Boolean Factoring and Decomposition of Logic Networks

Alan Mishchenko **Robert Brayton**

Department of EECS
University of California, Berkeley
{alanmi, brayton}@eecs.berkeley.edu

Satrajit Chatterjee

Intel Corporation, Strategic CAD Labs
Hillsboro, OR
satrajit.chatterjee@intel.com

Abstract

This paper presents new methods for restructuring logic networks based on fast Boolean techniques. The basis for these are 1) a cut based view of a logic network, 2) exploiting the uniqueness and speed of disjoint-support decompositions, 3) a new heuristic for speeding these up, 4) extending these to general decompositions, and 5) limiting local transformations to functions with 16 or less inputs so that fast truth table manipulations can be used in all operations. The use of Boolean methods lessens the structural bias of algebraic methods, while still allowing for high speed and multiple iterations. Experimental results on area reduction of K-LUT networks show an average additional reduction of 5.4% in LUT count, while preserving delay, compared to heavily optimized versions of the same networks.

1 Introduction

The traditional way of decomposing and factoring logic networks uses algebraic methods. These represent the logic of each node as a sum of products (SOP) and apply algebraic methods to find factors or divisors. Kerneling or two-cube division is used to derive candidate divisors. These methods can be extremely fast if implemented properly, but they are biased because they rely on an SOP representation of the logic functions, from which only algebraic divisors are extracted. A long-time goal has been to develop similarly fast methods for finding and using good Boolean divisors, independent of any SOP form.

We present a new type of Boolean method, which uses as its underlying computation, a fast method for disjoint support decomposition (DSD). This approach was influenced by the efficient BDD-based computation of complete maximum DSDs proposed in [6], but it has been made faster by using truth-tables and sacrificing completeness for speed. However, this heuristic, in practice, almost always finds the maximum DSD. This fast DSD computation is used as the basis for simple non-disjoint decompositions for finding Boolean divisors.

Methods based on these ideas can be seen as a type of Boolean rewriting of logic networks, analogous to rewriting AIG networks [27]. AIG rewriting has been very successful, partly because it can be applied many times due to its extreme speed. Because of this, many iterations can be used, spreading the area of change and compensating for the locality of AIG-based transforms. Similar effects can be observed with the new methods.

The paper is organized as follows. Section 2 provides the necessary background on DSD as well as a cut-based view of logic networks. Section 3 shows new results on extending DSD methods to non-disjoint decompositions. A particularly interesting set of applications is on *K*-input lookup-table (*K*-LUT) networks. Section 4 looks at reducing the number of LUTs on top of a high-quality LUT-mapping and high-effort resynthesis. Implementation details are discussed, followed by experimental results. Section 5 concludes the paper and discusses future applications and improvements.

2 Background

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. We use the terms *Boolean network*s, *logic networks* and *circuits* interchangeably. We use the term *K-LUT network* to refer to Boolean networks whose nodes are *K*-input lookup tables (*K*-LUTs).

A node *n* has zero or more *fanins,* i.e. nodes that are driving *n*, and zero or more *fanouts*, i.e. nodes driven by *n*. The *primary inputs* (PIs) are nodes of the network without fanins. The *primary outputs* (POs) are a specified subset of nodes of the network.

An *And-Inverter Graph* (AIG) is a Boolean network whose nodes are two-input ANDs. Inverters are indicated by a special attribute on the edges of the network.

A cut of node *n* is a set of nodes, called leaves, such that

1. Each path from any PI to *n* passes through at least one cut node.
2. For each cut node, there is at least one path from a PI to *n* passing through the cut node and not passing through any other cut nodes.

Node *n* is called the root of *C*. A *trivial cut* of node *n* is the cut $\{n\}$ composed of the node itself. A non-trivial cut is said to *cover* all the nodes found on the paths from the leaves to the root, including the root but excluding the leaves. A trivial cut does not cover any nodes. A cut is *K-feasible* if the number of its leaves does not exceed *K*. A cut C_1 is said to be *dominated* if there is another cut C_2 of the same node such that $C_2 \subset C_1$.

A *cover* of an AIG is a subset R of its nodes such that for every $n \in R$, there exists exactly one non-trivial cut $C(n)$ associated with it such that:

1. If *n* is a PO, then $n \in R$.
2. If $n \in R$, then for all $p \in C(n)$ either $p \in R$ or *p* is a PI.
3. If *n* is not a PO, then $n \in R$ implies there exists $p \in R$ such that $n \in C(p)$.

The last requirement ensures that all nodes in *R* are "used".

In this paper, we sometimes use an AIG accompanied with a cover to represent a logic network. This is motivated by our previous work on AIG rewriting and technology mapping. The advantage of viewing a logic network as a cover of an AIG is that different covers of the AIG (and thus different network structures) can be easily enumerated using fast cut enumeration.

The *logic function* of each node $n \in R$ of a cover is simply the Boolean function of *n* computed in terms of $C(n)$, the cut leaves.

This can be extracted easily as a truth table using the underlying AIG between the node and its cut. The truth table computation can be performed efficiently as part of the cut computation. For practical reasons, the cuts in this paper are limited to at most 16 inputs[1].

A completely-specified Boolean function *F essentially depends* on a variable if there exists an input combination such that the value of the function changes when the variable is toggled. The *support* of *F* is the set of all variables on which function *F* essentially depends. The supports of two functions are *disjoint* if they do not contain common variables. A set of functions is *disjoint* if their supports are pair-wise disjoint.

A *decomposition* of a completely specified Boolean function is a Boolean network with one PO that is functionally equivalent to the function. A *disjoint-support decomposition* (DSD - also called simple disjunctive decomposition) is a decomposition in which the set of nodes of the resulting network are disjoint. Because of the disjoint supports of the nodes, the DSD is always a tree (each node has one fanout). The set of leaf variables of any sub-tree of the DSD is called a *bound* set, the remaining variables a *free* set. A *single* disjoint decomposition of a function consists of one block with a bound set as inputs and a single output feeding into another block with the remaining (free) variables as additional inputs. A *maximal DSD* is one in which each node cannot be further decomposed by DSD.

It is known that internal nodes of a maximal DSD network can be of three types: AND, XOR, and PRIME. The AND and XOR nodes may have any number of inputs, while PRIME nodes have support size at least three and only a trivial DSD. For example, a 2:1 MUX is a prime node with three inputs. A DSD is called *simple* if it does not contain prime nodes.

Theorem 2.1 [4]. For a completely specified Boolean function, there is a unique maximal DSD (up to the complementation of inputs and outputs of the nodes).

There are several algorithms for computing the maximal DSD [6][39][23]. Our implementation follows [6] but uses truth tables instead of BDDs to manipulate Boolean functions.

3 General non-disjoint decompositions

A general decomposition has the form

$$F(x) = \hat{H}(g_1(a,b),\cdots,g_k(a,b),b,c)\,.$$

If |*b*|=0 it is called *disjoint* or *disjunctive* and if *k* = 1 it is called *simple*.

Definition: A function *F* has an (*a,b*)-decomposition if it can be written as $F(x) = H(D(a,b),b,c)$ where (a,b,c) is a partition of the variables *x* and *D* is a single output function.

An (*a,b*)-decomposition is a simple, but, in general, a non-disjoint decomposition. For general decompositions, there is an elegant theory [36] on their existence. Kravets and Sakallah [19] applied this to constructive decomposition using support-reducing decompositions along with a pre-computed library of gates. In our case, if |*a*| > 1, an (*a,b*)-decomposition is support reducing. Although less general, the advantage of (*a,b*)-decompositions is that their existence can be tested much more efficiently (as far as we know) by using cofactoring and the fast DSD algorithms of this paper. Recent work use ROBDDs to test the existence of decompositions, with the variables (*a,b*) ordered at the top of the BDD (e.g. see [38] for an easy-to-read description).

[1] The fast methods of this paper are based on bit-level truth table manipulations and 16 is a reasonable limit for achieving speed for this.

The variables *a* are said to be in a separate block and form the *bound* set, the variables *c* are the *free* set, and the variables *b* are the *shared* variables. If |*b*| = 0, the decomposition is a DSD. The function *D*(*a,b*) is called a *divisor* of *F*.

A particular cofactor of *F* with respect to *b* may be independent of the variables *a*; however, we still consider that *a* is in a separate block in this cofactor. We call such cofactors, bound set independent cofactors, or *bsi*-cofactors; otherwise *bsd*-cofactors.

Example: If $F = ab + \bar{b}c$, then $F_{\bar{b}} = c$ is independent of *a* i.e. it is a *bsi*-cofactor.

A BDD version of the following theorem can be found in [38] as Proposition 2 with *t* = 1.

Theorem 3.1: A function *F*(*a,b,c*) has an (*a,b*)-decomposition if and only if *each* of the $2^{|b|}$ cofactors of *F* with respect *b* has a DSD structure in which the variables *a* are in a separate block.

Proof. **If**: Suppose *F* has an (*a,b*)-decomposition; then $F(x) = H(D(a,b),b,c)$. Consider a cofactor $F_{b^j}(x)$ with respect to a minterm of b^j, say $b^j = b_1\bar{b}_2\bar{b}_3b_4$ for *k*=4. This sets *b* = 1,0,0,1, giving rise to the function,

$$F_{b^j}(a,c) = H(D(a,1,0,0,1),1,0,0,1,c) \equiv H_{b^j}(D_{b^j}(a),c)\,.$$

Thus this cofactor has a DSD with *a* separated.

Only if. Suppose all the *b* cofactors have DSDs with variables *a* in separate blocks. Thus $F_{b^j}(a,c) = H_j(D_j(a),c)$ for some functions H_j and D_j. We can take $D_j(a) = 0$ if b^j is bsi[2]. The Shannon expansion gives $F(a,b,c) = \sum_{j=0}^{2^{|b|}-1} b^j H_j(D_j(a),c)$. Define

$D(a,b) = \sum_{j=0}^{2^{|b|}-1} b^j D_j(a)$ and note that

$b^j H_j(D_j(a),c) = b^j H_j(\sum_{m=0}^{2^{|b|}-1} b^m D_m(a),c)$. Thus,

$$F(a,b,c) = \sum_{j=0}^{2^{|b|}-1} b^j H_j(\sum_{m=0}^{2^{|b|}-1} b^m D_m(a),c) = H(D(a,b),b,c)\,. \quad \textbf{QED}.$$

In many applications, such as discussed in Section 4, the shared variables *b* are selected first, and the bound set variables *a* are found using Theorem 3.1 to search for a largest set *a* that can be used.

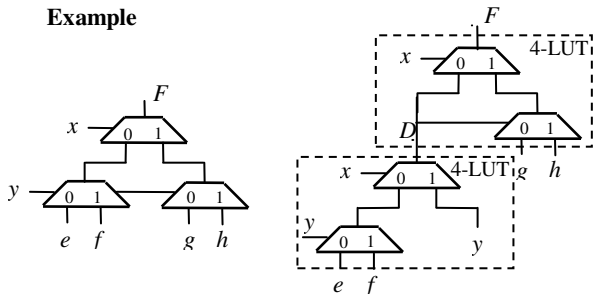

Figure 3.1. Mapping 4:1 MUX into two 4-LUTs.

[2] The choice of taking $D_j(a) = 0$ is arbitrary. We could have equally well taken $D_j(a)$ to be any function of *a*

Consider the problem of decomposing a 4:1 MUX into two 4-LUTs. A structural mapper starting from the logic structure shown in Figure 3.1 on the left would require three 4-LUTs, each containing a 2:1 MUX. To achieve a more compact mapping, we find a decomposition with respect to $(a,b)=((e,f,y),x)$. The free variables are $c=(g,h)$. This leads to cofactors $F_{\overline{x}}=\overline{y}e+yf$ and $F_x=\overline{y}g+yh$. Both $F_{\overline{x}}$ and F_x have (*e,f,y*) as a separate block[3]. Thus, $D_0=\overline{y}e+yf$ and $D_1=y$, while $H_0=D_0$ and $H_1=\overline{D}_1g+D_1h$. Thus we can write $F=\overline{x}H_0+xH_1=\overline{x}(D_0)+x(\overline{D}_1g+D_1h)$. Replacing D_0 and D_1 with $D=\overline{x}(\overline{y}e+yf)+x(y)$, we have $F=\overline{x}D+x(\overline{D}g+Dh)$. This leads to the decomposition shown on the right of Figure 3.1. As a result, a 4:1 MUX is realized by two 4-LUTs.

We will use the notation $f\cong g$ to denote that *f* equals *g* up to complementation. The following theorem characterizes the set of all (*a,b*)-divisors of a function *F*.

Theorem 3.2. Let *F* have an (*a,b*)-decomposition with an associated divisor $D(a,b)=\sum_{j=0}^{2^{|b|}-1}b^jD_j(a)$. Then $\hat{D}(a,b)=\sum_{j=0}^{2^{|b|}-1}b^j\hat{D}_j(a)$ is also an (*a,b*)-divisor if and only if $\hat{D}_j(a)\cong D_j(a),\forall j\in J$, where *J* is the set of indices of the bsd-cofactors of *F*.

Proof. **If:** Suppose $\hat{D}_j(a)\cong D_j(a),\forall j\in J$. If $\hat{D}(a,b)=\sum_{j=0}^{2^{|b|}-1}b^j\hat{D}_j(a)$ where $\hat{D}_j(a),\ j\notin J$ is an arbitrary function, we have to show that $\hat{D}(a,b)$ is a divisor of $F(a,b,c)$. We are given that $D(a,b)=\sum_{j=0}^{2^{|b|}-1}b^jD_j(a)$ is a divisor of $F(a,b,c)$. Thus, there exists *H*, such that $F=H(D(a,b),b,c)=\sum_{j=0}^{2^{|b|}-1}b^jH_{b^j}(D_j(a),c)=\sum_{j\in J_1\cup J_2}b^jH_{b^j}(D_j(a),c)+\sum_{j\notin J}b^jH_{b^j}(D_j(a),c)$ where $J_1=\{j\mid\hat{D}_j(a)=D_j(a)\}$ and $J_2=\{j\mid\hat{D}_j(a)=\overline{D}_j(a)\}$. Clearly $J=J_1\cup J_2$. Now define the operator $H'_k(\bullet,c)=H_k(\overline{\bullet},c)$, i.e. it takes a function and inverts its first argument. Thus, $H_j(D_j(a),c)=H_j(\hat{D}_j(a),c),\ j\in J_1$ and $H_j(D_j(a),c)=H'_j(\hat{D}_j(a),c),\ j\in J_2$. Finally, $H_j(D_j(a),c)=H_j(\hat{D}_j(a),c),\ j\notin J$, since H_j does not depend on the first variable. Thus

$$F=\sum_{j\in J_1}b^jH_{b^j}(\hat{D}_j(a),c)+\sum_{j\in J_2}b^jH'_{b^j}(\hat{D}_j(a),c)+\sum_{j\notin J}b^jH_{b^j}(\hat{D}_j(a),c).$$

Thus $F=\hat{H}(\hat{D}(a,b),b,c)$, where

$$\hat{H}\equiv\sum_{j\in J_1}b^jH_{b^j}(\bullet,c)+\sum_{j\in J_2}b^jH'_{b^j}(\bullet,c)+\sum_{j\notin J}b^jH_{b^j}(\bullet,c).$$

Therefore $\hat{D}(a,b)$ is an (*a,b*)-divisor of *F*.

Only if: Assume that $F=H(D(a,b),b,c)$ and $F=\hat{H}(\hat{D}(a,b),b,c)$. Cofactoring each with respect to $b^j,\ j\in J$, yields $F_{b^j}=H_{b^j}(D_{b^j}(a),c)$ and $F_{b^j}=\hat{H}_{b^j}(\hat{D}_{b^j}(a),c)$. Thus $F_{b^j}(a,c)$ has a DSD with respect to *a*, and by Theorem 2.1, $D_{b^j}(a)\cong\hat{D}_{b^j}(a)$ for $b^j,\ j\in J$. **QED**

Example: $F=ab+\overline{b}c=(ab+\overline{a}\overline{b})b+\overline{b}c$ The bsd-cofactors are $\{b\}$ and the bsi-cofactors are $(\overline{b}\}$. *F* has (*a,b*)-divisors $D^1=ab$ and $D^2=(ab+\overline{a}\overline{b})$, which agree in the bsd-cofactor *b*, i.e. $D^1_b(a)=D^2_b(a)$. In addition, $D^3=\overline{D^1}=\overline{a}+\overline{b}$ is a divisor because $F=\overline{D^3}+\overline{b}c$.

In contrast to the discussion so far, the next result deals with finding common Boolean divisors among a *set* of functions.

Definition: A set of functions, $\{F_1,\cdots,F_n\}$ is said to be (*a,b*)-*compatible* if each has an (*a,b*)-divisor, and $\forall j\in J_1\cap\cdots\cap J_n$, $D^1_{b^j}(a)\cong D^2_{b^j}(a)$, where J_i is the set of *bsd b*-cofactors of F_i

Note that compatibility is not transitive, but if {F_1, F_2, F_3} is pair-wise (*a,b*)-compatible, then the set {F_1, F_2, F_3} is (*a,b*)-compatible and by the next theorem, they all share a common (*a,b*)-divisor.

Theorem 3.3[4]. There exists a common (*a,b*)-divisor of $\{F_1,\cdots,F_n\}$ if and only if the set $\{F_1,\cdots,F_n\}$ is pair-wise (*a,b*)-compatible.

Proof. For simplicity, we show the proof for *n* = 2.

If: Suppose F_1 and F_2 are (*a,b*)-compatible. Then $F_1(a,b,c)=H_1(D^1(a,b),b,c)$ and $F_2(a,b,c)=H_2(D^2(a,b),b,c)$ and $D^1_{b^j}(a)\cong D^2_{b^j}(a)$ for all bsd { b^j } for both F_1 and F_2. Define $\tilde{D}_{b^j}(a)=D^1_{b^j}(a)$ for such b^j. If b^j is bsd for F_1 and bsi for F_2, let $\tilde{D}_{b^j}(a)=D^1_{b^j}(a)$. If b^j is bsd for F_2 and bsi for F_1 let $\tilde{D}_{b^j}(a)=D^2_{b^j}(a)$. Otherwise, let $\tilde{D}_{b^j}(a)=0$. Clearly, by Theorem 3.2, $\tilde{D}(a,b)=\sum_{j=0}^{2^{|b|}-1}b^j\tilde{D}_{b^j}(a)$ is an (*a,b*)-divisor of both F_1 and F_1.

Only if: Suppose a common (*a,b*)-divisor exists, i.e. $F_1(a,b,c)=H_1(\tilde{D}(a,b),b,c)$ and $F_2(a,b,c)=H_2(\tilde{D}(a,b),b,c)$. Then both F_1 and F_2 have (*a,b*)-divisors such that $D^1_{b^j}(a)\cong D^2_{b^j}(a)$ for $j\in J_1\cap J_2$, namely, $D^1=D^2=\tilde{D}$. **QED**

Thus a common divisor of two functions with shared variable *b* can be found by cofactoring with respect to *b,* computing the maximum DSDs of the cofactors, and looking for variables *a* for which the associated cofactors are compatible.

[3] In F_x, the DSD is a trivial one in which each input is a separate block. Since the variables (*a,b*) do not appear in F_x, they can be considered as part of the separate block containing *y*. Similarly, in $F_{\overline{x}}$ the entire cofactor is a separate block.

[4] As far as we know, there is no equivalent theorem in the literature.

4 Rewriting *K*-LUT networks

We consider a new method for rewriting *K*-LUT networks, using the ideas of Section 3, and discuss a particular implementation with experimental results.

4.1 Global view

The objective is to rewrite a local window of a *K*-LUT mapped network. The window consists of a root node, *n*, and a certain number of transitive fanin (TFI) LUTs. The TFI LUTs are associated with a cut *C*. The local network to be rewritten consists of the LUT for *n* plus all LUTs between *C* and *n*. Our objective is to decompose the associated function of *n*, $f_n(C)$, expressed using the cut variables, into a smaller number of LUTs. For convenience, we denote this local network N_n.

An important concept is the *maximum fanout free cone* (MFFC) of N_n. This is defined as the set of LUTs in N_n, which are only used in computing $f_n(C)$. If node *n* were removed, then all of the nodes in MFFC(*n*) could be removed also. We want to re-decompose N_n into fewer *K*-LUTs taking into account that LUTs not in MFFC(*n*) must remain since they are shared with other parts of the network. Since it is unlikely that an improvement will be found when a cut has a small MFFC(*n*), we only consider cuts with no more than *S* shared LUTs. In our implementation *S* is a user-controlled parameter that is set to 3 by default.

Given *n* and a cut *C* for *n*, the problem is to find a decomposition of $f_n(C)$ composed of the minimum number *N* of *K* (or less) input blocks. For those N_n where there is a gain (taking into account the duplication of the LUTs not in the MFFC), we replace N_n with its new decomposition.

The efficacy of this method depends on the following:

- the order in which the nodes *n* are visited,
- the cut selected for rewriting the function at *n*.
- not using decompositions that worsen delay,
- creating a more balanced decomposition,
- pre-processing to detect easy decompositions[5].

We describe the most important aspect of this problem, which is finding a maximum support-reducing decomposition of a function *F*. Other details of the algorithm can be found in [32] and in the source code of ABC [5] implementing command *lutpack*.

The proposed algorithm works by cofactoring the non-decomposable blocks of the DSD of *F* and using Theorem 3.1 to find a Boolean divisor and bound variables *a*. The approach is heuristic and does not guarantee to find a decomposition with the minimum number (*N*) of *K*-input blocks. The heuristic is to extract a maximum support-reducing block at each step based on the idea that support reduction leads to a good decomposition. In fact, any optimum implementation of a network in *K*-LUTs must be support reducing if any fanin of a block is support reducing for that block. However, in general, it may not be maximum support-reducing, but this is the heuristic we use.

4.2 Finding the maximum support-reducing decomposition

The approach is to search for common bound-sets of the cofactor DSDs where cofactoring is tried with respect to subsets of variables in the support of *F*. If all subsets are attempted, the proposed greedy approach reduces the associated block to a minimum number of inputs. However, in our current implementation, we heuristically trade-off the quality of the decomposition found for runtime spent in exploring cofactoring sets. A limit is imposed on (a) the number of cofactoring variables, and (b) the number of different variable combinations tried. Our experiments show that the proposed heuristic approach usually finds a solution with a minimum number of blocks.

The pseudo-code in Figure 4.1 shows how the DSD structures of the cofactors can be exploited to compute a bound set that leads to the maximum support reduction during constructive decomposition.

The procedure **findSupportReducingBoundSet** takes a completely-specified function *F* and the limit *K* on the support size of the decomposed block. It returns a good bound-set, that is, a bound-set leading to the decomposition with a maximal support-reduction. If a support-reducing decomposition does not exist, the procedure returns NONE.

```
varset findSupportReducingBoundSet( function F, int K )
{
    // derive DSD for the function
    DSDtree Tree = performDSD( F );
    // find K-feasible bound-sets of the tree
    varset BSets[0] = findKFeasibleBoundSets( F, Tree, K );
    // check if a good bound-set is already found
    if ( BSets[0] contains bound-set B of size K )
        return B;
    if ( BSets[0] contains bound-set B of size K -1 )
        return B;
    // cofactor F w.r.t. sets of variables and look for the largest
    // support-reducing bound-set shared by all the cofactors
    for ( int V = 1; V ≤ K – 2; V++ ) {
      // find the set including V cofactoring variables
      varset cofvars = findCofactoringVarsForDSD( F, V );
      // derive DSD trees for the cofactors and compute
      // common K-feasible bound-sets for all the trees
      set of varsets BSets[V] = {∅};
      for each cofactor Cof of function F w.r.t. cofvars {
        DSDtree Tree = performDSD( Cof );
        set of varsets BSetsC =
                    computeBoundSets(Cof, Tree, K-V);
        BSets[V] = mergeSets( BSets[V], BSetsC, K-V );
      }
      // check if at least one good bound-set is already found
      if ( BSets[V] contains bound-set B of size K-V )
        return B;
      // before trying to use more shared variables, try to find
      // bound-sets of the same size with fewer shared variable
      for ( int M = 0; M ≤ V; M++ )
        if ( BSets[M] contains bound-set B of size K-V-1 )
          return B;
    }
    return NONE;
}
```

Figure 4.1. Computing a good support-reducing bound-set.

First, the procedure derives the DSD tree of the function itself. The tree is used to compute the set of all feasible bound-sets whose size does not exceed *K*. Bound-sets of larger size are not interesting because they cannot be implemented using *K*-LUTs. For each of the bound-sets found, decomposition with a single output and no shared variables is possible. If a bound-set of size *K* exists, it is returned. If such bound set does not exist (for example, when the function has no DSD), the second best would be to have a bound-set of size *K*-1. Thus, the computation enters a loop, in

[5] e.g. a MUX decomposition of a function with at most 2*K*-1 inputs with cofactors of input size *K*-2 and *K*.

which cofactoring of the function with respect to several variables is tried, and common support-reducing bound-sets of the cofactors are explored.

When the loop is entered, cofactoring with respect to one variable is tried first. If the two cofactors of the function have DSDs with a common bound-set of size *K*-1, it is returned. In this case, although the decomposed block has *K* variables, the support of *F* is only reduced by *K*-2 because the cofactoring variable is shared and the output of the block is a new input. If there is no common bound-set of size *K*-1, the next best outcome is one of the following:

1. There is a bound-set of size *K*-2 of the original function.
2. There is a common bound-set of size *K*-2 of the two cofactors with respect to the cofactoring variable.
3. There is a common bound-set of size *K*-2 of the four cofactors with respect to two variables.

The loop over *M* at the bottom of Figure 4.1 tests for outcomes (1) and (2). If these are impossible, *V* is incremented and the next iteration of the loop is performed, which is the test for the outcome (3).

In the next iteration over *V*, cofactoring with respect to two variables is attempted and the four resulting cofactors are searched for common bound-sets. The process is continued until a bound-set is found, or the cofactoring with respect to *K*-2 variables is tried without success. When *V* exceeds *K*-2 (say, *V* is *K*-1), the decomposition is not support-reducing, because the composition function depends on shared *K*-1 variables plus the output of the decomposed block. In other words, the decomposition takes away *K* variables from the composition function and returns *K* variables. In this case, NONE is returned, indicating that there is no support-reducing decomposition.

Example. Consider the decomposition of function *F* of the 4:1 MUX shown in Figure 3.1 (left). Assume *K* = 4. This function does not have a non-trivial DSD, that is, its DSD is composed of one prime block. The set of *K*-feasible bound-sets is trivial in this case: { {∅}, {*a*}, {*b*}, {*c*}, {*d*}, {*x*}, {*y*} }. Clearly, none of these bound-sets has size *K* or *K*-1. The above procedure enters the loop with *V* = 1. Suppose *x* is chosen as the cofactoring variable. The cofactors are $F_{\bar{x}} = \bar{y}a + yb$ and $F_x = \bar{y}c + yd$. The *K*-1-feasible bound-sets are {{∅}, {*a*}, {*b*}, {*y*}, {*a, b, y*}}, and {{∅}, {*c*}, {*d*}, {*y*}, {*c, d, y*}}. A common bound-set {*a, b, y*} of size *K*-1 exists. The loop terminates and this bound-set is returned, resulting in the decomposition in Figure 3.1 (right).

4.3 Experimental results

The proposed algorithm is implemented in ABC [5] as command *lutpack*. Experiments targeting 6-input LUTs were run on an Intel Xeon 2-CPU 4-core computer with 8Gb of RAM. The resulting networks were all verified using the combinational equivalence checker in ABC (command *cec*) [28].

The following ABC commands are included in the scripts used to collect the experimental results, which targeted area minimization while preserving delay:

- *resyn* is a logic synthesis script that performs 5 iterations of AIG rewriting [27] trying to improve area without increasing depth
- *resyn2* is a script that performs 10 iterations of a more diverse set of AIG rewritings than those of *resyn*
- *choice* is a script that allows for accumulation of structural choices; *choice* runs *resyn* followed by *resyn2* and collects three snapshots of the network: the original, the final, and the one after *resyn,* re3sulting in a circuit with structural choices.
- *if* is an efficient FPGA mapper using priority cuts [31], fine-tuned for area recovery (after a minimum delay mapping) and using subject graphs with structural choices[6]
- *imfs* is an area-oriented resynthesis engine for FPGAs [30] based on changing a logic function at a node by extracting don't cares from a window and using Boolean resubstitution to rewrite the node function using possibly new inputs
- *lutpack* is the new resynthesis described in this section.

The benchmarks used in this experiment are 20 large public benchmarks from the MCNC and ISCAS'89 suites used in previous work on FPGA mapping [22][11][29][7].

Table 1 shows four experimental runs. We use the exponent, *n*, notation to denote iteration of the expression in parenthesis *n* times, e.g. (*com1; com2*)^3 means iterate (*com1; com2*) three times.

- "Baseline" = (*resyn; resyn2; if*). It corresponds to a typical run of technology-independent synthesis followed by default mapping
- "Choices" = *resyn; resyn2; if;* (*choice; if*)^4.
- "Imfs" = *resyn; resyn2; if;* (*choice; if; imfs*)^4.
- "Lutpack" = *resyn; resyn2; if;* (*choice; if; imfs*)^4; (*lutpack*)^2.

The table lists the number of primary inputs ("PIs"), primary outputs ("POs"), registers ("Reg"), area calculated as the number of 6-LUTs ("LUT") and delay calculated as the depth of the 6-LUT network ("Level"). The ratios in the tables are the ratios of geometric averages of values reported in the columns.

The *Baseline* and *Choices* columns have been included to show that repeated re-mapping has a dramatic impact over strong conventional mapping (*Baseline*). However, the main purpose of the experiments is to demonstrate the additional impact that the proposed command *lutpack* has on top of this very strong flow. Thus we only focus on the last line of the table, which compares *lutpack* against the strongest result obtained using other methods (*imfs*). Given the power of *imfs,* it is somewhat unexpected that *lutpack* can achieve an *additional* 5.4% reduction in area[8].

This additional area reduction speaks for the orthogonal nature of *lutpack* over *imfs*. While *imfs* tries to reduce area by analyzing alternative resubstitutions at each node, it cannot efficiently compact large fanout-free cones that may be present in the mapping. The latter is done by *lutpack*, which iteratively collapses fanout-free cones with up to 16 inputs and re-derives new implementations using the minimum number of LUTs.

The runtime of one run of *lutpack* did not exceed 20 sec for any of the benchmarks reported in Table 1. The total runtime of the experiments was dominated by *imfs.* (It has been tuned only recently for higher speed). Also, it should be pointed out that Table 1 illustrates only two passes of *lutpack* as a final processing, but several iterations where *lutpack* is in the iteration loop (e.g. (*choice; if; imfs; lutpack*)^4 often show similar additional gains.

5 Conclusions and future work

The paper presented a fast algorithm for decomposition of logic functions. We focused on an application to area-oriented resynthesis of *K*-LUT structures. The new algorithm, *lutpack*, is based on cofactoring and disjoint-support decomposition and is

[6] The mapper was run with the following settings: at most 12 6-input priority cuts are stored at each node; five iterations of area recovery are performed, three with area flow and two with exact local area.

[7] In the above set, circuit s298 was replaced by i10 because the former contains only 24 6-LUTs

[8] Some readers may suspect that the result after *imfs* can be improved on easily. We can only invite them to take any of the benchmarks (which are publicly available) and come up with a better result than that recorded in the *imfs* column.

much faster than previous solutions that rely on BDD-based decomposition and Boolean satisfiability. It achieved an additional 5.4% reduction in area, when applied to a network obtained by iterated high-quality technology mapping and another type of powerful resynthesis using don't cares and windowing.

Future work in this area will include:

- Improving the DSD-based analysis, which occasionally fails to find a feasible match and is the most time-consuming part.
- Exploring other data structures for cofactoring and DSD decomposition, to allow processing of functions with more than 16 inputs. This will improve the quality of resynthesis.

Some possible future applications include;

1. computing all decompositions of a function and using them to find common Boolean divisors among all functions of a logic network,
2. speeding up a network by finding fanin cones on critical paths, and collapsing and re-factoring them,
3. merging fanin blocks of two functions, where the blocks share common supports.
4. extracting a common Boolean divisor from a pair of functions (using Theorem 3.3),
5. mapping into fixed macro-cells, where the divisors must have a fixed but given structure as exemplified by Altera Stratix II [3] or Actel ProASIC3 devices [2].

Acknowledgements

This work was supported in part by SRC contracts 1361.001 and 1444.001, NSF contract CCF-0702668, and the California MICRO Program with industrial sponsors Actel, Altera, Calypto, IBM, Intel, Intrinsity, Magma, Synopsys, Synplicity, Tabula, and Xilinx. The authors are indebted to Stephen Jang of Xilinx for his masterful experimental evaluation of the proposed algorithms. We thank Slawomir Pilarski and Victor Kravets for careful readings and useful comments, and to an anonymous reviewer for pointing out critical errors in the original manuscript. This work was done while the third author was at Berkeley.

References

[1] A. Abdollahi and M. Pedram, "A new canonical form for fast Boolean matching in logic synthesis and verification", *Proc. DAC '05*, pp. 379-384.

[2] Actel Corp., "ProASIC3 flash family FPGAs datasheet," http://www.actel.com/documents/PA3_DS.pdf

[3] Altera Corp., "Stratix II device family data sheet", 2005, http://www.altera.com/literature/hb/stx/stratix_section_1_vol_1.pdf

[4] R. L. Ashenhurst, "The decomposition of switching functions", *Proc. Intl Symposium on the Theory of Switching*, Part I (Annals of the Computation Laboratory of Harvard University, Vol. XXIX), Harvard University Press, Cambridge, 1959, pp. 75-116.

[5] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 70911. http://www.eecs.berkeley.edu/~alanmi/abc/

[6] V. Bertacco and M. Damiani. "The disjunctive decomposition of logic functions". *Proc. ICCAD '97*, pp. 78-82.

[7] R. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.

[8] D. Chai and A. Kuehlmann, "Building a better Boolean matcher and symmetry detector," *Proc. DATE '06*, pp. 1079-1084.

[9] S. Chatterjee, A. Mishchenko, and R. Brayton, "Factor cuts", *Proc. ICCAD '06*, pp. 143-150. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iccad06_cut.pdf

[10] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD '05*, pp. 519-526. http://www.eecs.berkeley.edu/~alanmi/publications/2005/iccad05_map.pdf

[11] D. Chen and J. Cong. "DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs," *Proc. ICCAD '04*, 752-757.

[12] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," *Proc. FPGA'99*, pp. 29-36.

[13] A. Curtis. *New approach to the design of switching circuits*. Van Nostrand, Princeton, NJ, 1962.

[14] D. Debnath and T. Sasao, "Efficient computation of canonical form for Boolean matching in large libraries," *Proc. ASP-DAC '04*, pp. 591-596.

[15] A. Farrahi and M. Sarrafzadeh, "Complexity of lookup-table minimization problem for FPGA technology mapping," *IEEE TCAD*, Vol. 13(11), Nov. 1994, pp. 1319-1332.

[16] C. Files and M. Perkowski, "New multi-valued functional decomposition algorithms based on MDDs". *IEEE TCAD*, Vol. 19(9) , Sept. 2000, pp. 1081-1086.

[17] Y. Hu, V. Shih, R. Majumdar, and L. He, "Exploiting symmetry in SAT-based Boolean matching for heterogeneous FPGA technology mapping", *Proc. ICCAD '07*.

[18] V. N. Kravets. *Constructive Multi-Level Synthesis by Way of Functional Properties*. Ph. D. Thesis. University of Michigan, 2001.

[19] V. N. Kravets and K. A. Sakallah, "Constructive library-aware synthesis using symmetries", *Proc. of DATE*, pp. 208-213, March 2000.

[20] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE TCAD*, Vol. 16(8), 1997, pp. 813-833.

[21] A. Ling, D. Singh, and S. Brown, "FPGA technology mapping: A study of optimality", *Proc. DAC '05*, pp. 427-432.

[22] V. Manohara-rajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," *Proc. IWLS '04,* pp. 14-21.

[23] Y. Matsunaga. "An exact and efficient algorithm for disjunctive decomposition". *Proc. SASIMI '98*, pp. 44-50.

[24] K. Minkovich and J. Cong, "An improved SAT-based Boolean matching using implicants for LUT-based FPGAs," *Proc. FPGA'07*.

[25] A. Mishchenko and T. Sasao, "Encoding of Boolean functions and its application to LUT cascade synthesis", *Proc. IWLS '02*, pp. 115-120. http://www.eecs.berkeley.edu/~alanmi/publications/2002/iwls02_enc.pdf

[26] A. Mishchenko, X. Wang, and T. Kam, "A new enhanced constructive decomposition and mapping algorithm", *Proc. DAC '03*, pp. 143-148.

[27] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC'06*, pp. 532-536.

[28] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *Proc. ICCAD '06*, pp. 836-843. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iccad06_cec.pdf

[29] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs". *IEEE TCAD*, Vol. 26(2), Feb 2007, pp. 240-253. http://www.eecs.berkeley.edu/~alanmi/publications/2006/tcad06_map.pdf

[30] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "SAT-based logic optimization and resynthesis". Submitted to FPGA'08. http://www.eecs.berkeley.edu/~alanmi/publications/2008/fpga08_imfs.pdf

[31] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*. http://www.eecs.berkeley.edu/~alanmi/publications/2007/iccad07_map.pdf

[32] A. Mishchenko, S. Chatterjee, and R. Brayton, "Fast Boolean matching for LUT structures". ERL Technical Report, EECS Dept., UC Berkeley. http://www.eecs.berkeley.edu/~alanmi/publications/2007/tech07_lpk.pdf

[33] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," *Proc. FPGA '98*, pp. 35-42.

[34] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. S. Zhang.

"Decomposition of multiple-valued relations". *Proc. ISMVL'97*, pp. 13-18.

[35] S. Plaza and V. Bertacco, "Boolean operations on decomposed functions", *Proc. IWLS' 05*, pp. 310-317.

[36] J. P. Roth and R. Karp, "Minimization over Boolean graphs", *IBM J. Res. and Develop.*, 6(2), pp. 227-238, April 1962.

[37] S. Safarpour, A. Veneris, G. Baeckler, and R. Yuan, "Efficient SAT-based Boolean matching for FPGA technology mapping," *Proc. DAC '06*.

[38] H. Sawada, T. Suyama, and A. Nagoya, "Logic synthesis for lookup tables based FPGAs using functional decomposition and support minimization", *Proc. ICCAD,* 353-358, Nov. 1995.

[39] T. Sasao and M. Matsuura, "DECOMPOS: An integrated system for functional decomposition," *Proc. IWLS'98*, pp. 471-477.

[40] N. Vemuri and P. Kalla and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs", *ACM TODAES*, Vol. 7, 2002, pp. 501-525.

[41] B. Wurth, U. Schlichtmann, K. Eckl, and K. Antreich. "Functional multiple-output decomposition with application to technology mapping for lookup table-based FPGAs". *ACM Trans. Design Autom. Electr. Syst*. Vol. 4(3), 1999, pp. 313-350.

Table 1. Evaluation of resynthesis applied after technology mapping for FPGAs ($K = 6$).

Designs	PI	PO	Reg	Baseline		Choices		Imfs		Imfs + Lutpack	
				LUT	Level	LUT	Level	LUT	Level	LUT	Level
alu4	14	8	0	821	6	785	5	558	5	453	5
apex2	39	3	0	992	6	866	6	806	6	787	6
apex4	9	19	0	838	5	853	5	800	5	732	5
bigkey	263	197	224	575	3	575	3	575	3	575	3
clma	383	82	33	3323	10	2715	9	1277	8	1222	8
des	256	245	0	794	5	512	5	483	4	480	4
diffeq	64	39	377	659	7	632	7	636	7	634	7
dsip	229	197	224	687	3	685	2	685	2	685	2
ex1010	10	10	0	2847	6	2967	6	1282	5	1059	5
ex5p	8	63	0	599	5	669	4	118	3	108	3
elliptic	131	114	1122	1773	10	1824	9	1820	9	1819	9
frisc	20	116	886	1748	13	1671	12	1692	12	1683	12
i10	257	224	0	589	9	560	8	548	7	547	7
pdc	16	40	0	2327	7	2500	6	194	5	171	5
misex3	14	14	0	785	5	664	5	517	5	446	5
s38417	28	106	1636	2684	6	2674	6	2621	6	2592	6
s38584	12	278	1452	2697	7	2647	6	2620	6	2601	6
seq	41	35	0	931	5	756	5	682	5	645	5
spla	16	46	0	1913	6	1828	6	289	4	263	4
tseng	52	122	385	647	7	649	6	645	6	645	6
geomean				1168	6.16	1103	5.66	716	5.24	677	5.24
Ratio1				**1.000**	**1.000**	**0.945**	**0.919**	**0.613**	**0.852**	**0.580**	**0.852**
Ratio2						**1.000**	**1.000**	**0.649**	**0.926**	**0.614**	**0.926**
Ratio3								**1.000**	**1.000**	**0.946**	**1.000**