Boolean Decomposition for AIG Optimization

Lucas Machado
Department of Computer Science
Universitat Politècnica de Catalunya
Barcelona, Spain
lmachado@cs.upc.edu

Jordi Cortadella
Department of Computer Science
Universitat Politècnica de Catalunya
Barcelona, Spain
jordi.cortadella@upc.edu

ABSTRACT

Restructuring techniques for And-Inverter Graphs (AIG), such as rewriting and refactoring, are powerful, scalable and fast, achieving highly optimized AIGs after few iterations. However, these techniques are biased by the original AIG structure and limited by single output optimizations. This paper investigates AIG optimization for area, exploring how far Boolean methods can reduce AIG nodes through local optimization. Boolean division is applied for multi-output functions using two-literal divisors and Boolean decomposition is introduced as a method for AIG optimization. Multi-output blocks are extracted from the AIG and optimized, achieving a further AIG node reduction of 7.76% on average for ITC99 and MCNC benchmarks.

Keywords

Boolean decomposition; AIG; KL-cuts; local optimization

1. INTRODUCTION

Logic synthesis is a key process in design automation, generating an optimized netlist of logic gates from an RTL representation, and it is often divided in two classes: technology independent and technology dependent [6]. Recently, technology independent algorithms using AIGs have been proposed, enabling efficient and scalable optimizations [10].

Restructuring methods such as refactoring [10], rewriting [13], and balancing [5] are powerful, obtaining highly optimized AIGs after few iterations. Still, these techniques are usually constrained by single output transformations, and iterations with technology mapping [7, 12] are often used to improve structurally biased results.

AIG rewriting and refactoring perform local transformations, extracting the local context with K-cuts [16], windows or maximum fanout-free cones (MFFCs). K-cuts can be considered a superior method to extract local context compared to windowing [7, 10], as it is possible to *control the support* of the Boolean functions to be optimized, while identifying a region of the circuit that *depends on this support*.

Algorithms based on K-cut enumeration have been proposed, such as factor cuts [4] and priority cuts [14], reducing the search space and enabling cuts with more nodes and inputs. Also, *multi-output* blocks based on K-cuts were presented [8, 9], which extract the *complete local context*.

This paper studies technology-independent transformations that reduce the AIG size by exploring the use of Boolean decomposition. This is done by expanding the idea of two-literal divisors [15] to multi-output functions (see Fig. 1).

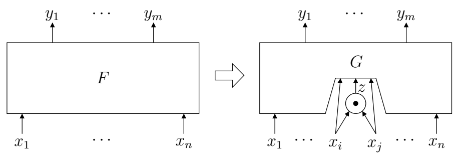

Figure 1: Decomposition using two-literal Boolean divisors.

The principle is as follows. A multi-output function

$$(y_1, \dots, y_m) = F(x_1, \dots, x_n)$$

can be decomposed into another multi-output function

$$(y_1, \dots, y_m) = G(x_1, \dots, x_n, z)$$

with z being a two-literal divisor ($z = x_i \odot x_j$) and $\odot$ being a Boolean operator (such as an AIG node). G is supposed to be a simpler function than F, which can be obtained by Boolean division. A multi-output Boolean function can be recursively decomposed using this paradigm, and the result can be represented as an AIG network.

The main purpose of this paper is to explore how far Boolean decomposition can go beyond the existing AIG rewriting methods. Unfortunately, scalability is an important issue when dealing with Boolean methods. Obviously, collapsing large networks into one-node functions and then decomposing is not computationally affordable.

This paper takes a significant leap forward regarding [15]:

- Boolean decomposition with two-literal divisors is generalized to be applicable to netlists with multiple outputs, instead of individual single-output functions.
- The selection of divisors is customized to increase the logic shared among multiple outputs.
- A set of filters to reduce the search space is presented.
- Scalability is addressed by iteratively applying Boolean decomposition to KL-cuts [9] of the AIG (see Fig. 2).

Note that a KL-cut represents a portion of the circuit that depends on the same set of variables. The *key idea* is to use this property of KL-cuts to identify divisors that are useful for several functions, sharing more logic and reducing area.

In [11], a resynthesis method that uses Boolean decomposition is performed on netlists of FPGA LUTs, by identifying and decomposing MFFCs. However, [11] does not perform a complete Boolean decomposition, as it is limited to a simplified version of Disjoint Support Decomposition. In [17], windows are enumerated and don't cares identified in the network are used to simplify these windows, while in [18] the algebraic decomposition method is improved by assigning previously calculated don't cares using a set of rules. In this work, there is neither search nor previous calculation of don't care conditions, as the optimization using don't cares occurs solely inside the KL-cut logic.

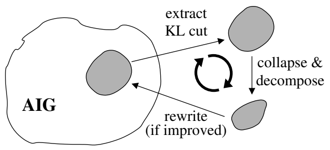

Figure 2: Iterative rewriting of AIG cuts.

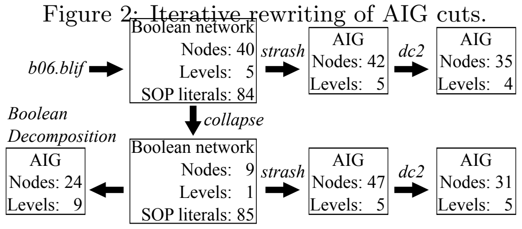

Figure 3: Optimization flow using different methods for *b06*.

The results reported in the paper have been often obtained by applying computationally intensive methods (e.g., many divisors, many cuts). Bear in mind that our goal now is to establish the bounds potentially reachable by future work that could use smart *oracles* to drive the search. Some preliminary criteria are discussed in the paper. Still, the method proposed in this paper can be interesting for highly repetitive structures or area-critical components.

2. MULTI-OUTPUT DECOMPOSITION

Boolean decomposition is known to obtain good-quality results at the expense of a high computational cost. Finding good divisors is the most challenging task. Different approaches can be proposed to prune the search, e.g., use two-literal divisors [15], consider polarity information to ignore unpromising divisions, or reuse algebraic factored forms to select divisors. Considering all these simplifications, is it still possible to obtain good results?

To illustrate the method, consider the set of functions and its DC-set in (1). Any combination of two literals of F can be selected as a Boolean divisor. The number of literals is defined as the cost function, therefore the cost of F is 12.

$$F = \begin{cases} f_1 = b\cdot x + \bar{c}\cdot\bar{x} \\ f_2 = a\cdot x + d\cdot\bar{x} \\ f_3 = a\cdot x + b\cdot e \end{cases} \quad DC = \bar{a}\cdot b\cdot x + a\cdot\bar{b}\cdot x + a\cdot b\cdot\bar{x} \quad (1)$$

A reduction of 3 literals is achieved by performing a multi-output Boolean division using the divisor $y = a\cdot b$, generating the new set of functions and DC-set in (2). Note that this divisor is not easily extracted from the set of functions F: f_1 does not have the variable a and f_2 does not have the variable b. Still, all functions are reduced in 1 literal due to the effective use of the DC-set.

$$F_{new} = \begin{cases} f_1' = y + \bar{c}\cdot\bar{x} \\ f_2' = y + d\cdot\bar{x} \\ f_3' = y + b\cdot e \end{cases} \quad DC_{new} = DC + (y \oplus (a\cdot b)) \quad (2)$$

For further decomposition steps, it is possible to perform a DC-set projection, removing variables from the DC-set that are not in the support of F, and decreasing the computational effort of the Boolean division. By projecting DC_{new} to the support of F_{new}, $DC_{proj} = \bar{b}\cdot y$ is generated.

2.1 AIG optimization example

In order to demonstrate the potential benefits of the approach introduced in this paper, the circuit *b06* of ITC99 benchmarks suite [1] is used. The optimization flow and the results for the different solutions are depicted in Fig. 3.

The input circuit is a Boolean network represented in a BLIF file. An AIG with 42 nodes is obtained after decomposing the Boolean network with algebraic factorization and structural hashing (*strash* command in ABC [3]). After iteratively applying algebraic transformations using *dc2* command in ABC, the number of nodes is reduced to 35.

An alternative approach would start by collapsing the initial network, which results in a Boolean network with one node for each output. Decomposing these nodes results in a larger AIG with 47 nodes, but applying iterative algebraic transformations reduces it to 31 nodes.

The method proposed in this paper applies Boolean decomposition on top of the collapsed network using an approach inspired by [15]. In this work, multi-output decomposition is used by iteratively selecting the Boolean divisor that minimizes the literals of the functions factored forms [2]. This approach is able to achieve a better logic sharing, obtaining an AIG with only 24 nodes.

Also, a set of filters is applied to reduce the search space of divisors. By reducing the amount of two-level minimizations compared to [15], runtime is reduced 40 times on average, without sacrificing the quality of the results. Note that the results in this work cannot be directly compared with [15], which only presents the decomposition of single-output functions and it is not applied for circuit optimization.

3. BACKGROUND

3.1 Functions, unateness and containment

A completely specified Boolean function f is a mapping from an n-dimensional ($n \geq 0$) into a 1-dimensional Boolean space: $\{0,1\}^n \to \{0,1\}$. The positive (negative) *cofactor* $f|_{x=1}$ ($f|_{x=0}$) of f with respect to the variable x is a function obtained by assigning x to one (zero) in f.

A Boolean function f is positive (negative) unate in the variable x if $f|_{x=1} \supseteq f|_{x=0}$ ($f|_{x=0} \supseteq f|_{x=1}$), where $\supseteq$ is the set operation for inclusion. Otherwise, f is considered binate in x. This is the concept of *unateness* [6], intended for completely specified functions.

An incompletely specified function (ISF) g is a mapping from an n-dimensional ($n \geq 0$) into a 1-dimensional Boolean space: $\{0,1\}^n \to \{0,1,*\}$, where $*$ denotes a *don't care* value. The subdomains of g that evaluate to 1, 0 and $*$ are the ON-set, OFF-set and DC-set, and can be represented by the completely specified functions g_{on}, g_{off} and g_{dc}.

Containment [19] is a generalization of the concept of unateness for ISFs. The variable x in the positive polarity is *contained* in g if

$$(g_{dc}|_{x=1} \cup g_{on}|_{x=1}) \supseteq g_{on}|_{x=0}$$

and in the negative polarity if

$$(g_{dc}|_{x=0} \cup g_{on}|_{x=0}) \supseteq g_{on}|_{x=1},$$

where the operators $\cup$ and $\supseteq$ are the set operations of union and inclusion, respectively.

3.2 And-Inverter Graphs and K-cuts

An *And-Inverter Graph* is a directed acyclic graph where each node has either 0 incoming edges - the *primary inputs* (PI) - or 2 incoming edges - the AND nodes. Each edge can be negated or not. Some nodes are also *primary outputs* (PO). Sequential elements are considered as PI/PO pairs. An AIG example is depicted in Fig. 4. The dotted lines are negated edges, the circles are AND nodes, the rectangles at the bottom are PIs and at the top are POs.

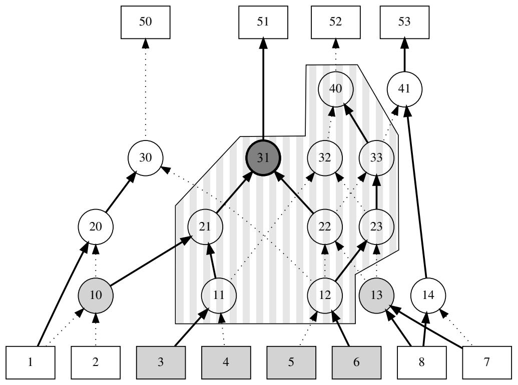

Figure 4: Example of KL-cut computation.

A *cut* of a node n in a graph G is a set of nodes c such that every path between a PI and n contains a node in c. A cut is said to be irredundant if no subset of it is also a cut. A *K-cut* [16] of a graph G is an irredundant cut of K or fewer nodes. Consider the two sets of cuts A and B and the auxiliary set operation $\bowtie$ described in (3).

$$A \bowtie B \equiv \{a \cup b \mid a \in A, b \in B, \mid a \cup b \mid < K\} \qquad (3)$$

The $\bowtie$ set operation removes the redundant cuts, and it is commutative, as the union set operation $\cup$ is also commutative. Let $\Phi_K(n)$ to be the set of K-cuts of $n \in G$ and, if n is an AND node, let n_1 and n_2 to be its inputs. Then, $\Phi_K(n)$ is defined recursively [4], as described in (4).

$$\Phi_K(n) = \begin{cases} \{\{n\}\}, & n \text{ is a PI} \\ \{\{n\}\} \cup \{\Phi_K(n_1) \bowtie \Phi_K(n_2)\}, & \text{otherwise} \end{cases} \qquad (4)$$

3.3 KL-cuts

K-cuts can be an efficient way to represent a graph region regarding a single output. However, several K-cuts may be necessary to cover regions with multiple outputs, *duplicating logic*. A *KL-cut* [8,9] identifies a multi-output region in order to overcome this issue. A KL-cut is a sub-graph G_{KL} of a graph G with K inputs and L outputs. It is represented as two sets of nodes: the inputs G_K, and the outputs G_L.

If a node n belongs to a path between $n_K \in G_K$ and $n_L \in G_L$, being $n \notin G_K$, then n is contained in G_{KL}. Notice that all nodes in G_L are contained in G_{KL}, and G_{KL} does not contain any node of G_K. In this work, the number of outputs is not restricted in KL-cuts enumeration. Therefore, for every K-cut of a node n, there is a *unique* KL-cut G_{KL}.

The nodes that are part of G_{KL} are identified by traversing forward the graph G from G_K. A node n is part of G_{KL} if at least one of the K-cuts of n is a subset of or equal to G_K. A node of G_{KL} is contained in G_L if it is also a PO, or if it has a fanout to a node not contained in G_{KL}.

KL-cut enumeration example: Figure 4 depicts $G_K = \{3, 4, 5, 6, 10, 13\}$ with its nodes in light gray, which is one of the K-cuts at node 31, in dark gray. The KL-cut G_{KL} is obtained by traversing the AIG forward from G_K, identifying the sub-graph hatched in Fig. 4. Nodes 31 and 40 are also POs, and nodes 12 and 33 have fanout to nodes not contained in G_{KL}, therefore $G_L = \{12, 31, 33, 40\}$ is defined. Note that the logic of G_L nodes can be described as Boolean functions that depend on the same support: the G_K nodes.

3.4 Boolean division

Given the Boolean functions f and d, if it is possible to express f as $f = q \cdot d + r$, where $\cdot$ and $+$ represent the Boolean AND and OR operators respectively, then f can be divided by d, with q and r being the quotient and remainder of the division, respectively. This division operation can be performed by algebraic or Boolean methods.

A common approach to perform Boolean division is using two-level minimizers that accept don't care information [6]. A new variable x is added and the division is performed by adding the satisfiability don't care (SDC) expression $x \oplus d$ to the DC-set of f, where $\oplus$ represent the Boolean exclusive-OR operator, followed by a two-level minimization.

Example: Consider the function $f = \bar{a} \cdot b \cdot c + d \cdot (a + b)$ represented as a 6-literal factored form. It is possible to rewrite f as $f = \bar{a} \cdot b \cdot c + d \cdot x$ by performing algebraic division with the divisor $x = a + b$. Boolean division can be performed by incorporating $x \oplus (a + b)$ in the DC-set and doing a two-level minimization results in $f = \bar{a} \cdot c \cdot x + d \cdot x$, that can be represented as the 4-literal factored form $f = (\bar{a} \cdot c + d) \cdot x$.

Table 1: Results obtained through AIG transformations.

Name	Initial		(a) After *dc2*		(b) After collapse + *dc2*		Diff nodes (a) and (b)
	N	LV	N	LV	N	LV	
b04	546	24	487	21	871	16	78.85%
b05	830	54	459	23	4095	19	792.16%
b06	42	5	35	4	31	5	-11.43%
b07	365	27	331	22	566	22	71.00%
b08	155	20	131	15	119	9	-9.16%
b09	136	12	123	10	197	12	60.16%
b10	180	11	162	9	175	8	8.02%
b11	611	28	452	21	1085	18	140.04%
b12	1002	17	947	14	1399	13	47.73%
b13	261	12	220	11	224	10	1.82%
b14	6069	60	3924	100	-	-	-
b15	8432	65	7030	95	-	-	-
alu4	2654	14	1573	15	625	14	-60.27%
apex2	2960	17	991	17	142	14	-85.67%
bigkey	3081	10	2847	10	3302	10	15.98%
clma	11938	40	4842	38	527	16	-89.12%
diffeq	2575	40	2137	41	-	-	-
ex1010	7681	17	4664	15	2337	14	-49.89%
ex5p	1731	15	928	19	204	8	-78.02%
i10	3675	50	1637	36	-	-	-
misex3	2454	13	1267	15	754	14	-40.49%
pdc	7757	19	3219	19	1717	18	-46.66%
seq	2780	14	1373	13	1516	16	10.42%
spla	6660	19	2298	16	525	14	-77.15%
tseng	1927	47	1763	41	-	-	-
Mean I	1391.64	21.26	921.52	19.48	-	-	-
Ratio 1	1.00	1.00	0.65	0.92	-	-	-
Mean II	1090.95	17.03	698.38	14.93	550.87	12.78	-
Ratio 2	1.00	1.00	0.64	0.88	0.50	0.75	-
Ratio 3	-	-	1.00	1.00	0.79	0.86	-

4. AIG TRANSFORMATIONS RESULTS

This section presents the AIG size reduction achieved by AIG transformations for a set of ITC99 and MCNC benchmarks [1]. Each benchmark is read and tranformed into an AIG through algebraic factorization. Then structural hashing is performed, obtaining the number of AIG nodes (N) and levels (LV) shown in column "*Initial*" of Table 1.

In order to obtain a highly optimized AIG, the *dc2* command is executed iteratively until no changes are observed. This reduces the number of nodes for the majority of cases, as seen in column "*After dc2*". The number of levels is usually reduced, but not for all cases. Geometric mean I and the ratio 1 refer to the complete set of benchmarks.

An alternative experiment is performed by first collapsing the netlist, just after reading the input file. The collapsing operation cannot finish for all benchmarks due to its complexity. In the cases it can finish, the AIGs are ob-

```
1:  procedure BOOLEANDECOMPOSITIONAIG(AIG, cutParams)
    Input: An AIG network, and parameters to enumerate KL-cuts
2:      for each node $N$ in AIG in topological order do
3:          for each kcut $C$ of node $N$ from kcut enumeration do
4:              obtain the klcut from $C$ in AIG
5:              if the klcut is accepted based on cutParams then
6:                  $F$ = set of klcut functions
7:                  $DC = \emptyset$ /* DC-set of $F$ functions */
8:                  divisors = BOOLDECOMPOSE($F$, $DC$)
9:                  if size of divisors < size of klcut then
10:                     new_klcut = AIG network of divisors
11:                     replace klcut by new_klcut in AIG
12:                     restart kcut enumeration
```

Figure 5: AIG optimization using Boolean decomposition.

tained through algebraic factorization and structural hashing. Then, the *dc2* command is run iteratively until no changes are observed, generating the results shown in column "*After collapse + dc2*". Geometric mean II and the corresponding ratios 2 and 3 refer to the benchmarks in which collapsing could finish.

The final number of nodes varies from a 89% reduction (*clma*) to a 792% increase (*b05*). Collapsing the netlist may result in a larger AIG (see Sect. 2), and the AIG transformations may not obtain the same results as before collapsing, as these modifications are biased by the AIG structure. In the *b05* benchmark, the initial description has very good logic sharing between outputs, which is lost after collapsing. The shared logic is not recovered due to local optimization limitations, as some of the benchmark outputs depend in a large set of inputs. For other cases, collapsing enables significant AIG reduction by removing redundant logic.

5. AIG OPTIMIZATION APPROACH

This section introduces a new AIG optimization approach, based on Boolean decomposition with two-literal divisors [15]. Boolean methods are known to be inefficient and not scalable, but also to obtain better results when compared with algebraic methods. In this work, the Boolean decomposition method [15] is applied to multi-output functions and runtime is improved without losing quality of results. Still, the algorithm is not scalable for large circuits, and therefore applied via local optimization.

5.1 AIG local optimization using KL-cuts

A pseudo-code of the AIG optimization strategy is shown in Fig. 5. The procedure *booleanDecompositionAIG* receives the AIG and the parameters to enumerate the KL-cuts, which define the number of nodes and inputs, for example.

From the outputs to the inputs, each node is visited (line 2), and for each node all K-cuts enumerated based on the parameters are tried (line 3). In line 4, the KL-cut is obtained from the K-cut, and if accepted based on the parameters (line 5), the Boolean decomposition is performed on top of the KL-cut Boolean functions (line 8).

If the Boolean decomposition result is smaller than the number of nodes of the KL-cut, it replaces the logic in the AIG (line 11). Note that *boolDecompose* returns the set of divisors used to decompose the KL-cut functions, which can be easily translated to an AIG network. Also, since the AIG is changed when there is a KL-cut replacement, the previous K-cut enumeration has to be restarted (line 12).

5.2 Boolean decomposition

The algorithm for *boolDecompose* is presented in Fig. 6, which recursively performs the Boolean decomposition on a

```
1:  function BOOLDECOMPOSE($F$, $DC$)
    Input: A set of functions $F$ and their DC-set
    Return: A set of divisors that form an AIG
    /* Each recursive call generates one divisor */
2:      /* Step (I) - If all functions are decomposed, return */
3:      if all functions $f \in F$ are trivial then return
4:
5:      /* Step (II) - Generate the candidate Boolean divisors */
6:      numLiterals = 0  /* Number of literals of all functions */
7:      $D = \emptyset$              /* Set of generated divisors */
8:      for each non-trivial $f \in F$ do
9:          $a$ = algebraic factored form of $f$
10:         numLiterals += number of literals in $a$
11:         $D = D \cup$ {two-literal leaves of $a$}
12:
13:     /* Step (III) - Perform Boolean division for each divisor */
14:     $x$ = new variable  /* Variable of the new divisor */
15:     bestDivisor = $\emptyset$
16:     for each divisor $d \in D$ do
17:         divLiterals = 0  /* Number of literals after division */
18:         for each non-trivial $f \in F$ do
19:             /* $DC(f)$ is the DC-set of $f$ */
20:             $DC_{proj} = DC(f)$ projection onto $f$ and $d$ support
21:             $DC_{div} = DC_{proj} + (x \oplus d)$
22:             Flits, Rlits = number of literals in $f$ factored form
23:             if $d$ is accepted based on $f$ vars polarities then
24:                 /* $R(f)$ is the result of the division of $f$ by $d$ */
25:                 $R(f)$ = twoLevelMinimization($f$, $DC_{div}$)
26:                 Rlits = literals in $R(f)$ factored form
27:             if Rlits > Flits then
28:                 /* If $R(f)$ has more literals, or division was not
29:                    performed, select $f$ as part of the solution */
30:                 $R(f) = f$; divLiterals += Flits
31:             else
32:                 divLiterals += Rlits
33:         /* If division reduced literals, update best result */
34:         if divLiterals < numLiterals then
35:             for each non-trivial $f \in F$ do
36:                 $bestR(f) = R(f)$
37:             $bestDivisor = d$; numLiterals = divLiterals
38:
39:     /* Step (IV) - Set the functions for next recursive call */
40:     for each non-trivial $f \in F$ do
41:         $newF(f) = bestR(f)$
42:         $newDC(f) = DC(f) + (x \oplus bestDivisor)$
43:     /* Return the best divisor and make a new recursive call */
44:     return {$bestDivisor$} $\cup$ BOOLDECOMPOSE($newF$, $newDC$)
```

Figure 6: Boolean decomposition procedure.

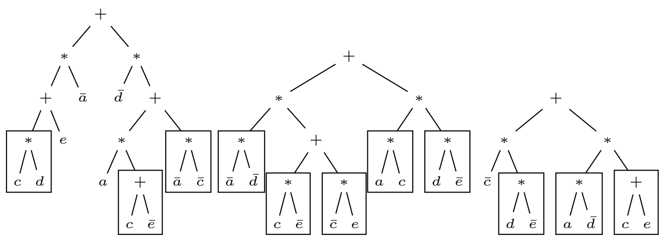

Figure 7: Factored form trees from *b06* benchmark.

set of functions F. The algorithm is divided into four steps: (I) detection of trivial cases, (II) generation of candidate Boolean divisors and definition of the cost function, (III) selection of the best divisor via Boolean division, and (IV) preparation of the next recursive call.

At line 3, detection of trivial cases (I) is performed, identifying when all functions in F are decomposed. The algorithm is executed recursively until this condition is satisfied.

Step (II) starts by obtaining the algebraic factored form for each function in F (line 9). The cost function to be minimized is defined as the sum of literals of all functions in factored form (*numLiterals*, line 10). Then, the two-literal leaves of the factored form trees are generated as candidate divisors for each function in F (see Fig. 7).

Boolean division is performed for all functions in F using each divisor in D, in order to calculate the cost function for all divisors. The best divisor (III) is the one that achieves the largest reduction in number of literals.

As seen in Sect. 3.4, Boolean division is performed by adding the SDC of a divisor to the DC-set of a function and running two-level minimization. As the DC-set may contain variables not relevant to the division, a DC-set projection is performed to the support of the function f and divisor d (line 20). The projected DC-set is accumulated with the SDC generated by the evaluated divisor (line 21), generating the DC-set used in the two-level minimization (DC_{div}).

The division may be filtered due to the polarities of the variables (see Sect. 5.3). If the division is filtered or if the number of literals in the division result $R(f)$ is greater than in f (line 27), then f is used as part of the current solution (line 30). If the number of literals is reduced after a division (line 34), the best solution is updated: the division result (line 35), the divisor and the number of literals (line 37).

The next iteration is prepared at step (IV). The ON-set (line 41) and the DC-set (line 42) obtained by the best divisor are used in the next recursive call (line 44).

Note that the two-level minimization could be performed as a multi-output function. However, running single output two-level minimizations is preferred as it is more efficient, divisions can be filtered based on the variables polarities and the divisions that increase literals can be discarded for each output individually.

5.3 Filters to reduce runtime

Select divisors from factored forms. In comparison to [15], only pairs of literals obtained from the leaf nodes of the factored form trees are selected as potential divisors, instead of trying all possible pairs of variables and polarities. The divisors are selected from all output functions of the KL-cut. For the benchmarks analyzed with our method, the quality of results was not affected by applying this filter, while the optimization runtime was significantly improved.

To illustrate the divisor selection, factored form trees obtained from *b06* functions are depicted in Fig. 7. The two-literal leaves highlighted in Fig. 7 are the divisors selected. Notice that only one polarity is investigated, e.g., if the divisor $c + \bar{e}$ is chosen, its negated version $\bar{c} \cdot e$ is disregarded.
Use variable polarity information. This filter is used to avoid exploring divisions with unpromising polarities between the divisor and the divided function. Table 2 describes the divisors that are accepted based on its support and the polarities of the variables in the divided function. A total of 849 Boolean divisions are performed during the Boolean decomposition of *b06* when using this filter versus 943 without it (and 13829 divisions would be done without any filter).

Table 2: Divisors accepted based on the divided function f.

Divisor variables a and b w.r.t. f	Divisors accepted
$a \notin f$ support or $b \notin f$ support	None
a is binate or b is binate	$(a \cdot b)$, $(\bar{a} \cdot \bar{b})$, $(a \cdot \bar{b})$, $(\bar{a} \cdot b)$
a and b have the same polarity	$(a \cdot b)$, $(\bar{a} \cdot \bar{b})$
a and b have different polarities	$(a \cdot \bar{b})$, $(\bar{a} \cdot b)$

Table 3: AIG results of Boolean decomposition.

Name	Initial		(a) ABC smallest		(b) Boolean Decomp.		Runtime (s)	Diff. nodes (a) and (b)
	N	LV	N	LV	N	LV		
b04	546	24	487	21	442	21	222	-9.24%
b05	830	54	459	23	409	29	140	-10.89%
b06	42	5	31	5	23	9	0.55	-25.81%
b07	365	27	331	22	320	27	125	-3.32%
b08	155	20	119	9	113	10	2	-5.04%
b09	136	12	123	10	117	11	4	-4.88%
b10	180	11	162	9	156	10	14	-3.70%
b11	611	28	452	21	427	24	178	-5.53%
b12	1002	17	947	14	920	15	212	-2.85%
b13	261	12	220	11	207	12	3	-5.91%
b14	6069	60	3924	100	3810	120	14421	-2.91%
b15	8432	65	7030	95	6656	107	14592	-5.32%
alu4	2654	14	625	14	570	16	268	-8.80%
apex2	1960	17	142	14	128	15	5	-9.86%
bigkey	3081	10	2847	10	2396	15	2209	-15.84%
clma	11938	40	527	16	449	17	46	-14.80%
diffeq	2575	40	2137	41	2015	51	407	-5.71%
ex1010	7681	17	2337	14	2306	15	2004	-1.33%
ex5p	1731	15	204	8	197	8	127	-3.43%
i10	3675	50	1637	36	1530	37	882	-6.54%
misex3	2454	13	754	14	731	14	288	-3.05%
pdc	7757	19	1717	18	1543	18	1371	-10.13%
seq	2780	14	1373	13	1320	16	626	-3.86%
spla	6660	19	525	14	436	15	211	-16.95%
tseng	1927	47	1763	41	1696	42	445	-3.80%
Mean	1391.5	21.2	614.4	17.5	566.7	19.9	147.4	-7.76%
Ratio 1	1.000	1.000	0.442	0.826	0.407	0.939	-	-
Ratio 2	-	-	1.000	1.000	0.922	1.137	-	-

The polarity of the variables can be obtained using the concept of unateness, which is defined for completely specified functions. Unateness can only be used in the first iteration of Boolean decomposition, when the DC-set is empty. After the first iteration, the DC-set contains the SDCs of the previously selected divisors, and the concept of containment [19] must be used (see Sect. 3.1).

6. EXPERIMENTAL RESULTS

Table 3 shows the AIGs metrics (N: nodes, LV: levels) before and after Boolean Decomposition. Column *Initial* reports the metrics of the AIGs after input and structural hashing, and column *ABC smallest* reports the AIGs with least number of nodes from Table 1.

The AIG optimization via Boolean decomposition is applied on top of the AIGs of column *ABC smallest.* KL-cuts with K=8 and unbounded L are enumerated in order to obtain smaller parts of the AIG with complete local context. Also, the number of nodes of the KL-cuts is restricted

Table 4: Technology mapping results.

Name	FPGA (LUT4)				Standard cell (μm^2, ns)			
	(a) ABC		(b) Bool. Dec.		(a) ABC		(b) Bool. Dec.	
	N	LV	N	LV	Area	Delay	Area	Delay
b04	183	7	177	8	14041	1879	12735	1833
b05	222	9	201	11	14350	2057	12687	2545
b06	17	2	17	2	1191.9	567	993	754
b07	145	8	146	8	9828	1949	9645	2205
b08	55	4	50	5	3659	873	3460	934
b09	55	4	53	5	3554	876	3716	1060
b10	73	5	75	5	5076	881	4893	957
b11	179	7	175	8	13215	1916	12551	2086
b12	452	6	455	6	28188	1322	27922	1409
b13	95	4	94	4	6890	975	6482	1046
b14	1608	33	1544	37	112586	8639	112277	10068
b15	3021	33	2922	33	204689	8110	195332	9398
alu4	293	7	277	7	18114	1308	16446	1450
apex2	65	6	61	6	4218	1300	3884	1398
bigkey	1254	3	921	4	81867	901	73250	1327
clma	277	6	249	6	15934	1446	13968	1512
diffeq	824	14	754	14	61892	3658	57616	4129
ex1010	1132	7	1118	7	66649	1314	65849	1415
ex5p	129	3	122	4	6356	784	6090	789
i10	724	14	688	14	47974	3222	45120.5	3187
misex3	363	6	363	6	21716	1222.7	21162.1	1218
pdc	904	7	821	7	50301	1585	45429	1592
seq	716	6	673	7	40718	1232	38957	1438
spla	262	5	222	6	15108	1218	12991	1303
tseng	758	13	741	13	50583	3598	49251	3561
Mean	284.35	6.84	268.64	7.39	18356	1607	17217	1859
Ratio	1.000	1.000	0.945	1.081	1.000	1.000	0.938	1.157

to 30, therefore having a very limited scope of optimization. Boolean decomposition is performed on top of the KL-cut outputs functions, only replacing the KL-cut logic if the number of nodes is reduced. The experiments were run on an Intel Core i7 processor with 4GB of RAM. All AIGs passed formal verification using ABC command *cec*.

The column "*Boolean Decomp.*" reports the results obtained after performing two iterations of the Boolean decomposition method. By using Boolean decomposition, it was possible to reduce the number of AIG nodes further by 7.76% on average, with important results such as 25.81% (*b06*), 16.95% (*spla*), 15.84% (*bigkey*) and 14.8% (*clma*).

Our approach is able to identify a better logic sharing, therefore increasing the number of levels, which is not controlled by our method. Still, there is an increase of up to 1 level for 15 out of 25 benchmarks evaluated. Also, the average number of levels is still smaller than the *Initial* results.

Table 4 shows technology mapping performed for FPGAs and standard cells using the AIGs from Table 3. Area reduction was observed simply by changing the input with smaller AIGs. Mapping to LUT4s was performed with the ABC command "*if -K 4*", obtaining 5.5% area reduction on average. Mapping to standard cells was performed with the ABC command "*map*" using the library "*GSCLib_3.0.lib*" of [1], obtaining 6.2% area reduction on average.

7. CONCLUSIONS

This paper introduced an approach to explore the potential use of Boolean decomposition in the optimization of AIGs. The experiments show promising results, with an average reduction of 7.76% in AIG nodes. Scalability is one of the aspects that requires more investigation. We envision a synthesis system in which smart oracles could guide the search for divisors based on simple correlation metrics between functions and divisors.

As future work, there are some directions that could be explored. For example, different types of cuts and more combinations of divisors could be studied. Using other models of flexibility (Boolean relations) could also be considered. Delay is another important aspect that is not considered in this work, but could be incorporated, controlling the number of levels and reducing the resulting circuit delay.

8. ACKNOWLEDGMENTS

We thank Mayler Martins for helpful discussions. The present work was performed with the support of CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil. This work has been partially supported by funds from the Spanish Ministry for Economy and Competitiveness and the European Union (FEDER funds) under grant TIN2013-46181-C2-1-R, and the Generalitat de Catalunya (2014 SGR 1034).

9. REFERENCES

[1] C. Albrecht. IWLS 2005 benchmarks. http://iwls.org/iwls2005/benchmarks.html, 2005.

[2] R. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *Proc. of ISCAS*, pages 49–54, 1982.

[3] R. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40. Springer, 2010.

[4] S. Chatterjee, A. Mishchenko, and R. Brayton. Factor cuts. In *Proc. of ICCAD*, pages 143–150. ACM, 2006.

[5] J. Cortadella. Timing-driven logic bi-decomposition. *IEEE TCAD*, 22(6):675–685, 2003.

[6] G. De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[7] P. Fišer and J. Schmidt. Improving the iterative power of resynthesis. In *Proc. of DDECS*, pages 30–33. IEEE, 2012.

[8] L. Machado, M. G. A. Martins, V. Callegaro, R. P. Ribas, and A. I. Reis. KL-cut based digital circuit remapping. In *Proc. of NORCHIP*, pages 1–4. IEEE, 2012.

[9] O. Martinello Jr, F. S. Marques, R. P. Ribas, and A. I. Reis. KL-cuts: a new approach for logic synthesis targeting multiple output blocks. In *Proc. of DATE*, pages 777–782. EDAA, 2010.

[10] A. Mishchenko and R. Brayton. Scalable logic synthesis using a simple circuit structure. In *Proc. of IWLS*, pages 15–22, 2006.

[11] A. Mishchenko, R. Brayton, and S. Chatterjee. Boolean factoring and decomposition of logic networks. In *Proc. of ICCAD*, pages 38–44. IEEE, 2008.

[12] A. Mishchenko, R. Brayton, J. R. Jiang, and S. Jang. Scalable don't-care-based logic optimization and resynthesis. *ACM TRETS*, 4(4):34, 2011.

[13] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *Proc. of DAC*, pages 532–535. ACM, 2006.

[14] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton. Combinational and sequential mapping with priority cuts. In *Proc. of ICCAD*, pages 354–361. IEEE, 2007.

[15] N. Modi and J. Cortadella. Boolean decomposition using two-literal divisors. In *Proc. of VLSID*, pages 765–768. IEEE, 2004.

[16] P. Pan and C. Lin. A new retiming-based technology mapping algorithm for LUT-based FPGAs. In *Proc. of FPGA*, pages 35–42. ACM, 1998.

[17] H. Savoj and R. Brayton. The use of observability and external don't cares for the simplification of multi-level networks. In *Proc. of DAC*, pages 297–301. IEEE, 1990.

[18] C. Scholl. Multi-output functional decomposition with exploitation of don't cares. In *Proc. of DATE*, pages 743–748. IEEE, 1998.

[19] L. Wang and A. Almaini. Multilevel logic simplification based on a containment recursive paradigm. In *IEE Proceedings Computers and Digital Techniques*, volume 150, pages 218–226. IET, 2003.