Faster Logic Manipulation for Large Designs

Alan Mishchenko Robert Brayton

Department of EECS, University of California, Berkeley

{alanmi, brayton}@eecs.berkeley.edu

Abstract

When logic transformations, such as circuit restructuring, technology mapping, and post-mapping optimization, are repeatedly applied to large hardware designs, millions of relatively small (6-16 input) Boolean functions have to be efficiently manipulated. This paper focuses on a novel representation of these small functions, in terms of their disjoint-support decomposition (DSD) structures. A new DSD manipulation package is developed, which allows for faster logic manipulation compared to known methods.

1. Introduction

The last decade has seen gradual emergence of a new logic synthesis based on (a) simple circuit structure, (b) local transforms, and (c) iterative optimization.

This paradigm shift has led to the development a logic synthesis and formal verification system ABC [2], which is actively used in academia and industry. For example, many of the participants of HWMCC'12 [4] build parts of their verification tools on top of ABC, while several commercial offerings use ABC as an internal logic optimization/ mapping/verification engine.

A differentiator of ABC is the use of And-Inverter Graphs (AIGs) [9] for logic representation. Optimization, mapping, post-mapping resynthesis, and even Boolean reasoning (SAT solving) are performed by considering K-feasible cuts in the AIG and applying various operations, such as decomposition, matching, and CNF generation, to the K-input Boolean functions of these cuts. Each cut can be seen as a logic node in the old-style synthesis exemplified by SIS [21]. By operating on an AIG with flexible cut/node boundaries, ABC achieves the effect of optimizing multiple networks, comparing the results and choosing the best.

While the synthesis in ABC has been found promising compared to the traditional synthesis [6][7] in terms of quality and scalability for large designs, it has limitations:

Limiting scope in order to save memory: In a typical AIG-based computation, such as AIG rewriting [5][13] or technology mapping [14], structural cuts are computed along with the truth tables of their Boolean functions. Deriving truth tables takes about 20% of the total cut enumeration time for 6-input cuts, 100% for 10-input cuts, and nearly 300% for 12-input cuts. Memory needed to store the cut functions also grows exponentially with the cut size. This often limits optimization to use small enough cut sizes for the runtime and memory usage to be reasonable.

Degrading quality in order to save runtime. To detect various Boolean properties of cut functions represented by truth tables, complicated and time-consuming computations are required. For example, in recent work on LUT structure mapping [17], truth table computation and Boolean matching for cut functions of 12 or more inputs takes 50X longer than cut enumeration alone. This runtime increase is prohibitive for large designs, especially when the LUT structure mapping is used in the inner loop of a synthesis system similar to [16]. This is why the quality of mapping is often compromised in order to gain speed of computation.

Both of these limitations are related to the use of truth tables to represent Boolean functions of K-feasible cuts. A well-known alternative to truth tables is Binary Decision Diagrams (BDDs) [7]. However, BDDs have the same drawbacks when it comes to manipulating millions of relatively small functions appearing in hardware designs:

- They are even slower to construct than truth tables, due to CPU cache misses inherent in the use of hashing (both unique table and computed table are hash tables).
- Although BDDs are typically more compact than truth tables for functions of 10-16 inputs, for some practical functions, such as multipliers and their parts, BDDs can be large, which slows down the computation.
- *Most importantly, both truth tables and BDDs obscure Boolean properties, such as NPN-classification, decomposability, symmetries, unateness, linearity, etc.* In other words, timing-consuming and complex computations should be performed on truth tables and BDDs to find these properties.

In this paper, we introduce and explore a different data-structure and computation engine based on disjoint support decompositions (DSDs). This engine outperforms both truth tables and BDDs for handling relatively small (6-16 input) Boolean functions arising when local transforms are applied to large industrial designs.

The new representation allows for scalable, low-memory computation, *while making Boolean properties, such as decomposability, explicit and readily available in practical applications, such as circuit restructuring and technology mapping.* An added advantage is that the new representation allows for caching computed results, similar to an efficient BDD package.

The contribution of this paper is a synthesis methodology based on DSDs and an efficient package for storing, manipulating, and analyzing DSD structures for Boolean functions. The components of this methodology are:

- Canonical form [1], which makes Boolean properties of a function explicit, unlike truth tables and BDDs.
- Algorithm to compute DSD from cofactors [3] without resorting to Karnaugh maps and decomposition charts.

- Algorithm for deriving DSD structures directly from the circuit, by performing operations on them [18], starting from DSD structures for elementary variables.
- Caching intermediate results, which is important since the percentage of functions with full and 1-step DSD is high and they are often repeated.

The rest of the paper is organized as follows. Section 2 contains some necessary background. Section 3 introduces and motivates DSD structures. Section 4 describes a DSD package that can be used to manipulate Boolean functions in a variety of applications. Experimental results are given in Section 5 and Section 6 concludes the paper.

2. Background

2.1 Boolean network

A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to logic gates and edges corresponding to wires connecting the nodes. It is assumed that each node has a unique integer called *node ID*.

A node *n* has zero or more fanins, i.e. nodes driving *n*, and zero or more fanouts, i.e. nodes driven by *n*. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes of the network, connecting it to the environment. A fanin (fanout) cone of node *n* is a subset of nodes of the network, reachable through the fanin (fanout) edges of the node.

2.2 And-Inverter Graph

And-Inverter Graph (AIG) is a Boolean network whose nodes can be classified as follows:

- One constant 0 node.
- Combinational inputs (primary inputs, flop outputs).
- Internal two-input AND nodes.
- Combinational outputs (primary outputs, flop inputs).

The fanins of internal AND nodes and combinational outputs can be complemented. The complemented attribute is represented as a bit mark on a fanin edge, rather than a separate single-input node.

Due to their compactness and homogeneity, AIGs have become a de-facto standard for representing sequential miters in formal verification.

An *AIG manager* is a data-structure used to store AIGs in a software implementation of verification engines. A typical AIG manager enforces the following invariants during AIG construction and manipulation:

- Constants are propagated (the constant 0 can only feed into a combinational output).
- Duplicated fanins of AND nodes are not allowed.
- Fanins of AND nodes are ordered using their node IDs.
- AND nodes are structurally hashed (each AND node has a unique pair of fanins, possibly complemented).
- All AIG nodes are stored in a topological order (that is, fanins of each node precede the node itself).

2.3 Structural cuts

A cut *C* of a node *n* is a set of nodes of the network, called *leaves* of the cut, such that each path from a PI to *n* passes through at least one leaf. Node *n* is called the root of cut *C*. The cut *size* is the number of its leaves. A trivial cut is the node itself. A cut is *K-feasible* if the number of leaves does not exceed *K*. A local function of an AIG node *n*, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in *n* and expressed in terms of the leaves, *x*, of a cut of *n*.

An *onset minterm* is a complete assignment of input variables, which makes the function evaluate to 1.

Two functions are NPN-equivalent if one of them can be obtained from the other by negation and permutation of the inputs and outputs.

3. Disjoint-support decomposition

3.1 DSD structures

Disjoint-support decomposition (DSD) [1][3] represents a completely-specified Boolean function as a tree of nodes with non-overlapping supports and inverters as optional complemented attributes on the edges. This tree is called the *DSD structure* of a function. DSD structure is *full* if all logic nodes are elementary gates (AND, XOR, MUX). This definition is motivated by the fact that the elementary gates appear often in the designs and therefore DSD structures composed of them capture a practical subset of functions. If a full DSD exists for a given function, it is unique up to a permutation of inputs and positions of inverters on the edges. For example, $F = ab + cd$ has full DSD containing three ANDs and three inverters; $F = !(!(ab)\ !(cd))$.

The following observation can be made about *small practical Boolean functions*, that is, Boolean functions of 4-12 inputs appearing as cut functions in typical hardware designs. 60-80% of these have a full DSD. The majority of the remaining functions satisfies the following property: There exists at least one input variable, such that the cofactors w.r.t. this variable have full DSDs. Such functions are called *1-step DSD* functions to distinguish them from *complex* functions, which require more than one cofactoring step to get to full DSDs.

Unlike truth tables and BDDs, DSD structures have the advantage of making Boolean properties of the function explicit. Several examples are:

- symmetric variables of a full DSD structure are found by detecting symmetric nodes (AND/XOR) at the leaves of the structure;
- decomposability of a full DSD into a network of K-LUTs is checked by computing K-feasible cuts, without an expensive search for a K-feasible bound-set;
- a DSD structure can be made canonical by permuting branches according to some ordering principle, and by removing complemented attributes from the inputs and the output, resulting in an NPN canonical form.

If the function has a 1-step DSD, its analysis is bit more involved, as it requires simultaneous exploration of a pair of DSD trees of the cofactors, as shown in [15]. However, this analysis is still substantially simpler than computations using other representations.

What if the function is *complex*, that is, has no full DSD or 1-step DSD? In many practical applications, complex functions can be ignored, given that they appear rarely. For example, in technology mapping or AIG rewriting, cuts whose size exceeds the given limit are routinely filtered out. Added to these can be rare cuts whose functions are complex. In other applications, where all functions have to be considered, properties of complex functions can be computed using standard methods or inferred from the properties of the structural divisors that have DSDs.

3.2 Operations on DSD structures

Although there exists many algorithms to find the DSD structure of a completely-specified Boolean function using BDDs [3][10][20], SOPs [12], or AIGs [10], they are not efficient enough when applied to functions of K-feasible cuts in the AIG. Indeed, if we need to compute a truth table or a BDD before a DSD structure can be found, we run into problems with memory and runtime, as discussed above. Also, even if the runtime of a decomposition algorithm is negligible when applied to one small function, computation slows down when it is applied to millions of functions.

Our computation of DSDs for cut functions is based on an earlier work [18], which allows us to compute the DSD of a cut function by considering DSDs of its fanins.

To understand how it works, we first review the computation of truth tables during topological cut enumeration in the AIG [17]. The cut of the root node is derived by merging two fanin cuts. The truth table of the resulting cut is computed by permuting the fanin truth tables to match the variable ordering of the resulting cut, followed by a bit-wise AND operation with possible complementation due complementation of the fanins.

Similarly, if functionality of the fanin cuts is represented using DSD structures, the DSD structure of the resulting cut is the conjunction of the fanin DSD structures. If the supports of the two fanin cuts are mutually disjoint, the resulting DSD structure is obtained by adding an AND to two existent ones. If fanins cuts have overlapping supports, the algorithm is shown below (see [18] for details):

- create a shared representation by putting a two-input AND gate on top of the two DSD structures;
- isolate the reconvergent core of the shared representation by replacing disjoint-support branches of the fanin DSD trees by free variables;
- collapse the reconvergent core into a single Boolean function and find the DSD of this function if it exists; otherwise, a non-decomposable node is introduced;
- create the resulting DSD structure by composing the free variables of the DSD of the recovergent core with the corresponding disjoint-support branches.

This computation lends itself to an efficient implementation. Moreover, intermediate results can be cached and reused, as it is done in a BDD package.

Example: Consider conjunction of two DSD structures: $F = ab + cd$ and $G = aef$, with one shared variable (a). Thus $x = cd$, $y = ef$, $H = FG = (ab + x)ay = ay(b+x)$. The result is a DSD structure: $H = aef(b + cd)$.

3.3 Advantages of DSDs over truh tables

This section motivates the use of DSD structures by showing that they tend to be better than truth tables in terms of memory, runtime, and quality of results.

Memory: Consider mapping a 1M-node AIG into K-LUTs, which is a key step in optimization engines, such as incremental re-synthesis, LUT mapping, LUT structure mapping, and CNF computation. For K = 10, memory for a truth table takes 128 bytes. The technology mapper [14] stores one cut with truth table at each node, which results in about 48MB used for cuts and 128MB for truth tables. It can be seen that truth tables dominate memory requirements of the mapper for cuts with K > 8.

Assuming that about 90% of 10-input functions have full or 1-step DSDs, their DSD structures can be stored in a shared DSD manager, containing only NPN classes of functions, which is typically only about 3-5% of the total number of functions enumerated during cut computation. Mapping of cut inputs into the input variables of the DSD manager is a permutation of 10 integers between 0 and 9. It takes one byte per input variable, plus one integer needed to store the number to the canonical structure in the DSD manager. As a result, 1M cuts take about 14MB, an 9x reduction compared to memory used for truth tables.

Runtime: A substantial fraction of runtime during cut-enumeration is spent computing truth tables of the cuts. This runtime grows exponentially in the number of cut variables. As noted above, for all cuts whose fanins have disjoint supports, the DSD representation can be computed in constant time. For other cuts, the procedure described in Section 2.2 should be used. The runtime of this procedure is exponential in the size of the reconvergent core of the circuit structure, which is typically much smaller than K. This leads to an exponential reduction in runtime. Caching of computed results can further reduce this runtime.

Quality: All transformations (such as restructuring and Boolean decomposition) are more natural (if not trivial) when DSD structures are used to represent the functions. For example, Table 3 in the experimental results section shows that LUT structure mapping [19] runs 6X faster with DSDs, compared to truth tables. This is because there is no need to perform repeated truth table-based computations while searching for a good subset of variables. Decomposition of Boolean functions with full DSD can be found by one traversal of their DSD structures. Decomposition of those with 1-step DSD can be found by a coordinated traversal of the two DSD structures of the cofactors, which is quadratic in the number of tree nodes. Rare complex functions are ignored in favor of smaller cuts having a DSD. Except for the computation of DSD structures of the cut functions, truth tables are not used in the process of decomposition.

4. DSD manager

This section describes a software package, called *DSD manager,* for storing Boolean functions using their DSD structures, performing Boolean operations, and caching intermediate computations. The Boolean operations include various checks applied to DSD structures, for example, the result of matching a function with a LUT structure [19].

4.1 Primitives

Primitives of the DSD manager are the following:

- One constant 0 node.
- One primary input node *n*.
- Multi-input AND and XOR nodes with ordered fanins.
- Three-input MUX nodes.

- Multi-input PRIME nodes whose Boolean functions do not have DSDs.

These primitives look similar to those of the AIG manager, listed in Section 2.2, with several important differences. For example, logic nodes have several different types and can have more than two fanins. Also, structural canonicity is stricter than that of an AIG manager.

4.2 Canonicity

A DSD package is similar to a BDD package in that both enforce a strong notion of canonicity on the representation of a Boolean function. In particular, BDD package represents a function as a DAG of ITE (MUX) operators. This DAG is canonical for a given variable order [8]. A DSD package, on the other hand, canonicizes a DSD structure by enforcing several invariants described below.

4.2.1 Shifting variables

No matter what specific variables a given function depends on, they are always expressed using one Boolean variable. This is only possible for trees (as opposed to DAGs) whose nodes have fanin ordering (see Section 4.2.4 below).

Example: Function AND(*a*, XOR(*b*, *c*), MUX(d, e, f)) has canonical form AND(*n*, XOR(*n, n*), MUX(*n, n, n*)). Since the ordering of fanins of each DSD node is fixed, the actual formula for this function is derived from its canonical form by replacing variable *n* in the canonical form by variables *a, b, c,* etc, in their natural order.

Example: Both functions $F = (a \oplus b)c$ and $G = d(e \oplus f)$ share the same canonical form AND(*n*, XOR(*n, n*)). The reason why AND(*n*, XOR(*n, n*)) is chosen instead of AND(XOR(*n, n*), *n*), is explained in Section 4.2.4.

4.2.2 Propagating complements

Complemented attributes are not allowed at the primary inputs of a DSD structure, but allowed at the intermediate nodes if these attributes cannot be moved to the primary inputs or the output of a node.

Example: Function AND(*a*, !AND(*b, c*)) has canonical form AND(*n*, !AND(*n, n*)) because the complemented attribute, denoted as an exclamation mark, cannot be moved to the primary inputs.

Example: Function AND(!*a*, !XOR(*b*, *c*)) has canonical form AND(*n*, XOR(*n, n*)). Indeed, the internal complemented attribute can be moved to the primary inputs because !XOR(*b,c*) = XOR(!*b,c*). The attributes at the primary inputs can be removed resulting in a function that is NPN-equivalent to the original one.

Example: AND(*a*, MUX(*b*, !AND(*c,d*), !AND(*e,f*))) has canonical form AND(*n*, !MUX(*n*, AND(*n,n*), AND(*n,n*))) because the complemented attributes at the data inputs of a MUX propagate to the output of the MUX.

4.2.3 Collapsing operators

An XOR node cannot have another XOR node as a fanin. An AND node cannot have another AND node as a fanin if this fanin has no complemented attribute.

Example: Function AND(*a*, AND(*b*, !AND(c, *d*)) has canonical form AND(*n*, *n*, !AND(*n, n*)) because the topmost two AND nodes can be collapsed.

4.2.4 Ordering fanins

The following (quite arbitrary) rules are used to order fanins of symmetric DSD nodes (AND and XOR):

- The node fanins are ordered by their support size.
- If there is a tie, AND precedes XOR precedes MUX precedes PRIME.
- If there is a tie, a non-complemented fanin precedes a complemented fanin.
- If there is a tie, the fanins' fanins are ordered and compared in their selected order. If the recursive comparison fails to produce a unique order, the fanins' DSD structures are isomorphic and therefore their order is immaterial.

Example. Function $F = ab \oplus cd$ has canonical form XOR(AND(*n,n*), AND(*n,n*)), in which the order of fanins of XOR is immaterial because they are isomorphic.

Fanins of three-input MUX nodes are ordered as follows: *control*, *data0*, *data1*. Fanins of PRIME nodes are ordered by the algorithm used in [22] to (semi-)canonicize Boolean functions of these nodes.

Although the ordering rules are arbitrary, they are consistently applied by the DSD manager. This results in computing the same canonical form when applied to any Boolean function in the given NPN class.

4.3 Boolean operations

The representation of an arbitrary Boolean function in terms of its DSD structure includes the following parts: (1) integer ID of the node in the DSD manager representing the canonical form of its DSD structure; (2) optional complemented attribute of the output; and (3) PN-configuration showing how to permute/complement inputs of the canonical form to obtain the given function. The first two parts can be combined by specifying the node ID together with the complemented attribute.

When a Boolean operation is performed on two functions, their NPN-configurations are used to find the common variable order. Then the Boolean operation is performed on the canonical forms of these functions, as described in Section 3.2. Finally, the result is expressed as a canonical structure in the DSD manager, and its node ID is returned to the user, along with the resulting NPN-configuration.

Example: Consider computing the conjunction of functions $F = cd + ef$ and $G = ca!b$, with canonical forms !AND(!AND(*n, n*), !AND(*n, n*)) and AND(*n, n, n*). The PN-configurations are (*c, d, e, f*) and (*c, a,* !*b*), respectively. Indeed, if we substitute variables (*c, d, e, f*) into the first canonical form in this order, we get $F = cd + ef$. Now we find the common variable order and detect that *c* is a shared variable. A Boolean operation is performed on the canonical forms: $F^{canon} = ab + cd$ and $G^{canon} = aef$, and the result is $H^{canon} = F^{canon} \wedge G^{canon} = aef(b + cd)$, as shown in the example of Section 3.2. Converting it back into the original variable order, we get $H = F \wedge G = ca!b(d + ef)$. This function is presented in the DSD manager as the DSD structure AND(*n, n, n*, !AND(*n*, !AND(*n, n*))) with the PN-configuration (*c, a, !b, d, e, f*).

4.4 Computed table

Given that the vast majority of functions and operations in practical applications appear many times, the DSD package benefits from caching intermediate results. To this end, Boolean operations (such as AND or XOR) performed on the canonical form, as described in Section 4.3, are cached and the cache is looked up when an operation is performed.

4.5 Handling non-DSD blocks

An efficient NPN semi-canonicizer has been developed as part of the previous work on harvesting and reusing AIG structures for practical Boolean functions [22]. This canonicizer can be applied to the non-decomposable functions and non-decomposable blocks of DSD structures to bring them into a semi-canonical form. As a result, when two DSD structures with non-decomposable blocks can be reduced to each other by permuting and complementing their inputs/output, they are stored as a single structure.

Since the number of NPN classes of non-decomposable functions appearing in hardware designs is limited even for smaller cuts, the increase in the size of the DSD manager due to their presence does not pose a problem.

If the number of these functions becomes prohibitive for larger cuts, cut enumeration can be restricted to keep only those cuts whose non-decomposable blocks have sizes that do not exceed a given limit (for example, 6 inputs). Since cuts with large non-decomposable nodes occur infrequently, skipping them does not degrade the quality of results more than other heuristics, such as removing non-K-feasible cuts.

5. Experimental results

The proposed DSD manager is implemented in ABC [2]. The current version is being tested by computing DSD structures of Boolean functions of K-feasible cuts that are encountered in the LUT mapper *if* [14].

The following three experiments are reported:

5.1 Statistics for functions of K-feasible cuts

Table 1 contains cumulative statistics for a suite of industrial synthesis benchmarks. Table 2 contains cumulative statistics for public benchmarks suites MCNC, ISCAS, and ITC. For each benchmark, a fixed number (32) of K-feasible cuts was enumerated at each AIG node. Next, Boolean functions of the cuts were collected and analyzed. The following statistics are reported for each cut size:

- The number of cuts considered (Column “Cuts”).
- The percentage of unique Boolean functions relative to the number of all cuts (Column “Funcs”).
- The percentage of NPN semi-canonical classes relative to the number of all cuts (Column “NPNs”).

In each of the next two sections, columns “Full”, “1-Step” and “Complex” show percentages of functions with full DSD, 1-step DSD, and complex functions. The first 3-column section shows percentages relative to the number of NPN classes, while the second 3-column section is relative to the number of all cuts.

As an example, consider cut size 6 in Table 1 for industrial benchmarks. The first column denoted “Full” shows that 6.15% of NPN classes of functions of 6-input cuts have a full DSD. The second “Full” column shows that 87.68% of 6-input cuts have functions with a full DSD.

It is interesting to observe that industrial benchmarks have fewer NPN classes and more DSD-decomposable functions, compared to the academic ones. Possibly this has to do with the fact that 100+ public benchmarks come from many different sources and were processed by different tools, resulting in a more diverse population of Boolean functions.

5.2 Case study: LUT structure mapping

Table 3 shows preliminary experimental results for several public benchmarks, comparing the runtime of mapping into LUT structure “666” composed of three 6-LUTs [19] using truth tables or DSD structures to perform Boolean matching. This application was selected as a case-study because (a) LUT structure mapping mitigates structural bias and improves the quality of regular LUT-mapping, (b) it is useful for new generations of programmable devices, (c) it requires heavy Boolean manipulation, and (d) its current implementation in ABC is prohibitively slow for large designs.

Table 3 shows that Boolean matching for cut functions with the 16-input LUT structure is on average 30X faster with DSD structures than with truth tables. DSDs are faster to compute for large cut sizes, which leads to a 6X reduction in the total runtime of mapping.

Table 4 contains a sampling of full DSD structures appearing in MCNC benchmark *i10*. The notation used in the table is as follows: (*ab*) denotes AND(*a,b*); [*ab*] denotes XOR(*a,b*), and <*abc*> denotes MUX(*a, b, c*) = *ab* + !*ac*. Note that, for 2- and 3-input functions, the table contains *all* canonical structures with a full DSD.

6. Conclusions and future work

This paper presents preliminary evidence of the efficiency of a new data-structure and computation engine, *DSD structures*, for fast manipulation of numerous small Boolean functions arising in large industrial designs.

A new software package is currently being developed to enable efficient manipulation of DSD structures. The package is similar to a BDD package in that it exploits logic sharing among structures and enforces a strong notion of canonicity on the represented functions. The package also allows for caching of Boolean operations applied to DSD structures and Boolean properties of these structures (such as decomposability), which can lead to substantial speedups in practical implementations.

The case study of applying DSD structures to matching of Boolean functions with LUT structures suggests that the use DSD structures can lead to a substantial speedup in logic synthesis and technology mapping.

We believe that the (re)discovery of DSD structures as a vehicle of efficient manipulation of small practical Boolean functions is an important new development in logic synthesis which will enable significant improvements in several synthesis applications.

7. REFERENCES

[1] R. L. Ashenhurst, "The decomposition of switching functions". *Computation Lab*, Harvard University, 1959, Vol. 29, pp. 74-116.

[2] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

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

[4] A. Biere, HWMCC 2012, http://fmv.jku.at/hwmcc12/

[5] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification", *Proc. ICCAD '04*, pp. 42-49.

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

[7] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli, "Multilevel logic synthesis", *Proc. IEEE*, Vol. 78, Feb.1990.

[8] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comp*., Vol. C-35, No. 8 (August, 1986), pp. 677-691.

[9] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps", *Proc. DAC'97*, pp. 263-268.

[10] H.-P. Lin, J.-H. R. Jiang, and R.-R. Lee. "Ashenhurst decomposition using SAT and interpolation," *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, S. Khatri and K. Gulati (Editors) , Springer 2011, pp. 67-85.

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

[12] S.-I. Minato and G. De Micheli, "Finding all simple disjunctive decompositions using irredundant sum-of-products forms". *Proc. ICCAD'98*, pp. 111-117.

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

[14] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*, pp. 354-361.

[15] A. Mishchenko, R. K. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks", *Proc. ICCAD'08*, pp. 38-44.

[16] A. Mishchenko, N. Een, R. K. Brayton, S. Jang, M. Ciesielski, and T. Daniel, "Magic: An industrial-strength logic optimization, technology mapping, and formal verification tool". *Proc. IWLS'10*, pp. 124-127.

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

[18] S. Plaza and V. Bertacco, "Boolean operations on decomposed functions", *Proc. IWLS '05*.

[19] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen, "Mapping into LUT structures", *Proc. DATE'12*.

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

[21] E. Sentovich, et al, "SIS: A system for sequential circuit synthesis", *Tech. Rep. UCB/ERI, M92/41*, ERL, Dept. of EECS, Univ. of California, Berkeley, 1992.

[22] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis", *Proc. ICCAD'12*.

Table 1: Percentages of unique functions, NPN classes, and DSDs for different cut sizes (industrial benchmarks).

	% of functions and classes			% of DSDs relative to NPN classes			% of DSDs relative to the number of cuts		
Cut size	Cuts	Funcs	NPNs	Full	1-step	Complex	Full	1-step	Complex
6	30037253	0.88	0.04	6.15	64.32	29.53	93.16	6.24	0.60
8	33637730	2.25	0.24	10.91	58.42	30.67	86.28	11.07	2.65
10	35044059	3.09	0.59	17.40	49.96	32.64	80.78	14.22	5.00
12	35641114	3.73	0.97	21.03	42.08	36.89	75.06	16.87	8.07
14	35910126	4.20	1.26	22.02	36.86	41.12	71.06	17.71	11.23

Table 2: Percentages of unique functions, NPN classes, and DSDs for different cut sizes (public benchmarks).

	% of functions and classes			% of DSDs relative to NPN classes			% of DSDs relative to the number of cuts		
Cut size	Cuts	Funcs	NPNs	Full	1-step	Complex	Full	1-step	Complex
6	19854401	1.68	0.07	4.74	58.38	36.88	89.72	9.32	0.96
8	21167936	4.66	0.56	8.27	47.22	44.51	78.48	16.36	5.16
10	21340473	6.71	1.46	11.57	37.79	50.63	69.96	19.60	10.44
12	21766880	7.92	2.23	12.55	32.69	54.76	63.51	20.93	15.56
14	22002227	8.89	2.92	11.99	28.40	59.61	60.36	19.25	20.39

Table 3: Runtime comparison for LUT structure mapping with truth tables (left) and DSDs (right).

Name	PI	PO	AND	Match (TT), sec	Total (TT), sec	Match (DSD), sec	Total (DSD), sec
i10	257	224	2675	9.44	14.13	0.21	2.41
pj1	1769	1063	16285	54.37	82.69	2.19	11.52
aqua	1941	4805	25058	122.23	169.67	4.05	22.85
Geo				1.000	1.000	0.031	0.147

Table 4: A sampling of full DSD structures appearing in MCNC benchmark i10.

2 inputs	3 inputs	4 inputs	5 inputs	6 inputs
(ab)	(abc)	(abcd)	(abcde)	(abcdef)
[ab]	(a!(bc))	(a!(b!(cd)))	(a[(bc)(de)])	(ab[(cd)(ef)])
	[abc]	[abcd]	(a!(!(bc)!(de))	(!(abc)!(def))
	[a(bc)]	(a[b(cd)])	(a<b(cd)e>)	[a<b[cd][ef]>]
	(a[bc])	(a<bcd>)	<a(bc)(de)>	<a(bc)<def>>
	<abc>	<ab[cd]>	<a<bc!(de)>>	<(ab!(cd))ef>