UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

BRUNO DE OLIVEIRA SCHMITT

Fast Extract with Cube Hashing

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engeneering

Advisor: Prof. Dr. André Inácio Reis
Coadvisor: Dr. Alan Mishchenko

Porto Alegre
December 2016

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

ACKNOWLEDGEMENTS

I would like to thank my mother, Gilda, who never gave up on me in the most difficult times and my father, Francisco, who always made me notice how important is to have a good education and helped me keep focus. I also thank my brothers, Fernando and Felipe, who were always there for me. Without my family support, I do not think I would have made this far.

I am grateful to my adviser, André Inácio Reis, for introducing me to the field of logic synthesis and EDA, for the invaluable advice he gave me, and for giving me the opportunity to join his logic synthesis group.

I would like to thank Alan Mishchenko for the valuable time spent with me and insightful discussions, which were not only about logic synthesis but also about life. I must also acknowledge helpful discussions with Victor Kravets, which improved the results of this work.

A special thanks to Robert Brayton who, together with Alan, welcomed me in Berkeley and shared his wisdom with me.

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 run-time is degraded for networks whose primary outputs are expressed in terms of primary inputs and have Boolean functions with thousands of cubes. This work 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 better run-time and scalability of the proposed algorithm, compared to the available solutions.

Keywords: Logic synthesis. optimization. extraction. combinational circuits.

FXCH

RESUMO

O algoritmo *fast-extract* é um método algébrico bem conhecido para a fatoração e decomposição de expressões booleanas. Entretanto, seu método de busca por divisores, que utiliza a comparação de pares de cubos, o torna demasiadamente lento para redes cujas saídas primárias são expressas em termos de entradas primárias e que tenham funções booleanas com milhares de cubos. Este trabalho descreve uma nova implementação do algoritmo *fast-extract*, *fxch*, com complexidade linear no número de cubos. A redução na complexidade é atingida com a utilização de uma tabela hash, utilizada para encontrar bons divisores. Os resultados experimentais em benchmarks industriais mostram tempo de execução e escalabilidade superiores, em comparação com as soluções disponíveis.

Palavras-chave: Síntese Lógica, otimização, extração, circuitos combinarionais.

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	And-Inverter graph
DAG	Directed acyclic graph
EDA	Electronic design automation
FX	Fast extract
FXCH	Fast extract with cube hashing
IC	Integrated circtuit
PLA	Programmable logic array
RTL	Register-transfer level
SOP	Sum of products

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Since its introduction in the early 1960s, integrated circuits (IC) have been at the center of technology advances in improving human life. The continuous downscaling of transistor dimensions, predicted by Intel co-founder Gordon E. Moore and known as Moore's Law, and the development of a powerful set of electronic design automation (EDA) tools have enabled a stupendous growth in the complexity, capability, and ubiquity of digital systems. At this time of 2016, the number of transistor in a single IC can be as many as several billion. To cope with this ever-increasing complexity, designers use techniques such as abstraction and hierarchy.

In general, EDA algorithms, techniques, and tools can be partitioned into three broad categories: logic design automation, verification and test, and physical design automation. In a typical design flow logic and physical design automation are somewhat disconnected in that logic design automation is performed before physical design automation, while the various components and aspects of the verification and test category are dispersed within both logic and physical design automation processes. (WANG; CHANG; CHENG, 2009)

The present work focuses on logic design automation, more specifically in the logic synthesis step on which the principal goal is to translate digital circuit designs (WAGNER; REIS; RIBAS, 2006) from the behavioral domain to the structural domain. For example, given a digital design at the register-transfer level (RTL), logic synthesis transforms it into a gate-level or transistor-level implementation. Furthermore, logic synthesis also explores different ways to implement logic functions optimally with respect to some desired design constraint.

Logic synthesis can be further divided into two phases: technology independent optimization and technology dependent optimization. In technology independent logic synthesis, combinational logic optimization consists of two-level and multilevel logic minimization. The use of two-level logic, however, is limited because not all Boolean functions can be efficiently represented as a sum-of-products (SOP), which is a two-level representation comprised of AND gates in the first logic level and OR gates in the second. On the other hand, a multilevel representation is often faster and smaller as it allows the reuse of sub-circuits which in turn gives more degrees of freedom in implementing a Boolean expression.

This way, logic synthesis can in many cases start by generating an initial two level

logic description. Indeed, one of the first approaches that students learn in computer engineering and computer science courses is to synthesize two level expressions (KLOCK et al., 2007; KLOCK; RIBAS; REIS, 2010). However, in current EDA tools, algorithms that are more elaborate and scale better are used in practice (COUDERT, 1994). These two level expressions have to be transformed into multi-level expressions, through factoring of the initial equations (VASUDEVAMURTHY; RAJSKI, 1990). This produces a multilevel form of technology independent implementation of the circuit, such as an AND-Inverter-Graph. The technology independent implementation is then mapped to a particular technology library (MARTINS et al., 2015), producing a mapped netlist while considering timing constraints (MACHADO et al., 2013). The mapped netlist is used for place and route during the physical design of the circuit (PUGET et al., 2015; FLACH et al., 2016).

In my final dissertation work, I will attack the problem of identifying common sub-expressions, also known as divisors (or factors), in Boolean functions. The identification of common sub-expressions has been a key component of logic synthesis tools since the early days of multilevel synthesis (BRAYTON; HACHTEL; SANGIOVANNI-VINCENTELLI, 1990). The process is known as decomposition if it creates a new intermediate variable; 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. This process produces a multi-level circuit implementation from a set of two level equations given as input. Factoring/decomposition algorithms, such as (VASUDEVAMURTHY; RAJSKI, 1990), can be used to reduce the complexity of an available multilevel network whose nodes are represented by sum-of-products (SOPs), or it can construct a new multilevel network from the SOP representing a multiple-output Boolean function. Notice that some factorization approaches have good quality but do not scale for large circuits (MARTINS et al., 2010), while others are able to scale better but are restricted to some classes of Boolean functions (CALLEGARO et al., 2013; CALLEGARO et al., 2014). In my work presented here, I will be addressing a factoring algorithm that scales when treating very large sum-of-products as input, as demonstrated through obtained results. The proposed algorithm is not restricted to any class of Boolean functions, being able to factorize any given SOP.

1.1 Motivation

Almost three decades have passed since the appearance of the *fx* algorithm. 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 run-time for lower memory usage, was appropriate. Since then, the transistor count in the designs has increased by three orders of magnitude (INTEL, 2008) while the price of memory decreased by four orders of magnitude (MCCALLUM, 2016), thereby rendering the traditional *fx* approach unsuitable for today's designs. In general, due to the pressure to reduce the run-time of EDA 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 work 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 run-time, reducing complexity from quadratic to linear in the number of cubes. The motivation for this work, a limited early-stage implementation, and incomplete experimental results appeared in a workshop publication (MISHCHENKO; BRAYTON, 2015).

It is important to notice that the work described herein generated a conference article to be presented in January 2017 at The 22nd Asia and South Pacific Design Automation Conference (ASPDAC) in Chiba, Japan. This confirms that the proposed work is an important contribution to the state of the art for factoring/decomposition.

1.2 Organization

The rest of this work is organized as follows. Chapter 2 gives background on Boolean functions, Boolean networks, algebraic decomposition and the original *fx* algorithm. Chapter 3 describes the implementation of *fxch* with a brief discussion of cube hashing and why skipping some cubes to reduce the hash table size is a bad idea. Chapter 4 gives the experimental setup and discusses the experimental results. Finally, Chapter 5 concludes the dissertation.

2 BACKGROUND

In this chapter we define some necessary background concepts for the understanding of this work. In section 2.1 we revise the axioms of the Boolean algebra. This is necessary because our work will rely only on a sub-set of these axioms to produce a factoring method that scales to large circuits, while maintaining good quality. Boolean functions and networks are described in section 2.2 and 2.3, respectively. Section 2.4 discusses the concept of optimization of Boolean networks. Section 2.5 presents an overview of the most common logic transformations used in the optimization of Boolean networs. Next sections describe some of these transformations in more detail, including Boolean division (secction 2.6), algebraic division (section 2.7) and algebraic extraction (section 2.8). Finally, the original fast extract algorithm, which is our main reference and that is improved through our work, is presented in section 2.9.

2.1 Boolean Algebra

An algebraic system defined by the quintuple $(\mathbb{B}, +, \cdot, 0, 1)$, in which $\mathbb{B}$ is as set; $+$ and $\cdot$ are binary operations on $\mathbb{B}$; and 0 and 1 are distinct member of $\mathbb{B}$, is a Boolean algebra if the axioms in Table 2.1 adapted from those given by (HUNTINGTON, 1904) are satisfied.

Table 2.1: The axiomatic definition of Boolean algebra

Axiom	OR(+) form	AND($\cdot$) form
Commutative	$a + b = b + a$	$a \cdot b = b \cdot a$
Distributive	$a + (b \cdot c) = (a + b) \cdot (a + c)$	$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
Identities	$a + 0 = a$	$a \cdot 1 = a$
Complements	$a + \overline{a} = 1$	$a \cdot \overline{a} = 0$

This axiomatic definition of Boolean algebra is sound and complete (HUNTINGTON, 1904), that is to say, logic arguments or formulas proved by these set of axioms are valid (soundness), and all true logic arguments are provable (completeness).

Next, Table 2.1 offer the reader a list of properties - valid for arbitrary elements a, b, c in a Boolean algebra - that are useful for manipulating Boolean expression.

Table 2.2: Some properties of Boolean algebra

Property	OR(+) form	AND($\cdot$) form
Associativity	$(a+b)+c = a+(b+c)$	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$
Idempotence	$a + a = a$	$a \cdot a = a$
Absorption	$a + (a \cdot b) = a$	$a \cdot (a + b) = a$
Annihilation	$a + 1 = 1$	$a \cdot 0 = 0$
De Morgan's Laws	$\overline{(a+b)} = \overline{a} \cdot \overline{b}$	$\overline{(a \cdot b)} = \overline{a} + \overline{b}$
Involution	$\overline{\overline{a}} = a$	

2.2 Boolean Functions

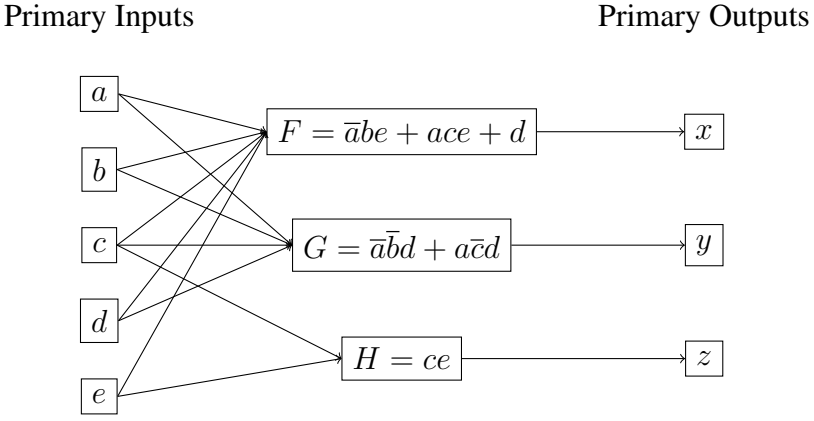
A **Boolean variable**, x, is a variable that takes one of the two values from the domain $\mathbb{B} = \{false, true\}$, or $\{0, 1\}$. A **positive literal** is the Boolean variable, x, and a **negative literal** is its complement, $\overline{x}$. The Boolean AND of k literals is a **cube**, or product, 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 d don't care literal values covers 2^d 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 multiple-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. A function f is said **cube-free** if no cube divides it evenly.

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 does not have a

Figure 2.1: Example of a Boolean network.



cube pair such that one cube contain the other.

2.3 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 **fan-out** of n_1.

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

2.4 Optimization of Boolean Networks

The goal of combinational logic optimization is to obtain an equivalent representation of a Boolean network optimal with respect to some design constraints. Typically, these design constraints are area and delay. When dealing with SOPs, a two-level logic representation, the area and delay are proportional to the size of the cover. Hence, achieving irredundant covers corresponds to optimizing both area and delay. In multilevel logic, however, minimal-area implementations generally don't correspond to minimal delay ones and vice versa. (MICHELI, 1994).

Multilevel logic optimization algorithms must take into account the trade-off be-

tween area and delay. To that end, it is necessary to extract from the Boolean network estimatives of both. It is evident that in a multilevel logic circuit the area occupied is devoted to logic gates and wires. Unfortunately, when doing technology independent optimization, it is necessary to estimate the logic gates information. A popular way to estimate area, in this case, is to relate it to the number of literals of a factored form representation (MICHELI, 1994).

Delay optimization consists in minimizing the delay in the slowest path, also known as the critical path. Delay estimation also suffers from the lack of information when doing technology independent optimization since gate delays as a function of its fan-out is not known. An unrefined way of estimating delay is to model it as a unit per level of logic.

2.5 Logic Transformations

Many different methods have been proposed to tackle the multilevel optimization problem outlined in the last section. This problem, however, is believed to be intractable due to its inherent complexity. Thus, most of these methods are heuristic, as opposed to exact methods, which are, generally, impractical even for a medium-size network.

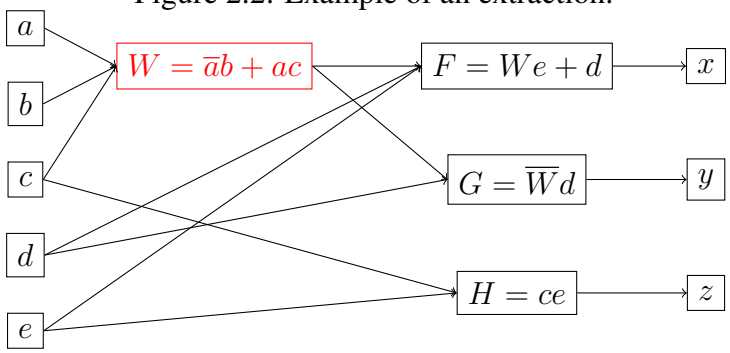
The heuristic methods improve the network through logic transformations that preserve the input/output network behavior. Fortunately, most logic transformations are defined so that network equivalence is guaranteed and does not need to be checked. Using them, however, has two significant drawbacks. First, due to the additional degrees of freedom associated with the use of multilevel networks, it is hard, if not impossible, to claim that, given a set of transformations, all equivalent networks can be explored by applying some sequence of transformations. Thus, achieving an optimal solution, or even a feasible one - given a set of constraints, may not be possible. Second, since different sequences of transformations lead to different results, it may be the case that these results correspond to local optimums, which, in many cases, deliver sub-optimal solutions to the global problem.

There are five key transformations for manipulating Boolean networks: decompositions, extraction, factoring, substitution, and elimination. From these, this works is almost exclusively concerned about extraction.

Extraction is the process of identifying common sub-expressions and using them to create new intermediate functions, which are associated with new variables, and re-

expressing the original functions in term of the original as well as the new variables. Figure 2.2 illustrates result of extracting the expression $\overline{a}b + ac$ from the Boolean network represented in figure 2.1

Figure 2.2: Example of an extraction.



The extraction process creates and introduces new nodes to the logic network, but, as a result, it simplifies each of the original logic functions. The optimization problem associated with this transformation lies within finding a good set of common subexpressions such that the resulting network is optimal in an appropriate sense.

An operation analogous to "division" is key for applying extraction to a logic network. In fact, "division" plays a key role in many other transformations and, consequently, multilevel logic optimization. The next section will present the concept of Boolean division.

2.6 Boolean Division

When optimizing logic functions, it is important to define an operation which, when given functions f and p, finds functions q and r such that $f = pq + r$. Since Boolean algebra does not have a multiplicative inverse, in mathematical terms it can not have a division operation. The described operation, however, is similar to the division operation of other algebraic systems and so is called a "division" of f by p which generates a "quotient q" and a "remainder r". The function p is either called a Boolean divisor of f if r is not null or a Boolean factor of f otherwise. For example, the following function:

$$F = \overline{a}be + ace + de \tag{2.1}$$

has e as a factor:

$$\begin{aligned} q_e &= \overline{a}b + ac + d \\ r_e &= 0 \end{aligned} \tag{2.2}$$

and $\overline{a}b + ac$ as a divisor:

$$\begin{aligned} q_{\overline{a}b+ac} &= e \\ r_{\overline{a}b+ac} &= de \end{aligned} \tag{2.3}$$

The number of Boolean divisors and factors of a given logic function f can be very large. Indeed, any function containing f is a Boolean factor of f, and with at least one minterm common with f is a Boolean divisor of f. Moreover, for a given division operation, the resulting q and r may depend on the specific representation of f and p. Thus, a problem of choosing the best factor or divisor to extract stems from the large domain available for search. Next section presents a solution which restricts the domain to a particular subset of expressions and so making the division operation unique and much easier to compute.

2.7 Algebraic Division

In (BRAYTON; MCMULLEN, 1982), the authors suggested simplifying the Boolean model by dropping from consideration some assumptions of Boolean algebra. Namely, the complements, the AND form of the distributive laws and the use of don't care sets. The resulting simplified model enables the optimization of logic networks through the use of general properties of polynomial algebra. This is known as algebraic manipulation of Boolean expression.

The successful use of this simplified Boolean model, also known as the algebraic model, requires representing the Boolean functions by algebraic expressions. An algebraic expression is defined as a single cube containment free set of cubes. For example, given an expression $F = a + ab$, F is not an algebraic expression because cube a contains cube ab. Moreover, since the algebraic model does not define complements, negative and positive literals of the same variable are treated as unrelated. Thus, there is not a

distinction between literals and variables.

Under the algebraic model, a function p is a divisor of f if there exist q and r such that $f = pq + r$, where $p \neq 0$, p and q have disjoint support, and the remainder r is minimal. Under this condition on the remainder, the quotient q is, in fact, unique. Furthermore, as in other algebraic operations, the disjoint support condition is necessary to preserve the single-cube containment free property of the results. Indeed, this prevents the generation of cubes that are covered by other cubes as well as cubes with both polarities of the same variable (WANG; CHANG; CHENG, 2009), i.e. $a\overline{a}$ and $a + \overline{a}$, which the algebraic model can not detect. We shall call this operation algebraic division. The set of primary divisors of f is defined as $P(f) = \{f/c \mid c \text{ is a cube}\}$.

2.8 Algebraic Extraction

The algebraic extraction process identifies common sub-expression and manipulates the Boolean network accordingly. The identification of sub-expressions relies on the search of common algebraic divisors (or divisors), which is done by considering an appropriate subsets of the divisors of each expression in the logic network.

The notions of kernels and co-kernels of expressions, introduced in (BRAYTON; MCMULLEN, 1982), plays a major role in the extraction process, especially when extraction multiple-cubes expressions. They provide means for finding divisors among two or more expressions using only algebraic operations.

2.8.1 Kernels and Algebraic Divisors

The kernels of an expression f are the cube-free primary divisors of f. A cube c used to obtain a kernel k such that $k = f/c$ is called a co-kernel of k. Note that, single-cube primary divisors are not kernels because they are not cube-free. The following example shall illustrate all these concepts:

Given the function $f = a\overline{b}c + acd + de$. The division of f by variable a results in $f/a = \overline{b}c + cd + de$, where $\overline{b}c + cd$ is the quotient and de the remainder. Since $\overline{b}c + cd$ is evenly divisible by c it is not cube-free, therefore it is not a kernel. Similarly, the division of f by variable e, yields a quotient of d, which is a single-cube and so not cube-free. On the other hand, the division of f by cube ac yields a quotient of $\overline{b} + d$ which is cube-free.

Thus, $\overline{b} + d$ is a kernel of f and ac its corresponding co-kernel. Since f is cube-free, it is considered a kernel of itself with the corresponding co-kernel being 1.

The set of kernels of a function is key in for detecting common multiple-cubes sub-expressions. Their relation is precisely stated in (BRAYTON; MCMULLEN, 1982) as a theorem which says that given two expressions f and g, and their respective set of kernels $K(f)$ and $K(g)$, f and g have a multiple-cube common divisor if and only if there exists kernels $k_f \in K(f)$ and $k_g \in K(g)$ such that $k_f \cap k_g$ has two or more cubes, i.e $k_f \cap k_g$ is not a single cube.

I refer the reader to (BRAYTON; MCMULLEN, 1982) for the description of a method that computes the sets of kernels for two or more logic expressions, and then intersects them to find common sub-expressions. A specialization of this approach that restricts kernels to double-cube divisors was introduced in (VASUDEVAMURTHY; RAJSKI, 1990) and will be described in the next section.

2.8.2 The Extraction Process

Given a set of algebraic expression, the extraction process is performed by repeatedly enumerating the divisors of each expression. It begins by applying the distributive law to enumerate a restricted set of common divisors, followed by selecting a divisor d and deriving, for each expression that has it as a divisor, a quotient q and a remainder r, such that $f = d \cdot q + r$. This process is iterated, as it is applied to d, q and r recursively as long as they have non-trivial divisors. The result of the extraction largely depends on the initial sum-of-products form and on finding "good" candidate divisors.

2.9 The Original Fast-Extract Algorithm

The decomposition algorithm described in (VASUDEVAMURTHY; RAJSKI, 1990) 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 (VASUDEVAMURTHY; RAJSKI, 1990).

From now on, we limit our discussion to the *fx* implemented in ABC (ABC: A system for sequential synthesis and verifiation), 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 a 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 its logic level.
- All divisors are stored in a priority queue, which is repeatedly accessed 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.

3 FAST-EXTRACT WITH CUBE HASHING

In this section, key aspects of the proposed *fxch* algorithm are presented. The pseudo-code of *fxch* is shown in Algorithm 1. The algorithm is based on *grouping* (section 3.1) of identical cubes for different outputs (in the case of the multiple-output SOP) and on efficient *hashing* of cubes and sub-cubes (section 3.2) during the extraction of divisors. The algorithm computes the set of all double-cube divisors up to four literals (section 3.3). The algorithm may create undesirable degenerate divisors, which require special treatment (section 3.4). Finally, the extraction procedure is described in section 3.5.

Algorithm 1: Fast-Extract with Cube Hashing

```
Input  : the original multiple-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 multilevel network
```

3.1 Cube Grouping

When decomposing 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* and *fxch*. One way to overcome this inefficiency is by grouping identical single-output cubes, resulting in a multiple-output SOP representation. The example below illustrates the current short-comings and the proposed mitigation. Consider the truth table and the respective SOP representation shown in Figure 3.1.

Figure 3.1: Truth table and the respective SOP representation originally used by ABC

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	$\overline{x_3}$

Notice the existence of two cube pairs with identical inputs, meaning that the representation shown in Figure 3.1 does not take advantage of the fact that some cubes differ only in their outputs. It has been demonstrated experimentally that the number of such cubes can be significant, and therefore processing cubes separately has an adverse impact on both run-time and memory usage. This work uses a better representation to mitigate this problem, as illustrated in Figure 3.2, where each cube representation consists of an input pattern and an output pattern.

Figure 3.2: New SOP representation used by FXCH

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	$\overline{x}_3$

3.2 Cube Hashing

The main technical contribution of this work 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 make 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 hits observed in the hash table enable us to find cubes that differ

in only one literal. This approach is linear in the number cubes.

An extension of the presented approach can be used 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 deletion 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 equal sub-cubes correspond to the *base*, i.e. the common part of the original cubes which remains after divisor extraction. Figure 3.3 illustrates these concepts by factoring a two-cube SOP expression.

Figure 3.3: Decomposition of a two-cube SOP expression
$F = (x_1\overline{x}_2x_3) + (x_1x_3x_4)$

Sub-cubes hash table		
Lit	$cube_1$	$cube_2$
1	$\overline{x}_2x_3$	x_3x_4
2	x_1x_3	x_1x_4
3	$x_1\overline{x}_2$	x_1x_3

$divisor = \overline{x}_2 + x_4$
$base = x_1x_3$

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 deletion 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:

$$F = x_1x_3(\overline{x}_2 + x_4)$$

As pointed out in (MISHCHENKO; BRAYTON, 2015), 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 run-time 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.

3.3 Divisors Functions

Unlike the preliminary implementation of *fxch* in (MISHCHENKO; BRAYTON, 2015), this 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 (KRAVETS, 2015).

The complete set of possible double-cube divisors with complements is summarized in Theorem 1 in (VASUDEVAMURTHY; RAJSKI, 1990). 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 the limit of four literals to divisors. It must be noted, however, that to handle *degenerate divisors*, *all* double-cube divisors with up to four literals must be checked.

3.4 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 handled properly, they lead to the selection of a divisor among other candidates based on an inaccurate assessment of divisor costs, and also the extraction process would not decompose all possible cubes, thereby reducing the quality of results.

Table 3.1: Set of possible extractions.

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

3.5 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 be 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 3.1 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 create a new cube and may invalidate one of the original cubes, depending on whether the resulting output pattern is equal to zero.

4 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 3.1.

4.1 Experimental Setup

This work implemented the algorithm described in Chapter 3 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, This work used a set of multiple-output PLA tables taken from an instruction decoder (EPFL Benchmarks, MULTI-OUTPUT PLA Behnchmakrs, 2015). These benchmarks demonstrate that the importance of factoring increases as the circuit size increases (KRAVETS, 2015). 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.

The present work compared against *jee* (KRAVETS, 2015) and ABC's available implementation of the original *fx* algorithm (VASUDEVAMURTHY; RAJSKI, 1990). 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 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.

4.2 Impact of Cube Grouping

To evaluate the impact of cube grouping presented in Section 3.1, two versions of *fxch* were implemented: one uses cube grouping ("CG") while the other does not. The experiment uses the decoder PLA tables as input to both versions. Table 4.1 lists the collected results for both implementations regarding run-time and peak memory usage. In the last row of the table, we present the arithmetic averages of the reduction ratios about not using cube grouping.

Table 4.1: The impact of using cube grouping

	Run-time (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**

The results show the benefits of using cube grouping when decomposing multiple-output Boolean functions; its use reduces both run-time 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.

4.3 Synthesis Results

The results of decomposing for each test case, are given in Table 4.2. The first column identifies each test case by the number of inputs and outputs. Table 4.2 presents an evaluation of the results in terms of run-time (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 about using ABC's original *fx* are given; they were calculated using the arithmetic mean.

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 to improve

Table 4.2: Logic synthesis results: comparison of *fxch*, *fx* and *jee*

	fxch in ABC				*fx* in ABC				*jee* factoring			
Design	t, sec	m, Mb	#nodes	#lvl	t, sec	m, Mb	#nodes	#lvl	t, sec	m, Mb	#nodes	#lvl
37/143	2.95	113	3835	22	18.18	11	4695	24	4.3	37	3587	24
38/67	1.16	59	3438	19	2.14	7	3727	18	2.0	25	3366	20
128/43	1.90	105	3051	18	2.43	9	3702	18	2.3	22	3191	18
128/53	1.63	102	2708	18	2.09	9	3261	19	2.0	23	2944	19
128/55	2.03	105	3079	18	2.46	10	3905	18	2.1	22	3069	20
128/69	2.72	107	3415	19	4.60	14	4295	20	2.7	28	3326	20
128/94	4.56	120	5140	21	9.50	20	6271	22	6.3	46	5266	24
128/104	3.98	121	4916	20	7.61	17	5853	21	5.7	44	4926	23
128/160	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**

run-time at the expense of the quality of results.

The following conclusions can be made from Table 4.2:

- In run-time, *fxch* is overall superior, but comes at the cost of using more memory. As the test-cases get larger, both the run-time 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*.

4.4 Scalability

This experiment has two objectives. The first is to examine the scalability of *fxch* as the problem size increases, while minimizing the effect of pre-processing techniques used by each algorithm. The experiment consists of synthesizing circuits whose output is one 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 4.3 gives the results for the 8 largest functions. It uses the same metrics as in the previous experiment in the evaluation. The name in the first column identifies an instance of the primes function by the number of inputs.

The results demonstrate the great run-time scalability of *fxch* for large problem instances. It completes the most complex test case containing $23,000$ cubes in 13 seconds.

Table 4.3: Results of the synthesis of the primality testing circuits.

	fxch in ABC				*fx* in ABC				*jee* factoring			
#inputs	t, sec	m, Mb	#nodes	#lvl	t, sec	m, Mb	#nodes	#lvl	t, sec	m, Mb	#nodes	#lvl
11	0.02	7	455	13	0.03	2	471	13	-	3	492	13
12	0.04	13	739	14	0.13	2	771	14	0.1	5	825	14
13	0.10	25	1355	15	0.53	2	1440	15	0.3	9	1419	15
14	0.25	50	2046	16	2.07	3	2401	16	1.4	20	2287	16
15	0.62	100	3670	17	7.99	5	4174	17	8.5	58	3989	17
16	1.55	202	6289	18	30.87	11	7448	18	41.2	151	6491	18
17	3.91	407	11413	19	129	23	11650	19	66.7	157	12096	19
18	12.97	827	17260	20	507	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**

Compared to *jee*, it takes on average 93% less run-time. *fxch* also provides the best node count savings. However, as the problem size increases, memory usage grows quickly.

5 CONCLUSION

This work presented a new, fully functional and very efficient implementation of the *fxch* algorithm for decomposition and factorization of Boolean expressions, which can handle degenerate divisors, redundant SOPs and, to some extent, single-cube containment. Improved run-time is the main advantage of the proposed method. Furthermore, when dealing with large single-output Boolean functions, the quality of results is better regarding the AIG node count.

It also presented a new way of representing multiple-outputs SOPs in ABC. The use of cube grouping as described in Section 3.1 has a substantial positive impact on performance. For now, however, the use of it is restricted to our *fxch* implementation, hence the need to ungroup the results before returning it to be processed by other ABC commands.

It is important to notice that this work generated a conference article that was accepted for publication in ASPDAC 2017. This confirms that the developed work can be considered as a contribution to the state of the art for factoring/decomposition.

REFERENCES

ABC: A system for sequential synthesis and verifiation. 2005. Http://www.eecs.berkeley.edu/ alanmi/abc/. Retrieved October 31, 2016.

BRAYTON, R. K.; HACHTEL, G. D.; SANGIOVANNI-VINCENTELLI, A. L. Multilevel logic synthesis. **Proceedings of the IEEE**, v. 78, n. 2, p. 264–300, Feb 1990. ISSN 0018-9219.

BRAYTON, R. K.; MCMULLEN, C. The decomposition and factorization of boolean expressions. **Proceedings of the International Symposium on Circuits and systems**, p. 49–54, May 1982.

CALLEGARO, V. et al. A domain-transformation approach to synthesize read-polarity-once boolean functions. **Journal of Integrated Circuits and Systems**, v. 9, p. 60–69, 2014.

CALLEGARO, V. et al. Read-polarity-once boolean functions. In: **2013 26th Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.: s.n.], 2013. p. 1–6.

COUDERT, O. Two-level logic minimization: An overview. **Integr. VLSI J.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 17, n. 2, p. 97–140, out. 1994. ISSN 0167-9260.

FLACH, G. et al. Drive strength aware cell movement techniques for timing driven placement. In: **Proceedings of the 2016 on International Symposium on Physical Design**. New York, NY, USA: ACM, 2016. (ISPD '16), p. 73–80. ISBN 978-1-4503-4039-7.

HUNTINGTON, E. V. Sets of independent postulates for the algebra of logic. **Transactions of the American Mathematical Society**, American Mathematical Society, v. 5, n. 3, p. 288–309, 1904.

INTEL. **Microprocessor Quick Reference Guide**. 2008. Http://www.intel.com/pressroom/kits/quickreffam.htm. Retrieved October 31, 2016.

KLOCK, C.; RIBAS, R.; REIS, A. Karma: um ambiente para o aprendizado de síntese de funções booleanas. **Brazilian Journal of Computers in Education**, v. 18, n. 02, p. 33, 2010. ISSN 2317-6121.

KLOCK, C. E. et al. Karma: A didactic tool for two-level logic synthesis. In: **2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)**. [S.l.: s.n.], 2007. p. 59–60.

KRAVETS, V. N. Application of a key x2013;value paradigm to logic factoring. **Proceedings of the IEEE**, v. 103, n. 11, p. 2076–2092, Nov 2015. ISSN 0018-9219.

MACHADO, L. et al. Iterative remapping respecting timing constraints. In: **2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2013. p. 236–241. ISSN 2159-3469.

MARTINS, M. et al. Open cell library in 15nm freepdk technology. In: **Proceedings of the 2015 Symposium on International Symposium on Physical Design**. New York, NY, USA: ACM, 2015. (ISPD '15), p. 171–178. ISBN 978-1-4503-3399-3.

MARTINS, M. G. A. et al. Boolean factoring with multi-objective goals. In: **2010 IEEE International Conference on Computer Design**. [S.l.: s.n.], 2010. p. 229–234. ISSN 1063-6404.

MCCALLUM, J. C. **Memory Prices (1957—2016)**. 2016. Http://www.jcmit.com/memoryprice.htm. Retrieved October 31, 2016.

MICHELI, G. D. **Synthesis and optimization of digital circuits**. [S.l.]: McGraw-Hill Higher Education, 1994.

MISHCHENKO, A.; BRAYTON, R. K. A linear divisor extraction algorithm. **Proceedings of the International Workshop on Logic and Synthesis**, Jun 2015.

MULTI-OUTPUT PLA benchmarks. 2015. Http://lsi.epfl.ch/benchmarks. Retrieved October 31, 2016.

PUGET, J. C. et al. Jezz: An effective legalization algorithm for minimum displacement. In: **Proceedings of the 28th Symposium on Integrated Circuits and Systems Design**. New York, NY, USA: ACM, 2015. (SBCCI '15), p. 19:1–19:5. ISBN 978-1-4503-3763-2.

VASUDEVAMURTHY, J.; RAJSKI, J. A method for concurrent decomposition and factorization of boolean expressions. In: **Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on**. [S.l.: s.n.], 1990. p. 510–513.

WAGNER, F. R.; REIS, A. I.; RIBAS, R. P. **Fundamentos de circuitos digitais.** Porto Alegre: Sagra Luzzatto, 2006.

WANG, L.-T.; CHANG, Y.-W.; CHENG, K.-T. T. (Ed.). **Electronic Design Automation: Synthesis, Verification, and Test**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. ISBN 9780080922003.