Boolean Operations on Decomposed Functions

Stephen Plaza   Valeria Bertacco
splaza@umich.edu   *valeria@umich.edu*
Advanced Computer Architecture Lab
The University of Michigan
Ann Arbor, MI 48109

ABSTRACT

Disjoint support decompositions (DSDs) are a way of exposing the inherent hierarchical structure of a Boolean function. By decomposing a function in its disjoint support components it is possible to reduce its complexity by considering it as the simple composition of smaller blocks that are disjoint, that is, they do not share any input variable. Exposing a function's decomposability properties has the potential to enable optimizations in various application domains of computer-aided design, from synthesis to verification. Moreover, it provides the potential for generating a compact representation of the function. Algorithms have been proposed that can decompose a function into a decomposition tree of the finest granularity components, once its Binary Decision Diagram (BDD) is given. However, no solutions have been suggested so far for applying Boolean operators directly on decomposed forms that do not require one to reconstruct first the BDD of the operand functions involved and then construct the BDD of the result.

This paper proposes a novel algorithm to perform Boolean operations directly on a decomposed form. Our algorithm can construct complex DSDs by performing multiple Boolean operations directly on other simpler DSDs, without ever constructing the BDD of the functions. The implications of the contribution are twofold: 1) we can maintain a lower memory profile than previous algorithms generating DSDs as a consequence of not constructing the BDDs before decomposition, and 2) by using a decomposed form during Boolean function manipulation, it is possible to exploit directly the function's hierarchical structure, exposed by the decomposed form, for further optimizations.

We show results indicating that we always maintain a lower memory profile compared to previous decomposition techniques while achieving competitive runtime performance. We also show that our algorithm generates a representation of a Boolean function that can require significantly less memory than a BDD representation. Thus, our algorithm could provide a compact alternative to BDDs in memory-critical applications.

1. INTRODUCTION

The compact representation of Boolean functions and their efficient manipulation is a central requirement for many applications of computer-aided design and beyond. Binary Decision Diagrams (BDD) [1] are a widespread representation based on directed-acyclic graphs (DAG). For many Boolean functions arising in practical situations, the size of the BDD representation is quite compact, and so is its memory footprint, thus enabling the manipulation of very complex functions. Moreover, Boolean operations between BDDs can be performed efficiently, enabling the construction of BDDs representing very complex functions by combining simpler ones. However, in several cases, the size of the BDD representation is prohibitively large, an example are multipliers, which have provably exponential BDD size [2]. One of the limitations of BDDs is that they represent a function as a monolithic graph, which usually does not expose any apparent internal structure of the function. This lack of structure in the representation is a contributing factor to the memory explosion that is seen in BDDs, because, often, common sub-functions go undetected and thus their representation cannot be re-used. This is also a cause of the high sensitivity that BDDs present with respect to variable ordering [3]. In fact, a good ordering could group together variables of a frequently used subfunction allowing for sharing of its representation, while a bad ordering would prevent this from happening.

Research that aims at recognizing the internal structure of a given Boolean function has been ongoing since the inception of design automation. Within this field, Disjoint Support Decompositions, (DSDs), play a major role [4, 5, 6]. Disjoint Support Decompositions are a way of decomposing a Boolean function by identifying sub-function components that do not share any input variables, and expressing the main function in terms of those components. It has been shown [7] that, although theoretically only a small fraction of all Boolean functions are decomposable, most functions arising in practice present good decomposition properties. Canonical forms and algorithms have been defined that can find the finest granularity decompositions of a Boolean function, once its BDD representation is known, and construct a corresponding decomposition tree [8, 9].

The ability to apply Boolean operators directly on decomposed forms of a function enhances applications benefit from decompositions by improving their performance and lowering their memory profile. Exposing the decomposed form of a function is advantageous for several reasons. First, it's a method to obtain a multiple-level implementation of a function, which can be exploited during synthesis. Decompositions have been applied in synthesis problems [10, 11, 6], and placement and routing, where they provide an automatic method for clustering together logic cells that depend on the same subset of inputs, thus enabling shorter interconnect lengths and increased routability [12]. Attempts [13, 14] to generate synthesized circuits directly from BDD forms have only been successful when applied to small functions. In the case of complex functions, a typical synthesis procedure would exploit the hierarchical structure of the functions derived from the designer's description; however, a synthesis tool that maps BDDs directly to logic gates would lose this structure information and produce a much larger circuit. By using DSDs, a hierarchy can be detected automatically and this problem alleviated. Second, DSDs expose parallelism in the computation of the function that can be exploited in formal verification and simulation. For example, the evaluation of complex digital functions can benefit from the inherent parallelism that is exposed by the disjoint support components of a function [15, 16]. Finally, by performing Boolean operations directly on DSDs, the requirement of generating an initial BDD representation to build a DSD, as in [8, 9], is unnecessary.

However, so far no algorithm has been suggested to apply Boolean operators on the decomposed form of a function, thus, it is always necessary to transform the function back to its BDD representation in order to perform any Boolean manipulation. Each direction of this transformation involves the execution of algorithms that have quadratic complexity on the size of the BDDs involved. DSDs provide the potential for detecting and re-using common sub-functions independently of the variable order, thus generating a more compact representation. Moreover, these decompositions are independent of the chosen variable ordering, in contrast with BDDs, reducing the variability of the memory profile.

In this paper, we introduce a novel algorithm that supports closure of DSDs under Boolean operations. In other words, our algorithm is an efficient approach for building the DSD of a function from other DSDs, thus eliminating the need for constructing the BDD beforehand. In addition to providing valuable decomposition information about the Boolean functions without requiring an initial BDD, the algorithm requires a minimal amount of memory for BDDs and thus provides considerable improvements over previous work [9] without serious runtime performance degradation. Our ability to efficiently and compactly represent Boolean functions indicates that our algorithm could be used instead of BDDs for memory-intensive applications.

In the remainder of this paper, we first review the previous work in this area and provide the necessary background on disjoint support decompositions. We then present our Boolean manipulation algorithm that operates on decomposed functions, called DecO (from Decomposed Operations). We conclude by analyzing performance in the experimental results section and discussing future research.

2. BACKGROUND AND PREVIOUS WORK

2.1 Binary Decision Diagrams (BDD)

Binary Decision Diagrams [1] are a canonical and efficient data structure to represent and manipulate Boolean functions through directed acyclic graphs. Each node in the graph has two outgoing edges, and it represents a function F by its Shannon decomposition: $F = \overline{x}F_0 + xF_1$, where x is the variable labeling the node and F_0 and F_1 are the BDDs pointed to by the outgoing edges. One important constraint in BDDs is that the variable ordering for the labels in the graph has to be fixed. The size of a BDD is highly dependent on the selected variable ordering, thus algorithms to discover good orderings have been studied extensively [3, 17]. Good orderings allow the construction of BDDs whose size is linear in the number of variables in the support of the function. It has been shown, however, that in the worst case the complexity of a BDD is exponential [2]. Applications of BDDs have been explored and adopted in virtually every area of computer-aided design and beyond (some examples are given in [18, 19]), although many applications continue to be limited by the memory required by BDDs.

2.2 Disjoint Support Decompositions (DSD)

In the most general case, finding the disjoint support decomposition of a function F, consists of discovering simpler functions K and A_i such that:

$$F(x_1, .., x_n) = K(A_1(x_1, .., x_{A_1}), A_2(x_{A_1+1}, .., x_{A_2}), ..) \quad (1)$$

with $\mathcal{S}(A_i) \cap \mathcal{S}(A_j) = \emptyset$, $\forall i, j$.

In other words, a function can be defined as the composition of a **kernel** function, K, with the functions A_i, which have disjoint supports. In general, a function has several disjoint support decompositions, which can be superimposed to obtain decompositions with finer granularity. Moreover, it is possible to recursively search for disjoint support decompositions for the functions A_i to produce even smaller components. At the limit, F can be represented as a tree of functions, with the inputs x_i being the leaves of the tree. We call this tree the **decomposition tree**. It has been shown that the maximal decomposition tree (that is, the tree of the finest granularity, where each node cannot be further decomposed) for a function is unique and independent of the variable ordering [4, 8].

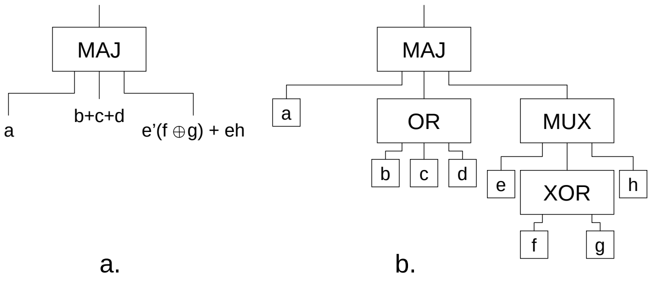

Figure 1: Decompositions for Example 1: a) A simple decomposition and b) The complete decomposition tree.

Example 1. Function $F = \mathrm{MAJ}(a, b+c+d, \overline{e}\overline{f}g+\overline{e}f\overline{g}+eh)$ can be decomposed as in Figure 1.a) where the kernel function is MAJ and there are three disjoint components $A_1 = a$, $A_2 = b + c + d$ and $A_3 = \overline{e}\overline{f}g + \overline{e}f\overline{g} + eh$. By decomposing recursively the functions A_i, we obtain a decomposition tree as in Figure 1.b). □

Note that the leaves of the tree are the input variables of the function, thus the number of internal nodes in a decomposition tree is linearly bound by the size of the support of the function, that is the number of input variables. In contrast, a BDD may have a number of internal nodes that is exponential on the size of the support.

Techniques for finding disjoint decompositions have been studied for the past 40 years. Traditional approaches are based on using decomposition tables to partition the inputs into disjoint sets as in [4, 20], while more recent techniques proceed by computing the DSD starting from a BDD representation [21, 8, 22]. In [8], the authors define a canonical

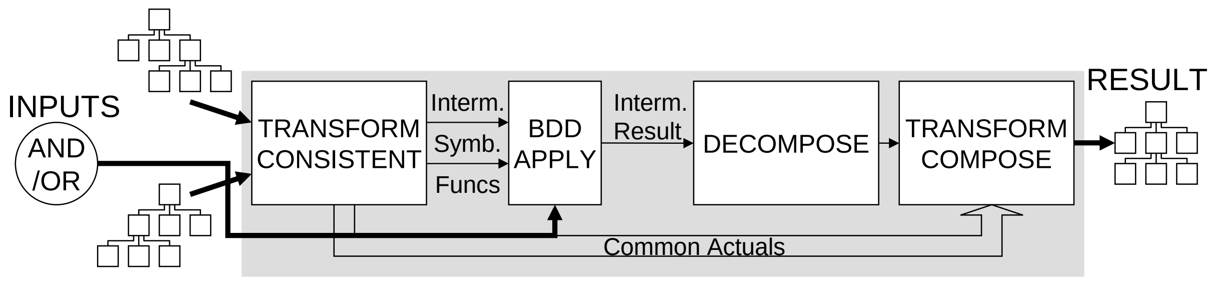

Figure 2: The DecO algorithm. Two input DSDs are expanded to have consistent root nodes. The Boolean operation is then applied to these nodes. Finally, the result is decomposed and the lower part of the tree is reconnected.

form for decomposition trees, and present an exact algorithm that constructs the decomposition tree by traversing a BDD bottom-up and has quadratic complexity on the size of the BDD. When considering each internal BDD node representing a function F, if the decompositions of the two cofactors F_0 and F_1 are known, the decomposition of F can be constructed by selecting one of a set of construction rules based on the decomposition characteristics of the cofactors. If the application domain requires that Boolean operations be performed between DSDs, the traditional approach requires that the DSDs are transformed back to BDD form, an operation that requires applying Boolean function composition at each internal node of the decomposition tree, which also has quadratic complexity. In the next section we present an algorithm that allows the application of Boolean operators directly on the DSD of a function, thus avoiding the transformations to and from BDD.

Anatomy of a DSD node.

Nodes in a decomposition tree can be of two types, based on the type of kernel function: **associative** (that is, AND, OR, XOR) and **prime**. Prime functions are functions that cannot be decomposed further into disjoint components. For example, *majority* and *multiplexor* functions cannot be divided any more. It follows that prime decompositions always have more than two inputs since all two-input functions are trivially associative operators. Each block in a decomposition tree maintains all the relevant information: its type, the list of its component functions, that is, its **actuals list**, and a symbolic representation of its specific kernel function, called **symbolic kernel**.

Example 2. Consider the Boolean function rooted in the top block of Figure 1. The top block, which is a MAJ function, is a prime decomposition type. The inputs, a, $OR(b, c, d)$, and $MUX(e, XOR(f, g), h)$ make up the actuals list for the top block. Finally, the symbolic kernel is a 3-input majority function where a different variable is used to represent a, $OR(b, c, d)$, and $MUX(e, XOR(f, g), h)$. □

Note that there are in general several ways to decompose an n-input associative operator. For instance, with reference to Figure 1.a), $b + c + d$ block could be decomposed as $\mathrm{OR}(b, \mathrm{OR}(c, d))$ or $\mathrm{OR}(b, c, d)$ or $\mathrm{OR}(\mathrm{OR}(b, c), d)$. The decomposition tree is canonical if the maximal-inputs associative operator is always chosen, as in Figure 1.b) [8].

The kernel function of a DSD node is described through symbolic kernels–Boolean functions encoding the specific relation among the actuals lists elements of the node [9]. The variables in the support of a symbolic kernel function have a one-to-one correspondence with the elements in the actuals list of the node. The variables selected to be part of the kernel function are chosen based on optimization criteria of the DSD construction. In this regard, DecO uses the same strategy as in Staccato [9] and selects each kernel variable based on the lowest ranked variable (closest to the constant node in a BDD) of each actuals list element. The kernel is called symbolic exactly because of this mapping between kernel variables and actuals list.

Example 3. The symbolic kernel for the topmost node in the decomposition tree of Figure 1.b, is $K_F = \mathrm{MAJ}(a, d, h)$, where the support set $\{a, d, h\}$ (the lowest ranked variable in each actuals list member) has a one-to-one correspondence with the actuals list elements, $\{a, OR(...), MUX(...)\}$. □

3. BOOLEAN MANIPULATION OF DSDS

We present our novel algorithm, DecO, to apply Boolean operators to Disjoint Support Decomposition forms. DecO takes as input two functions represented by DSDs and produces a resulting DSD that is a canonical maximal decomposition tree, and represents the result function of the Boolean operation. The complexity of the algorithm is quadratic on the size of the symbolic kernels of the operands involved in the computation, which are commonly much smaller functions than the complete BDDs of the operands. DecO does not require a complete transformation from a DSD representation to a BDD representation to perform Boolean manipulations. Thus, applications that benefit from the ability to manipulate DSD representations for run-time optimizations can exploit DecO to their advantage. In the remainder of this section, we first overview the DecO flow and then discuss each phase of the algorithm in detail.

3.1 The DecO Flow

The general algorithm in DecO for symbolic Boolean manipulation is outlined in Figure 2. Each step of the algorithm will be explained in detail in the following sections. In general, the algorithm takes as input two decomposition trees and a Boolean operator. Each input DSD is transformed into a non-maximal decomposition so that the root symbolic kernels store non-intersecting actuals lists. The Boolean operation is then applied to these two root kernels, and the result is decomposed using the algorithm in [9]. The resulting decomposition goes through another transformation to reconnect the original actuals and produce the final resulting DSD. The general idea in DecO is not to eliminate

BDD manipulation completely, but to perform manipulation on the smallest possible BDDs. DecO is able to produce smaller BDDs through symbolic kernels by exploiting the inherent decomposability of Boolean functions. The efficiency of DecO is dependent on the intermediate BDDs and the smaller these are, the faster the algorithm execution. For completely undecomposable functions, the intermediate BDDs themselves are often undecomposable, and thus DecO performs manipulation on complete BDDs.

3.2 Constructing Consistent Decompositions

The first phase of DecO is `TRANSFORM CONSISTENT` as shown in Figure 2. The general goal is to generate consistent root DSD nodes that will be the inputs for the second phase, `BDD APPLY`. Although the input DSDs already have root symbolic kernels, these kernels might not be the proper root functions for the second phase. In fact, it is possible for the root symbolic kernels of two distinct functions to share a common symbolic variable, which represents distinct actuals elements in the two kernels. If we were to apply a Boolean operation directly on the kernels, the result would not be correct. The underlying reason for this type of situation is that, while all the components of a DSD are disjoint among each other, the two input operands of DecO may share parts of their support. The objective of `TRANSFORM CONSISTENT` is to expand the root kernels until all the actuals list elements are disjoint functions. The next two sections will describe this process.

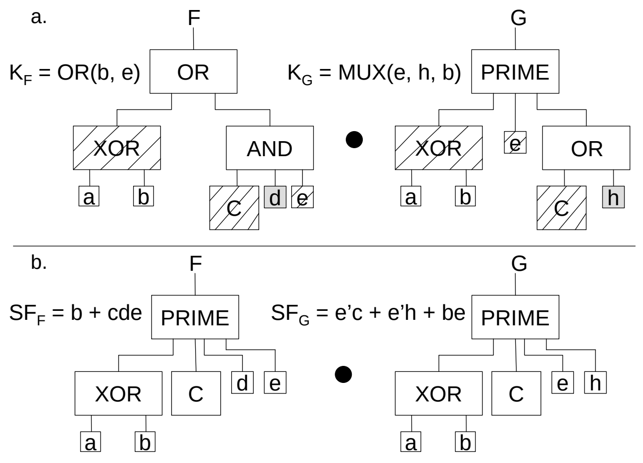

Figure 3: a) Decomposition trees of two input functions for a Boolean operation. b) Their corresponding pseudo-decompositions after applying `TRANSFORM CONSISTENT`.

Consistent Variable Mapping

Consider the example of Figure 3.a) that illustrates how this transformations takes place, for readability purposes, the symbolic kernels of the internal nodes are not specified but can be readily deducted. We assume that the variable order is lexicographic with a at the highest rank. Moreover, we assume that a function represented by an upper-case letter like C has variable c as its lowest rank support variable. Notice that the symbolic kernels, K_F and K_G contain variables that are common to both BDDs. However, variable, e, symbolically maps to Cde in F and to variable e in G. In this figure, C is an arbitrary function with support disjoint from all the other blocks in the diagram. In addition, variable h in G symbolically maps to function $C + h$, whose support intersects with the function mapped to e in F.

The first phase of DecO transforms both operand functions so that the symbolic kernels at the root of the decomposition have support variables which are consistent, that is, there is 1) a unique function corresponding to each distinct support variable, and 2) there is no intersection among the support of the actuals list elements. An example is shown in Figure 3.b), where F and G are no longer DSDs, but pseudo-decompositions. The functions of the root tree nodes are called symbolic functions.

Definition 1. *A* **pseudo-decomposition** *of F is a disjoint support decomposition that may or may not be further decomposed.*

The maximal disjoint support decompositions described in Section 2 are a special case of pseudo-decompositions. Likewise, symbolic kernels are a special case of symbolic functions. We use the symbol SF_F to refer to the symbolic function for a pseudo-decomposition of F.

In general, many distinct pseudo-decompositions are possible for any pair of Boolean functions. Among this set, we choose the finest granularity pseudo-decomposition which produces a consistent variable mapping.

Definition 2. *Given two symbolic functions SF_1 and SF_2 and their actuals list AL_1 and AL_2, SF_1 and SF_2 have a* **consistent variable mapping** *iff: $\forall f_i \in AL_1$ and $\forall f_j \in AL_2$ one of the following two conditions hold:*

1. $f_i = f_j$
2. $\mathcal{S}(f_i) \cap \mathcal{S}(f_j) = \emptyset$.

In other words, SF_1 and SF_2 have a consistent variable mapping iff each actuals list element of SF_1 is either identical or disjoint from each element of SF_2.

As shown in Figure 3.b), after `TRANSFORM CONSISTENT` each variable maps to a distinct disjoint component. This mapping enables us to perform Boolean operations with SF_F and SF_G.

Finding Consistent Variable Mappings

The algorithm described briefly below takes two decompositions and produces two pseudo-decompositions that have a consistent variable mapping while retaining as much of the original structures as possible. The consistent variable mapping algorithm traverses both DSD trees and identifies two types of blocks: common blocks and exclusive blocks. This phase of DecO has linear complexity on the size of the decomposition tree. With reference to the example in Figure 3, common blocks are represented by hashed blocks.

Definition 3. *A* **common block** *is a decomposition node that occurs in both decompositions and whose parent node is unique among all the nodes in the two decomposition trees.*

With reference to Figure 3.a), by traversing the decomposition tree, one can identify that $a \oplus b$, C, e are blocks common to both decomposition. Notice, that a and b occur in both decompositions but are not *common blocks* because their parent node is common to both decompositions. The second type of blocks are called **exclusive blocks** and are the blocks that are shaded in the figure.

Definition 4. *Given two decomposition trees F and G, an* **exclusive block** *in G is a decomposition node whose support is disjoint from the support of F and whose parent node is not disjoint from F. Similarly, an exclusive block in F is disjoint from G.*

In Figure 3.a), variable h is in the decomposition of G and it has disjoint support from F, hence h is an exclusive block (for notational convenience, input variables to the decomposition trees will be considered nodes in this algorithm). The common and exclusive blocks identified in this manner constitute the new actuals list components of the pseudo decompositions that we are going to build. Pseudo-decompositions are then constructed simply by applying function composition operations to the symbolic kernels that are above the selected actuals list. In Figure 3.b), the root nodes of the pseudo-decomposition are indicated by SF_F and SF_G.

3.3 Manipulating Symbolic Functions

The second phase of the algorithm in Figure 2, `BDD APPLY`, takes two BDDs and a Boolean operator, such as AND or OR, and produces an intermediate result. The input BDDs are the intermediate symbolic functions produced by `TRANSFORM CONSISTENT`. The result is obtained by performing BDD operations as defined in [1]. The complexity of this phase is quadratic with respect to the size of the BDDs.

Example 4. If we define the function J to be the result of performing the AND of SF_F and SF_G from the example in Figure 3, the output of `BDD APPLY` would be the BDD representation for $J = (b + cde)(be + e'c + eh) = bce' + be'h + be$. $\square$

Notice that J treats c and b like variables despite the fact that they symbolically map to larger functions. Thus, when the pseudo-decompositions have many large disjoint components, the BDD manipulation is done on corresponding symbolic functions that are much smaller than the BDDs that are represented by the pseudo-decompositions.

3.4 Decomposition

The third phase of Figure 2, `DECOMPOSE`, takes the intermediate BDD result from `BDD APPLY` and decomposes that BDD using the algorithm in [8]. The algorithm for decomposition has complexity that is quadratic with respect to the size of the intermediate BDD result. Figure 4.a) illustrates the decomposition tree for the intermediate result J, that was calculated in the previous section The symbolic kernel, K_J, is also given.

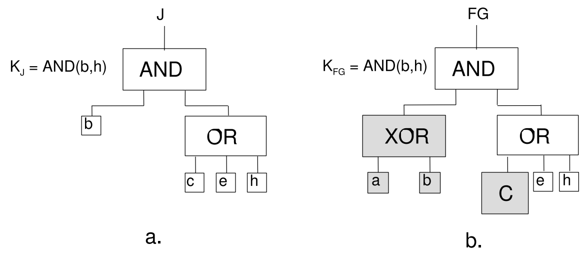

Figure 4: a) The decomposition for the intermediate function J. b) The resulting decomposition of $F \cdot G$ after `TRANSFORM COMPOSE`.

As with the previous phase, the efficiency of `DECOMPOSE` is improved when the pseudo-decompositions contain several large disjoint components. If this occurs, the decomposition algorithm is performed on an intermediate BDD that is much smaller than the BDD created without exploiting the inherent decomposition structure.

3.5 Composing

The final phase of Figure 2, `TRANSFORM COMPOSE`, takes the decomposition tree of J and the common actuals elements generated by `TRANSFORM CONSISTENT` and produces the final result, that is, the decomposition for $F \cdot G$. In order to obtain a canonical decomposition, it is now sufficient to connect together the decomposed function obtained from `DECOMPOSE` with the actuals list elements that we generated in the first phase. Each variable in the support of the `DECOMPOSE` output has a mapping to a distinct function as found in `TRANSFORM CONSISTENT`. The result, for example in Figure 3, is shown in Figure 4.b).

Sometimes, the result of composing the actuals list members into the decomposed function obtained from `DECOMPOSE` is a non-maximal solution. This situation may arise when `TRANSFORM CONSISTENT` breaks down associative operators. However, the algorithm to regenerate maximally expanded associative operator nodes is a simple algorithm whose complexity is linearly related to the size of the decomposition tree which is in turn linearly related to the number of support variables.

4. EXPERIMENTAL RESULTS

The purpose of this section is to demonstrate the effectiveness and efficiency of DecO. In particular, we show that DecO consistently achieves memory improvement over previous techniques that build DSDs from initial BDDs by utilizing the inherent structure in Boolean functions. Furthermore, our results indicate that DecO has the potential to be a compact alternative to BDDs, which is especially important for memory-limited applications. In addition to achieving a low memory profile, we show that DecO is a much more efficient solution in terms of run-time performance than previous solutions manipulating complete BDDs and then decomposing the solution. Finally, our results illustrate that DecO still has room for optimization and suggest opportunities for future research.

We implemented DecO and ran experiments on a large set of testbenches. For each experiment, we compute the DSD representation for each output using the DecO algorithm. The experiments were run on a 3.2GHz Pentium 4 processor with 1GB of memory. DecO uses the CUDD package [23] to perform the BDD operations required in `BDD APPLY`. Finally, DecO uses the algorithm defined in [9] to perform the decomposition required in `DECOMPOSE`.

4.1 DecO Statistics

The circuits tested by DecO are given in Tables 1 and 2. The testbenches are from ISCA-LogicSynthesis, IWLS, VIS suites, and the Data Cache Unit from Sun Microsystems' picoJava processor [24]. The VIS circuits were generated by using vl2mv, a tool within VIS [25]. For the sequential circuits, we considered only their combinational portions by removing latches. The Data Cache Unit was produced by synthesizing and flattening the netlist with DesignCompiler by Synopsys.

Table 1 shows circuits that have limited inherent decomposability, while Table 2 gives results for circuits that are very decomposable. DecO is implemented to achieve memory savings and run-time improvements for decomposable circuits while not suffering degradations for circuits with little decomposability. Both tables show various properties of the circuits tested along with memory statistics and runtime information for DecO.

The first five columns report circuit statistics: name of the circuit, number of primary inputs and outputs. The *dec.out* column represents the number of outputs in the circuit that are decomposable. An output is considered undecomposable when the decomposition tree is one single block and the actuals list consists of only primary inputs. The next column, *blocks*, gives the number of nodes in the decomposition trees of the circuit. In general, a circuit with few blocks will be considered more decomposable than a circuit with a smaller number of blocks. The next column, *%dec.ops*, shows the percentage of Boolean operations `TRANSFORM CONSISTENT` does not completely flatten the decomposition. Decomposable circuits tend to have higher percentages in this column. Even for circuits with little decomposability in Table 1, %dec.ops is still greater than zero indicating that many of the intermediate operations required to build the circuit involve decomposable operands. The circuits are ordered according to this percentage. The next three columns represent the amount of memory used by DecO for each circuit. The first two of these columns give the memory required by DecO to represent the BDD of the symbolic kernels and the decomposition tree respectively. The third column gives the total memory required by DecO. Notice that the amount of memory required to represent the decomposition tree is considerably less than that required to represent the symbolic kernels. The final column gives the execution time for DecO in seconds. This is the amount time needed to build the circuit using the DecO algorithm. The result is a disjoint support canonical representation for each output of the circuit.

The following two sections will give perspective to these results by showing relative memory improvement compared to Staccato [9] and CUDD and by showing speedup when compared to a naive scheme for performing Boolean operations on DSDs.

4.2 Memory Performance

Figure 5 shows the relative memory savings achieved by DecO when compared to Staccato, a package that generates DSDs after first constructing the complete BDDs. The results illustrate that DecO always requires less or the same amount of memory as Staccato. Since the decomposition trees of both algorithms use the same amount of memory, the difference in results is due to the compact representation of symbolic kernels over BDDs. This demonstrates the advantages of utilizing inherent structure in Boolean functions to reduce BDD memory requirements.

The difference between the memory requirements of DecO and Staccato is indicative of the degree to which decompositions allow DecO to eliminate BDD storage. Thus, for the more decomposable circuits, DecO tends to achieve substantial memory savings requiring less than half the memory in four circuits. When there is little decomposability, DecO does not require more memory than Staccato. The potential of DecO can be further illustrated by comparing the

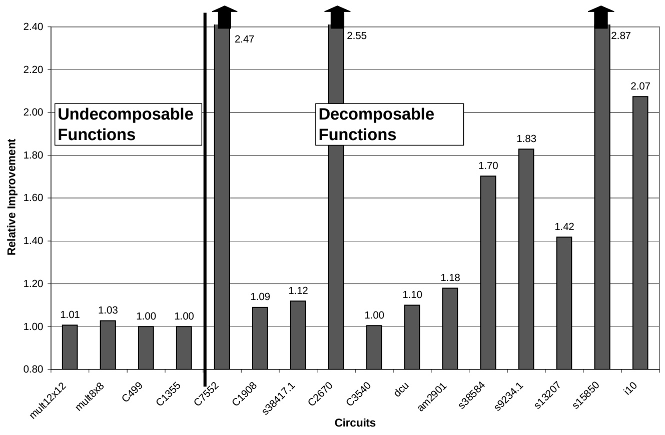

Figure 5: Memory improvement of DecO compared to Staccato.

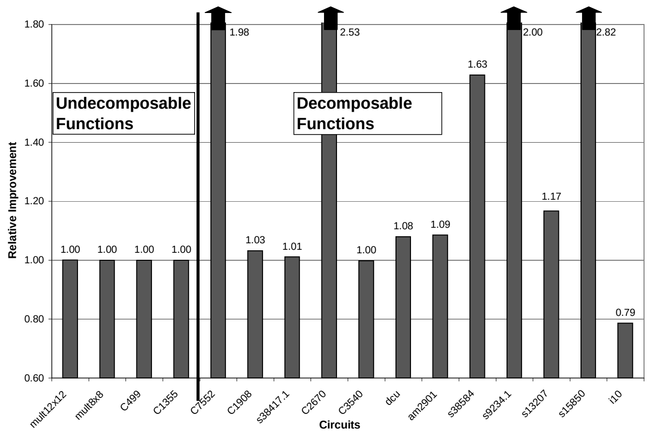

Figure 6: Memory improvement of the symbolic kernel component of DecO compared to CUDD.

total memory required to represent all the symbolic kernels in DecO to the BDD memory required in CUDD. For this analysis we do not take into account the decomposition tree memory since we showed in Table 1 and 2 that it is an insignificant fraction of the overall memory.

When comparing CUDD to the symbolic kernel components of DecO, consistent memory savings are found as shown in Figure 6. The notable exception is circuit i10, where there is practically no overlap between the decompositions of each output function, while the CUDD outputs find many sharing opportunities. Note that, undecomposable circuits have identical memory requirements to CUDD since the symbolic kernel is equal to the complete BDD. DecO utilizes the decomposable structure of Boolean functions through symbolic kernels to provide a compact representation. The complete BDDs do not take advantage of this structure present in circuits. Because several applications that require BDDs often experience memory explosion, using DecO to represent Boolean functions instead of BDDs could provide an effective compact alternative.

4.3 Runtime Performance

To analyze the efficiency of the DecO algorithm, we compared our approach to a naive approach that simply takes the input DSDs, generates a complete BDD, performs the

Circuit	in	out	dec.out	blocks	%dec.ops	DecO(KB)			DecO(s)
						S.Ker	Dec.Tree	Total	
mult12x12	24	24	4	29	16	9573	4	9577	534.90
mult8x8	16	16	4	21	26	145	3	148	1.29
C499	41	32	0	32	42	427	12	439	103.85
C1355	41	32	0	32	47	458	12	470	110.72

Table 1: Statistics for circuits with limited decomposability.

Circuit	in	out	dec.out	blocks	%dec.ops	DecO(KB)			DecO(s)
						S.Ker	Dec.Tree	Total	
C7552	207	108	107	518	52	151	23	174	0.87
C1908	33	25	7	94	57	93	7	100	8.87
s38417.1	1494	1571	1313	6153	61	5048	292	5340	197.30
C2670	233	140	119	453	62	57	7	64	0.91
C3540	50	22	14	56	63	444	6	450	8.85
dcu	700	265	231	3118	71	5976	46	6022	275.98
am2901	95	150	138	280	73	6077	49	6126	98.62
s38584	1464	1730	1611	11761	74	175	192	367	9.74
s9234.1	172	174	169	1275	77	17	18	35	0.15
s13207	700	790	783	2787	79	42	49	91	1.21
s15850	611	684	651	8877	83	101	76	177	11.26
i10	257	224	224	2241	87	1601	51	1652	19.80

Table 2: Statistics for highly decomposable circuits.

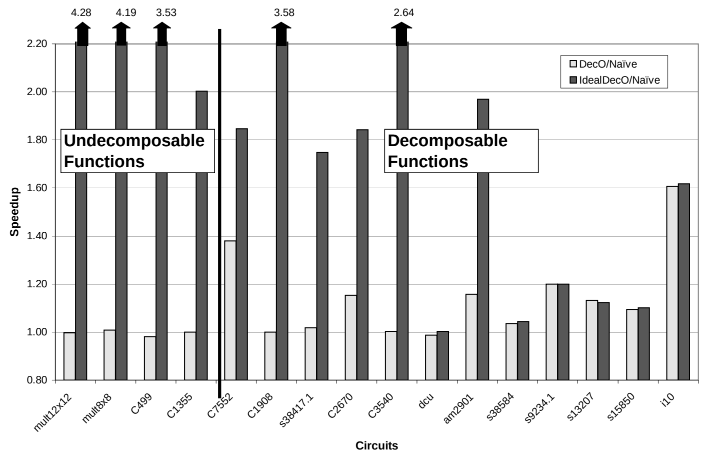

Figure 7: The light colored-bar shows the speedup of DecO over a naive approach that uses CUDD and Staccato. The dark-colored bar shows the potential speedup of DecO by showing the speedup of a theoretically optimal version of DecO, IdealDecO, compared to the naive approach.

Boolean operation on the complete BDDs using CUDD, and decomposes the result with Staccato. The light-colored bar in Figure 7 illustrates the speedup DecO achieves compared to this naive approach. As with the memory results, the greatest improvements occur for functions that are decomposable. In these situations, the intermediate functions are much smaller in DecO than in the naive approach. Hence, `BDD APPLY` and `DECOMPOSE` execute more efficiently. Also, DecO performs similarly to the naive approach for undecomposable circuits, which illustrates that the overhead involved with `TRANSFORM CONSISTENT` and `TRANSFORM COMPOSE` is minimal.

On the other hand, Tables 1 and 2 illustrate that not all operations performed are decomposable and thus do not benefit from the DecO algorithm. To illustrate the full potential of DecO, we compared the naive algorithm to a theoretical version of DecO, IdealDecO, shown in Figure 7 and illustrated by the dark-colored bar. This ideal version only uses DecO when the operation is decomposable as defined in Section 4.1. Otherwise it just uses CUDD and does not generate a DSD at those intermediate points. The result is a much more efficient solver that performs much better on undecomposable cases and also on decomposable cases while maintaining approximately the same memory profile as the original version of DecO.

Although IdealDecO was not implemented as a general strategy as it is very difficult to always know whether an operation will be decomposable ahead of time, IdealDecO provides a useful realistic goal to reach. This could motivate further research in developing heuristics in DecO that turn off the DecO algorithm when the rate of undecomposable operations reaches a certain threshold.

5. CONCLUSIONS

We presented a novel algorithm that supports Boolean operations directly on Disjoint Support Decompositions (DSDs) without necessarily requiring a transformation of DSDs back to BDDs. The algorithm produces a representation for a function that is more compact than previous DSD representations (like Staccato) and is often more compact than using BDDs for highly decomposable functions. Furthermore, these operations can be performed relatively efficiently especially for decomposable functions. Moreover, our results indicate that there is still room for more run-time improvement in DecO by heuristically disabling the DecO algorithm for operations that involve operands with no decomposability. The algorithm developed provides a more descriptive representation of the functions involved in the Boolean manipulation than maintaining complete BDDs and can therefore be useful for many application areas that benefit from the decomposition properties of Boolean functions, such as technology mapping and area optimizations in synthesis. In addition, DecO could provide a compact alternative to traditional BDD manipulation in the verification domain where memory is often the limiting factor.

6. REFERENCES

[1] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[2] Randal E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40:205–213, 1991.

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

[4] Robert L. Ashenhurst. The decomposition of switching functions. In *Proceedings of the International Symposium on the Theory of Switching*, Part I 29, pages 74–116, 1957.

[5] Robert K. Brayton and Curt McMullen. The decomposition and factorization of boolean expressions. In *ISCAS, Proceedings of the International Symposyium on Circuits and Systems*, pages 49–54, 1982.

[6] Rajeev Murgai, Yoshihito Nishizaki, Narendra V. Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *DAC, Proceedings of Design Automation Conference*, pages 620–625, June 1990.

[7] Tsutomu Sasao and Munehiro Matsuura. DECOMPOS: An integrated system for functional decomposition. In *International Workshop on Logic Synthesis*, pages 471–477, 1998.

[8] Valeria Bertacco and Maurizio Damiani. The disjunctive decomposition of logic functions. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 78–82, November 1997.

[9] Stephen Plaza and Valeria Bertacco. Staccato: Disjoint support decompositions from bdds through symbolic kernels. In *ASP-DAC, Asia and South Pacific Design Automation Conference*, January 2005.

[10] Kevin Karplus. Using if-then-else dags to do technology mapping for field-programmable gate arrays. Technical Report UCSC-CRL-90-43, Baskin Center for Computer Engineering & Information Sciences, 1990.

[11] Tsutomu Sasao. Multiple-valued decomposition of generalized boolean functions and the complexity of programmable logic arrays. *IEEE Transactions on Computers*, C-30(9):635–643, September 1981.

[12] Thomas Kutzschebauch and Leon Stok. Layout driven decomposition with congestion consideration. In *DATE, Design, Automation and Test in Europe Conference*, pages 672–676, March 2002.

[13] Per Lindgren, Mikael Kerttu, Mitch Thornton, and Rolf Drechsler. Low power optimization technique for BDD mapped circuits. In *ASP-DAC, Asia and South Pacific Design Automation Conference*, pages 615–621, 2001.

[14] Valeria Bertacco, Sin-Ichi Minato, Peter Verplaetse, Luca Benini, and Giovanni De Micheli. Decision diagrams and pass-transistor logic synthesis. In *International Workshop on Logic Synthesis*, 1997.

[15] Valeria Bertacco and Kunle Olukotun. Efficient state representation for symbolic simulation. In *DAC, Proceedings of Design Automation Conference*, June 2002.

[16] Gianpiero Cabodi, Paolo Camurati, Luciano Lavagno, and Stefano Quer. Disjunctive partitioning and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In *DAC, Proceedings of Design Automation Conference*, pages 728–733, June 1997.

[17] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[18] Alan Hu. Formal hardware verification with BDDs: An introduction. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 677–682, 1997.

[19] Randal E. Bryant. Symbolic boolean manipulation with ordered binary–decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[20] V. Yun-Shen Shen, Archie C. McKellar, and Peter Weiner. A fast algorithm for the disjunctive decomposition of switching functions. *IEEE Transactions on Computers*, C-20(3):304–309, 1971.

[21] Kevin Karplus. Using if-then-else dags for multi-level logic minimization. In *Proceedings of Advanced Research in VLSI*, pages 101–118, 1989.

[22] Yusuke Matsunaga. An exact and efficient algorithm for disjunctive decomposition. In *SASIMI*, pages 44–50, October 1998.

[23] CUDD-2.3.1. *http://vlsi.Colorado.edu/~fabio.*

[24] Sun Microsystems. PicoJava technology. *http://www.sun.com-/microelectronics/communitysource/picojava.*

[25] The VIS Group. VIS: A system for verification and synthesis. In *International Conference on Computer Aided Verification*, pages 428–432, July 1996.