FBDD: A Folded Logic Synthesis System

Dennis Wu and Jianwen Zhu
Electrical and Computer Engineering
10 King's College Road
{wudenni, jzhu}@eecg.toronto.edu

Technical Report TR-07-01-05
July 2005

Abstract

Despite decades of efforts and successes in logic synthesis, algorithm runtime has rarely been taken as a first class objective in research. As design complexity soars and million gate designs become common, as deep submicron effects dominate and frequently invoking logic synthesis within a low-level physical design environment, or a high-level architectural exploration environment become mandatory, it becomes necessary to revisit the fundamental logic synthesis infrastructure and algorithms. In this paper, we demonstrate FBDD, an open sourced, Binary Decision Diagram (BDD) based logic synthesis package, which employs several new techniques, including folded logic transformations and two-variable sharing extraction. Towards the goal of scaling logic synthesis algorithms, we show that for standard benchmarks, and for field programmable gate array (FPGA) technology, FBDD can produce circuits with comparable area with commercial tools, while running one order of magnitude faster.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

FBDD: A Folded Logic Synthesis System

Dennis Wu and Jianwen Zhu
Electrical and Computer Engineering
10 King's College Road
{wudenni, jzhu}@eecg.toronto.edu

***Abstract*— Despite decades of efforts and successes in logic synthesis, algorithm runtime has rarely been taken as a first class objective in research. As design complexity soars and million gate designs become common, as deep submicron effects dominate and frequently invoking logic synthesis within a low-level physical design environment, or a high-level architectural exploration environment become mandatory, it becomes necessary to revisit the fundamental logic synthesis infrastructure and algorithms. In this paper, we demonstrate FBDD, an open sourced, Binary Decision Diagram (BDD) based logic synthesis package, which employs several new techniques, including folded logic transformations and two-variable sharing extraction. Towards the goal of scaling logic synthesis algorithms, we show that for standard benchmarks, and for field programmable gate array (FPGA) technology, FBDD can produce circuits with comparable area with commercial tools, while running one order of magnitude faster.**

I. Introduction

Logic synthesis, the task of optimizing gate level networks, has been the corner stone of modern electronic design automation methodology since the 1990s. As the chip size grows exponentially, and as the logic synthesis task increasingly becomes coupled with physical design, the synthesis runtime has emerged as a new priority, in addition to the traditional metrics of synthesis quality, including area, speed and power. To this end, there is a growing interest in migrating from an algebraic method, exemplified by SIS [1], to a Binary Decision Diagram (BDD) based method, exemplified by BDS [2]. Compared with the former, which uses cube set as the central data structure for design optimization, the latter exploits the compactness and canonicality of BDD so that Boolean decomposition, Boolean matching and don't care minimization can be performed in an efficient way. Despite these advantages, our experiments on publicly available packages show that BDD-based methods are not yet competitive with cube set based methods in terms of area quality. A major reason for this shortcoming is the lack of a sharing extraction strategy. Sharing extraction is the process of extracting common functions among gates in the Boolean network to save area. Their usefulness have long been proven in cube set based systems. One example implementation is kernel extraction, which has been central in producing low area designs in the SIS [1] synthesis package and commercial tools. In contrast, BDD-based systems have provided relatively low support for sharing extraction.

In this paper, we document the key techniques employed in a recently released logic synthesis package, FBDD (acronym for folded binary decision diagram), which achieved competitive synthesis quality, and significantly outperformed publicly available logic synthesis packages in runtime at the time of release. In particular, it achieved one order of magnitude speedup over commercial FPGA logic synthesis tool, while maintaining comparable area measured in lookup table count (LUT) on academic benchmarks.

Our first technique attempts to address synthesis quality: a sharing extraction algorithm that directly exploits the structural properties of BDDs. More specifically, we make the following contributions. First, we demonstrate that by limiting our attention to a specific class of extractors (similar to limiting to kernels in the classic method), namely two-variable disjunctive extractors, effective area reduction can be achieved. Second, we show that an exact, polynomial time algorithm can be developed for the full enumeration of such extractors. Third, we show that just like the case of kernels, there are inherent structures for the set of extractors contained in a logic function, which we can use to make the algorithm incremental and as such, further speed up the algorithm. Our experiments indicate that an overhead of merely 6% is needed to run our sharing extraction algorithm, whereas 25% area reduction can be achieved.

Our second technique attempts to scale synthesis runtime: a new logic transformation strategy, called logic folding, that exploits the regularities commonly found in circuits. Regularities, or the repetition of logic functions, are not only abundant in array-based circuits such as arithmetic units, but can also be found in random logic as well. Using the simple metric of regularity, (# of gates) / (# of gate types), we typically find the regularity of datapath circuits to be on the order of several hundreds. The introduction of the BDD, with its fast equivalence checking properties, has made fast detection of regularity in circuits possible. On the BDD, the equivalence of two functions can be confirmed in constant time. Our logic synthesis system aggressively applies this new capability throughout the entire synthesis flow. With regularity information at hand, logic transformations applied to one logic structure can be easily shared wherever the logic structure is repeated. This dramatically reduces the number of logic transformations required for synthesis, and as a result, improves runtime.

The remainder of the paper is organized as follows. Section II introduces related works. Section III describe our sharing extraction technique. Section IV describes logic folding technique. In Section V, we briefly describe the logic synthesis software package within which the proposed techniques are implemented. In Section VI, experimental results are presented before we draw conclusions in Section VII.

II. Related Works

Two decades of literature makes it impossible to cover all important works. In this section we only focus on those most relevant to our proposed techniques.

It is important to distinguish two sharing extraction strategies. *Active* sharing first enumerates all candidate extractors for each gate, and then evaluates the extractors within the network before the actual extractions are committed. *Passive* sharing first performs decomposition, whose purpose, like sharing extraction, is to break large gates down into smaller ones. It differs in that decompositions are judged by area savings with respect to a single gate, without considering external opportunities for sharing. The passive sharing mechanism can thus only save area only by removing existing, redundant gates in the network after decomposition. Although active sharing is preferred for better area result, BDD-based synthesis systems often perform passive sharing due to the availability of efficient decomposition algorithms.

The most widely used sharing extraction algorithm, the cubeset based kernel extraction by Brayton and McMullen [3], performs active sharing. This algorithm has been improved in different generations of the Berkeley logic synthesis tools, including MIS [4] and SIS [1]. Their latest tool, MVSIS [5], generalizes the previous binary valued logic network optimization framework into a multi-valued network optimization framework. It has since served as a testbed for new logic synthesis algorithms. In particular, it used hybrid data structures, including the traditional cube set, BDDs, and AND-INVERTER graphs, to represent discrete functions under different contexts. The sharing extraction algorithm employed is reportedly a fast implementation of the algebraic method used in SIS [5]. Sawada et al [6] describe a BDD-based equivalent for kernel extraction. While they use BDDs to represent logic functions, they are represented in Zero-Suppressed Decision Diagram (ZDD) form, which implicitly represents cubesets. Essentially, the algorithm is cubeset based and cannot use the advantages of the BDD as described earlier.

Yang and Ciesielski's BDS [2] takes an approach to synthesis that moves away from cubesets altogether. They identify good decompositions by relying heavily on structural properties of BDDs. For example, 1, 0 and X dominators produce algebraic AND, OR and XOR decompositions respectively. They also describe structural methods for non-disjunctive decomposition based on their concept of a generalized dominator. They also perform other non-disjunctive decompositions, such as variable and functional mux decompositions. After performing a complete decomposition of the circuit, they perform sharing extraction by computing BDDs for each node in the Boolean network, in terms of the primary inputs. Nodes with equivalent BDDs can be shared. Vemuri, Kalla and Tessier [7] described an adaptation of BDS to the FPGA technology by performing elimination in the unit of maximum fanout free cones, and variable partitioning techniques for k-LUT feasible decomposition. Mishchenko et al [8] developed a BDD-based synthesis system centered on the Bi-decomposition of functions. They give a theory for when strong or weak bi-decompositions exist and give expressions for deriving their decomposition results. Their sharing extraction step is interleaved with decomposition so that sharing can be found earlier, avoiding redundant computations. They also retain don't care information across network transformations to increase flexibility in matching. For obvious reasons, the passive form of sharing extraction employed in the above systems produces area results inferior to kernel extraction.

The key observation exploited by our sharing extraction algorithm is the cofactor structure of a Boolean function with respect to a set of Boolean variables, commonly referred to as the bound set. This is somewhat related to the line of work on Boolean symmetries [9]–[14]. Recently efficient procedures of Boolean symmetry detection have been devised [15]–[17]. However, Boolean symmetries are defined by the equivalence of cofactor pairs. They have so far been used only for finding support-reducing bi-decomposition, not for sharing extraction [18]. On the other hand, we detect [19], as it becomes apparent later, if the *cardinality* of the set of distinct cofactors equals to two, a stronger condition than Boolean symmetries.

First proposed in [20], our logic folding mechanism is related to regularity extraction explored in datapath synthesis. Odawara [21] pioneered a structural approach where the one dimensional bit slice structure of the design is identified. This effort was extended by Nijssen et al [22],Arikati et al [23], and Kutzschebauch [24] to two dimensions. These efforts can be characterized by a seed growth approach where initially seeds, or slices, each containing a single gate with the same gate type as other slices, are first selected. Adjacent gates are then merged into the slice when their structural signatures, or are found to match the signatures of other gates with respect to the other slices.

Another method formulate regularity extraction as a covering problem. Like technology mapping similar to the one used in technology mapping [25], early efforts [26], [27] assume the availability of a template library. Later effort by Chowdhary et al [28] extracts maximal templates automatically. Since the number of the templates can be exponential, they use heuristics that bound the template count to quadratic complexity. Hassoun and McCreary [29] perform a preprocessing step before the generation of templates, which decomposes the circuit graph into a parse tree of structured subgraphs, called clans.

The majority of the regularity extraction efforts target towards identifying and *preserving* regularity during the physical synthesis phase. Only Kutzschebauch [24] explicitly used the regularity information to improve logic synthesis. While he was able to improve the post synthesis regularity of circuits by on average 57%, the run time, on average, increased by 8%. Compared to the previous methods, all of which can be considered some form of structural matching, our approach exploits the canonical property of BDDs, which allows us to track regularity at high speed throughout the synthesis process.

III. Sharing Extraction

Our sharing extraction algorithm shares similarities with the well known kernel extraction algorithm, used in cube set

based systems. Like kernel extraction, our algorithm performs sharing extraction in a two-step flow. In the first step, the candidate extractors are *enumerated* for each gate in the network. For practicality, not *all* extractors can be enumerated because they are too numerous. In kernel extraction, extractors are limited to those of the algebraic kind, because they can be found efficiently on the cube set. Similarly, we limit our extractors to disjunctive extractors because they can be found efficiently on the BDD. Later the runtime of the algorithm is improved by limiting extractors to two variables, which we shall show, does not adversely affect the area quality.

In the second step, extractors that are shared by multiple gates are *selected* for sharing. Committing some extractors destroys others, so ordering is important in choosing the extractors that have the most impact. One method that works well, is to select the extractors greedily, based on the size of the extractor and the number of times the extractor is repeated. The remainder of this chapter will focus on the extractor enumeration problem only.

We use the following convention for notations. Uppercase letters F,G,H represent functions. Lowercase letters a,b,c represent the variables of those functions. $Supp(F)$ is the support set of F. We call the function produced by taking function F and setting its variable x to the constant 1, the positive cofactor of F with respect to x and is denoted by $F|_x$. Similarly, the function produced by taking function F and setting its variable x to the constant 0 is called the negative cofactor of F with respect to x, and is denoted by $F|_{\overline{x}}$. $[F,C]$ represents an incompletely specified function with F as its completely specified function, and C as its care set. $\Downarrow$ represents the *restrict* binary operator. $F \Downarrow C$ tries to compute a completely specified function of minimum size for incompletely specified function $[F,C]$.

A. Functional Extraction

Given two functions F and E, the extraction process breaks F into two simpler functions, extractor E and remainder R. The extractor is the portion of the function we hope to share.

$$F(X) = R(e,X_R) \quad (1)$$
$$R(e,X_R) = e \cdot R_1(X_R) + \overline{e} \cdot R_2(X_R) \quad (2)$$
$$e = E(X_E) \quad (3)$$

X is the support set of F. X_E is the support set of E. X_R is the support set of R. $X_E \bigcup X_R = X$.

Our meaning for *remainder* is different from the common meaning of remainder computed in division, and used by Brayton and McMullen [3]. For them, remainder refers to the part of the dividend that is left over when the dividend is not evenly divisible by the divisor ($F = D \cdot Q + R$). In our terminology, a remainder refers to the function remaining after extraction, which produces the same signal as F.

Both R_1 and R_2 have multiple solutions. The range of solutions can be characterized by an incompletely specified function $[F,C]$, where F is a completely specified solution and C is the care set. One solution is $R_1 = F$ and $R_2 = F$. The care conditions are obtained by noting that the value of R_1 is only relevant when E is true and R_2 is only relevant when E is false.

$$R_1 = [F,E] \quad (4)$$
$$R_2 = [F,\overline{E}] \quad (5)$$

We want a completely specified solution that minimizes the complexity of R_1 and R_2. To do this, we assign the don't care conditions in a way that minimizes the resulting node count for the BDDs of R_1 and R_2. This problem was found to be NP complete [30] but a solution can be obtained using one of several don't care minimization heuristics. We use the restrict operator to obtain completely specified functions for R_1 and R_2.

$$R_1 = F \Downarrow E \quad (6)$$
$$R_2 = F \Downarrow \overline{E} \quad (7)$$

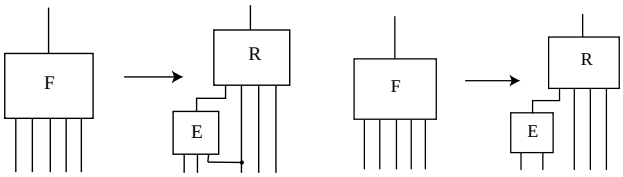
The final equations for the remainder and extractor are shown below:

$$R(e,X_R) = e \cdot (F \Downarrow E) + \overline{e} \cdot (F \Downarrow \overline{E})$$
$$e = E(X_E)$$

B. Disjunctive Extraction

The last section described how to compute the remainder for an arbitrary function and extractor. In this section we describe a specialized extraction algorithm tailored to disjunctive extractors.

An extractor is *disjunctive* if it does not share support with its remainder. In contrast, an extractor is *conjunctive* if it does share support with its remainder. Examples of disjunctive and conjunctive extraction are shown in Figure 1. Disjunctive extractions are ideal for area, because they form a perfect partition of the function, where the remainder and extractor produced have no redundancies between them. However, disjunctive extractors are not always available, in which case, conjunctive extraction is required to break the function down.



(a) Conjunctive Extraction. (b) Disjunctive Extraction.

Fig. 1. Conjunctive vs. Disjunctive Extraction.

It is important to note that by limiting the solution space to disjunctive extractors, some sharing opportunities will be missed. Restricting candidate extractors is necessary however, because the generalized sharing extraction problem is NP

hard. Nevertheless, disjunctive extractors are good candidates because they can be found and matched quickly. We show experimentally that they are effective in reducing area.

1) Extractor Types: In addition to being disjunctive, the extractor considered must satisfy additional properties to avoid repeating redundant computations. The first restriction is that extractors be *prime*, that is, an extractor of size N must not be disjunctively extractable by a function of size less than N. For example, *abcd*, can be extracted by *abc*, however, since *abc* can also be extracted by *ab*, *abc* is not considered a valid extractor. The motivation for this restriction is to reduce processing and memory consumption. Without this restriction a function may have on the order of $O(N^N)$ disjunctive extractors, where N is the number of variables of the function. With the restriction, the number of disjunctive extractors is limited to $O(N^2)$. Prime extractors can be shared recursively to find larger, non-prime, extractors.

Another condition that must be satisfied, is that the extractor must evaluate to 0 when the input vector is $\mathbf{0}$. An extractor and its complement produce the same extraction; computing both is redundant. The condition ensures that only one polarity of the extractor is considered. Forcing extractors to satisfy $E(\mathbf{0}) = 0$, ensures that $\overline{E}$ is not considered, since $\overline{E}(\mathbf{0}) = 1$.

In total, three properties must be satisfied for an extractor of size N to be valid.

Condition 1: The extractor forms a disjunctive extraction.
Condition 2: The extractor cannot be extracted by a function of less than N variables.
Condition 3: The extractor evaluates to '0' when all inputs are false.

2) Enumerating Extractors: The enumeration step identifies all disjunctive extractors for every gate in the Boolean network. Once enumerated, the extractors are matched to find extractors that are shared by multiple gates. The enumeration algorithm works by considering all combinations of variables. For each combination of variables, the algorithm determines whether the set of variables can form a disjunctive extractor. An example of a function and its valid extractors is shown below.

Example 1: The valid extractors of $F = abc + d + e$ are $\Phi(F) = \{ab, bc, ac, d+e\}$.

The disjunctive extractors in Example 2 are more difficult to find by inspection.

Example 2: The valid extractors of $F = \overline{a}b\overline{e} + a\overline{b}\overline{e} + \overline{a}b\overline{f} + a\overline{b}\overline{f} + cef$ are $\Phi(F) = \{a \oplus b, ef\}$.

It turns out, $E = a \oplus b$ is a disjunctive extractor with $R = E\overline{e} + E\overline{f} + cef$ as the corresponding remainder. And $E = ef$ is a disjunctive extractor with $R = \overline{a}b\overline{E} + cE$ as the corresponding remainder. While identifying extractors by inspection may seem difficult, there is a relatively efficient algorithm to find disjunctive extractors on the BDD. The answer may be efficiently computed on the BDD by checking for equivalence between certain cofactors of F.

Theorem 1: Let E be an N variable disjunctive extractor of F. Let $S = \{S_0, \cdots, S_{2^N-1}\}$ be the set of all minterms of E. Then E is a disjunctive extractor of F iff all cofactors of F with respect to the minterms in S map to exactly two functions (R_1 and R_2).

The cofactor condition can be checked fairly quickly. Cofactors with respect to a cube can be determined in $O(|G|)$ time. And cofactors can be compared, on the BDD, in constant time. For fixed N, the overall complexity of determining if a set of variables can be disjunctively extracted is $O(|G|)$.

The algorithm for determining if a set of variables form a valid, disjunctive extractor works as follows: All 2^N cofactors of F with respect to the N variables considered for extraction, are computed. If all cofactors map to exactly two functions, and the extractor satisfies the three conditions for valid extractors, then the set of variables form a valid, disjunctive extractor. Condition 1 is met by construction. To meet Condition 2, extractors of smaller size are enumerated before extractors of larger size. Variables that belong to extractors of smaller size will not be considered when finding extractors of larger size. For example, all two variable extractors are found first. Those variables that belong to a two variable extractor are not considered when enumerating three variable extractors. This ensures no three variable extractor contains a two variable extractor. Finally, functions are inverted if they do not meet the 3rd condition.

Once a set of variables X_E is determined to have a valid, disjunctive extraction, the exact function to be extracted is computed. The cofactors of F with respect to the minterms of X_E, map to exactly two functions, R_1 and R_2. The extractor is computed by setting the on-set of the extractor to be the sum of those minterms c where $F|_c = R_1$. It follows that the off-set is composed of those minterms c where $F|_c = R_2$. The extractor is then complemented, if required, to satisfy the third condition of valid extractors, $E(\mathbf{0}) = 0$. The algorithm to find disjunctive extractors is shown in Figure 1.

The enumeration algorithm is now applied to the difficult function in Example 2. For $X_E = \{a, b\}$, the four cofactors of F with respect to these two variables are listed below:

$$F|_{\overline{a}\overline{b}} = cef \quad (8)$$
$$F|_{\overline{a}b} = \overline{e} + \overline{f} + cef \quad (9)$$
$$F|_{a\overline{b}} = \overline{e} + \overline{f} + cef \quad (10)$$
$$F|_{ab} = cef \quad (11)$$
$$(12)$$

All cofactors map to exactly two functions, so we know from Theorem 1 that $\{a, b\}$ can be extracted disjunctively. The two cofactors are $R_1 = \overline{e} + \overline{f} + cef$ and $R_2 = cef$. Two minterms produce cofactor R_1. They are the cofactors with respect to minterms $\overline{a}b$ and $a\overline{b}$. These two minterms form the on-set of the extractor. Hence the extractor is $E = a \oplus b$. The Remainder is $R = \overline{E}(cef) + E(\overline{e} + \overline{f} + cef)$. Computing the cofactors with respect to variables $\{e, f\}$ determines that ef is also a disjunctive extractor. All other combinations of variables result in a mapping to more than two cofactors of F, and hence those variable combinations do not produce disjunctive extractors.

3) Matching Extractors: Once the extractors are enumerated, the extractors are matched to determine if there is sharing. The candidate extractors are matched by computing an integer signature for the extractors. The signature is a bit

Algorithm 1: Finding Disjunctive Extractors

```
findExtractors( F ) {                                          1
    L = ∅;                                                     2
    forall( variable combinations X_E of F ) {                 3
        R1 = ∅;                                                4
        R2 = ∅;                                                5
        E = '0';                                               6
        isDisjunctive = TRUE;                                  7
        forall( minterms C of X_E ) {                          8
            if( R1 ≠ F|C ∧ R2 ≠ F|C ) {                        9
                if( R1 = ∅ )                                   10
                    R1 = F|C;                                  11
                else if( R2 = ∅ )                              12
                    R2 = F|C;                                  13
                else{                                          14
                    isDisjunctive = FALSE;                     15
                    break;                                     16
                }                                              17
                                                               18
                if( R2 = F|C )                                 19
                    E = Or(E, C);                              20
            }                                                  21
            if( isDisjunctive = FALSE )                        22
                break;                                         23
        }                                                      24
        if( E(0) = 1 )                                         25
            E = Ē;                                             26
                                                               27
        L = L+E;                                               28
    }                                                          29
    return L ;                                                 30
}                                                              31
```

vector, where each bit represents a minterm of the extractor. The bit is a '1' if the minterm belongs to the on-set of the extractor, '0' if it belongs to the off-set. Because extractors are limited to support of size 5, at most 32 bits are required to represent a function. Comparing extractors for equivalence amounts to comparing the signatures (an integer comparison) and the support of the extractors.

The value of the signature depends on the ordering of the support variables. For example, the signature for function $F = abc + d$, with variable order {a,b,c,d} is *1110101010101010*, while the signature for F with variable order {d,c,b,a} is *1111111110000000*. To ensure that the variable orders are consistent, the support is sorted by the address location of the variables before the signature is computed.

Each extractor is rated by a cost function based on the size of the extractor and number of instances in the circuit. At each step, the extractor with the lowest cost is accepted in a greedy fashion. The cost function is computed as $C = N-(N-1)*M$, where N is the support size and M is the number of matches found.

4) Analysis: The sharing extraction algorithm just described presents a novel BDD based method for finding extractors actively; a task that previously was not available for BDD based systems. It differs from traditional sharing extraction techniques in two major ways. First it allows the entire synthesis flow to be BDD based. The compactness and efficiency with which Boolean operation can be performed on the BDD, ultimately translates into faster runtimes than could be achieved with cube sets. Second, this algorithm is able to extract Boolean gates such as XOR gates, which was difficult to do on cube sets.

Although the algorithm described is well suited for BDDs, it can not be as efficiently performed on cube sets. While computing cofactors on the cube sets is also a linear time operation with respect to the size of the logic function, there is difficulty in comparing cofactors for equivalence. Unlike in the BDD, the cube set representation of logic functions are not canonical and comparison of logic functions cannot be performed in constant time. Even if this difficulty could be overcome, the runtime benefits of using BDDs would be lost when performing the sharing extraction technique on cube sets.

While the sharing extraction algorithm described can theoretically handle extractors of arbitrary size, for practical purposes, the size of extractors considered must be restricted. Large extractors present a difficulty in computational complexity because the number of variable combinations that need to be considered grows exponentially with the size of the extractor. There are $O(N^M/M^M)$ variable combinations to consider for extractors of size M and functions of size N. The complexity becomes unmanageable fairly quickly for values of M greater than 5. In the next section we show that the algorithm works well for small values of M and give further improvements to reduce runtime.

C. Disjunctive Two Variable Extraction

Sharing Extraction with large extractors is slow because the number of variable combinations that need to be considered grows exponentially with the size of the extractor. In this section, the extractor size is limited to two variables. In addition to reducing the number of variable combinations considered for extraction, two variables extractors have some added properties that make their extraction incremental and fast. It will be shown experimentally that only a negligible area penalty is experienced when ignoring large extractors.

1) Extractor Types: All two variable functions are considered potential extractors. A two variable function has four unique input values. Each of these input values have two possible output values. That makes $4^2 = 16$ unique, two variable, functions. The one and zero constants and the single variable functions ($F = a$, $F = \overline{a}$, $F = b$ and $F = \overline{b}$) make six trivial functions. These functions cannot produce useful extractions. The ten remaining functions are listed below:

F	=	ab	F	=	$\overline{a}+\overline{b}$
F	=	$a+b$	F	=	$\overline{a}\overline{b}$
F	=	$a \oplus b$	F	=	$a \overline{\oplus} b$
F	=	$a\overline{b}$	F	=	$\overline{a}+b$
F	=	$\overline{a}b$	F	=	$a+\overline{b}$

The right five functions are complements of the left five. Because both polarities of the extractor are used in the remainder, extracting a function in its regular or complemented

TABLE I
COFACTOR CONDITIONS FOR TWO VARIABLE, DISJUNCTIVE EXTRACTION

Condition	Extractor	Remainder
$F\|_{a\overline{b}} = F\|_{\overline{a}b} = F\|_{\overline{a}\overline{b}}$	AND	$R = eF\|_{ab} + \overline{e}F\|_{a\overline{b}}$
$F\|_{ab} = F\|_{a\overline{b}} = F\|_{\overline{a}b}$	OR	$R = eF\|_{ab} + \overline{e}F\|_{\overline{a}\overline{b}}$
$F\|_{ab} = F\|_{\overline{a}b} = F\|_{\overline{a}\overline{b}}$	AND10	$R = eF\|_{a\overline{b}} + \overline{e}F\|_{ab}$
$F\|_{ab} = F\|_{a\overline{b}} = F\|_{\overline{a}\overline{b}}$	AND01	$R = eF\|_{\overline{a}b} + \overline{e}F\|_{ab}$
$F\|_{ab} = F\|_{\overline{a}\overline{b}}$ & $F\|_{a\overline{b}} = F\|_{\overline{a}b}$	XOR	$R = eF\|_{\overline{a}b} + \overline{e}F\|_{ab}$

form will produce exactly the same extraction. Therefore, we only need to consider one half of the ten extractors. In total, there are five, two variable, functions to consider.

2) Computing Extractors: Theorem 1 describes conditions required for the existence of disjunctive extractors of arbitrary size. It follows that the theorem also applies for extractors of size 2. Here we show how extractors are computed for the specific case of two variable, AND extractors. The procedure shown here can be applied to derive extractions for all other two variable extractors.

Theorem 2: $E = ab$ is a disjunctive extractor of F iff $F|_{\overline{a}\overline{b}} = F|_{a\overline{b}} = F|_{\overline{a}b}$.

Similarly, cofactor conditions for the other four, two variable extractors exist as well. These cofactor conditions are listed in Table I.

In order to determine if two variables (a,b) form a disjunctive extractor, four cofactors of F with respect to a and b are computed. The cofactors are computed once and reused in the detection of each of the five extractor types. On the BDD, with it's recursive cofactoring structure, single cube cofactors are computed in $O(|G|)$ time, where G is the number of BDD nodes.

The complete extraction search algorithm is shown in Algorithm 2. The for loop iterates $O(N^2)$ times, where N is the number of variables, and each time performs a $O(|G|)$ cofactor operation. Thus the total worst case complexity for finding the disjunctive, two variable extractors of a function is $O(N^2 \cdot |G|)$.

Algorithm 2: Finding Disjunctive, Two Variable Extractors

```
findExtractors( F ) {
    forall( pairs of variables (a,b) ) {                    32
        A = F|ab;                                           33
        B = F|ab̄;                                           34
        C = F|āb;                                           35
        D = F|āb̄;                                           36
                                                            37
        if( B = C = D )                                     38
            return disjunctive AND (ab) extractor ;         39
        else if( A = B = C )                                40
            return disjunctive OR (a+b) extractor ;         41
        else if( A = B = D )                                42
            return disjunctive ab̄ extractor ;               43
        else if( A = C = D )                                44
            return disjunctive āb extractor ;               45
        else if( A = D and B = C )                          46
            return disjunctive XOR (a⊕b) extractor ;        47
    }                                                       48
}                                                           49
```

D. Incrementally Finding Extractors

In this section we discuss techniques that speed up the extraction algorithm further. The first improvement uses the property that disjunctive, two variable extractors of a function continue to be disjunctive, two variable extractors in their remainders. Instead of rediscovering these extractors, they can be copied over.

Theorem 3: Let E_1 and E_2 be disjunctive, two variable extractors of F. $Supp(E_1) = \{a,b\}$, $Supp(E_2) = \{c,d\}$ and $Supp(E_1) \bigcap Supp(E_2) = \oslash$. If R is the remainder of F extracted by E_1, then E_2 is a disjunctive, two variable extractor of R.

Thus we can obtain some disjunctive, two variable extractors of R by copying them from F. We call these extractors "copy" extractors. Copy extractors do not account for *all* disjunctive, two variable extractors of R. The extractors missed are those formed with substitute variable e. To find these extractors, cofactor conditions between e and every other variables of R must be checked. Extractors found in this way are called "new e" extractors. These two types of extractors, in fact, account for all disjunctive, two variable extractors of R. The benefit is that extractors of R can be obtained through "copy" and "new e" extractors. This is faster than computing the extractors directly.

Theorem 4: Let R be the remainder of F disjunctively extracted by two variable function E_1. E is a disjunctive, two variable extractor of R iff E is a "copy" extractor or "new e" extractor.

The complexity of transferring extractors from F to R is $O(N^2)$. The complexity for finding new extractors involving variable e is $O(N|G|)$. The total complexity for finding extractors for a remainder is $O(N^2 + N|G|)$. The incremental algorithm only applies when finding extractors for *remainders*. When finding extractors for functions whose parent extractors have not been computed, the $O(N^2|G|)$ complexity still applies.

E. Transitive Property of Extractors

The $O(N^2|G|)$ complexity required to find the initial set of extractors can be reduced if we are willing to relax the condition that *all* disjunctive, two variable extractors be found.

Theorem 5: $E_1(a,b)$ and $E_2(b,c)$ are disjunctive, two variable extractors of $F \Rightarrow \exists\ E_3(a,c)$ such that $E_3(a,c)$ is a disjunctive, two variable extractor of F.

Given two disjunctive extractors $E_1(a,b)$ and $E_2(b,c)$, the gate type for extractor $E_3(a,c)$ can be derived from the gate types of E_1 and E_2. Essentially, the polarity of the common variable (b) must be the same. The rules are shown in Table II. Any combinations of gate types not listed in Table II are illegal because both cannot be disjunctive extractors of a function.

The significance of the transitive property is that it is cheaper to discover a disjunctive extractor using the transitive property than it is to compute the extractor directly. Using the transitive property, a fast heuristic can be used to find

Source Gate 1	Source Gate 2	Derived Gate
ab	bc	ac
$a+b$	$b+c$	$a+c$
$a \oplus b$	$b \oplus c$	$a \oplus c$
ab	$b\overline{c}$	$a\overline{c}$
$\overline{a}b$	bc	$\overline{a}c$
$\overline{a}b$	$b\overline{c}$	$a+c$
$a\overline{b}$	$\overline{b}c$	ac
$a+b$	$\overline{b}c$	$\overline{a}c$
$a\overline{b}$	$b+c$	$a\overline{c}$

TABLE II
DERIVING GATE TYPES.

disjunctive extractors. In our previous algorithm, the $O(N^2|G|)$ complexity arose from the need to explicitly find extractors between every pair of variables. Using the transitive property of extractors, we only look for extractors between variables that are adjacent in the BDD order. This reduces the number of pairs we consider from $O(N^2)$ to $O(N)$. The transitive property then, is applied across successively adjacent extractors to find additional extractors. The new algorithm relies on a heuristic: If two variables a and b form a disjunctive extractor, then they are likely to satisfy one of two conditions:

> Condition 1: They are adjacent in the BDD variable order.
> Condition 2: They are separated by variables that form disjunctive extractors with their adjacent variables.

This is not a rule, however, as disjunctive, two variable extractors can be formed that do not satisfy the above conditions. The heuristic works well however, because variables that form disjunctive extractors are likely clustered together in the BDD variable order because it reduces node count. What we have is a trade off between finding all extractors, and finding them quickly. In our experimental results however, the tradeoff in using this heuristic is minimal, degrading area quality by only 0.1%.

IV. FOLDED LOGIC TRANSFORMATIONS

In the design of large circuits, design reuse through hierarchy or repetition of logic structures is often applied to reduce design effort. This theme can be used to speed up synthesis speed. In this section we exploit the inherent regularity in logic circuits to share the transformation results between equivalent logic structures, called logic folding, with the focus of improving runtime.

Since logic transformations typically operate on one or two gates at a time, we use a simpler form of regularity that only considers equivalence between *single* gates, or *pairs* of gates. When matching single gates, we are interested in whether a gate is functionally equivalent to other gates in the network. When dealing with pairs of gates, we are also interested in how the pair is interconnected. Capturing this regularity information enables us to detect instances in the circuit where the same logic transformation is applied more than once. Noting that many circuits exhibit a fair amount of regularity, and noting that many logic transformations depend solely on the logic functions of the gates, we propose to use logic folding to identify regularities in the circuit, to share the logic transformations and improve runtime.

Prior to the BDD, equivalence checking between two Boolean functions was expensive. The cube set representation, faced two hurdles when performing equivalence checking. First, cube sets are not canonical and the cost of putting them in a canonical form is expensive. Second, even if cube sets could be made canonical, they still required the comparison of the cubes in order to confirm equivalence. This is much slower than the constant time requirement for equivalence checking with the BDD. Because logical equivalence between gates could not be determined easily, each gate stored it's own copy the logic function, and no attempt to share the logic function representations was made.

In our circuit representation, we take advantage of the canonical property of BDDs by separating gates from their logic functions. The logic functions are stored in a global function manager where the N variables of a function are mapped to the bottom N generic variables of the function manager. When a new gate is constructed, its logic function is constructed in the global function manager. If the BDD for the logic function finds a match, then the function is shared and the gates are grouped together into an *equivalent class*. Otherwise a new function is added to the global manager.

BDD based matching has its limitations. For example, logic functions can still be equivalent under input permutations. This limitation can be removed by Boolean matching methods reported in [31]–[33]. Our finding is that the further gain of Boolean matching is rather minimal. Another potential limitation is the amount of regularity found is dependent on how the circuit is decomposed. Take the eight input AND gate for example. A balanced decomposition will result in a total of two (folded) decompositions, while a one-sided decomposition requires six decompositions. A similar problem occurs during the elimination stage, where a highly regular circuit may have its regularity collapsed away.

Despite the limitations described above, the potential benefit of logic folding is significant. Gates can be grouped by logical equivalence and logic transformations performed on one gate can be shared among all members of the group. If a match is found early on, there are savings on the immediate logic transformation, as well as on all downstream logic transformations. Folding is essentially free. The cost of folding is to copy BDDs to and from the global BDD manager, but this copying is required anyways when isolating a BDD for variable reordering.

There are also logic transformations however that work on two gates at a time. For example, elimination collapses one gate into its fanout. In this section we describe how sharing transformations can be extend to pairs of gates. Two gate pairs P_1 and P_2, have the same logic transformation result when the gate pairs meet two requirements. First, the Boolean function for each of the two gates in P_1 must match with the corresponding gates in P_2. (i.e. $P_1.gate1.bool = P_2.gate1.bool$ and $P_1.gate2.bool = P_2.gate2.bool$). With the Boolean functions of the gates already matched in the shared function manager (described earlier) this problem is easily solved by using a hash table with the two Boolean functions of gate pair as

the hash key. Secondly, we need to match how the gates of a gate pair are interconnected. In particular, we need to identify which variables are shared, and the positions that the shared variables take in the support sets. This information is called support configuration. When the Boolean functions and support configurations of two gate pairs match, their transformation results will be the same.

A. Support Configurations

Support configuration tells us how variables are shared between the two gates. It does not record information about where the support comes from, but rather what position that shared variable takes in the support sets. Therefore, two gate pairs can have very different support sets but identical support configurations. Before explaining how support configuration is computed, we make a few assumptions that are required of the gates. First, no gate has repeating input variables in its support. And second, no gate has constant values in its support. Both conditions can be met by sweeping the circuit for these instances, and simplifying a gate whenever repeated or constant variables are found in its support.

Let S_1 and S_2 be two support sets. A support configuration is an unordered set of pairs where each pair corresponds to a shared variable. The first element of each pair represents the position of the shared variable in S_1 and the second element of each pair represents the position of the shared variable in S_2. The support configuration can be computed in linear time with respect to the size of the support sets.

Example 3: Let $S_1 = \{a,d,b,c\}$ and $S_2 = \{c,d,e,f,g\}$. The arrangement is shown in Figure 2. Their support configuration is $C(S_1,S_2) = \{(1,1),(3,0)\}$.

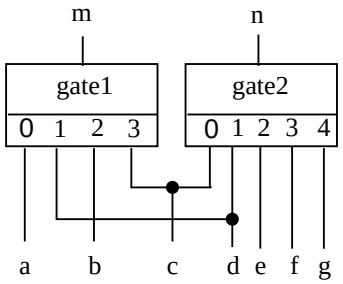


Fig. 2. Support Configuration Example.

The purpose of computing support configurations is to find matching with support configurations in other gate pairs. Support configuration matching is performed very frequently. Whenever a gate pair is created, its support configuration must be compared with all other existing gates pairs for equivalence. A simple way to compare two support configurations is to do a linear traversal of their lists. However, this is a significantly slower than the constant time, pointer comparison done with Boolean matching.

B. Characteristic Function

We present a faster way to compare support configurations by computing a characteristic function. In our formulation of the characteristic function, we use BDDs to represent the elements of a set. An element is represented as a Boolean function of $log_2(N)$ variables, where N is the number of elements in the set. Let $x_0,\cdots,x_{K-1}$, be the K variables of the characteristic functions. Then the elements of the set are assigned as follows:

$$P(0,X) = x_{K-1}\cdots x_1 \cdot x_0 \quad (13)$$
$$P(1,X) = x_{K-1}\cdots x_1 \cdot \overline{x_0} \quad (14)$$
$$P(2,X) = x_{K-1}\cdots \overline{x_1} \cdot x_0 \quad (15)$$
$$P(3,X) = x_{K-1}\cdots \overline{x_1} \cdot \overline{x_0} \quad (16)$$
$$etc\cdots \quad (17)$$

$P(i,X)$ is used to denote the characteristic function for ith element using the variables $X = x_0,\cdots,x_{K-1}$. This representation grows logarithmically with the size of the set, and each element is represented by a single cube. The characteristic functions for the elements are combined to form a support configuration characteristic function.

Let S_1 be a support set of size $|S_1|$. Let S_2 be a support set of size $|S_2|$. Let $C(S_1,S_2) = \{(x_1,y_1),(x_2,y_2),\cdots,(x_K,y_K)\}$ be their support configuration, where K is the number of shared variables. Let X be a set of $log_2(|S_1|)$ variables. Let Y be a set of $log_2(|S_2|)$ variables (independent of X).

Then the support configuration characteristic function is computed as,

$$\begin{aligned} Q &= P(x_1,X)P(y_1,Y) + P(x_2,X)P(y_2,Y) \\ &+ \cdots + P(x_{K-1},X)P(y_{K-1},Y) \end{aligned}$$

The memory requirements for this representation are quite modest; the number of variables of the characteristic function is $log_2(|S_1|) + log_2(|S_2|)$. The major advantage with the characteristic function representation, however, comes from the fact that when stored as a BDD, equivalence between support configurations can be confirmed in constant time.

C. Folded Transformations

A logic expression can be expressed in a number of ways, with some expressions being more compact than others. The goal of simplification is to minimize the complexity of a logic function in an effort to reduce area. In BDD based logic synthesis, one measure of the complexity is the size of its BDD. This size is very sensitive to the variable order chosen and many techniques have been devised to select a variable order that minimizes the node count. BDD based simplification amounts to applying variable reordering on a logic function of a gate, and remapping its support set accordingly. Using the property that two logically equivalent gates have the same result after simplification, *folded simplification* is performed on one logic function and the result applied to all instances of that function.

Likewise, the *folded decomposition* algorithm works as follows. Each decomposition is performed one equivalent class at a time. The BDD for the equivalent class is decomposed into two or more smaller BDDs. If these BDDs are not found in the

global BDD manager, new equivalent classes are created for them. Otherwise, the existing equivalent classes are used. The gates are updated to reflect the changes. If the new equivalent classes can be decomposed further, they are added to the heap, used to order the decompositions in non-increasing size of their support set. Decomposing BDD's in this order ensures that no decompositions are repeated.

Elimination is the process of merging nodes on the Boolean network with the goal of removing inter-gate redundancies. An adjacent pair of gates form an elimination pair $\langle G1, G2, pos\rangle$, which consists of a parent gate *G1*, child gate *G2* and a position *pos*. *pos* is the position of the variable in the parent gate that is to be substituted by the child function.

Two elimination pairs, P_1 and P_2, produce the same elimination result if the logic functions of its gates are the same, the position where they connect is the same, and their support configurations are the same. i.e. $P_1.G_1 = P_2.G_1$, $P_1.G_2 = P_2.G_2$, $P_1.pos = P_2.pos$ and $C(P_1.G_1, P_1.G_2) = C(P_2.G_1, P_2.G_2)$. A hash table is used to identify elimination pairs with the same gate functions, position *pos* and support configuration. When an elimination pair is created, it is matched against the hash table. Eliminations pairs that match are grouped together. Therefore, the elimination result can be computed only once and shared to achieve *folded elimination*.

Folded sharing extraction is more involved: There are two separate computations that can take advantage of regularity. The first computation is the enumeration of extractors. Equivalent functions will produce the same list of disjunctive extractors which can be shared by all instances of the function. This is the cost of computing cofactors between all adjacent variables of the function, to determine if they can be disjunctively extracted. The extractors found are written in terms of generic variables, not in terms of absolute support. This computation is done only once. Then the extractor list is enumerated in terms of absolute support for each instance of the function.

Consider the functions $F = ab + cd$ and $G = lm + ad$. In terms of generic variables, the logic functions are identical, $H = x_0x_1 + x_2x_3$. The extractors are enumerated on the logic function to produce the following extractors (x_0x_1, x_2x_3). At this point, the expensive process of computing the cofactor conditions has been completed. The extractors for F and G are then enumerated by remapping the generic variables to actual support. F has extractors ab, cd and G has extractors lm, ad.

The second computation where regularity can be taken advantage of is the computation of remainders. When an extractor is selected for sharing, it must be extracted from it's parent function to produce a remainder. This requires the expensive process of computing the extractor, and then simplifying the function through variable reordering. For an extraction that breaks F down into remainder R and extractor E, extractions that involve the same F and use the same relative positions of the variables of E in F, produce the same remainders. In the example given earlier, the remainder for F when extracting (a,b) is the same as the remainder for G when extracting (l,m) because the logic functions for F and G are the same, and the relative position of the variables of their extractors are the same (using variables (x_0, x_1)). Thus the remainder $R = e + v_2v_3$ is computed once only, and shared by both instances F and G.

V. Open Sourced FBDD Package

The proposed techniques are implemented in a complete, BDD-based logic synthesis system, called FBDD, that targets combinational circuit optimization. Its complete source code can be downloaded at [34]. FBDD takes as input a gate level description of a circuit in the Berkeley Logic Interchange Format (BLIF) [1]. FBDD then applies a set of algorithms to minimize area while also breaking the circuit down into basic gates in preparation for technology mapping. The output produced, is an area optimized, technology independent circuit in BLIF or structural Verilog format. The BLIF output enables a path from FBDD to academic, standard cell or FPGA technology mappers. The industry standard Verilog output allows for integration with a wider array of tools, including commercial tools.

In addition to the new optimization techniques described in this paper, and a comprehensive set of decomposition algorithms reported in BDS [2], FBDD performs many of the steps present in a typical BDD-based synthesis flow. *Logic minimization* is performed through BDD variable reordering using the sifting heuristic [35]. The sifting heuristic, like bubble sort, swaps adjacent variables in search of the minimum BDD size. The *sweep* stage, further simplifies gates by propagating constant values, and merging support that is repeated more than once within a single gate. In the *elimination* stage, gates of the network are selectively collapsed in an attempt to remove inter-gate redundancies. Finally, the *sharing extraction* and *decomposition* steps are interleaved. Sharing extraction is applied first to find as much disjunctive sharing as possible. Decomposition then breaks down gates where sharing extraction cannot, such as where conjunctive decompositions are required. As decompositions are applied, new good extractors may be created and sharing extraction is re-applied.

FBDD contains over 31000 lines of code, written in the C programming language. Low level BDD storage and manipulation are handled with the CUDD package [36], developed by Fabio Somenzi at the University of Colorado at Boulder. Automated scripts that drive the standard cell and technology mappers, and collect result statistics, are also provided.

VI. Experimental Results

We perform four sets of experiments. In Section VI-A, we describe experiments to justify and quantify the benefits of our sharing extraction algorithm. In Section VI-B, we demonstrate the *asymptotic gains* of logic folding on a set of synthetic benchmarks. In Section VI-C, we demonstrate more *realistic gains* of logic folding on standard benchmarks. In Section VI-D, we perform a comprehensive comparative study on area and runtime against academic and commercial logic synthesis packages.

Two standard benchmark suites are used in the experiments. We use the MCNC benchmarks [37], whic are highly reported

in academic publications, to enable comparison with other works. We use only the combinational, multi-level examples with approximate gate counts of 500 or more for testing. Even so, the circuits obtained from MCNC are relatively small by today's standards. To complement them, we also report results on the ITC99 benchmarks [38], which include a set of large processor cores. For example, it includes subsets of the Viper and 80386 processor cores which offer test cases that are 13 times larger than those found in the MCNC benchmarks.

The area of synthesized circuits is reported as sum of the areas of the gates in the circuit, after technology mapping. For standard cell technology, the SIS Mapper is used to perform technology mapping to the *lib2.genlib* standard cell library from the MCNC benchmark. Another common measure of area is literal count, however, we use the area after technology mapping as our metric because the tools (FBDD, SIS, BDS) target different levels of decomposition, which has an effect on literal count. For FPGA technology, we use the area-oriented *Praetor* mapper in the UCLA RASP package [39], and map the designs to LUTs of size four.

A. Sharing Extraction

To justify our restriction to only extractors of two variables, we analyze the computational effort required to find extractors of various sizes, in Section VI-A.1, and compare that to their area improvement. In Section VI-A.2 we compare exhaustive, all pairs, two variable, extractor enumeration to fast adjacent pairs of variables enumeration. Finally, a comparison of FBDD with sharing extraction versus FBDD without sharing extraction is given in Section VI-A.3.

1) Maximum Extractor Size: The runtime of sharing extraction grows exponentially with the size of the extractors considered. We stated that the runtime could be improved by limiting extractors to two variables without much sacrifice in area. In this section we give empirical evidence to support that claim.

The large examples of the MCNC benchmark were synthesized using varying maximum extractor sizes of two to five. The exact algorithm, which enumerates *all* variable combinations, is used. The runtimes spent on sharing extraction are summed together and shown in Figure 3. From the figure, a steep trade off between maximum extractor size and runtime can be seen. Sharing extraction with extractors of five variables is over nine times slower than with extractors of two variables. With such large runtimes, sharing extraction dominates the overall runtime of synthesis.

To determine the effect that maximum extractor size has on area, we performed logic synthesis with a maximum extractor size of 5, and collected information on the distribution of extractor sizes found. The number of shared extractors found for circuits in the MCNC benchmark are collected and reported in Table III. Each number in the table indicates the number of *prime* extractors found, for a given size. An extractor of size K is *prime* if it cannot be disjunctively extracted by a function of size less than K.

The data shows that two variable extractors clearly make up the majority. On average, extractors of size three through five make up only 0.33% of the shared extractors found, with two variable extractions making up the rest. This is not an entirely obvious result. The number of prime extractors with N variables grows super exponentially with respect to N. As analyzed earlier, there are 5 two variable valid extractors (prime extractors of positive polarity). This number grows to 52 three variable valid extractors and 28620 four variable valid extractors. If circuits were composed of random circuits, the proportion of two variable extractors would be much less. In practice, circuits are typically composed of highly structured logic, such as AND, OR and XOR gates, which can be disjunctively extracted using two variable extractors.

The area results produced using the varying maximum extractor sizes are shown in Figure 4. With relatively few large extractors available for sharing, the effort put into their detection has little effect on area results. Since the computational cost of finding large extractors is high, and the area gain almost non-existent, the runtime of sharing extraction can safely be improved by ignoring large extractors, without significantly

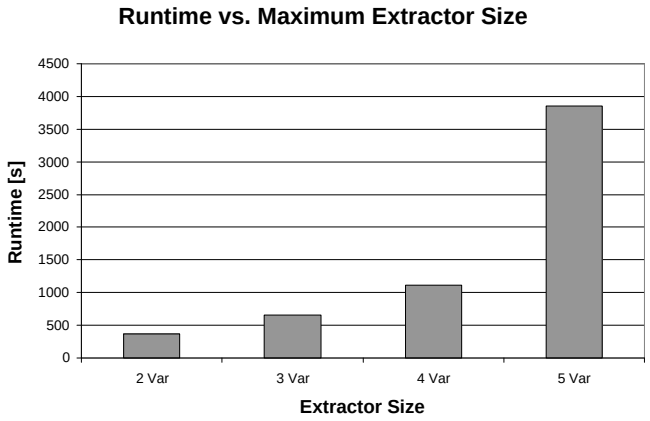


Fig. 3. Runtime vs. Maximum Extractor Size.

TABLE III
DISTRIBUTION OF SHARED EXTRACTOR SIZES.

Circuit	2 Var	3 Var	4 Var	5 Var
C1355	12	0	0	0
C1908	176	0	0	0
C2670	163	9	0	1
C3540	198	0	0	0
C5315	502	2	3	0
C6288	0	0	0	0
C7552	492	2	0	0
alu4	70	7	4	0
dalu	707	10	0	0
des	1697	0	0	0
frg2	619	0	0	0
i10	785	0	0	0
i8	2096	0	0	0
i9	566	0	0	0
k2	1911	0	0	0
pair	116	6	0	0
rot	73	3	0	0
t481	2733	0	0	0
too_large	359	0	0	0
vda	888	0	0	0
x3	234	0	0	0
Total	14397	39	7	1

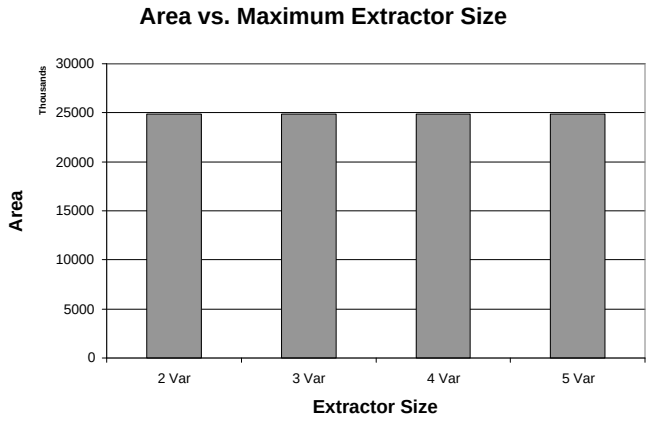
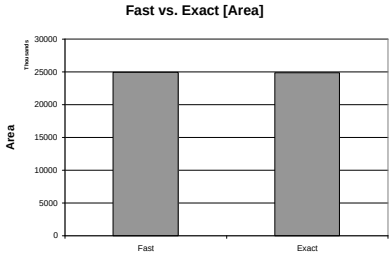
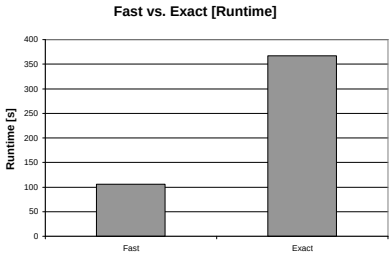


Fig. 4. Area vs. Maximum Extractor Size.



(a) Area



(b) Runtime

Fig. 5. Fast vs. Exact Two Variable Extraction.

affecting area.

2) Fast Two Variable Extraction: Further improvements in runtime are possible when the extractor size is fixed at two. Extractors can be found incrementally and transitively which alleviate the need to process extractors between all pairs of variables. While algorithmically faster, the fast extraction algorithm is inexact and may miss some sharing opportunities, however the loss was found to be minimal. In total, the fast extraction algorithm runs 2.5 times faster than the exact algorithm, while inflating area by merely 0.08%.

3) Sharing Extraction vs. No Sharing Extraction: Finally, to determine the impact that sharing extraction has on area, we obtain area results produced using FBDD both with and without sharing extraction enabled. For the best area and runtime balance, we use the fast, two variable sharing extraction algorithm in these tests. The area results are shown in Figure 6. The benefit experienced from sharing extraction is highly dependent on the circuit type. Circuits *k2* and *vda* experiencing savings of over 100%, while a few circuits do not benefit from sharing extraction at all. Overall however, most circuits do experience benefit from sharing extraction, with the average area savings found to be a substantial 28%.

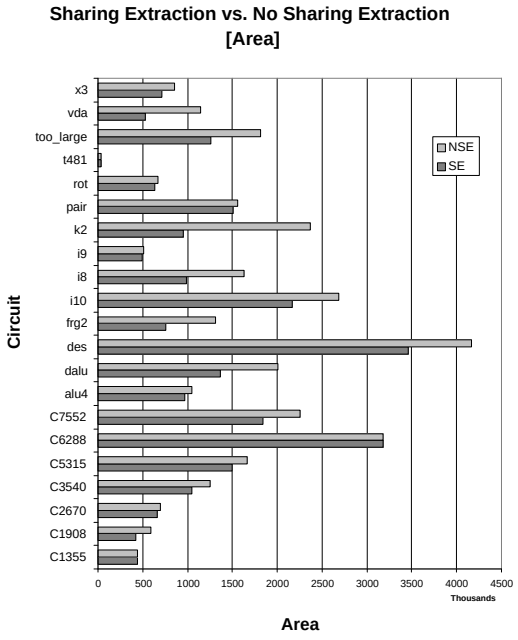


Fig. 6. Sharing Extraction vs. No Sharing Extraction [Area].

As an added benefit, our sharing extraction algorithm also improves the overall runtime of logic synthesis. The runtime results for FBDD with and without sharing extraction is shown in Figure 7. Adding sharing extraction capabilities to logic synthesis has resulted in a runtime improvement of 82%! This is possible because sharing extraction is interleaved with decomposition, which work together in breaking the circuit down into basic gates. Each transformation that is handled with sharing extraction means that one less decomposition is required. Our findings indicate that the computational cost of performing sharing extraction is less than the cost of decomposition. The result, is a synthesis system with both substantially improved area and runtime.

B. Asymptotic Benefits of Logic Folding

Logic folding shares logic transformations to reduce runtime. But the effectiveness of this method depends on the proportion of sharable to non-sharable costs. We report the number of transformations performed, and the number of transformations that could be shared. In order to measure the effectiveness of the method on *individual* transformations, we perform this study on each of the major synthesis stages - decomposition, elimination and sharing extraction.

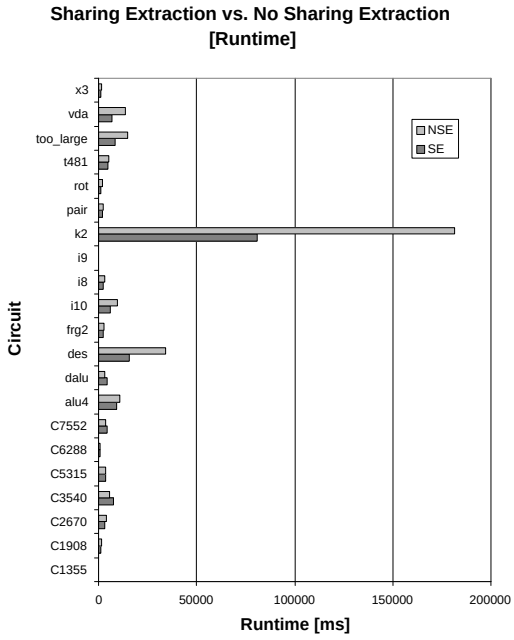


Fig. 7. Sharing Extraction vs. No Sharing Extraction [Runtime].

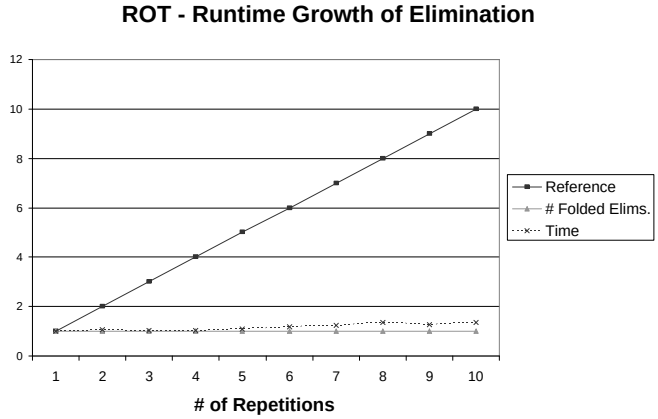


Fig. 8. ROT.blif - Elimination.

In this section we are interested in seeing how folded synthesis performs in the *best case*. To do this we generate benchmark circuits by instantiating varying number of copies of a template circuit. For the template circuit we use rot.blif from the MCNC benchmark. In this way, regularity is increased while the logic content of the circuit remains the same. In total, there are ten benchmark circuits with the number of instances of rot varying from one to ten. We report the runtime growth for the elimination, decomposition and sharing extraction stages against this synthetic benchmark.

The runtime growth of elimination is shown in Figure 8. The "Time" and "# of Folded Elims" plots show normalized values, which emphasize growth instead of absolute value, to enable their comparison. The normalized values are computed as $V(N)_{normalized} = V(N)/V(1)$, where N is the number of instances. A "Reference" line reflects the total time required for elimination if each instance of rot were processed individually. From the graph, it shows that "# of Folded Elims." remains constant for all circuits repetitions, due to the fact that additional instances can share the eliminations computed for the first.

The total runtime of elimination can be broken down into sharable and non-sharable components. In elimination, the sharable components consist of collapsing BDDs with the compose operation, and the simplification of the composed function that follows. The shared components are computed once, and the result shared with all compatible elimination pairs. The non-shared parts consist of computing the characteristic functions for the support configurations, and updating the gate instances in the Boolean network as eliminations are committed. As regularity is increased with each added repetition, the cost of computing the sharable components remain unchanged while the cost of the non-sharable components grows linearly. In practice, the sharable costs may grow as well because the sequence in which transformations take place in each template instance cannot be guaranteed to be the same. But in practice, any increase in the sharable costs are minimal.

The normalized values for the actual time, plotted in Figure 8, closely follows the "# of Folded Elims.". At 10 repetitions, the actual time has grown to a mere 1.36, illustrating that the sharable costs dominate the overall cost of elimination. The cost of the non-sharable component, while not negligible, grows far more slowly than if regularity were not used.

The runtime growth for decomposition, as shown in Figure 9 has characteristics similar to the growth for elimination. Again, the count for the number of folded transformations remains relatively constant for all numbers of repetitions. Although this time, the plot is not perfectly constant, due to differences in the way each instance is synthesized. For decomposition, the sharable portion consists of computing the various BDD decomposition algorithms. The non-sharable portion consists of updating the gates for each instance as decompositions are applied. The non-sharable costs makes up an even smaller fraction of the total cost when compared to elimination where support configurations were computed. As a result, the growth rate for actual time spent on decomposition grows even slower than that of elimination. At 10 repetitions, only an 18% increase in the runtime of decomposition is experienced.

Sharing extraction has two separate, sharable computations. The first sharable computation, called SE1 for reference, is the enumeration of disjunctive extractors. Equivalent functions will produce the same list of disjunctive extractors which can be shared by all instances of the function. This is the cost of computing cofactors between all adjacent variables of the function to determine if they can be disjunctively extracted. It does not include, however, enumerating the extractors in terms of absolute support, which must be performed for each gate individually.

The second sharable component, called SE2 for reference, is the computation of remainders. When an extractor is selected for sharing, it must be extracted from its parent function to produce a remainder, which requires the expensive process

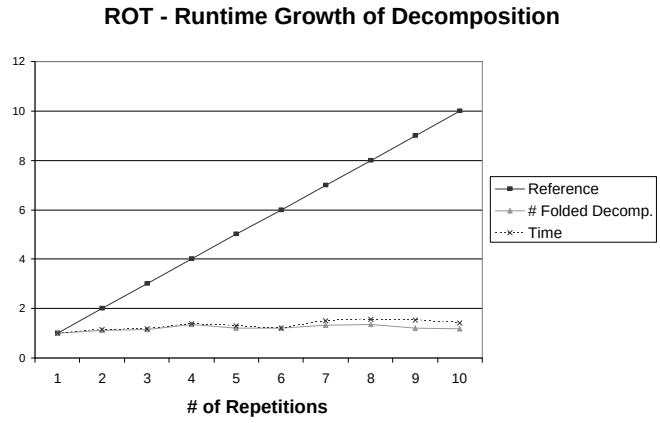


Fig. 9. ROT.blif - Decomposition.

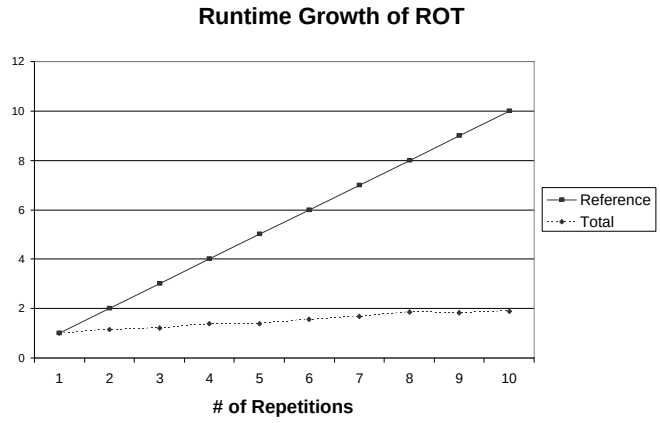


Fig. 11. Runtime Growth of ROT.

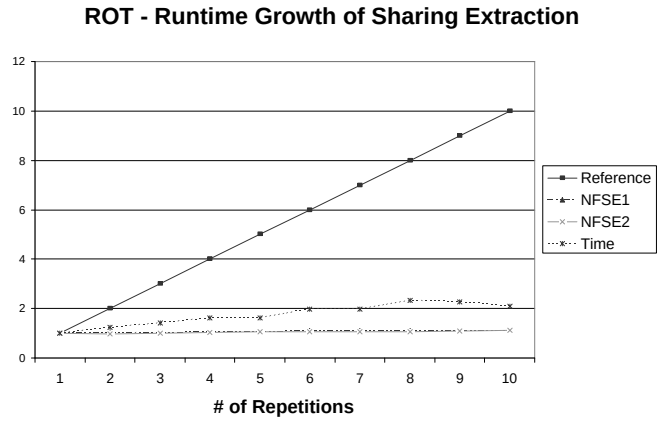


Fig. 10. ROT.blif - Sharing Extraction.

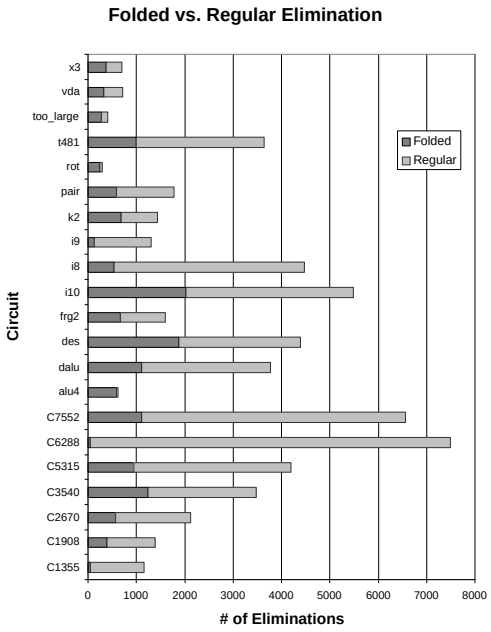


Fig. 12. Folded vs. Regular Elimination.

of simplification through variable reordering. For an extraction that breaks F down into remainder R and extractor E, extractions that involve the same F and use the same relative positions of the variables of E in F, produce the same remainders.

The growth for the number of folded computations for each of SE1 and SE2 are shown in Figure 10. The number of folded computations remains virtually flat for both plots. At 10 repetitions, only 10% more folded SE1 computations and 11% more folded SE2 computations are required. Due to the high non-sharable cost of manipulating large lists of extractors, the actual runtime grows quite noticeably. The run time of the sharing extraction component is doubled when synthesizing 10 instances. However, the overall growth is still far smaller than if each computation were performed individually.

The total, overall runtime, is shown in Fig. 11. It has growth similar with the three major synthesis steps described earlier. 2403 ms were required to synthesize a circuit with 10 repetitions of rot.blif where 12800 ms would have been required if each instance were synthesized individually; a runtime savings of 81%.

C. Realistic Benefits of Logic Folding

Now we look at how folded synthesis performs under a set of comprehensive benchmarks, which are more representative of realistic circuits. We run the benchmarks through FBDD and count the number of folded and regular transformations performed. Dividing the number of regular transformations by the number of folded transformations gives an indication of the runtime improvement that can be expected. It is essentially an upper bound on the achievable speedup.

Figure 12 shows the number of regular versus folded eliminations counted. On average, the folded approach requires 3.85 times less eliminations than the regular approach. Reductions varied between 1.07 times in *alu4* to 136.25 times in circuit *C6288*.

Figure 13 shows the number of regular versus folded decom-

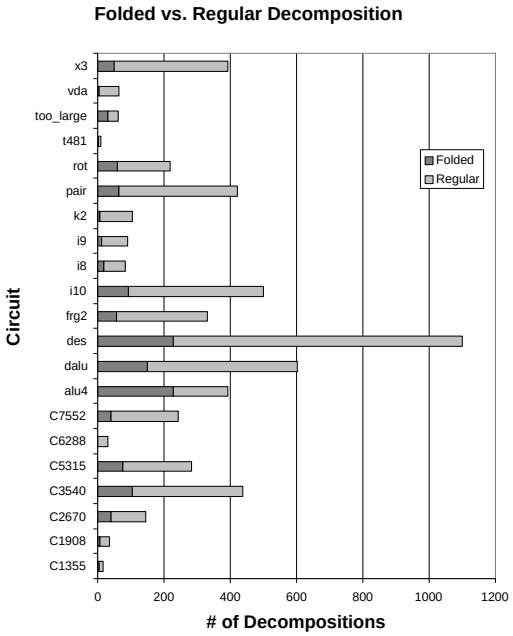


Fig. 13. Folded vs. Regular Decomposition.

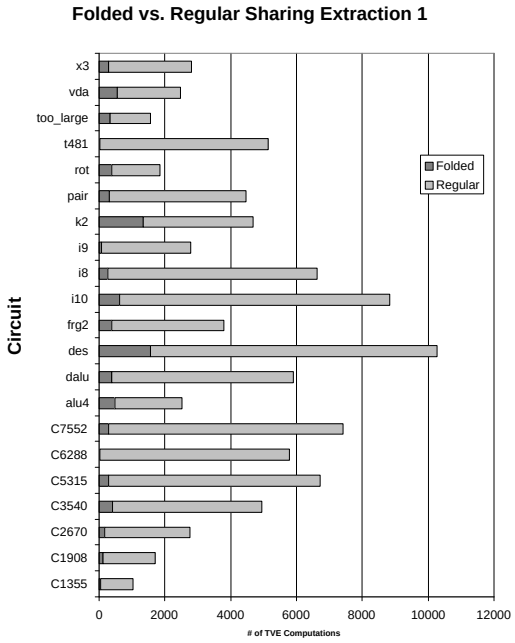


Fig. 14. Folded vs. Regular Extractor Enumeration.

positions counted. On average, the folded approach requires 4.35 times less decompositions than the regular approach. Reductions varied between 1.73 times in circuit *alu4* to 30 times in circuit *C6288*.

The number of regular and folded extractor enumerations are shown in Figure 14. On average, 11 extractor enumerations share one computation. The number of regular and folded remainder computations are shown in Figure 15. On average, 3 remainders share one computation.

D. Comparative Study

In this section, we study the area and runtime performance of FBDD against publicly available logic synthesis packages, including SIS (version 1.2 [40]), BDS (version 1.2 [41]), BDS-PGA (version 2.0 [42]), as well as commercial logic synthesis tools, Xilinx ISE ver. 7.1.01i [43] and Altera Quartus ver. 5.0 sp1 [44][1].

A word of caution is needed in interpretting the presented results correctly and fairly. It is well known that a logic synthesis technique may work well on some benchmarks but not so well on others. For that reason many packages provide command line options, or synthesis scripting capabilities to allow the user to customize their optimization strategies. In this study we use only the standard script, or the default command line options that target area minimization. For ISE, default options are used with the optimization goal is set to area, and effort level set to high. The Spartan3 device (xc3s1500-4-fg676) is selected as the target device. For Altera Quartus,

[1]The latest version of MVSIS at the time of our study, MVSIS 2.0, still hangs on some benchmarks and was not included in the study for fairness. It was suggested [5] that MVSIS produces area comparable to SIS and around 3-5 times faster than SIS.

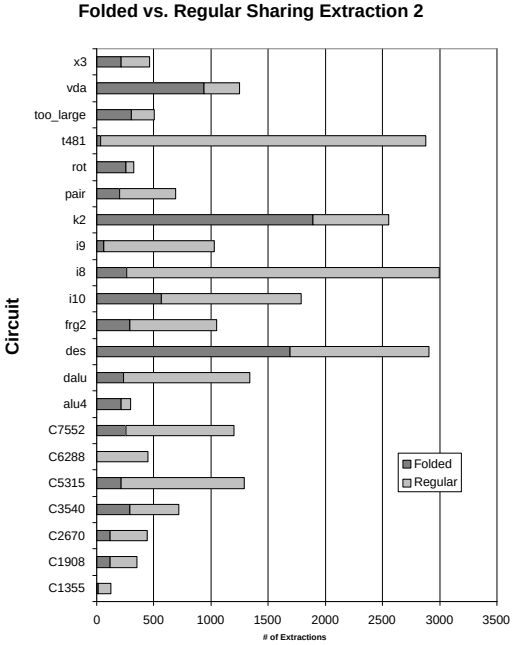


Fig. 15. Folded vs. Regular Remainder Computation.

default options are selected with the optimization goal set to area. The target device is a Cyclone II (EP2C70F896I8). When comparing area and runtime, those benchmarks that fail or do not terminate within *four hours* are discarded from the final statistics.

Table IV gives results on the MCNC benchmark suite for academic tools. For SIS, we used the script.rugged script, which is commonly used for area minimization. FBDD manages to run 59.8X, 1.8X and 3.8X faster than SIS, BDS, BDS-PGA respectively.

Table V gives results on the MCNC benchmark suite for commercial tools. Since technology mapping cannot be separated from logic synthesis in ISE and Quartus, the runtimes reported include time for technology mapping. In FBDD's case, the runtime includes the time used by Praetor. FBDD manages to run 19X and 33X times faster than ISE and Quartus respectively. It also uses 19.4% less LUTs than ISE and 1.7% less LUTs Quartus.

Table VI shows the ITC benchmark results for academic tools. Since BDS and BDS-PGA do not process sequential circuits, their results on this suite cannot be included. Also, since the script.rugged script takes excessively long for the ITC benchmark, we used script.algebraic instead. It is interesting to compare FBDD against SIS on these larger benchmarks: FBDD achieves similar area as SIS for standard cells, and slightly better area on FPGAs. However, the runtime is 31.7 times faster.

Table VII compares ITC benchmark results for FBDD and commercial tools. For the ITC benchmark, FBDD is 50.1X faster than ISE and 4.9X faster than Quartus. FBDD uses 2.8% less LUTs than ISE and 2.4% more LUTs than Quartus.

VII. Conclusions

From our study, we observe that compared to the classic, cubeset based logic synthesis targeting standard cells, FBDD produces inferior area results on the smaller MCNC benchmark suite, but comparable results on the much larger ITC benchmark suite. FBDD however runs orders of magnitude faster. This discrepancy of area performance on MCNC disappeared on FPGAs, suggesting that FBDD might be spending the right amount of effort on sharing extraction. Compared with other BDD-based logic synthesis systems, FBDD has consistently improved on both area and runtime, for both standard cells and FPGAs.

Still at the early stage of research, FBDD suffers from a number of limitations. For example, delay oriented optimization is not yet in place and it is still not clear to what extent the logic folding strategy may affect such optimizations. Nevertheless, we believe FBDD has made a step forward towards the goal of scaling logic synthesis algorithms. Based on our experience, we believe tremendous opportunities exist along this direction. For example, folded transformations manage to cut the runtime of decomposition and sharing extraction by multiple orders of magnitude. The bottleneck currently lies with elimination, which occupies 70% of the runtime. The overall runtime of FBDD can be dramatically improved if this last bottleneck can be scaled down by the same amount as other transformations. It is our hope that a truly scalable synthesis system can open the door for future research on integrating logic synthesis downwards with physical synthesis, and upwards with high level and system level synthesis.

References

[1] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanaha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuits synthesis," Tech. Rep. UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, 1992.

[2] C. Yang, M. Ciesielski, and V. Singhal, "BDS: A BDD-based logic optimization system," in *Proceeding of the 37th Design Automation Conference*, 2000, pp. 92–97.

[3] R. Brayton and C. McMullen, "The decomposition and factorization of boolean expressions," in *ISCAS Proceedings*, 1982, pp. 49–54.

[4] R. K. Brayton, R. L. Rudell, and A. L. Sangiovanni-Vincentelli, "MIS: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 6, pp. 1062–1081, 1987.

[5] D. Chai, J.-H. Jiang, Y. Jiang, A. Mishchenko, and R. Brayton, "MVSIS 2.0 user's manual," Tech. Rep., Department Electrical and Computer Science, University of California, Berkeley, CA 94720, 2004.

[6] H. Sawada, S. Yamashita, and A. Nagoya, "An efficient method for generating kernels on implicit cube set representations," in *International Workshop on Logic Synthesis*, 1999.

[7] N. Vemuri, P. Kalla, and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 4, pp. 501–525, October 2002.

[8] A. Mishchenko, B. Steinbach, and M. Perkowski, "An algorithm for bi-decomposition of logic functions," in *Proceeding of the 38th Design Automation Conference*, 2001, pp. 103–108.

[9] C. R. Edwards and S. L. Hurst, "A digital synthesis procedure under function symmetries and mapping methods," *IEEE Transactions on Computer*, vol. C-27, no. 11, pp. 985–997, 1978.

[10] B.-G. Kim and D. L. Dietmeyer, "Multilevel logic synthesis of symmetric switching functions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 436–446, April 1991.

[11] M. Chrzanowska-Jeska, W. Wang, J. Xia, and M. Jeske, "Disjunctive decomposition of switching functions using symmetry information," in *Proceedings of IEEE SBCCI2000 International Symposium on Integrated Circuits and System Design*, Manaus, Brazil, September 2000, p. 67.

[12] V. N. Kravets and K. A. Sakallah, "Generalized symmetries in boolean functions," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, November 2000, pp. 526–532.

[13] Y.-T. Lai, S. Sastry, and M. Pedram, "Boolean matching using binary decision diagrams with applications to logic synthesis and verification," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, November 1992, pp. 452–458.

[14] F. Mailhot and G. De Micheli, "Technology mapping using boolean matching and don't care sets," in *Proceedings of the European Design Automation Conference*, 1990, pp. 212–216.

[15] D. Moller, J. Mohnke, and M. Weber, "Detection of symmetry of boolean functions represented by robdds," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, November 1993, pp. 680–684.

[16] S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, November 1994, pp. 628–631.

[17] A. Mishchenko, "Fast computation of symmetries in boolean functions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 11, pp. 1588–1593, November 2003.

[18] J. S. Zhang, M. Chrzanowska-Jeske, A. Mishchenko, and J. R. Burch, "Detecting support-reducing bound sets using two-cofactor symmetries," in *Proceedings of the Asia and South Pacific Design Automation Conference*, Shanghai, China, January 2005.

[19] D. Wu and J. Zhu, "BDD-based two-variable sharing extraction," in *Proceedings of the Asia and South Pacific Design Automation Conference*, Shanghai, China, January 2005.

[20] D. Wu and J. Zhu, "Folded logic decomposition," in *International Workshop on Logic and Synthesis*, Laguna Beach, California, June 2003.

TABLE IV

COMPARATIVE RESULTS FOR MCNC BENCHMARK (ACADEMIC).

	SIS script.rugged			BDS 1.2			BDS-pga 2.0			FBDD 1.0		
Circuit	Runtime	ASIC Area	FPGA Area	Runtime	ASIC Area	FPGA Area	Runtime	ASIC Area	FPGA Area	Runtime	ASIC Area	FPGA Area
9symml	3900	158	62	240	345	110	240	294	98	219	57	15
C1355	3000	420	78	650	441	76	620	446	79	190	431	79
C17	0	7	2	0	8	2	0	8	2	15	11	2
C1908	4000	434	111	1300	450	122	1290	468	131	765	415	110
C2670	2900	584	182	-	-	-	-	-	-	1377	591	171
C3540	8300	1062	388	1510	1101	350	2400	1065	343	17176	1123	336
C432	31700	177	67	250	222	77	1490	329	97	300	289	71
C499	3000	420	78	1480	434	81	1450	433	78	146	427	79
C5315	4400	1341	455	-	-	-	-	-	-	2046	1384	456
C6288	13600	2809	509	2800	3066	601	2230	2994	538	480	3106	494
C7552	29900	1794	451	4710	2103	474	5450	2020	511	4350	1918	477
C880	1100	356	119	1070	352	107	1060	392	119	400	371	100
alu2	9800	302	126	340	344	99	360	318	78	3615	494	151
alu4	43700	599	230	2630	957	269	2310	994	297	6708	1036	312
apex6	1100	620	200	970	823	250	530	757	255	771	755	241
apex7	300	205	66	280	271	76	140	244	90	219	270	79
b1	0	7	3	0	8	3	0	10	3	13	6	3
b9	100	116	44	110	113	44	70	116	48	133	118	40
c8	200	104	39	110	162	40	50	107	36	158	131	41
cc	100	55	19	70	64	23	30	62	25	38	57	19
cht	100	142	39	130	165	39	60	147	38	21	149	38
cm138a	100	25	9	10	34	9	0	36	9	18	36	9
cm150a	100	45	13	3300	37	13	3320	37	13	99	38	13
cm151a	0	19	8	0	35	8	0	19	7	38	20	8
cm152a	0	16	6	240	21	7	10	16	6	31	16	6
cm162a	100	38	13	20	42	12	0	45	14	29	39	12
cm163a	0	35	10	10	39	12	10	43	12	24	38	12
cm42a	100	30	10	10	38	10	10	38	10	19	37	10
cm82a	0	17	4	0	23	4	0	20	4	15	19	4
cm85a	100	41	14	20	45	12	10	45	12	52	44	14
cmb	0	46	17	30	51	16	10	54	19	39	60	16
comp	200	105	33	28070	141	30	27970	141	31	156	116	31
cordic	100	51	14	60	75	20	90	79	19	168	66	16
count	100	111	45	100	133	45	50	133	39	36	136	45
cu	100	51	18	40	58	20	10	62	20	55	54	18
dalu	29000	751	277	-	-	-	-	-	-	1769	1240	355
decod	100	44	18	30	52	18	20	44	18	18	44	18
des	38800	2851	1130	-	-	-	6100	4014	1340	7848	3569	1137
example2	700	280	106	320	364	119	150	331	122	181	325	113
f51m	200	88	23	40	86	27	30	99	23	62	107	25
frg1	1000	125	49	280	109	37	230	110	40	208	48	19
frg2	5800	637	250	1870	1065	395	1070	1162	445	1338	804	290
i1	200	46	16	30	50	18	10	50	17	29	47	16
i10	139000	1928	682	4210	2581	787	3110	2495	849	4366	2211	712
i2	1800	170	72	4550	180	71	1510	205	72	4056	171	69
i3	300	106	46	170	106	46	120	103	46	108	106	46
i4	61500	180	70	240	183	71	410	199	74	842	180	70
i5	300	210	66	180	260	73	180	237	75	101	213	67
i6	700	345	113	560	525	145	320	461	144	91	381	107
i7	1000	559	173	1670	683	222	520	680	201	140	473	169
i8	5700	886	338	3130	1295	457	2420	1296	534	2333	1012	367
i9	1700	502	193	2030	729	280	660	682	276	337	621	194
k2	16400	1012	403	-	-	-	-	-	-	9462	1002	382
lal	300	90	30	110	101	35	50	109	42	102	94	32
majority	0	10	2	0	12	3	0	10	3	19	10	2
mux	100	38	13	3000	37	13	3030	37	13	98	37	13
my_adder	300	165	32	4090	212	42	2010	177	32	36	141	32
pair	5400	1376	440	4110	1527	476	3420	1482	478	1390	1581	488
parity	100	44	5	0	40	5	0	40	5	24	44	5
pcle	100	63	20	40	67	20	10	67	20	31	64	20
pcler8	100	87	29	60	90	30	20	90	30	37	87	29
pm1	100	42	18	30	50	18	20	50	21	40	50	20
rot	1900	595	213	10360	738	248	9850	642	225	679	632	222
sct	300	68	17	80	96	29	40	90	32	82	90	23
t	100	7	244	0	10	2	0	7	2	13	7	2
t481	24400	611	244	1130	32	5	-	-	-	15021	34	5
tcon	0	18	8	10	45	8	10	18	8	11	18	8
term1	1400	151	40	360	203	66	200	179	60	1122	278	89
too_large	5397100	264	105	41300	2034	760	174390	1813	695	3840	1684	507
ttt2	500	193	67	180	235	66	100	216	68	253	238	70
unreg	100	73	33	80	103	32	50	99	32	20	97	32
vda	9800	525	226	1660	689	270	590	865	366	1248	549	220
x1	700	266	113	720	355	141	400	335	137	608	384	133
x2	100	42	16	40	46	17	20	57	16	54	67	17
x3	1600	650	220	1020	691	209	670	753	232	553	702	217
x4	800	346	131	630	526	161	300	492	175	318	399	129
z4ml	100	33	9	0	30	6	10	39	8	26	31	6
Total	5911800	28858	9782							98833	33768	10005
BDS Total				138880	28474	8489				76331	25980	7504
BDS-pga Total							263320	31838	10057	69158	29514	8636
Norm	59.8X	85.5%	97.8%	1.8X	109.6%	113.1%	3.8X	107.9%	116.5%	1.0X	100.0%	100.0%

TABLE V
COMPARATIVE RESULTS FOR MCNC BENCHMARK (COMMERCIAL).

	Xilinx ISE		Altera Quartus		FBDD 1.0	
Circuit	Runtime	FPGA Area	Runtime	FPGA Area	Runtime	FPGA Area
9symml	16955	82	7777	9	219	15
C1355	16533	78	5555	74	290	79
C17	12122	2	2222	2	15	2
C1908	21666	129	8888	103	865	110
C2670	27844	234	12222	132	1677	171
C3540	37755	396	25555	323	17676	336
C432	16411	89	6666	73	400	71
C499	14755	78	5555	74	246	79
C5315	49133	477	32222	395	2746	456
C6288	82011	696	38888	507	1680	494
C7552	83000	494	38888	459	5250	477
C880	16322	109	10000	114	600	100
alu2	19411	157	21111	121	3815	151
alu4	30488	292	28888	252	7208	312
apex6	23055	322	23333	222	1071	241
apex7	17800	82	15555	77	319	79
b1	12188	2	7777	2	13	3
b9	14566	47	10000	36	133	40
c8	14200	52	12222	35	258	41
cc	13266	20	10000	17	38	19
cht	13177	46	11111	38	121	38
cm138a	12711	10	11111	9	18	9
cm150a	12900	14	12222	11	99	13
cm151a	12444	7	11111	5	38	8
cm152a	12433	6	12222	6	31	6
cm162a	12966	14	13333	14	29	12
cm163a	12566	11	13333	11	24	12
cm42a	12466	10	14444	10	19	10
cm82a	12288	6	13333	4	15	4
cm85a	12955	11	15555	12	52	14
cmb	13077	15	16666	16	39	16
comp	15655	30	16666	31	156	31
cordic	13711	21	16666	20	168	16
count	13588	43	16666	39	136	45
cu	13055	19	17777	16	55	18
dalu	46211	366	60000	406	2269	355
decod	12744	25	18888	18	18	18
des	81333	1524	112222	1156	9548	1137
example2	16611	115	31111	110	481	113
f51m	13333	25	28888	41	162	25
frg1	16477	43	30000	41	208	19
frg2	28922	339	58888	273	1738	290
i1	12844	16	33333	16	129	16
i10	81855	777	85555	683	5466	712
i2	15955	79	36666	69	4256	69
i3	13388	54	35555	46	208	46
i4	16233	75	37777	94	942	70
i5	14600	75	40000	110	201	67
i6	15577	144	41111	129	291	107
i7	16422	195	50000	167	440	169
i8	31733	371	76666	350	2833	367
i9	18555	212	55555	197	637	194
k2	50222	764	98888	437	9862	382
lal	14777	40	61111	28	202	32
majority	12222	2	57777	2	19	2
mux	12588	16	58888	11	98	13
my_adder	13177	32	60000	32	136	32
pair	34311	508	100000	507	2090	488
parity	12288	5	58888	5	124	5
pcle	13022	20	61111	19	31	20
pcler8	13522	29	62222	30	37	29
pm1	13433	18	65555	14	40	20
rot	24533	237	93333	208	979	222
sct	14633	39	78888	21	182	23
t	12400	2	80000	2	13	2
t481	46011	73	123333	357	15021	5
tcon	12377	8	77777	8	11	8
term1	21477	130	98888	82	1222	89
too_large	508155	277	152222	407	4540	507
ttt2	16855	77	106666	53	353	70
unreg	12955	48	91111	33	120	32
vda	30955	377	126666	253	1448	220
x1	21422	141	117777	135	808	133
x2	13155	17	122222	15	54	17
x3	25155	347	151111	215	853	217
x4	20933	192	150000	116	518	129
z4ml	13033	12	138888	9	26	6
Total	2171911	11947	3761111	10174	114133	10005
Norm	19.0X	119.4%	33.0X	101.69%	1.0X	100.0%

TABLE VI
COMPARATIVE RESULTS FOR ITC BENCHMARK (ACADEMIC).

	SIS script.algebraic			FBDD 1.0		
Circuit	Runtime	ASIC Area	FPGA Area	Runtime	ASIC. Area	FPGA Area
b01blif	0	67744	13	49	50576	13
b02blif	0	41296	4	25	26448	4
b03blif	100	266336	52	108	136416	52
b04blif	800	885312	174	935	619904	162
b05blif	1200	861648	220	1249	806432	222
b06blif	100	82592	10	40	65888	10
b07blif	400	636144	140	312	437552	132
b08blif	200	266800	48	269	219008	56
b09blif	100	264944	48	119	163328	49
b10blif	200	283968	78	243	228288	67
b11blif	800	740544	174	944	762816	169
b12blif	2700	1659728	402	4984	1201760	383
b13blif	400	540096	97	252	353104	88
b14blif	77900	6895504	1931	37014	5959616	1816
b14_1blif	50800	6057520	1626	18294	5189840	1622
b15blif	11553500	10691488	3185	43882	9205760	2901
b15_1blif	261200	10607040	3094	44739	8549664	2779
b17blif	-	-	-	161595	28667776	9222
b17_1blif	-	-	-	122957	26631744	8529
b20blif	502600	13910720	3887	69114	12431024	3695
b20_1blif	351300	12562800	3295	52185	11405120	3298
b21blif	542200	14544544	4159	81417	13083872	3946
b21_1blif	379100	12728448	3364	53043	11097488	3334
b22blif	1739700	21203408	5890	66777	18879232	5694
b22_1blif	1234600	19035136	5029	51099	17011168	4897
Total				811645	173183824	53140
SIS Total	16699900	134833760	36920	527093	117884304	35389
Norm	31.7X	114.4%	104.3%	1.0X	100.0%	100.0%

[21] G. Odawara, T. Hiraide, and O. Nishina, "Partitioning and placement technique for cmos gate arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 3, pp. 355–363, May 1987.

[22] R. Nijssen and J. Jess, "Two-dimensional datapath regularity extraction," in *Proceeding of the International Symposium on Physical Design*, 1996.

[23] S. R. Arikati and R. Varadarajan, "A signature based approach to regularity extraction," in *Proceedings of the International Conference on Computer Design*, San Jose, 1997.

[24] T. Kutzschebauch and L. Stok, "Regularity driven logic synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, 2000, pp. 439–446.

[25] K. Keutzer, "Dagon: Technology binding and local optimization by DAG matching," in *Proceeding of the Design Automation Conference*, June 1987.

[26] D. Rao and F. Kurdahi, "On clustering for maximal regularity extraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 8, pp. 1198–1208, August 1993.

[27] M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, and J. Rabey, "Performance optimization using template mapping for datapath-intensive high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 8, August 1996.

[28] A. Chowdhary, S. Kale, P. Saripella, N. Sehgal, and R. Gupta, "A general approach for regularity extraction in datapath circuits," in *Proceedings of the International Conference on Computer-Aided Design*, 1998, pp. 332–339.

[29] S. Hassoun and C. McCreary, "Regularity extraction via clan-based structural circuit decomposition," in *Proceedings of the International Conference on Computer-Aided Design*, 1999, pp. 414–419.

[30] M. Sauerhoff and I. Wegener, "On the complexity of minimizing the OBDD size of incompletely specified functions," in *IEEE Transactions on Computer Aided Design*, 1996, pp. 1434–1437.

[31] D. Debnath and T. Sasao, "Fast boolean matching under permutation using representative," in *Asia and South Pacific Design Automation Conference, ASP-DAC'992001 IEEE/ACM*, 1999, pp. 359–362.

[32] Jovanka Ciric and Carl Sechen, "Efficient canonical form for boolean matching of complex functions in large libraries," in *2001 IEEE/ACM International Conference on Computer Aided Design*, 2001.

[33] J. Mohnke, P. Molitor, and S. Malik, "Application of bdds in boolean matching techniques for formal logic combinational verification," in *International Journal on Software Tools for Technology Transfer*, 2001, pp. 48–53.

[34] Toronto Synthesis Group, *FBDD Web Site*, `http://www.eecg.toronto.edu/~jzhu/fbdd.html`.

[35] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proceedings of the International Conference on Computer-Aided Design*, 1993, pp. 42–47.

[36] Fabio Somenzi, "Cudd: Cu decision diagram package release 2.3.1," Tech. Rep., Department of Electrical and Computer Engineering, University of Colorado at Boulder, 2001.

[37] Saeyang Yang, "Logic synthesis and optimization benchmarks user guide version 3.0," Tech. Rep., Microelectronics Center of North Carolina, P. O. Box 12889, Research Triangle Park, NC 27709, 1991.

[38] F. Corno, M. Sonza Reorda, and G. Squillero, "RT-level ITC 99 benchmarks and first atpg results," in *IEEE Design & Test of Computers*, 2000, pp. 44–53.

[39] UCLA VLSI CAD LAB, *RASP Website*, `http://ballade.cs.ucla.edu/software_release/rasp/htdocs/`.

[40] Berkeley CAD Group, *SIS Website*, `http://www-cad.eecs.berkeley.edu/software.html`.

[41] University of Massachusetts at Amherst, *BDS Website*, `http://www.ecs.umass.edu/ece/labs/vlsicad/bds/bds.html`.

[42] University of Massachusetts at Amherst, *BDS-PGA Website*, `hhttp://www.ecs.umass.edu/ece/tessier/rcg/bds-pga-2.0/`.

[43] Xilinx, *Xilinx Website*, `http://www.xilinx.com/products/design_resources/design_tool/`.

[44] Altera, *Quartus Website*, `http://www.altera.com/support/`

TABLE VII
COMPARATIVE RESULTS FOR ITC BENCHMARK (COMMERICIAL).

	Xilinx ISE		Altera Quartus		FBDD 1.0	
Circuit	Runtime	FPGA Area	Runtime	FPGA Area	Runtime	FPGA Area
b01blif	12722	12	8888	9	49	13
b02blif	12344	4	8888	4	25	4
b03blif	13766	29	8888	49	208	52
b04blif	21444	189	15555	165	1135	162
b05blif	29133	216	30000	186	1549	222
b06blif	12566	9	7777	9	40	10
b07blif	20933	146	14444	133	512	132
b08blif	14311	52	11111	43	369	56
b09blif	15400	63	8888	46	219	49
b10blif	15466	64	11111	62	343	67
b11blif	27011	205	20000	156	1244	169
b12blif	32755	399	26666	354	5484	383
b13blif	16233	93	10000	84	452	88
b14blif	409100	1912	157777	1830	41114	1816
b14_1blif	229688	1509	115555	1465	22094	1622
b15blif	431733	3274	235555	2995	54182	2901
b15_1blif	586844	3030	221111	2709	51939	2779
b17blif	36963422	9635	925555	9374	211195	9222
b17_1blif	4634011	9330	901111	8297	157157	8529
b20blif	1396422	3803	400000	3677	94414	3695
b20_1blif	860333	3189	266666	3052	70085	3298
b21blif	1463033	3941	413333	3837	101517	3946
b21_1blif	834822	3112	270000	3134	69843	3334
b22blif	2886644	5682	642222	5606	90177	5694
b22_1blif	1698111	4741	465555	4613	74599	4897
Total	52638255	54639	5196666	51889	1049945	53140
Norm	50.1X	102.8%	4.9X	97.6%	1.0X	100%

`software/download/altera_design/quartus_w%e/dnl-quartus_we.jsp`.

APPENDIX

Theorem 1: Let E be an N variable disjunctive extractor of F. Let $S = \{S_0, \cdots, S_{2^N-1}\}$ be the set of all minterms of E. Then E is a disjunctive extractor of F iff all cofactors of F with respect to the minterms in S map to exactly two functions (R_1 and R_2).

CASE: $\Leftarrow$

1.
$$F = S_0 \cdot F|_{S_0} + \cdots + S_{2^N-1} \cdot F|_{S_{2^N-1}} \quad \text{By Shannon's expansion}$$
Let $U = \{U_0, \cdots, U_{J-1}\}$ be the minterms of S such that $F|_{U_i} = R_1$, $0 \le i \le J-1$.
Let $V = \{V_0, \cdots, V_{K-1}\}$ be the minterms of S such that $F|_{V_i} = R_2$, $0 \le i \le K-1$.
And U and V form a partition of S; $U \cap V = \oslash$, $U \cup V = S$
$$F = (U_0 + \cdots + U_{J-1}) \cdot R_1 + (V_0 + \cdots + V_{K-1}) \cdot R_2$$
Since U and V form a partition of S, $U_0 + \cdots + U_{J-1} = (V_0 + \cdots + V_{K-1})'$.
A disjunctive extraction is possible by setting $e = U_0 + \cdots + U_{J-1}$ and $F = e \cdot R_1 + \overline{e} \cdot R_2$.

CASE: $\Rightarrow$

2. E is a disjunctive extractor of $F \Rightarrow F$ can be written as $F(X) = H(X_{\overline{E}}, e), e = E(X_E)$, where X is the support of F, X_E is the support of E, and $X_{\overline{E}} = X - X_E$.
$$\begin{aligned} F &= H(X_{\overline{E}}, e) \\ &= e \cdot H(X_{\overline{E}}, e)|_e + \overline{e} \cdot H(X_{\overline{E}}, e)|_{\overline{e}} \quad \text{By Shannon's expansion} \\ &= E(X_E) \cdot H(X_{\overline{E}}, e)|_e + \overline{E}(X_E) \cdot H(X_{\overline{E}}, e)|_{\overline{e}} \end{aligned}$$
Let $U = \{U_0, \cdots, U_{J-1}\}$ be the minterms that make up the on-set of $E(X_E)$.
Let $V = \{V_0, \cdots, V_{K-1}\}$ be the minterms that make up the off-set of $E(X_E)$.
$U \cup V$, enumerate all the minterms of variables in X_E.
$$F = (U_0 + \cdots + U_{J-1}) \cdot H(X_{\overline{E}}, e)|_e + (V_0 + \cdots + V_{K-1}) \cdot H(X_{\overline{E}}, e)|_{\overline{e}}$$
Enumerating the cofactors of F with respect to the minterms of X_E, we have, $F|_{U_0} = \cdots = F|_{U_{J-1}} = H(X_{\overline{E}}, e)|_e$ and $F|_{V_0} = \cdots = F|_{V_{K-1}} = H(X_{\overline{E}}, e)|_{\overline{e}}$. The cofactors of F with respect to the minterms of E map to exactly two functions.

3. Q.E.D.

Theorem 2: $E = ab$ is a disjunctive extractor of F iff $F|_{\overline{ab}} = F|_{a\overline{b}} = F|_{\overline{a}b}$.

PROVE: If $E = ab$ is a disjunctive extractor of F then $F|_{\overline{ab}} = F|_{a\overline{b}} = F|_{\overline{a}b}$.

1.
$$\begin{aligned} F &= R_1 ab + R_2 \overline{ab} \\ &= R_1 ab + R_2 \overline{a}b + R_2 \overline{a}\overline{b} + R_2 a\overline{b} \end{aligned}$$

2.
$$\begin{aligned} F|_{\overline{ab}} &= R_2 \\ F|_{a\overline{b}} &= R_2 \\ F|_{\overline{a}b} &= R_2 \end{aligned}$$

PROVE: If $F|_{\overline{ab}} = F|_{a\overline{b}} = F|_{\overline{a}b}$ then $E = ab$ is a disjunctive extractor of F.

3.
$$R_1 = [F, ab], \quad \text{from Equation 4.}$$
$F|_{a\overline{b}}$, $F|_{\overline{a}b}$, $F|_{\overline{a}\overline{b}}$ are don't cares. Set them to zero.
$$R_1 \Rightarrow F|_{ab}$$
R_1 does not contain a or b.

4.
$$\begin{aligned} R_2 &= [F, \overline{ab}] \\ &= [F|_{a\overline{b}} a\overline{b} + F|_{\overline{a}b}\overline{a}b + F_{\overline{a}\overline{b}}\overline{a}\overline{b}, \overline{ab}] \end{aligned}$$
Given $F|_{\overline{a}\overline{b}} = F|_{a\overline{b}} = F|_{\overline{a}b}$,
$$\begin{aligned} R_2 &= [F|_{a\overline{b}}(a\overline{b} + \overline{a}b + \overline{a}\overline{b}), \overline{ab}] \\ &= [F|_{a\overline{b}}\overline{ab}, \overline{ab}] \\ &\Rightarrow F|_{a\overline{b}} \end{aligned}$$
R_2 does not contain a or b.

5.
$$\begin{aligned} R &= eR_1 + \overline{e}R_2 \\ &= eF|_{ab} + \overline{e}F|_{a\overline{b}} \end{aligned}$$
The remainder contains neither a or b.

6. Q.E.D.

Theorem 3: Let E_1 and E_2 be disjunctive, two variable extractors of F. $Supp(E_1) = \{a,b\}$, $Supp(E_2) = \{c,d\}$ and $Supp(E_1) \bigcap Supp(E_2) = \oslash$. If R is the remainder of F extracted by E_1, then E_2 is a disjunctive, two variable extractor of R.

NOTE: Here we show this is true for the case where $E_1 = ab$ and $E_2 = a + b$. The same analysis can be applied to show the theorem is true for other combinations of disjunctive, two variable extractors.
CASE: $E_1 = ab$ and $E_2 = a + b$

1. E_1 is a disjunctive, two variable AND extractor $\Rightarrow F|_{a\overline{b}} = F|_{\overline{a}b} = F|_{\overline{a}\overline{b}}$
E_2 is a disjunctive, two variable OR extractor $\Rightarrow F|_{c\overline{d}} = F|_{\overline{c}d} = F|_{cd}$

2.
$$\begin{aligned} R|_{cd} &= (eF|_{ab} + \overline{e}F|_{a\overline{b}})|_{cd} \\ &= eF|_{abcd} + \overline{e}F|_{a\overline{b}cd} \end{aligned}$$

3.
$$\begin{aligned} R|_{c\overline{d}} &= (eF|_{ab} + \overline{e}F|_{a\overline{b}})|_{c\overline{d}} \\ &= eF|_{abc\overline{d}} + \overline{e}F|_{a\overline{b}c\overline{d}}, \qquad \text{Using } F_{c\overline{d}} = F_{cd}, \\ &= eF|_{abcd} + \overline{e}F|_{a\overline{b}cd} \end{aligned}$$

4.
$$\begin{aligned} R|_{\overline{c}d} &= (eF|_{ab} + \overline{e}F|_{a\overline{b}})|_{\overline{c}d} \\ &= eF|_{ab\overline{c}d} + \overline{e}F|_{a\overline{b}\overline{c}d} \qquad \text{Using } F_{\overline{c}d} = F_{cd}, \\ &= eF|_{abcd} + \overline{e}F|_{a\overline{b}cd} \end{aligned}$$

5. $R|_{cd} = R|_{c\overline{d}} = R|_{\overline{c}d} \Rightarrow c + d$ is a disjunctive extractor of R.
6. Q.E.D.

Theorem 4: Let R be the remainder of F disjunctively extracted by two variable function E_1. E is a disjunctive, two variable extractor of R iff E is a "copy" extractor or "new e" extractor.

PROVE: If E is a "copy" extractor or "new e" extractor of R, then E is a disjunctive, two variable extractor of R.

1. Theorem 3 says "copy" extractors are disjunctive, two variable extractors. "new e" extractors are disjunctive, two variable extractors by construction.

PROVE: If E is a disjunctive, two variable extractor of R then E is a "copy" extractor or "new e" extractor.

NOTE: Again, for compactness, we only prove this for the case where $E_1 = ab$ and $E_2 = c + d$.

2. $F|_{\overline{a}\overline{b}} = F|_{a\overline{b}} = F|_{\overline{a}b}$
$R = eF|_{ab} + \overline{e}F|_{a\overline{b}}$
$R|_{cd} = R|_{c\overline{d}} = R|_{\overline{c}d}$

3.
$$\begin{aligned} & R|_{cd} = R|_{c\overline{d}} = R|_{\overline{c}d} \\ & e(F|_{ab})|_{cd} + \overline{e}(F|_{a\overline{b}})|_{cd} \\ & = e(F|_{ab})|_{c\overline{d}} + \overline{e}(F|_{a\overline{b}})|_{c\overline{d}} \\ & = e(F|_{ab})|_{\overline{c}d} + \overline{e}(F|_{a\overline{b}})|_{\overline{c}d} \quad * \end{aligned}$$
$$\begin{aligned} &\Rightarrow (F|_{ab})|_{cd} = (F|_{ab})|_{c\overline{d}} = (F|_{ab})|_{\overline{c}d} \\ & (F|_{a\overline{b}})|_{cd} = (F|_{a\overline{b}})|_{c\overline{d}} = (F|_{a\overline{b}})|_{\overline{c}d} \\ &\Rightarrow (F|_a)|_{cd} = (F|_a)|_{c\overline{d}} = (F|_a)|_{\overline{c}d} \end{aligned}$$

4. From *, we have
$$\begin{aligned} & (F|_{\overline{a}\overline{b}})|_{cd} = (F|_{\overline{a}\overline{b}})|_{c\overline{d}} = (F|_{\overline{a}\overline{b}})|_{\overline{c}d} \\ & (F|_{\overline{a}b})|_{cd} = (F|_{\overline{a}b})|_{c\overline{d}} = (F|_{\overline{a}b})|_{\overline{c}d} \\ &\Rightarrow (F|_{\overline{a}})|_{cd} = (F|_{\overline{a}})|_{c\overline{d}} = (F|_{\overline{a}})|_{\overline{c}d} \end{aligned}$$

5. $F|_{cd} = F|_{c\overline{d}} = F|_{\overline{c}d}$
CASE: Both c and d are elements of F.
Then $c + d$ is a disjunctive, two variable extractor of F so $c + d$ is a "copy" extractor.
CASE: One of c or d is the e variable.
Then $c + d$ is a "new e" extractor.

6. Therefore, an OR extractor must be either a "copy" or "new e" extractor.
7. Q.E.D.

Theorem 5: $E_1(a,b)$ and $E_2(b,c)$ are disjunctive, two variable extractors of $F \Rightarrow \exists\ E_3(a,c)$ such that $E_3(a,c)$ is a disjunctive, two variable extractor of F.

Here we only consider the case where $E_1(a,b) = ab$ (an AND extractor). The analysis presented can be applied to prove the proposition is true for all functions of a and b.
Function $E_2(b,c)$ can be one of 5 extractor functions.
CASE: $E_2 = bc$
E_1 is a disjunctive AND extractor $\Rightarrow F|_{a\overline{b}} = F|_{\overline{a}b} = F|_{\overline{a}\overline{b}}$ (1)
E_2 is a disjunctive AND extractor $\Rightarrow F|_{b\overline{c}} = F|_{\overline{b}c} = F|_{\overline{b}\overline{c}}$ (2)

1.
$$\begin{aligned} F|_{a\overline{c}} &= (F|_{a\overline{c}})|_b b + (F|_{a\overline{c}})|_{\overline{b}}\overline{b} \\ &= F|_{ab\overline{c}}b + F|_{a\overline{b}\overline{c}}\overline{b} \qquad \text{using (1)} \\ &= F|_{\overline{a}\overline{b}\overline{c}}b + F|_{\overline{a}\overline{b}\overline{c}}\overline{b} \\ &= F|_{\overline{a}\overline{b}\overline{c}} \end{aligned}$$

2.
$$\begin{aligned} F_{\overline{a}c} &= (F|_{\overline{a}c})|_b b + (F|_{\overline{a}c})|_{\overline{b}}\overline{b} \\ &= F|_{\overline{a}bc}b + F|_{a\overline{b}c}\overline{b}, \qquad \text{using (1)} \\ &= F|_{\overline{a}\overline{b}c}b + F|_{\overline{a}\overline{b}c}\overline{b} \qquad \text{using (2)} \\ &= F|_{\overline{a}\overline{b}\overline{c}}b + F|_{\overline{a}\overline{b}\overline{c}}\overline{b} \\ &= F|_{\overline{a}\overline{b}\overline{c}} \end{aligned}$$

3. Similarly,
$$\begin{aligned} & F|_{\overline{a}\overline{c}} = F|_{\overline{a}\overline{b}\overline{c}} \\ \Rightarrow\ & F|_{\overline{a}c} = F|_{a\overline{c}} = F|_{\overline{a}\overline{c}} \\ \Rightarrow\ & E_3(a,c) = ac \text{ is a disjunctive AND extractor of } F. \end{aligned}$$

Similarly, if $E_2 = b\overline{c}$, then $F|_{ab} = F|_{a\overline{b}} = F|_{\overline{a}\overline{b}}$ which implies $E_3(a,c) = a\overline{c}$ is a disjunctive extractor of F.
CASE: $E_2 = b + c$
Here we show that $b + c$ cannot be a disjunctive extractor. Assuming that both ab and $b + c$ are disjunctive extractors results in the contradiction that F is independent of a.
E_1 is a disjunctive AND extractor $\Rightarrow F|_{a\overline{b}} = F|_{\overline{a}b} = F|_{\overline{a}\overline{b}}$ (1)
E_2 is a disjunctive OR extractor $\Rightarrow F|_{b\overline{c}} = F|_{\overline{b}c} = F|_{bc}$ (2)

4.
$$\begin{aligned} F|_{\overline{b}c} &= (F|_{\overline{b}c})|_a)a + (F|_{\overline{b}c})|_{\overline{a}})\overline{a} \\ &= F|_{a\overline{b}c}a + F|_{\overline{a}\overline{b}c}\overline{a}, \qquad \text{using (1)} \\ &= F|_{\overline{a}\overline{b}c}a + F|_{\overline{a}\overline{b}c}\overline{a} \\ &= F|_{\overline{a}\overline{b}c} \\ &= (F|_{\overline{b}c})|_{\overline{a}} \end{aligned}$$

5.
$$\begin{aligned} F|_{\overline{b}c} &= (F|_{\overline{b}c})|_{\overline{a}} \\ &\Rightarrow F|_{\overline{b}c} \text{ is not dependent on } a. \end{aligned}$$

6. Since $F|_{b\overline{c}} = F|_{\overline{b}c} = F|_{bc}$, $F|_{b\overline{c}}$ and $F|_{bc}$ also do not depend on a.

7.

$$\begin{aligned} F|_{\overline{b}\overline{c}} &= (F|_{\overline{b}\overline{c}})|_a a + (F|_{\overline{b}\overline{c}})|_{\overline{a}}\overline{a} \\ &= F|_{a\overline{b}\overline{c}} a + F|_{\overline{a}\overline{b}\overline{c}}\overline{a}, \qquad \text{using (1)} \\ &= F|_{\overline{a}\overline{b}\overline{c}} a + F|_{\overline{a}\overline{b}\overline{c}}\overline{a} \\ &= F|_{\overline{a}\overline{b}\overline{c}} \\ &= (F|_{\overline{b}\overline{c}})|_{\overline{a}} \\ &\Rightarrow F|_{\overline{b}\overline{c}} \text{does not depend on } a \end{aligned}$$

8. $F|_{bc}$, $F|_{b\overline{c}}$, $F|_{\overline{b}c}$, $F|_{\overline{b}\overline{c}}$ all do not depend on $a \Rightarrow F$ does not depend on a, a contradiction. Therefore, $b+c$ cannot be a disjunctive extractor of F when ab is a disjunctive extractor of F.

Similarly, $E_2 = \overline{b}c$ and $E_2 = b \oplus c$ cannot be disjunctive extractors of F when $E_1 = ab$ is a disjunctive extractor of F.

We have shown that for the two valid functions of E_2 $\exists\, E_3(a,b)$ that is a disjunctive extractor of F.

9. Q.E.D.