Fast-Extract with Cube Hashing

Bruno de O. Schmitt[1], Alan Mishchenko[2], Victor N. Kravets[3], Robert K. Brayton[2] and Andre I. Reis[1]

[1]Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS)
[2]Department of EECS, University of California, Berkeley
[3]IBM Thomas J. Watson Research Center, Yorktown Heights, NY

Abstract— The fast-extract algorithm is a well-known algebraic method for factoring and decomposing Boolean expressions. Since it uses pairwise comparisons between cubes to find factors, the runtime is degraded for networks whose primary outputs are expressed in terms of primary inputs and have Boolean functions with thousands of cubes. This paper describes a new implementation of the fast-extract algorithm, *fxch*, having complexity linear in the number of cubes. The reduction in complexity is achieved by hashing sub-cubes and using the hash table to find good factors to extract. Experimental results on industrial benchmarks show superior runtime and scalability of the proposed algorithm, compared to the available solutions.

I. Introduction

The problem of identifying common sub-expressions, also known as divisors (or factors), in Boolean functions has been a key component of logic synthesis tools since the early days of multi-level synthesis [1]. The process is known as decomposition if a new intermediate variable is created, otherwise it is called factoring. The goal of decomposition is to identify frequently used sub-expressions, implement them once, and share them across the entire network. A decomposition algorithm, such as [2], can be used to reduce the complexity of an available multi-level network whose nodes are represented by sum-of-products (SOPs), or it can construct a new multi-level network from the SOP representing a multi-output Boolean function.

Decomposition methods are divided into two groups: fast algebraic methods and slower but more powerful Boolean methods, which take advantage of Boolean properties of the function and use don't cares [1]. In practice, algebraic methods are widely used due to their lower complexity. These take as input a sum of products representation of a Boolean function and apply algebraic operations to find and extract divisors.

Algebraic methods traditionally use the notion of kernels [3] to find multi-cube divisors that are common to two or more expressions. The original fast-extract algorithm *fx* introduced in [2] is based on this notion but restricts the set of kernels to double-cube divisors and single-cube two-literal divisors.

A detailed overview of *fx* is deferred to Section II of this paper. Here it is enough to note that *fx* uses pairwise comparison between cubes to find divisors, which means that the complexity of *fx* is quadratic in the number of cubes.

Almost three decades have passed since the *fx* algorithm was introduced. At that time memory was not as cheap and designs not as large and complex as today; therefore the quadratic approach to algebraic decomposition, which trades faster runtime for lower memory usage, was appropriate. Since then, the transistor count in the designs has increased by three orders of magnitude [9] while the price of memory decreased by four orders of magnitude [8], thereby rendering the traditional *fx* approach unsuitable for today's designs. In general, due to the pressure to reduce the runtime of CAD tools, any algorithm whose complexity is greater than linear must be carefully evaluated while trying to obtain a linear version of the algorithm that is more scalable for large designs.

This paper focuses on presenting the new divisor extraction algorithm. The approach is called fast-extract with cube hashing, *fxch*. It uses cube hashing to trade increased memory usage for faster runtime, reducing complexity from quadratic to linear in the number of cubes. The motivation for this work and incomplete experimental results appeared in a workshop publication [4], while this paper is the first one to present the new algorithm in a peer-reviewed conference.

The rest of the paper is organized as follows. Section II gives background on Boolean functions, Boolean networks, algebraic decomposition and the original *fx* algorithm. Section III describes the implementation of *fxch* with a brief discussion about cube hashing and why skipping some cubes to reduce the hash table size is a bad idea. Section IV gives the experimental setup and discusses the experimental results. Finally, Section V concludes the paper.

II. Background

A. Boolean Functions

A **Boolean variable**, x, is a variable that takes one of the two values from the domain $\mathbb{B} = \{false, true\}$, or $\{0, 1\}$. A **positive literal** is the Boolean variable, x, and a **negative literal** is its complement, $\overline{x}$. The Boolean AND of k literals is a **cube**, or product, i.e. $c = l_1 \cdot l_2 \cdots l_k$. Let symbol "-" denote a **don't care literal value**. If a variable is not represented by a positive literal or a negative literal in a cube, then its value is said to be a don't care literal. A **minterm** is a cube, in which every variable is represented by either a negative or positive literal. It can be noted that a cube with n don't care literal values covers 2^n minterms.

Let $f(X) : \mathbb{B}^n \rightarrow \mathbb{B}$ be a **completely specified Boolean**

function of n variables $X = \{x_1, x_2, \ldots, x_n\}$. The **support** of f is the subset of variables that influence the output value of the function f. The set of minterms, for which f evaluates to 1 and to 0, defines the **on-set** and the **off-set** of f, respectively. Unless stated otherwise, we assume that a Boolean function is completely specified. In a multi-output Boolean function $f(X) : \mathbb{B}^n \rightarrow \mathbb{B}^m$, $m > 1$, each output $f_i, 1 \leq i \leq m$ is a Boolean function.

Even though *fxch* is capable of handling multi-outputs functions, for the sake of simplicity we shall continue defining terms for single-output functions. A cube is an **implicant** of f if it covers only minterms present in the on-set of f. A **prime implicant**, or **prime**, of f is an implicant, from which no positive or negative literal can be removed without intersection with the off-set of the function.

Any Boolean function can be represented as a two-level **sum of products** (SOP), which is a Boolean OR of implicants (i.e. $S = c_1 + c + \cdots + c_n$). A SOP is said to be **irredundant** if no implicant can be removed without changing its functionality. A cube c_1 is contained in cube c_2 if the set of minterms covered by c_1 is a subset of the minterms contained in c_2. A SOP is said to be **single-cube containment free** if it doesn't have a cube pair such that one cube contain the other.

B. Boolean Networks

A **Boolean network** (or circuit) is a directed acyclic graph (DAG) $G = (V, E)$ with nodes V and edges E. Every node is associated with a Boolean function and a Boolean variable, called the output variable, representing the node's output. The existence of an outgoing edge from node n_1 to node node n_2 means that the variable representing the output of n_1 is an input to the function represented by n_2. In this case, we say that n_1 is a **fanin** of n_2, or that n_2 is a **fanout** of n_1.

A node n might have zero or more fanins and zero or more fanouts. **Primary inputs** are nodes without fanins. **Primary outputs** are a subset of nodes that connect the networks to the environment.

C. Factoring of Boolean Functions

Algebraic factoring (or decomposition) is traditionally used to decompose the SOP representation of a Boolean function, f, into a multi-level Boolean network.

The effectiveness of this approach rests on the manipulation of SOPs as arithmetic expressions, hence the *algebraic* name. This substantially simplifies division and allows it to scale to SOPs with thousands of cubes but it also means not fully exploiting the rules of Boolean algebra, i.e. the rules of annihilation and idempotency are not used. The added convenience and efficiency limit the scope of transformations and lead to suboptimal results [5]. For a comprehensive background on algebraic factoring and other techniques used in synthesis of Boolean networks, we refer the reader to [1].

The algebraic factoring operation is performed by repeatedly applying algebraic decomposition to a given SOP representation. Each decomposition begins by applying the distributive law in order to enumerate a restricted set of common subexpressions called algebraic divisors (or divisors), followed by selecting a divisor d and deriving a quotient q and a remainder r, such that $f = d \cdot q + r$. This process is iterated, as factoring is applied to d, q and r recursively as long as they have non-trivial divisors.

The result of this operation largely depends on the initial sum-of-products form and on finding "good" candidate divisors. The concept of a *kernel* introduced in [3] is used to define necessary and sufficient conditions for logic sharing among logic expressions to exist, therefore providing an effective way of finding good candidate divisors.

The method described in [3] computes the sets of kernels for two or more logic expressions, and then intersects them to find common sub-expressions. A specialization of this method that restricts kernels to double-cube divisors was introduced in [2].

D. The Original Fast-Extract Algorithm

The decomposition algorithm described in [2] is widely known as *fx*. The practical value of this algorithm is in limiting kernels to single-cube double-literal divisors and double-cube divisors. The algorithm performs concurrent extraction of the divisors of all types. For an in-depth discussion of *fx*, we refer the reader to [2].

From now on, we limit our discussion to the *fx* implemented in ABC [6], which is an efficient implementation of the original *fx*. Its key characteristics are the following:

- The original functions are given in the SOP form.
- Single- and double-cube divisors are considered concurrently.
- Double-cube divisors are found using pairwise comparison between cubes of the same Boolean function. Therefore the total number of double-cube divisors in an expression with n cubes is bounded by n^2.
- The weight of each divisor is a function of the number of saved literals and of its logic level.
- All divisors are stored in a priority queue, which is repeatedly accessed in order to get the divisor with the highest weight.
- After each extraction, the SOP and the divisor weights are updated, and new divisors are added to the queue.

III. Fast-Extract with Cube Hashing

In this section, we delve into the key aspects of the proposed *fxch* algorithm. The pseudo-code of *fxch* is shown in Algorithm 1. The algorithm is based on *grouping* (sub-section A) of identical cubes for different outputs (in the case of the multi-output SOP) and on efficient *hashing* of cubes and sub-cubes (sub-section B) during the extraction of divisors. The algorithm computes the set of all double-cube divisors up to four literals (sub-section C). The algorithm may create undesirable degenerate divisors, which require special treatment (sub-section D). Finally, the extraction procedure is described in sub-section E.

A. Cube Grouping

When decomposing a number of Boolean functions, the current implementation in ABC generates an inefficient SOP representation that considers cubes for each output independently. This inefficient representation is given as input to *fx*

Algorithm 1: Fast-Extract with Cube Hashing

Input : the original multi-output SOP
Output: the network derived by the factoring process

begin
- process the SOP by grouping identical cubes;
- create the hash table containing sub-cubes;
- create the initial set of divisors candidates;
- **while** *there are non-trivial divisors* **do**
  - select the best divisor candidate;
  - find cubes affected by its extraction;
  - extract the divisor;
  - update the affected cubes;
  - update the sub-cube hash table;
  - update the set of divisor candidates;
- process SOP by ungrouping the cubes;
- **return** *the resulting multi-level network*

and *fxch*. One way to overcome this inefficiency is by grouping identical single-output cubes, resulting in a multi-output SOP representation. The example below illustrates the existing short-comings and the proposed mitigation. Consider the truth table and the respective SOP representation shown in Figure 1.

x_1	x_2	x_3	y_1	y_2
1	1	-	1	0
-	1	1	1	0
0	0	0	1	1
0	1	1	1	1
1	-	1	0	1
1	1	0	0	1

y_1	x_1	x_2	
y_1	x_2	x_3	
y_1	$\overline{x_1}$	$\overline{x_2}$	$\overline{x_3}$
y_1	$\overline{x_1}$	x_2	x_3
y_2	$\overline{x_1}$	$\overline{x_2}$	$\overline{x_3}$
y_2	$\overline{x_1}$	x_2	x_3
y_2	x_1	x_3	
y_2	x_1	x_2	$\overline{x_3}$

Fig. 1. Truth table and the respective SOP representation originally used by ABC

Notice the existence of two cube pairs with identical inputs, meaning that the representation shown in Figure 1 does not take advantage of the fact that some cubes differ only in their outputs. It can be shown experimentally that the number of such cubes can be large, and therefore processing them separately has a negative impact on both runtime and memory usage. We propose a better representation to mitigate this problem, as illustrated in Figure 2.

y_1	x_1	x_2	
y_1	x_2	x_3	
y_1y_2	$\overline{x_1}$	$\overline{x_2}$	$\overline{x_3}$
y_1y_2	$\overline{x_1}$	x_2	x_3
y_2	x_1	x_3	
y_2	x_1	x_2	$\overline{x_3}$

Fig. 2. New SOP representation used by FXCH

where each cube representation consists of an input pattern and an output pattern.

B. Cube Hashing

The main technical contribution of this paper is the introduction of an algorithm capable of finding useful algebraic divisors by hashing of sub-cubes, which is called *cube hashing*.

To better understand the concept behind this technique, consider the problem of finding all cubes in a given SOP, which differ only in one literal. A naive approach consists of comparing all cubes pairwise. The key insight used to develop a smarter approach systematically examines cubes that differ only in one literal, and removes that literal to makes the cubes equal. The approach first builds a hash table containing all cubes, and then for each cube removes one literal at a time, while inserting the resulting sub-cube into the hash table. The collisions observed in the hash table enable us to find cubes that differ in only one literal. This approach is linear in the number cubes.

We can use an extension of the presented approach in order to find divisors. All we have to do, is to hash (1) each cube itself, (2) all of its sub-cubes created by the removal of one literal and (3) all its sub-cubes created by the removal of a pair of literals. In this case a collision could mean the existence of a divisor that corresponds to the removed literals. On the other hand, the identical sub-cubes correspond to the *base*, i.e. the common part of the original cubes which remains after divisor extraction. Figure 3 illustrates these concepts by factoring a two-cube SOP expression.

$$F = (x_1 \cdot \overline{x_2} \cdot x_3) + (x_1 \cdot x_3 \cdot x_4)$$

Sub-cubes hash table		
Lit	$cube_1$	$cube_2$
1	$\overline{x_2} \cdot x_3$	$x_3 \cdot x_4$
2	$x_1 \cdot x_3$	$x_1 \cdot x_4$
3	$x_1 \cdot \overline{x_2}$	$x_1 \cdot x_3$

$divisor = \overline{x_2} + x_4$
$base = x_1 \cdot x_3$

Fig. 3. Decomposition of a two-cube SOP expression

The first column of the presented hash table indicates the position of the removed literal. The two colored cells indicate a collision. In this case, the removal of the second literal from the first cube generates the same sub-cube as the removal of the third literal from the second cube. A divisor is generated using the removed literals, while the base is equal to the sub-cube. After extraction, the resulting Boolean function is:

$$F = x_1 \cdot x_3 \cdot (\overline{x_2} + x_4)$$

As pointed out in [4], the hash table may become excessively large when working with a large SOP. Indeed, finding double-cube divisors for a cube requires hashing

$$1 + n_l + \frac{n_l * (n_l - 1)}{2}.$$

sub-cubes, where n_l is the number of literals in a given cube. Observing that initially many sub-cubes do not generate collisions, and thereby are deemed useless for factoring at that point, one could be misled into thinking that filtering such "useless" sub-cubes would be a good way to reduce the consumed memory. However, as we found out experimentally, this leads to a substantial runtime increase. Instead, it is advantageous to have all sub-cubes in the hash table at all times. Thus, when a divisor is extracted and the cubes are updated, it is only necessary to generate sub-cubes for the updated cubes.

C. Divisors Functions

Unlike the preliminary implementation of *fxch* in [4], our implementation can potentially use the set of all double-cube divisors with up to four literals. The algorithm restricts divisors to a small set of functions and this facilitates their ranking and logic sharing during decomposition [5].

The complete set of possible double-cube divisors with complements, is summarized in Theorem 1 in [2]. The set implies that using canonical basis NAND, XOR ($\oplus$), and MUX as divisor functions imposes a duality property in such divisors, meaning that the complement of a divisor is also a divisor. The constant-1 function is also included in the set in order to eliminate the redundant logic, since its appearance implies the existence of distant-1 cubes ($\overline{x_i} + x_i$). Thus, the divisor functions are restricted to the following:

1, NAND, XOR, MUX

This restriction also explains why we limit divisors to four literals. It must be noted, however, that in order to handle *degenerate divisors*, we do check *all* double-cube divisors with up to four literals.

D. Degenerate Divisors

There is a set of divisors that can deteriorate the outcome of extraction if not properly handled. In the previous subsection, we encountered one form of this divisors: the constant-1 divisor: ($\overline{x_i} + x_i$). In the earlier implementation of *fxch*, this divisor was ignored during extraction, meaning it was identified but not properly handled.

Other degenerate divisors appear during the extraction process because the input SOP is not prime and irredundant. In any case, the problem results in handling $x_i + \overline{x_i}x_k$, $x_i\overline{x_k} + x_k$ and $x_i + x_k$ as three different divisors, while in fact they are the same divisor $x_i + x_k$. If degenerate divisors are not properly handled, the selection of a divisor among other candidates is based on inaccurate assessment of divisor costs, and also the extraction process would not decompose all possible cubes, thereby reducing the quality of results.

E. Extraction

The extraction changes the input part of a cube. The literals present in the divisor are removed from the cube, and a properly complemented literal that identifies the divisor may added to it, depending on whether or not a new intermediate variable was created. The extraction may also change the output pattern of a cube and produce a new cube, or invalidate an existing one. The set of performed extractions is tabulated below according to the possibility of cube output patterns being equal (i.e. exact extraction) or not equal but with an intersection (i.e. inexact extraction).

Table I identifies the input and output patterns of a cube as x and y, respectively. The input part of a cube is treated as a set of literals. The output pattern is treated as a bit vector; the syntax of bit-wise operators from the C programming language is used to describe updating of both parts. If the extraction of a divisor from a pair of cubes is exact, the operation also invalidates one of them. Inexact extractions creates a new cube, and may invalidate one of the original cubes, depending on whether the resulting output pattern is equal to zero.

TABLE I
SET OF POSSIBLE EXTRACTIONS.

$y_1 = y_2$ (exact)	$y_1 \neq y_2$ (inexact)
$c_1 \equiv \{(x_1 \cap x_2) \cdot z, y_1\}$	$c_1 \equiv \{x_1, y_1 \,\&\sim y_2\}$
$c_2 \equiv nil$	$c_2 \equiv \{x_2, y_2 \,\&\sim y_1\}$
	$c_3 \equiv \{(x_1 \cap x_2) \cdot z, y_1 \,\& y_2\}$

IV. Experimental Results

We describe our experimental setup and compare *fxch* with other state-of-the-art methods. We also evaluate the usefulness of the cube grouping proposed in Section III-A.

A. Experimental Setup

We implemented the algorithm described in Section III as command *fxch* in ABC, an open-source tool for logic synthesis, technology mapping, and formal verification of logic circuits. ABC is also used to verify the resulting networks using combinational equivalence checking (command *cec*), which compares the AIG derived by factoring against the original multiple-output function represented by the SOP.

For comparison, we used a set of multiple-output PLA tables taken from an instruction decoder [7]. These benchmarks demonstrate that the importance of factoring increases as the circuit size increases [5]. The names of these benchmarks appear in the form "N_{PI}/N_{PO}", where N_{PI} and N_{PO} denote the number of primary inputs and primary outputs, respectively.

We compared against *jee* [5] and ABC's available implementation of the original *fx* algorithm [2]. For ease of comparison, a new switch was added to command *fx* in ABC (*fx -x*), which limits *fx* to use the same set of divisors containing up to four literals that are used by *fxch* and *jee*. After being factored, each PLA table is post-processed by ABC in order to generate an AIG representation of the optimized logic. The ABC structural hashing command *strash* is used to obtain the starting AIG representation that is further processed by ABC command *balance*. These normalization steps are applied to the output produced by the different decomposition algorithms before comparing them. In our experiments, *jee* was run as a single-threaded application so that its results are more directly comparable to the other algorithms.

B. Impact of Cube Grouping

To evaluate the impact of cube grouping presented in Section III-A, we implemented two versions of *fxch*: one uses cube grouping ("CG") while the other does not. In the experiment, we gave the decoder PLA tables to both versions. Table III lists the collected results for both implementations in terms of runtime and peak memory usage. The arithmetic averages of the reduction ratios relative to not using cube grouping are given in the last row of the table.

The results show the benefits of using cube grouping when decomposing multi-output Boolean functions; its use reduces

TABLE II
LOGIC SYNTHESIS RESULTS: COMPARING THE *fxch*, THE *fx* FACTORING IMPLEMENTATION IN ABC AND THE *jee* TOOL

Design			*fxch*				*fx* in ABC				*jee* factoring			
Name	#literals	#cubes	t, sec	m, Mb	#nodes	#lvl	t, sec	m, Mb	#nodes	#lvl	t, sec	m, Mb	#nodes	#lvl
37/143	151830	15129	2.95	113	3835	22	18.18	11	4695	24	4.3	37	3587	24
38/67	528674	53463	1.16	59	3438	19	2.14	7	3727	18	2.0	25	3366	20
128/43	350315	20154	1.90	105	3051	18	2.43	9	3702	18	2.3	22	3191	18
128/53	317523	18182	1.63	102	2708	18	2.09	9	3261	19	2.0	23	2944	19
128/55	449226	25590	2.03	105	3079	18	2.46	10	3905	18	2.1	22	3069	20
128/69	811265	46344	2.72	107	3415	19	4.60	14	4295	20	2.7	28	3326	20
128/94	1342845	75439	4.56	120	5140	21	9.50	20	6271	22	6.3	46	5266	24
128/104	1147195	65491	3.98	121	4916	20	7.61	17	5853	21	5.7	44	4926	23
128/160	2147268	114343	8.35	226	7358	23	21.02	30	8889	24	15.5	76	7268	24
		ratios:	**0.42**	**8.33**	**0.83**	**0.97**	**1**	**1**	**1**	**1**	**0.61**	**2.54**	**0.83**	**1.04**

TABLE III
THE IMPACT OF USING CUBE GROUPING

	Runtime (s)		Memory (Mb)	
	w/ CG	w/o CG	w/ CG	w/o CG
37/143	2.95	14.47	113.36	673.14
38/67	1.16	2.75	58.64	265.11
128/43	1.90	4.73	105.48	430.69
128/53	1.63	3.89	102.21	421.89
128/55	2.03	6.44	104.95	523.19
128/69	2.72	14.64	106.84	556.56
128/94	4.56	29.38	120.44	1013.58
128/104	3.98	24.25	120.61	915.76
128/160	8.35	56.71	226.32	1882.85
ratios:	**0.19**	**1**	**0.16**	**1**

both runtime and peak memory at the cost of a slightly more complicated implementation. Both implementations identify and extract the same divisors, meaning the qualities of the resulting networks are the same.

C. Synthesis results

The results of decomposing for each test case, are given in Table II. The first column identifies each test case by the number of inputs and outputs. The next two columns give the original number of SOP literals and the cube count. Table II presents an evaluation of the results in terms of runtime (column t), peak memory usage (column m), the number of nodes (column $\#nodes$), and the number of levels (column $\#lvl$) in the final AIG. At the bottom, reduction ratios relative to using ABC's original *fx* are given; they were calculated using the arithmetic mean.

It should be noted that each algorithm pre-processes the initial set of cubes in a different way. For instance, *fx* uses a technique which favors reduction of the number of cubes in order to improve runtime at the expense of the quality of results.

The following conclusions can be made from Table II:

- In runtime, *fxch* is overall superior, but comes at the cost of using more memory. As the testcases get larger, both the runtime advantage and the memory increase become more pronounced.
- The *fxch* implementation has on average 3% less logic levels than *fx* and 7% less than *jee*. The ability to handle degenerate divisors explains the node count improvement compared to *fx*, while using level-aware divisor weights explains the level improvement compared to *jee*.

D. Scalability

This experiment has two objectives. The first is to examine the scalability of *fxch* as the problem size increases, while mnimizing the effect of pre-processing techniques used by each algorithm. The experiment consists of synthesizing circuits whose output is 1 if and only if a given integer represented as an array of bits is a prime number. We used command *gen* in ABC to generate PLA tables of all functions representing prime numbers up to 18 bits and used them as input to *fxch*, *fx*, and *jee*.

Table IV gives the results for the 8 largest functions. The same metrics as in the previous experiment were used in the evaluation. The name in the first column identifies an instance of the primes function by the number of inputs. The next column gives the number of cubes, which is also the number of primes among all the natural numbers of the given bit-width.

The results demonstrate the great runtime scalability of *fxch* for large problem instances. It completes the most complex test case containing $23,000$ cubes in 13 seconds. Compared to *jee*, it takes on average 93% less runtime. *fxch* also provides the best node count savings. However, as the problem size increases, memory usage grows quickly.

V. CONCLUSION

The paper presents a new, fully functional and very efficient implementation of the *fxch* algorithm for decomposition and factorization of Boolean expressions, which is able to handle degenerate divisors, redundant SOPs and, to some extent, single-cube containment. Improved runtime is the main advantage of the proposed method. Furthermore, when dealing with large single-output Boolean functions, the quality of results is better in terms of the AIG node count.

TABLE IV
SYNTHESIS OF THE PRIMALITY TESTING CIRCUITS FOR A INTEGER NUMBER WITH $\#inputs$ BINARY DIGITS

Design		*fxch in ABC*				*fx* in ABC				*jee* factoring			
#inputs	#cubes	t, sec	m, Mb	#nodes	#lvl	t, sec	m, Mb	#nodes	#lvl	t, sec	m, Mb	#nodes	#lvl
11	309	0.02	7	455	13	0.03	2	471	13	-	3	492	13
12	564	0.04	13	739	14	0.13	2	771	14	0.1	5	825	14
13	1028	0.10	25	1355	15	0.53	2	1440	15	0.3	9	1419	15
14	1900	0.25	50	2046	16	2.07	3	2401	16	1.4	20	2287	16
15	3512	0.62	100	3670	17	7.99	5	4174	17	8.5	58	3989	17
16	6542	1.55	202	6289	18	30.87	11	7448	18	41.2	151	6491	18
17	12251	3.91	407	11413	19	2.09 min	23	11650	19	66.7	157	12096	19
18	23000	12.97	827	17260	20	8.27 min	72	22158	20	167.8	169	18144	19
	ratios:	**0.03**	**13.6**	**0.86**	**1**	**1**	**1**	**1**	**1**	**0.42**	**4.8**	**0.91**	**1**

The paper also describes a new way of representing multi-output SOPs in ABC. The technique known as cube grouping introduced in Section III-A has been shown to improve performance. Currently it is restricted to the *fxch* implementation and the resulting multi-output cubes are ungrouped before they are processed by other ABC commands.

ACKNOWLEDGEMENTS

The co-authors affiliated with UC Berkeley were partly supported by NSA grant Enhanced equivalence checking in crypto-analytic applications. They acknowledge industrial sponsors of BVSRC: IBM, Mentor Graphics, Verific, and Xilinx for their continued support. The authors also acknowledge a helpful discussion with Niklas Een several years ago, which led to the realization that cube hashing could be used to reduce the complexity of the fast-extract algorithm.

REFERENCES

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

[2] J. Vasudevamurthy and J. Rajski, "A method for concurrent decomposition and factorization of Boolean expressions", *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1990, pp. 510−513.

[3] R. K. Brayton and C.T. McMullen, "The decomposition and factorization of Boolean expressions", *in Proc Int. Symp. Circuits Syst.*, May 1982, pp. 29−54.

[4] A. Mishchenko and R. K. Brayton, "A linear divisor extraction algorithm", *Proc. IWLS*, July 2015.

[5] V. N. Kravets. "Application of a key-value paradigm to logic decomposition", *Proceedings of the IEEE*, 103(11):2076−92, Nov. 2015.

[6] Berkeley Logic Synthesis and Verification Group, Berkeley, Calif. "ABC: A system for sequential synthesis and verifiation", http://www.eecs.berkeley.edu/ alanmi/abc/.

[7] The EPFL Combinational Benchmark Suite, "Multi-output PLA benchmarks", http://lsi.epfl.ch/benchmarks.

[8] J. C. McCallum "Memory Prices (1957—2016)", *Retrieved October 31, 2016, from http://www.jcmit.com/memoryprice.htm*

[9] Intel Corp. "Microprocessor Quick Reference Guide" *Retrieved October 31, 2016, http://www.intel.com/pressroom/kits/quickreffam.htm*