FBDD: A Folded Logic Synthesis System

Dennis Wu, Jianwen Zhu

Department of Electrical and Computer Engineering
University of Toronto, Toronto, Ontario, Canada
{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.

1. 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 2 introduces related works. Section 3 describe our sharing extraction technique. Section 4 describes logic folding technique. In Section 5, we briefly describe the logic synthesis software package within which the proposed techniques are implemented. In Section 6, experimental results are presented before we draw conclusions in Section 7.

2. 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.

3. Sharing Extraction

Our sharing extraction algorithm, like kernel extraction, decomposes sharing extraction into 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 two variable, disjunctive extractors because they can be found efficiently on the BDD.

In the second step, common extractors are *selected* to share. Committing some extractors destroy 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 (frequency). In two variable extraction, all extractors have size of two, so selection is based solely on the frequency of the extractor. Extractors are matched in a hash table using a key based on their two variable support and gate type.

With the selection step described, the remainder of the section will focuses on the process of enumerating extractors. We use the following conventions 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. $F|_x$ is the cofactor of F with respect to x. $[F, C]$ represents an incompletely specified function with F as it's completely specified function and C as it's care set. $\Downarrow$ represents the restrict operation.

3.1 Functional Extraction

Given two functions F and E, the extraction process breaks F into two simpler functions, extractor E and remainder R.

$$
\begin{aligned}
F(X) &= R(e, X_R) && (1)\\
R(e, X_R) &= eR_1(X_R) + \overline{e}R_2(X_R) && (2)\\
e &= E(X_E) && (3)
\end{aligned}
$$

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

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 $R_2 = F$. We obtain the C conditions by noting R_1 is a don't care when E is false and R_2 is a don't care when E is true.

$$
\begin{aligned}
R_1 &= [F, E] && (4)\\
R_2 &= [F, \overline{E}] && (5)
\end{aligned}
$$

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. This problem was found to be NP complete [9] but a solution can be obtained using one of several don't care minimization heuristics. One well known heuristic, which has been shown to be fast, is the restrict operation [10]. Applying the restrict operator, the final equations for the remainder and extractor are shown below:

$$
\begin{aligned}
R(e, X_R) &= e(F \Downarrow E) + \overline{e}(F \Downarrow \overline{E})\\
e &= E(X_E)
\end{aligned}
$$

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	$R = eF\|_{\overline{a}b} + \overline{e}F\|_{ab}$

Table 1: Cofactor conditions for good extraction

3.2 Disjunctive Two Variable 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 good extractors.

DEFINITION 1. *Given function F, extractor E and remainder R, good extractors are two variable extractors whose variables are disjunctive from R. E and R are disjunctive when they do not share support.*

It is important to note, the limitations of good extractors will force us to miss some good sharing opportunities. Not all good sharing opportunities use disjunctive extractors. Nor are all disjunctive extractors the combination of two variable disjunctive extractors. Restricting candidate extractors is necessary however, to keep the runtime reasonable. Nevertheless, good extractors are good candidates because they can be found and matched quickly. We show experimentally that they are effective in reducing area.

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

$$
\begin{array}{rl}
F = ab & F = \overline{a} + \overline{b} \\
F = a + b & F = \overline{a}\overline{b} \\
F = a \oplus b & F = a \overline{\oplus} b \\
F = a\overline{b} & F = \overline{a} + b \\
F = \overline{a}b & F = a + \overline{b}
\end{array}
$$

The right five functions are compliments of the left five. They will produce the same extractors so half can be discarded. In total, there are five functions to consider when looking for good extractors.

The same procedure used for computing extraction for arbitrary functions and extractors (shown earlier) can also be applied to good extractors. However, a faster algorithm is available for the special case of good extractors. Essentially, good extractors require equivalence between certain cofactors of F.

THEOREM 1. *$E = ab$ is a good extractor of F iff $F|_{\overline{a}\overline{b}} = F|_{a\overline{b}} = F|_{\overline{a}b}$.*

The theorem states that a disjunctive, two variable, AND extractor can be detected by comparing three cofactors for equivalence. Cofactor conditions also exist for the four other extractor types, and are listed in Table 1.

The complete extractor search algorithm then proceeds by enumerating every variable pair ($O(N^2)$) of the gate, and for each pair an a $O(G)$ cofactor operation is performed. Thus the total worst case complexity for finding the good extractors of a function is $O(N^2G)$.

3.3 Incrementally Finding Extractors

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

THEOREM 2. *Let E_1 and E_2 be arbitrary good 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 good extractor of R.*

We call these extractors "copy" extractors. Copy extractors do not account for *all* good extractors of R. The good extractors missed are those formed with 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 good extractors of R. The benefit is that good extractors of R can be obtained through "copy" and "new e" extractors. This is faster than computing good extractors directly.

THEOREM 3. *Let R be the remainder of F extracted by E_1. E is a good 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 $O(NG)$. The total complexity for finding extractors for a remainder is $O(N^2 + NG)$. The incremental algorithm only applies when finding extractors for *remainders*. When finding extractors for functions whose parent extractors have not been computed, the $O(N^2G)$ complexity still applies.

3.4 Transitive Property of Good Extractors

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

THEOREM 4. *$E_1(a,b)$ and $E_2(b,c)$ are good extractors of $F \Rightarrow \exists\, E_3(a,c)$ such that $E_3(a,c)$ is a good extractor of F.*

The transitive property of good extractors allows us to reduce the complexity of finding good extractors. In our previous algorithm, the $O(N^2G)$ 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 good extractor, then they are likely to satisfy one of two conditions:

1. They are adjacent in the BDD variable order.

2. They are separated by variables that form good extractors with their adjacent variables.

This is not a rule however, as good extractors can be formed that do not satisfy the above conditions. The heuristic works well however, because variables that form good extractors are likely clustered together in the BDD variable order; 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%.

4. 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.

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.

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.

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

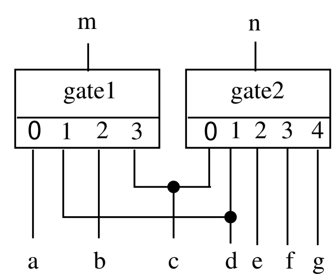

Figure 1: Support Configuration Example.

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.

4.2 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:

$$\begin{aligned} P(0,X) &= x_{K-1} \cdots x_1 \cdot x_0 && (6)\\ P(1,X) &= x_{K-1} \cdots x_1 \cdot \overline{x_0} && (7)\\ P(2,X) &= x_{K-1} \cdots \overline{x_1} \cdot x_0 && (8)\\ P(3,X) &= x_{K-1} \cdots \overline{x_1} \cdot \overline{x_0} && (9)\\ & \quad etc \cdots && (10) \end{aligned}$$

$P(i,X)$ 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.

4.3 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.

5. 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 [11]. 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 [12]. 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

Table 2: Comparative Area and Runtime Statistics.

	FBDD 1.0	SIS 1.2	BDS 1.2	BDS-PGA 2.0	Altera	Xilinx
LUT	100%	97.8%	113.1 %	116.5%	101.7%	119.4%
runtime	1.0X	59.8X	1.8X	3.8X	33.0X	19.0X

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 [13], 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.

6. Experimental Results

We use the MCNC benchmarks [14], 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.

For Field Programmable Gate Array (FPGA) technology, for academic benchmarks, FBDD 1.0 is able to produce comparable area result against commercial tools, while running one order of magnitude faster. The detailed results at the time of release (June, 2005), comparing against publicly accessible academic logic synthesis packages, including SIS version 1.2 [15], BDS version 1.2 [16], BDS-PGA version 2.0 [17], as well as commercial logic synthesis tools, Xilinx ISE ver. 7.1.01i [18] and Altera Quartus ver. 5.0 sp1 [19], are provided in both spreadsheet and chart formats on the FBDD website [11]. To summarize, FBDD 1.0 reports slight area gain for all packages except SIS, and significant speed up for all packages. Note that all packages are run with their **default options**. And benchmarks that fail or do not terminate within four hours in the other packages are discarded for the statistics. The readers are also referred to [20] for more experiments aiming to justify and quantify the individual benefits of sharing extraction and folded transformations.

7. Conclusions

From our study, we observe that compared to the classic, cube-set 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. Technical Report 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*, pages 92–97, 2000.

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

[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*, 6(6):1062–1081, 1987.

[5] D. Chai, J.-H. Jiang, Y. Jiang, A. Mishchenko, and R. Brayton. MVSIS 2.0 user's manual. Technical report, 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*, 7(4):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*, pages 103–108, 2001.

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

[10] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems*, pages 365–373, 1989.

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

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

[13] Fabio Somenzi. Cudd: Cu decision diagram package release 2.3.1. Technical report, Department of Electrical and Computer Engineering, University of Colorado at Boulder, 2001.

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

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

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

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

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

[19] Altera. *Quartus Website*. `http://www.altera.com/support/software/download/altera_design/quartus_w%e/dnl-quartus_we.jsp`.

[20] D. Wu and J. Zhu. FBDD: A folded logic synthesis system. Technical Report TR-07-01-05, Department of Electrical and Computer Engineering, University of Toronto, 2005.