohma, *Decompositions of Logical Functions Using Majority Decision Elements*, IEEE Trans. Electronic Computers, pp. 698-705, 1964.

[16] ABC Logic Synthesis Tool [Online]. Available: http://www.eecs.berkeley. edu/alanmi/abc/

[17] O. Coudert, J.C. Madre, *A unified framework for the formal verification of sequential circuits*, Proc. ICCAD, 1990

[18] O. Coudert, C. Berthet, J.C. Madre, *Verification of sequential machines using boolean functional vectors*, Proc. International Workshop on Applied Formal Methods for Correct VLSI Design, 1989.

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

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

[21] R. Chaudry *et al.*, *Area-oriented synthesis for PTL*, Proc. ICCD 1998.

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