UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

VINICIUS NEVES POSSANI

Parallel Algorithms for Scalable Logic Synthesis & Verification

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. André I. Reis
Coadvisor: Prof. Dr. Renato P. Ribas

Porto Alegre
April 2019

CIP — CATALOGING-IN-PUBLICATION

Neves Possani, Vinicius

Parallel Algorithms for Scalable
Logic Synthesis & Verification / Vinicius Neves Possani. – Porto Alegre: PPGC da UFRGS, 2019.

135 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2019. Advisor: André I. Reis; Coadvisor: Renato P. Ribas.

1. AIG rewriting. 2. Tech mapping. 3. Equivalence checking. 4. Parallel computing. I. I. Reis, André. II. P. Ribas, Renato. III. Título.

To my family and all those who transmitted me knowledge.

ACKNOWLEDGMENT

I would like to thank my parents Jonas Possani and Mariane Possani as well as my sister Taíse Possani for the education, support and all the efforts to enable my studies since the beginning of my life. I would like to thank my girlfriend Bárbara Cavalheiro for many years sharing our life time, giving support and love to each other. Thank to my grandparents and all my family in general, that always motivating me to go further studying.

A special thank to my advisor Professor Dr. André Reis and to my co-advisor Professor Dr. Renato Ribas for the knowledge, advisement and opportunities, making this work possible. Also, I would like to thank Professor Dr. Leomar da Rosa Jr. and Professor Dr. Felipe Marques for all the knowledge transmitted in the beginning of my career as a researcher at UFPEL.

I would like to thank Professor Dr. Keshav Pingali for receiving me in his research group at UT Austin during my internship. It was a very important part of my studies where I learned from Professor Pingali and his students. A special thanks to the colleague Yi-Shan Lu for all the collaborative work and to Julie Heiland for all the support during my internship at UT Austin.

I would like to thank Dr. Alan Mishchenko from UC Berkeley for kindly sharing his research knowledge and personal experience, helping me in several subjects during the development of this work.

A special acknowledgement to Professors Dr. Pierre-Emmanuel Gaillardon, Dr. Paulo Butzen and Dr. Marcelo Johann that participated as members of the jury in the Theses defense. Thanks for the high qualified evaluation of this work, based on interesting question and comments.

Finally, I am grateful for all my research colleagues from UFRGS and UFPEL for many years learning with each other in a very positive, creative and funny environment.

Thanks to Brazilian funding agencies CNPq, CAPES (PDSE) and FAPERGS that supported the research presented in this work.

ABSTRACT

The design of digital integrated circuits relies on gradually compiling a circuit specified by hardware description language into its physical implementation layout. Such a design flow is strongly dependent of a tool chain known as electronic design automation (EDA) tools. Currently, the high complexity of system-on-chips and the increasing demand for hardware accelerators are imposing new challenges on the EDA field. Parallel computing is a trend to enhance scalability of EDA tools using widely available multicore platforms. In order to benefit from parallelism, well-known EDA algorithms have to be reformulated and optimized for massive parallel environments. This work aims to enable parallelism during *logic synthesis and verification* phases of EDA flow, in order to significantly improve runtime while handling very large designs. We are rethinking algorithms such as multi-level logic optimization, technology mapping and combinational equivalence checking (CEC) to achieve extensive parallelism. Such algorithms are strongly correlated to each other in the design flow and work on directed-acyclic graphs called AIGs (AND-inverter graphs), which are irregular and sparse structures. The time spent for synthesis and verification of large design comprising millions of AIG nodes is becoming more critical, requiring several hour for synthesis and even more than a day for verification. Therefore, we are proposing a parallel flow based on a fine-grain parallel AIG rewriting method for multi-level logic optimization and a fine-grain parallel LUT-based technology mapping that enable fast solutions with competitive quality-of-results (QoR). Moreover, we are exploiting data sharing and data independence to enable parallelism in a modern CEC engine for faster verification. Experimental results have demonstrated that the proposed parallel methods are able to accelerate the design flow when compared to the ABC tool, which is an industrial-strength academic tool comprising state-of-the-art algorithms for logic synthesis and verification. When optimizing designs comprising millions of AIG nodes and running at 40 processor cores, the proposed parallel methods for AIG rewriting and technology mapping are up to 36x and 25x faster than the respective ABC commands *rewrite* and $\&if$, with similar QoR. The proposed parallel CEC is able to significantly reduce the verification runtime when compared to the ABC CEC engine and to a commercial verification tool. For example, we observed some expressive improvements where the CEC runtime went down from 19h to only 18min, representing a speedup of 63x.

Keywords: AIG rewriting. tech mapping. equivalence checking. parallel computing.

Algoritmos Paralelos para Síntese Lógica & Verificação Escalável

RESUMO

O projeto de circuitos integrados digitais consiste em gradualmente compilar um circuito especificado em uma linguagem de descrição de hardware em seu leiaute de implementação física. Este fluxo de projeto é fortemente dependente de uma cadeia de ferramentas conhecidas como ferramentas de automação de projetos eletrônicos (do inglês *EDA*). Atualmente, a alta complexidade de sistemas integrados em um chip e a crescente demanda por aceleradores de hardware estão impondo novos desafios no campo de EDA. Computação paralela é uma tendência para aumentar a escalabilidade de ferramentas de EDA usando plataformas multicore. Com o objetivo de usufruir do paralelismo, algoritmos de EDA bem estabelecidos precisam ser reformulados. O objetivo deste trabalho é viabilizar o paralelismo durante a fase de *síntese lógica e verificação* do fluxo de EDA, a fim de melhor o tempo de execuç ao manipular circuitos grandes. Neste trabalho, algoritmos para otimização lógica multi-nível, mapeamento tecnológico e checagem de equivalência combinacional (CEC) estão sendo repensados para viabilizar paralelismo massivo. Tais algoritmos estão correlacionados e operam sobre grafos acíclicos direcionados chamados AIGs (AND-inverter graphs), os quais são estruturas irregular e esparsas. O tempo gasto durante a síntese e verificação de grandes projetos com milhões de nodos e AIG está se tornando mais crítico, requerendo horas para a síntese e até mesmo mais de um dia para a verificação. Este trabalho propõe um fluxo paralelo baseado reescrita de AIG e mapeamento tecnológico baseado em LUTs, ambos com um grão fino de paralelismo, os quais viabilizam resultados rápidos sem sacrificar a qualidade dos resultados. Além disso, este trabalho explora compartilhamento de dados e independência de dados para habilitar paralelismo em um módulo moderno de CEC, permitindo a verificação mais rápida de circuitos grandes. Resultados experimentais demonstram que os métodos paralelos propostos têm potencial para acelerar o fluxo de projeto quando comparado a ferramenta ABC, a qual é uma ferramenta acadêmica com capacidade industrial para síntese lógica e verificação. Quando aplicados a circuitos compostos por milhões de nodos de AIG e executados em 40 núcleos de processamento, os métodos paralelos propostos para reescrita de AIG e mapeamento tecnológico são até 36x e 25x mais rápida do que o os respectivos comandos *rewrite* e $\&if$ da ferramenta ABC, com qualidade de resultados similares. O método proposto para CEC paralelo é capaz de reduzir significativamente o tempo de veri-

ficação comparado ao método de CEC da ferramenta ABC e a uma ferramenta comercial de verificação. Por exemplo, foram observados melhorias expressivas onde o tempo de execução foi reduzido de 19h para apenas 18min, representando uma aceleração de 63x.

Palavras-chave: Reescrita de AIG. mapeamento tecnológico. checagem de equivalência. computação paralela.

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	AND-Inverter Graph
ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagram
CAD	Computer Aided Design
CEC	Combinational Equivalence Checking
DAG	Directed Acyclic Graph
DFS	Depth-First Search
DSD	Disjoint-Support Decomposition
EDA	Electronic Design Automation
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
LUT	Look-Up Table
MFFC	Maximum Fanout-Free Cone
MIG	Majority-Inverter Graph
NPN	Negation-Permutaion-Negation
POS	Product-Of-Sums
QoR	Quality-of-Results
ROBDD	Reduced-Ordered Binary Decision Diagram
SAT	Satisfiability
SoC	System-on-Chip
SOP	Sum-Of-Products
TFI	Transitive Fanin
TFO	Transitive Fanout
VLSI	Very-Large Scale Integration

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

1 INTRODUCTION

The design of digital integrated circuits (ICs) relies on gradually compiling a circuit specified by hardware description language (HDL) into its physical implementation layout. Such a design flow is strongly dependent of a tool chain known as electronic design automation (EDA) tools (WANG; CHANG; CHENG, 2009). Currently, the high complexity of system-on-chips (SoCs) and the increasing demand for hardware accelerators are imposing new challenges on the EDA field. Moreover, shorter design cycles are directly related to productivity and project cost. In this sense, the concept of EDA 3.0 points to a new generation of parallel and distributed computer-aided design (CAD) tools, which must be able to quickly handle a huge amount of information like in big data and cloud computing services (STOK, 2018).

The increasing availability of supercomputers has leveraged novel possibilities for other fields such as EDA. Aiming to exploit massive parallelism, the next generation of EDA tools need to scale for dozens of processors. Even though part of the current EDA tools present a reasonable scalability, some of them do not scale beyond 12-16 threads and others need to be completely replaced (STOK, 2013).

EDA algorithms are responsible for solving complex optimization and decision problems. Generally speaking, such algorithms can be classified in four main categories: high-level synthesis, logic synthesis, physical synthesis and verification (HASSOUN; SASAO, 2002). On one side, optimization algorithms aim to improve the quality of current circuit description in some aspects, targeting to meet the design constraints and goals in terms of circuit power, performance and area. On the other side, verification algorithms are applied in several stages on the design flow for checking the circuit correctness. A great part of the problems solved by EDA tools rely on graph-based solutions. Similarly to other areas of computer science, EDA algorithms are commonly evaluated by considering their quality-of-results (QoR) and runtime as criteria. In this sense, to exploit parallelism in such algorithms is promising to decrease runtime and to enable QoR improvements within a feasible execution time.

1.1 Problem Definition

It is predicted that future generations of integrated circuits will contain trillions of logic gates (STOK, 2013; STOK, 2014). Therefore, fast and scalable EDA algorithms are fundamental in several stages of the design flow for both application specific integrated circuit (ASIC) and field programmable gate array (FPGA). In this thesis, we are focusing in three key problems from logic synthesis and verification field defined as follows.

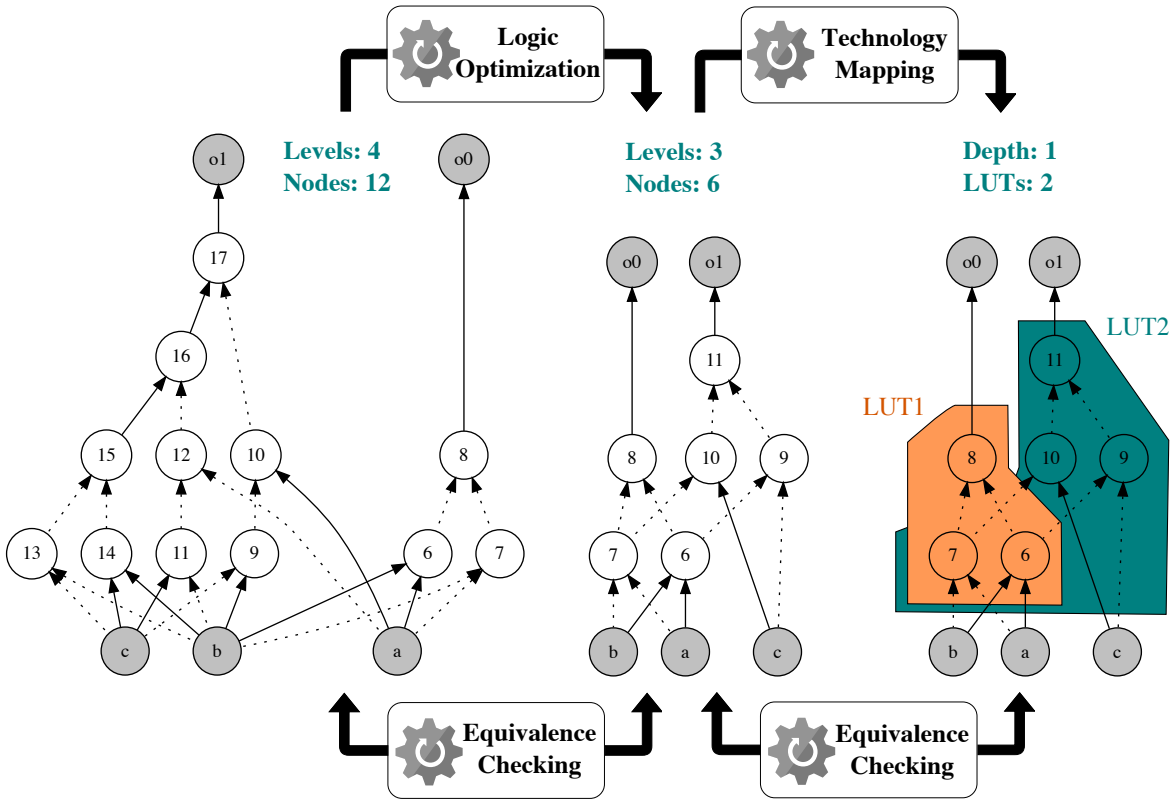


Figure 1.1: Initial logic network (a), optimized network delivered by the multi-level logic optimization (b) and mapped network delivered by the technology mapping into 3-input LUTs (c). A CEC engine verifies the design functionality at each step of the design flow.

Multi-Level Logic Optimization Problem: Given a logic network N, representing the circuit as a directed acyclic graph (DAG), obtain an equivalent network N' minimizing a cost function σ. Conventionally, the cost function σ is based on technology independent metrics such as the number of *nodes* and *levels* in a DAG. These metrics are used for guiding logic optimizations because they correlate to the circuit *area* and *delay*, respectively.

The overall goal of multi-level logic optimization is to prepare the logic network to be mapped into a given target technology, i.e., the technology mapping process,

(MICHELI, 1994). For instance, Fig. 1.1(a) shown a logic network represented as an *AND-iverter graph* (AIG), comprising 2-input AND operators and inverters. By applying logic optimizations on the AIG shown in Fig. 1.1(a), we can obtain a reduced AIG comprising six nodes and four levels, as illustrated in Fig. 1.1(b).

LUT-Based Technology Mapping Problem: Given a technology independent logic network N', obtain a logically equivalent network N'' representing a covering into k-input lookup tables (LUTs), while minimizing a given cost function α and respecting the given design constraints ϕ. A k-input LUT is a programmable cell which is able to implement any k-variable Boolean function. In a typical scenario, the technology mapping covers the logic network aiming to minimize the number of used LUTs while meeting the timing (delay) constraints (CHEN; CONG; PAN, 2006).

Consider the optimized AIG previously delivered by the multi-level optimization from Fig. 1.1(b). Intuitively, it is more convenient to perform the technology mapping on top of the optimized graph shown in Fig. 1.1(b) than the original graph shown in Fig. 1.1(a). For the sake of simplicity, in this example, we abstract the details on mapping goals and constraints. A typical LUT-based technology mapper enumerates many cuts (subgraphs) representing subcircuits which fit into a *k-input* LUT and elects the best cut for each node based on the corresponding cut area and delay costs (MISHCHENKO et al., 2007). The last task of the mapper is to select a subset of the best cuts which covers the logic network into k-LUTs, as shown in Fig. 1.1(c).

Combinational Equivalence Checking (CEC) Problem: Given two logic networks N and N', verify whether N is functionally equivalent to N'. CEC is commonly applied after multi-level logic optimizations and technology mapping to ensure the right circuit functionality between the original circuit specification and its optimized/mapped version (HASSOUN; SASAO, 2002).

Considering our general example depicted in Fig. 1.1, we first need to check and ensure logic equivalence between the initial logic network presented in Fig. 1.1(a) and its optimized version presented in 1.1(b). Therefore, the CEC engine is responsible for proofing the equivalence or non-equivalence of the networks under verification. If the networks are equivalent, then the optimized version can be carried to technology mapping. Analogously, the equivalence checked between the generic network used as input to the technology mapper, shown in Fig. 1.1(b), and the final mapped network, shown in Fig. 1.1(c). Thus, if the networks are equivalent, the mapped network is delivered to the physical synthesis phase of the design flow, e.g. placement and routing steps.

1.2 Motivational Examples

This section introduces three motivational examples demonstrating challenges on runtime, scalability and QoR across the logic synthesis and verification problems previously defined.

Motivation on Multi-Level Logic Optimization: Local transformations, such as *logic rewriting*, play an import role during the multi-level logic optimization in modern design flows. Typically, rewriting algorithms incrementally replace subgraphs of the logic network by a better representation which is retrieved from a table of precomputed subgraphs. Currently, the time spent to synthesize large AIGs with more than ten million nodes has become considerable, even when fast rewriting methods are applied.

A representative case is the command *rewrite* in the ABC tool, which is an industrial-strength academic tool for logic synthesis and formal verification (MISHCHENKO; CHATTERJEE; BRAYTON, 2006; Berkeley Logic Synthesis and Verification Group,). We have observed that this command takes approximately 18 minutes on a modern processor to perform a single iteration of rewriting for a 33-million-node AIG. The command *rewrite* is one of the main algorithms executed four times in the well-known ABC scripts *resyn2* and *dc2* for delay- and area-oriented synthesis, respectively. Moreover, these scripts are often applied many times to compensate for the local nature of the transformations. Hence, it may result in several hours of execution due to dozen incremental passes of algorithms such as AIG rewriting. These observations and the wide availability of multi-core processors motivate the investigation on how to enable parallelism for AIG rewriting.

Motivation on LUT-Based Technology Mapping: The multi-level logic optimizations have direct impact on the technology mapping quality due to the *structural bias*, which is the influence of the underlying logic representation on the quality of technology mapping (CHATTERJEE et al., 2006b). Moreover, there is a weak correlation between the cost metrics adopted during multi-level logic optimization and LUT-based mapping.

On the one hand, area-oriented logic optimizations aim to minimize as hard as possible the number of AIG nodes. On the other hand, during the LUT-based technology mapping, the number of LUTs in the mapped netlist determines the circuit area. In this context, a recent study demonstrated that, more nodes in the AIG (Gate Count) can enable LUT count reductions, as shown in Fig 1.2 (LIU; ZHANG, 2017). Therefore, there is much room for novel algorithms that decrease the gap between the multi-level logic optimization and technology mapping within a moderate runtime.

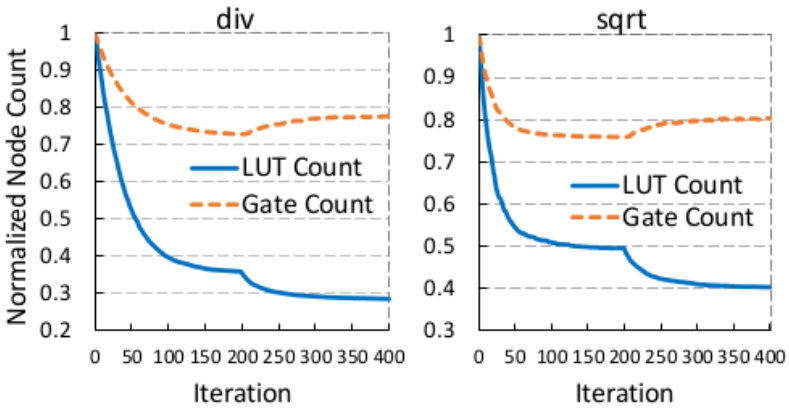


Figure 1.2: Miscorrelation between AIG node count (Gate Count) and LUT count during technology mapping. Source (LIU; ZHANG, 2017).

Motivation on Combinational Equivalence Checking: CEC is known to be a co-NP-complete problem which relay on solving several complex decision problems usually modeled as Boolean satisfiability (SAT). In order to demonstrate the complexity of CEC when dealing with large designs, we consider one instance of the problem. Two versions of the design, the original one and the one after the synthesis, containing 14 million and 9 million nodes, respectively, were represented as AIGs and given to command *&cec* in ABC (Berkeley Logic Synthesis and Verification Group,). ABC took more than 24 hours to prove equivalence, making it clear that verification becomes harder as the design size increases, reinforcing the need for parallel CEC in order to improve scalability of EDA tools.

It should be noted that scalable CEC techniques are used as an important building block in other applications, which depend on efficient computation of equivalence classes of internal nodes such as: removal of functionally equivalent logic in the design during the logic synthesis process; computation of structural choices, which enable area and delay improvements after technology mapping (CHATTERJEE et al., 2006a); sequential equivalence checking based on register/signal correspondence (BJESSE; CLAESSEN, 2000; MISHCHENKO et al., 2008); bridging circuits for the implementation and the specification in engineering change orders (ECOs) (KRISHNASWAMY et al., 2009); and a number of other utility packages. Therefore, it is desired to enable parallelism on CEC engines to achieve moderated execution time for all those applications which rely on CEC.

1.3 Proposal and Contributions

The scientific investigation addressed in this thesis is formulated as the following research question: *how to enable parallelism for logic synthesis and verification algorithms in order to improve runtime and scalability?* In this sense, this work aims to rethink logic synthesis and verification algorithms to achieve extensive parallelism in multi-level logic optimization based on AIG rewriting, LUT-based technology mapping and CEC.

The overall objective of this work is a challenging task, since the most part of logic synthesis comprises algorithms working on irregular data structures like sparse graphs and verification algorithms rely on SAT solving. Graph partitioning can be applied to enable the parallelism in such irregular algorithms. On the one hand, the graph partitioning can introduce a negative bias in the QoR when dealing with optimization problems like AIG rewriting and technology mapping. It is because the logic synthesis tool loses the global view of the network and it is hard to efficiently exploit optimizations in the boundaries among partitions. On the other hand, graph partitioning is more useful for decision problems like CEC, since it aims only to check functionality correctness regardless the circuit area and delay aspects. In this sense, we are adopting the most appropriated strategy of parallelism for each kind of problem.

For the parallel AIG rewriting and LUT-based technology mapping, which are optimization algorithms, we adopted the concept of *operator formulation* and the Galois system to exploit fine-grain parallelism without graph partitioning. The *operator formulation* is a data-centric abstraction of algorithms, which are described in terms of atomic actions on data structures (PINGALI et al., 2011). The Galois system provides a programming model in a multi-processor environment with shared memory to speculatively exploit parallelism on non-overlapping subgraphs (LENHARTH; PINGALI, 2015).

For the parallel CEC, which is a decision algorithm, we adopted a coarse-grain parallelism based on graph partitioning and the well established POSIX Threads (Pthreads) standard for shared-memory computing (BUTENHOF, 1997). Since CEC is a decision problem, it is not sensitive to degradation in the quality of results due to the graph partitioning. We revisit this discussion on graph partition effects later on in this work.

In both scenarios, considering the Galois system or Pthreads, the main challenges towards our objective rely on rethink algorithms and data structures to efficiently work in a parallel environment. Such algorithms comprise several shared data-structures and require cleaver solutions to ensure mutual exclusion without inserting runtime bottlenecks.

The major contributions, uniqueness and impact of this thesis are:

- We start revisiting the multi-level logic optimization by proposing a fine-grain parallel AIG rewriting. In this work, we introduce a set of principles that describe how to unlock the parallelism during AIG rewriting. The proposed method speculatively discovers non-overlapping subgraphs and rewrite them in parallel. Experimental results demonstrate that the proposed parallel AIG rewriting is able to speed up the multi-level logic optimization by scaling to dozens of threads. Our method rewrites AIGs comprising millions of nodes in few minutes without QoR degradation.
- We are proposing a fine-grain parallel technology mapping for LUT-based FPGAs. Experimental results have demonstrated that our parallel mapper is able to bring the mapping runtime form more than one hour to only few minutes with comparable QoR. By using the same principles of parallelization, the proposed AIG rewriting and the technology mapper can be integrated into a synergistic way. The proposed integration enables to perform logic rewriting optimizations driven by technology mapping cost functions. Our fast parallel mapper can be used to increase the exploration in the solution space for improving QoR while keeping a low runtime.
- Three novel models are introduced to enable massive parallelism for speeding up two crucial time-consuming CEC tasks, mitering and SAT sweeping. The proposed parallel CEC engine handles large designs comprising millions of AIG nodes, and scales to many threads unlocking the potential of parallel environments available through cloud computing. Experimental results showed significant runtime improvement when comparing to both single-threaded ABC and parallel commercial CEC engines. In some cases, the proposed solution reduces the CEC runtime from more than one day to only a few minutes/hours.
- The uniqueness of this work is that we are enabling parallelism to solve three important and complex problems for logic synthesis and verification, proposing effective parallelization strategies according to the problem characteristics. This is the first work in the literature to propose fine-grain parallel approaches allowing logic optimization and mapping of many graph nodes at the same time, rather than applying a graph partitioning for synthesizing and mapping each partition sequentially. Moreover, the proposed parallel CEC enables parallelism in two key point of the verification engine, accelerating the most critical components of the algorithm which rely on SAT solving. The proposed solutions present promising speedups when compared to state-of-the-art methods to solve these three problems.

- The impact of this work is related to both ASIC and FPGA design flows that can be directly benefited from the proposed parallel AIG rewriting and parallel CEC, since these two algorithms perform technology independent tasks. Moreover, it is worth to mention that there are common steps in the FPGA and ASIC technology mapping. In this sense, by efficiently enabling parallelism in FPGA side, we are on the path for enabling parallelism in ASIC side. Therefore, the contributions of this work can impact a real design flow as well as leverage novel research and development in parallel EDA.

1.4 Thesis Structure

We start presenting some background on parallel programming in Chapter 2, since enabling parallelism in software is in the core of this thesis. We conclude the chapter summarizing and justifying our choices of parallel programming models according to the problem characteristics.

The three main contributions of this work are presented in the three subsequent chapters of this manuscript. Each chapter presents a brief review on the addressed problem and motivations followed by a background section and a revision on related works. The proposed approaches and contributions are presented at the end of each chapter, supported by a section of experimental results.

- In Chapter 3, we introduce our parallel rewriting by presenting the challenges and the proposed strategies to enable efficient management of memory allocation, shared data-structures and routine interactions in a parallel environment.
- In Chapter 4, we present our parallel LUT-based technology mapping by manging data dependency during cut computation as well as introducing insights on how to integrate AIG rewriting and LUT mapping in a synergistic way.
- In Chapter 5, we define the three advanced models proposed in this work to enable parallelism for CEC. We discuss and demonstrate how the proposed models can be applied independently or combined to unlock CEC speedups.

We conclude this work in Chapter 6 by summarizing our contributions on the logic synthesis and verification field. We highlighting a set o principles for unlocking parallelism in the proposed algorithms. These principles can be useful for leveraging the parallelism in other related algorithms. Finally, we discuss future steps to extend this work.

2 REVIEW ON PARALLEL PROGRAMMING

In this chapter, we introduce some concepts on parallel computing while discussing different techniques and technologies for parallel programming. We discuss the lower level of abstraction and native approaches for parallel programming such as the standard POSIX Threads (Pthreds), passing for other general approaches up to specific graph analytic frameworks for implementing parallel graphs algorithms. The main objectives of this chapter is to provide the necessary background on parallelism for a better understanding of this work as well as to justify the choices for adopting some models to design parallel algorithms.

2.1 Background

2.1.1 Processes and Threads

Parallelism can be exploited in several different levels such as instruction-level, thread-level and others. In this work, we are interested in exploiting parallelism in thread-level so we are limiting the discussion to this scope. One source of parallelism in software arises from the decomposition of a program into subproblems which can be solved independently in parallel while keeping the original semantic of the program. Generally speaking, a *computer process* is an instance of a program executed and managed by an operating system (OS). In the shared-memory model, the parallelism can be exploited by decomposing and mapping a process to many independent lines of executions called *threads*. Threads share memory regions of the main process such as heap, code section, data section and each thread has its own stack section. In distributed system model, the parallelism is exploited by decomposing the computation into many process running in different host machines where the processes communication is usually performed by message passing. In the following, we refer to parallel tasks in a generic way, which can be viewed as threads in shared-memory model or processes in distributed systems.

2.1.2 Static and Dynamic Parallelization

Designing efficient parallel algorithms relies on a set of components and the choices on how to deal with these components are strongly related to the algorithm characteristics, data structures and the target parallel architecture (GRAMA; GUPTA; KUMAR, 2003). Some typical components involved in parallel algorithm design are:

- Identify portions of work that can be processed in parallel;
- Decompose the computation and assign pieces of work to parallel processors;
- Manage shared data structures accessed concurrently;
- Synchronize the parallel tasks;
- Balance the work load among processors.

The identification of available parallelism and the decomposition of computation into smaller tasks may be defined statically or dynamically. In the static task generation, the parallelism is discovered and decomposed in smaller tasks before the algorithm starts its execution. Usually, the static analysis is performed at compile time using task-dependency graphs for determining the parallelism.

In the dynamic task generation, the identification of parallelism and the computation decomposition are performed during the execution time. Therefore, the task dependencies are known as the computation is executed and the decomposition strategy is applied. There are several techniques for decomposing a problem in smaller and independent tasks like recursive decomposition, e.g. merge sort and graph partitioning.

The interaction among tasks can also be classified as static and dynamic. In static case, the interactions and the respective stages each interaction happens are known in advance before the program execution. Dynamic interactions happen during the program execution when it is not possible to determine in advance which tasks need to interact in a given order.

Mapping tasks onto processes/threads is strongly related to the task generation and interaction discussed before. The mapping aims to produce an efficient work load distribution. Therefore, static generated tasks can be mapped either dynamically or statically whereas dynamic generated tasks can be mapped only dynamically. There are several techniques for static mapping tasks based on independent computation and the adopted decomposition scheme. Dynamic mapping is usually more complicated to be designed, requiring the management of tasks dependencies at the execution time.

2.1.3 Parallelism Granularity

The granularity of the problem decomposition is related to the size and number of derived smaller tasks (HWANG; JOTWANI, 2011). In a *fine-grain* parallelism a problem is decomposed into many small tasks. On the other hand, *coarse-grain* parallelism decomposes the problem in few larger tasks. A *medium-grain* parallelism can also be considered, which lies on the middle of this granularity spectrum. The target problem characteristics and the number of available physical processors can be used for determining the most appropriated granularity scheme.

2.1.4 Mutual Exclusion

In a parallel environment several different threads may need to access a shared resource simultaneously, requiring a policy for ensuring the program consistence. Therefore, mutual exclusion is used for coordinating the access of a given resource in a serial way. Conventionally, a *mutex* can be used for implementing mutual exclusion, acting as a virtual lock in shared resources. Firstly, a thread need to acquire the lock of a given shared resource and then to perform the read/write operation on that. When the operation is finished, the lock must be released in order to enable that other threads access such resource. The interval between acquiring and releasing the lock is called *critical section* (GRAMA; GUPTA; KUMAR, 2003).

One possible drawback related to the mutex is the possibility of introducing *deadlock* and *livelock* in the program execution. A deadlock happens when two or more threads need to access shared resources and the locks are never available due to a cyclic dependence of locks or due to a thread stop its execution before releasing a given lock (COFFMAN; ELPHICK; SHOSHANI, 1971). A livelock may occurs when threads are trying to recover from a deadlock by releasing successively their already acquired locks without making any progress. These issues can also occur in distributed systems which need concurrency control or transactions.

Alternatively, one can consider to use *atomic operations* instead of mutex. However, atomic operations are not portable since they are strongly dependent on the processor instruction set and the adopted compiler specifications. Overall, programs and data structures which ensure mutual exclusion through a given technique are usually called *tread-safe*.

2.1.5 Performance Metrics

The *speedup* is a well established metric for evaluating how fast a parallel program executes when compared to the reference sequential program. Considering that a given problem is solved sequentially with runtime T_1 and it is solved in parallel using p processors with runtime T_p, the speedup is defined as T_1/T_p (EIJKHOUT, 2014). The Amdahl's Law is commonly used to predict the maximum speedup of a program running in multiple processors. This law is based on the observation that the theoretical speedup of a program is limited by the sequential segments of a program which cannot be parallelized, independently of how many processors are being used (AMDAHL, 1967).

The *scalability* of a parallel program is related to how the program runtime behaves as the number of processors and/or the problem size increases. The *strong scaling* is measured by increasing the number of processors while keeping the problem size constant. In practice, the strong scaling is the same as speedup presented above (EIJKHOUT, 2014). The *weak scaling* is measure by increasing both the number of processors and the problem size. In weak scaling, larger problems are being solved but the amount of work per processor is roughly invariant, since the number of processors and the problem size tend to grow according to each other.

2.1.6 Regular and Irregular Algorithms

Generally speaking, algorithms can be classified as *regular* and *irregular*. Regular algorithms operate on regular data structures such as dense array and matrix. In irregular algorithms, the computation is performed on sparse graph and trees organized on irregular pointer-oriented data structures without any particular organization. Therefore, exploiting parallelism on irregular algorithms is more complicated than in regular ones. Pingali *et al.* define the parallelism on such irregular structures as *amorphous data-parallelism* (PINGALI et al., 2009; PINGALI et al., 2011).

In general, it is hard to determine parallelism in irregular algorithms at compile time by statically performing a computation-centric analysis of the code. In several irregular applications the task dependencies and localities can be discovered only during the execution time. In this cases, the parallelism is strongly related to the input data and actions the algorithms perform on that. Therefore, a dynamic data-centric analysis tends to be more appropriated for exploiting parallelism in irregular algorithms (PINGALI et

al., 2011).

The logic synthesis and verification techniques investigated in this work rely on irregular graph-based data structures. Although graph partitioning can be applied to enable parallelism in some cases such as decision problems arising in combinational equivalence checking, it is not the best alternative for enabling parallelism in optimization problems such as AIG rewriting and technology mapping.

2.2 General Programming Models

Considering this scenario on regular and irregular algorithms, in the remaining of this chapter, we introduce different approaches for parallel programming while discussing the potential of each one to deal with the challenges of the irregular algorithms we are interested in this work.

2.2.1 POSIX Threads

Pthreads is a standard API for managing thread in shared-memory programming model, which is widely supported in several systems and multicore platforms (BUTENHOF, 1997). Through the Pthreads API the programmer can define sections of the code to be parallelized by creating threads and specifying the data and the operation each thread must execute. The API also provides interfaces to join thread back to the initial process as well as mutex-based functions which help the programmer to deal with mutual exclusion in critical sections.

One of the main advantages of Pthreads is that the programmer has freedom to manage threads in a low level without abstraction overheads. On the other hand, such a management comes at the cost of a high level of difficulty to design parallel programs. For instance, implementing parallel irregular algorithms using Pthreads requires the development of all necessary thread-safe data structures and strategies to figure out the parallelism dynamically. In this context, Pthreads does not provide abstractions for improving the development productivity and making the development easier.

However, in those cases where the massive data being handled by the program can be divided into independent partitions, Pthreads provides an efficient and simple way to distribute the computing among the processor cores. In this scenario, Pthreads help to fully exploit the processing resources of the target machine without to insert overheads due to the abstractions of a given programming model. Therefore, it is wise to adopt Pthreads for those applications in which data partitioning is feasible and it does not lead to QoR degradation. We revisit this discussion in Chapter 5, demonstrating how we apply graph partitioning and Pthreads to build the proposed engine for parallel CEC.

2.2.2 Other Approaches

There exist other general programming models to exploit parallel computing in shared-memory architectures. It is worth to mention the *Open Multi-Processing (OpenMP)* and the *Intel Threading Building Blocks (TBB)* approaches. OpenMP is an alternative to Pthreads by increasing the level of abstraction to programmers which only need to specify code sections and loops which can be parallelized in a shared-memory model (OpenMP, 1997; DAGUM; MENON, 1998). Basically, in this model, the parallelism is exploited at compile time by translating parallel loops determined by the programmer into data-parallel code. Intel TBB is a C++ library designed by Intel and can be viewed as a complement to other techniques (REINDERS, 2007). The library provides parallel executor such as $parallel\_for$, $parallel\_while$ and a set of concurrent (thread-safe) containers such as *queue*, *vector* and $hash\_map$, among others. The Intel TBB manages thread executions by creating and evaluating task-dependency graphs dynamically, being more appropriate for implementing irregular algorithms.

When thinking on distributed computing, the *Message Passing Interface (MPI)* is an interesting and generic alternative. MPI is a system which enables distributed and parallel computing by coordinating process communications through message passing (GROPP; LUSK; SKJELLUM, 1999). In distributed computing, many autonomous machines, also known as nodes, receive tasks and data from the initial MPI executor node. The MPI provides a standard for communicating and synchronizing processes based on protocols. MPI can be used together with shared-memory approaches such as Pthreads and OpenMP to exploit parallelism by distributing many tasks to a set of nodes which use mutithreading for processing those tasks. In this work, MPI can be viewed as a complementary solution to the shared-memory parallel algorithms proposed herein. In this sense, one can use MPI for distributing tasks among several processing hosts and applying the proposed parallel solutions for exploiting the power of multi-core architectures at each processing host.

2.3 Parallel Graph Analytic Tools

As previous discussed, many EDA tools are strongly based on a set of graph-based algorithms. Therefore, in (STOK, 2014), Stok suggests that it is time to learn how EDA can benefit from massive parallel frameworks used in graph analytics field. Therefore, we start presenting a detailed view of the Galois System followed by a brief overview of other graph analytics frameworks.

2.3.1 Galois System

Considering the motivations of this work as well as the demands and complexity involved in the development of parallel logic synthesis methods, currently, we consider the Galois programming model the most appropriate approach for accomplishing this task. In the following we present the main reasons for considering this programming model and, in the sequence, we present the organization and the features of Galois system (ISS Group, The University of Texas at Austin,).

- It is a model natively designed to exploit parallelism on irregular graph algorithms.
- It provides thread-safe data structures, supporting graph topology modifications.
- It enables fine-grain parallelism without the need of graph partitioning.
- It manages threads efficiently with low overheads due to abstractions.
- It is a programmer friendly model for write the code.

Galois is a shared-memory system that provides a data-centric programming model to exploit *amorphous data parallelism* in irregular graph algorithms (PINGALI et al., 2011). It is based on an abstraction of algorithms called the *operator formulation*. In this abstraction, there is a local view and a global view of algorithms.

- The local view is described by an *operator*, which specifies an action that modifies the state of the graph atomically. Each application of the operator is called an *activity*, and the region of the graph modified by an activity is called its *neighborhood*, as shown in Fig. 2.1.
- In general, there may be many places in a graph where an operator can be applied. If there is an order in which these operator applications must be performed, that is specified by the *schedule*, which provides a global view of the algorithm. For the

algorithms of interest in this work, operator applications may be performed in any order, so these are called *unordered* algorithms.

- Usually, each activity modifies only a small portion of the overall graph. Therefore, for unordered algorithms, activities that modify non-overlapping regions of the graph can be performed in parallel without changing the semantics of the program. A pair of activities with overlapping neighborhoods can be performed in either order but not concurrently.

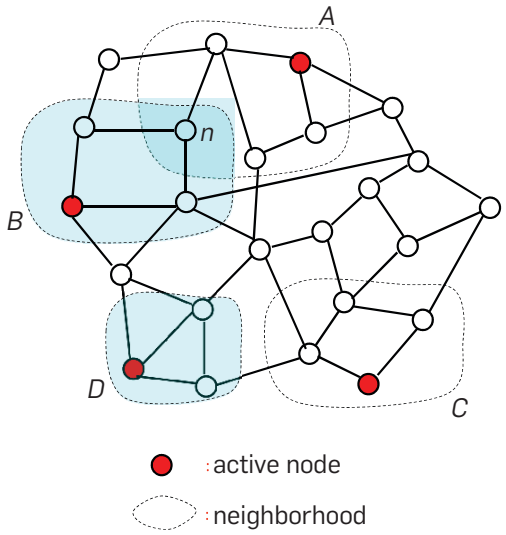


Figure 2.1: Abstraction of active nodes and their respective neighborhoods. Source (LENHARTH; NGUYEN; PINGALI, 2016)

From the programmer point of view, the Galois system provides C++ thread-safe data structures such as graphs and sets as well as parallel executors such as *for_each, do_all* (LENHARTH; PINGALI, 2015). The thread-safe graphs provided in Galois package supports different types of operations such as read-only, read-write and also topology modifications in the graph structure. Thread-safe sets are used to implement worklists to store active nodes. The parallel executor consumes nodes from the worklist and dynamically assigns them to threads. Operator execution is speculative and optimistic in the sense that the activities are assumed to be non-conflicting but Galois system dynamically treats and reschedules activities if conflicts happen. Galois scheduling is non-deterministic although it can be modified to work deterministically with some runtime cost. The dynamic management of thread conflicts is needed in irregular algorithms, unlike in regular algorithms in which non-conflicting activities can be found and scheduled statically.

Optimistic scheduling of activities is implemented as follows. Graph elements have exclusive locks to ensure mutual exclusion when threads are changing the graph.

Therefore, Galois system manages thread conflicts in the owners of locks from graph elements. Threads hold the abstract locks until the end of an activity or until a conflict is detected. For instance, consider two threads t_1 and t_2 that are processing the active nodes n_1 and n_2, respectively, and n_3 is a shared neighbor of n_1 and n_2. If t_1 acquires the lock of n_3, then t_2 will not be able to proceed the execution of its operator until the lock of n_3 is released. In these cases, Galois detects the conflict and aborts the execution of the operator at the active node n_2 by releasing its already acquired locks. Then, the active node n_2 is rescheduled to be processed later. Fig. 2.1 shows an example where the neighborhoods A and B share the node n, potentially leading to a thread conflict.

Intuitively, activities are processed atomically in an all-or-nothing approach in terms of the acquisition of the necessary locks. When the execution of the operator in a given active node is aborted due to conflicts, all computation performed at this point is lost. In this sense, it is desirable to design *cautious operators*, which first try to acquire all necessary locks in the neighborhood of the active node and only then perform the graph modifications. This way, after acquiring necessary locks, it is possible to ensure that the operator will be successfully executed without wasting time in the complex computation before all locks are available.

Although Galois offers a high level of abstraction for programmers, recent studies have compared Galois to a native thread implementation such as *pthreads* and have shown that the abstraction penalty is small. In (MOCTAR; BRISK, 2014), Moctar and Brisk proposed an efficient parallel FPGA router using the Galois system and they state to believe that the Galois model is the right solution for parallel CAD. This statement is based on the wide number of irregular graph-based algorithms used to solve problems in EDA. Compared to other graph analytics frameworks, we consider the Galois system the most promising approach according to the characteristics of the logic synthesis method we are concerned to parallelize.

2.3.2 Other Approaches

Among several other tools for parallel graph analytics, it is worth to mention *Pregel*, *Giraph* and *GraphX*. Pregel is a distributed system designed by Google in 2010 which introduced the concept of vertex programs in graph applications (MALEWICZ et al., 2010). Pregel is based on the bulk-synchronous parallel (BSP) model, which is based on supersteps of computation, communication and synchronization. In this model,

the program is expressed as a sequence of iterations where the vertices receive messages from previous iterations and send messages to the next iterations. At each iteration vertices can update their own values or the values of their outgoing edges. Giraph is also based on BSP model and designed on top of Apache Hadoop, which provides fault tolerant software infrastructure for distributed processing and managing of large sets of data (Apache Giraph,). GraphX, in turn, is a scalable framework which unifies data-parallel and graph-parallel computation in a distributed fashion. In this framework, the data can be viewed and manipulated by applying graph operations such as the vertex program proposed in Pregel or by applying relational algebra (XIN et al., 2013; Apache Spark GraphX,). The main limitation of these tools for the context of this work is related to poor operations to change the graph topology. In general, such operations include only reverse graph edges, subgraph extraction and other simple variations.

2.4 Summary

In this chapter, we introduced a background on parallel programming and presented an overview on a set of programming models. Overall, the most appropriate way to take advantage of the general programming models for exploiting parallelism in the irregular graph-based algorithms we are interested on is by applying graph partitioning to get data independence. For several graph-based problems the partition does not affect the final quality of results and it is an interesting strategy to enable parallelism. Since the CEC is a decision problem which can be decomposed into subproblems without penalties in the final solution, we adopted graph partitioning and Pthreads to unlock parallelism in this task. A detailed view of the CEC problem and the proposed solution are presented in Chapter 5.

Although OpenMP and Intel TBB provide friendly models for parallel programming, we consider that these models do not attend all the necessary demands to enable parallelism in the decision and optimization algorithms we are interested in this work. OpenMP is not so efficient for designing parallel irregular algorithms where data-independence and computing-independence cannot be easily discovered through static code analysis and loop decomposition. Intel TBB does not provide thread-safe data structures for graph representations in such a library. These models could be an alternative to implement the proposed parallel CEC engine based on graph partitioning. However, for this task, we consider that Pthreads provides better performance with a reasonable programming effort compared to OpenMP and Intel TBB.

The main concern in applying graph partitioning during logic synthesis based on AIGs is that many opportunities of logic optimization can be lost due to the boundaries among the partitions. Therefore, successive iterations of graph (re)partitioning are needed for enabling logic optimizations on the partition boundaries. Although graph partitioning is an alternative for coarse-grain parallelism, in this work, we are motivated to exploit a fine-grain parallelism which fits well with logic optimization based on AIG rewriting and technology mapping. Therefore, the Galois system was considered the most appropriated model to enable parallelism in these two optimization algorithms.

In the context of other graph analytics tools, although such tools have interesting infrastructures based on fault-tolerant distributed data and computing, the most part of them do not fit in our application demands. For instance, the great part of graph analytics tools do not provide the level of freedom for changing the graph topology as we need for

rewriting AIGs efficiently. The main reason is that usual graph analytics applications aims to analyze the data represented as a graph and perform simple read/write operations with limited graph modifications. On the one hand, current graph analytics tools are concerned to extract useful information from the data. On the other hand, AIG rewriting and functional technology mappers aim to change relative large subgraphs to optimize the DAG representation in some aspects. The Galois system supplies all the necessary demands to implement logic synthesis algorithms. Future improvements in the features for graph topology modifications can make tools such as Spark GraphX more suitable for applications like parallel logic synthesis and EDA in general.

3 MULTI-LEVEL LOGIC OPTIMIZATION

In this chapter, we revisit the *multi-level logic optimization* problem. The logic synthesis phase starts from a general representation, usually called logic network, which describes the logic behavior of a given circuit regardless technology and physical aspects. Logic networks are commonly represented as directed cyclic graphs (DAGs). Typically, multi-level logic optimizations aim to minimize the number of nodes (size) and levels (depth) in the given logic network. These cost functions are well-established since they are correlated to the circuit area and delay, respectively (BRAYTON; HACHTEL; SANGIOVANNI-VINCENTELLI, 1990; MICHELI, 1994). Multi-level logic optimization is an important and time-consuming task during logic synthesis because the underlying logic representation has direct impact on the QoR in the next of the design flow *i.e.*, the technology mapping (CHATTERJEE et al., 2006b; LIU; ZHANG, 2017) as well as placement and routing (WANG; CHANG; CHENG, 2009).

Local transformations, such as *logic rewriting*, play an import role in modern logic synthesis due to its interesting balance between runtime and quality-of-results (QoR). However, as we have discussed in the motivational examples in the introduction of this work, the time spent for multi-level logic optimization may represent several hours due to incremental passes of logic rewriting on large networks. In this sense, logic rewriting is the central optimization technique discussed in this chapter, where we are introducing a fine-grain parallel AIG rewriting. Experimental results show that the proposed approach is able to speed up the multi-level logic optimization of designs comprising millions of AIG nodes with similar QoR when compared to the reference serial method. Several other method based on the same rewriting technique can be accelerated by employing the principles we are introducing in this work.

The chapter starts proving a background on the necessary logic synthesis terms and concepts. In the sequence, we review well-established data structures and algorithms addressing multi-level logic optimization based on logic rewriting that gradually leveraged this field of research to its state-of-the-art. Finally, we introduce the parallel AIG rewriting which is the first contribution of this work.

3.1 Background

3.1.1 Combinational and Sequential Digital Circuits

A *combinational digital circuit* depends only on the current state of its input signal to determine the values of its outputs. A *sequential digital circuit* depends on the current and the previous input information to determine the values of its outputs. Usually, combinational circuits are the basis for arithmetic and control operations whereas sequential circuits are used for circuit synchronizations such as pipeline barriers and also for control based on finite-state machines (MICHELI, 1994). This work focuses on algorithms that are used to optimize the combinational part present in digital circuit, such as microprocessors or microcontrollers (DA ROSA JR et al., 2003). Besides that, this work is in the scope of digital synchronous circuits as opposed to asynchronous ones (MOREIRA et al., 2014). Optimizing combinational circuits can potentially reduce the area and power consumption (BUTZEN et al., 2010; WILTGEN et al., 2013) of the final circuit.

3.1.2 Boolean Function

A single-output and completely-specified Boolean function f is a mapping from a n-dimensional space B^n to a 1-dimensional space B, *i.e.* $f : B^n \rightarrow B$, where n is the number of input variables of $f(x_1, x_2, ..., x_n)$ and $B = \{0, 1\}$. The *support* of f refers to the subset of input variables that influence the function output value. A *literal* is an instance of a variable than can appears in the direct or complemented polarity, *e.g.* x_1 and $!x_1$, respectively. A *conjunction* of literals is a product, *e.g.* $x_1.!x_2.x_3$, also referred as *cube*. A *disjunction* of literals is a sum, *e.g.* $x_1 + !x_2 + x_3$.

3.1.3 AND-Inverter Graphs

There are several ways for representing Boolean functions such as truth tables, sum-of-products (SOP), product-of-sums (POS), factored forms, binary-decision diagrams (BDDs), AND-inverter graphs (AIGs), and others (BRYANT, 1986; MICHELI, 1994). For the sake of simplicity, we are giving focus to the AIG representation, since the algorithms proposed in this work are based on that.

An AIG is a directed acyclic graph containing four type of nodes: the constant, primary inputs (PI), primary outputs (PO), and 2-input AND (AND2) operators. Sequential elements such as latches and flip-flops can be viewed as special nodes or pseudo-PI/PO. A graph edge can present an optional attribute depending on whether the corresponding signal is complemented. The AIG is a *homogeneous* and *universal* circuit representation. It is *homogeneous* in the sense that all internal nodes represent the same operator (AND2). It is *universal* in the sense that any arbitrary Boolean function can be represented in the AIG format by applying the De Morgan's law. Fig. 3.1 illustrates an example of AIG where dashed edges represent inverters. A *majority-inverter graph* (MIG) is defined similarly. However, instead of AND2 gates, a MIG comprises three-input majority (MAJ3) gates (AMARÚ; GAILLARDON; MICHELI, 2014).

3.1.4 Fanin and Fanout

The set of nodes connected to the inputs of a given AIG node n is called the *fanins* of n. Analogously, the set of nodes connected to the outputs of n is called the *fanouts* of n. If there is a path from node n to n', then n is in the *transitive fanin (TFI)* of n' and n' is in the *transitive fanout (TFO)* of n. The *TFI cone* of a given node n includes n and all those nodes in the transitive fanin of n towards the primary inputs of the AIG. The *TFO cone* of n is analogous, including n and all those nodes in the transitive fanout of n towards the primary outputs of the AIG (MISHCHENKO; BRAYTON, 2006).

3.1.5 Maximum Fanout Free Cone

A *maximum fanout free cone* (MFFC) of a given AIG node n is a subset S of the predecessors of n such that every path from any node in S to a PO passes through n. In other words, the MFFC of a given node n contains all nodes that are, exclusively, used to define the logic function of n. For instance, the MFFC of node 15 in the AIG illustrated in Fig. 3.1(a) contains all the nodes inside the shaded region. It means that, if the node 15 is removed, all nodes in its MFFC can also be removed (MISHCHENKO; BRAYTON, 2006).

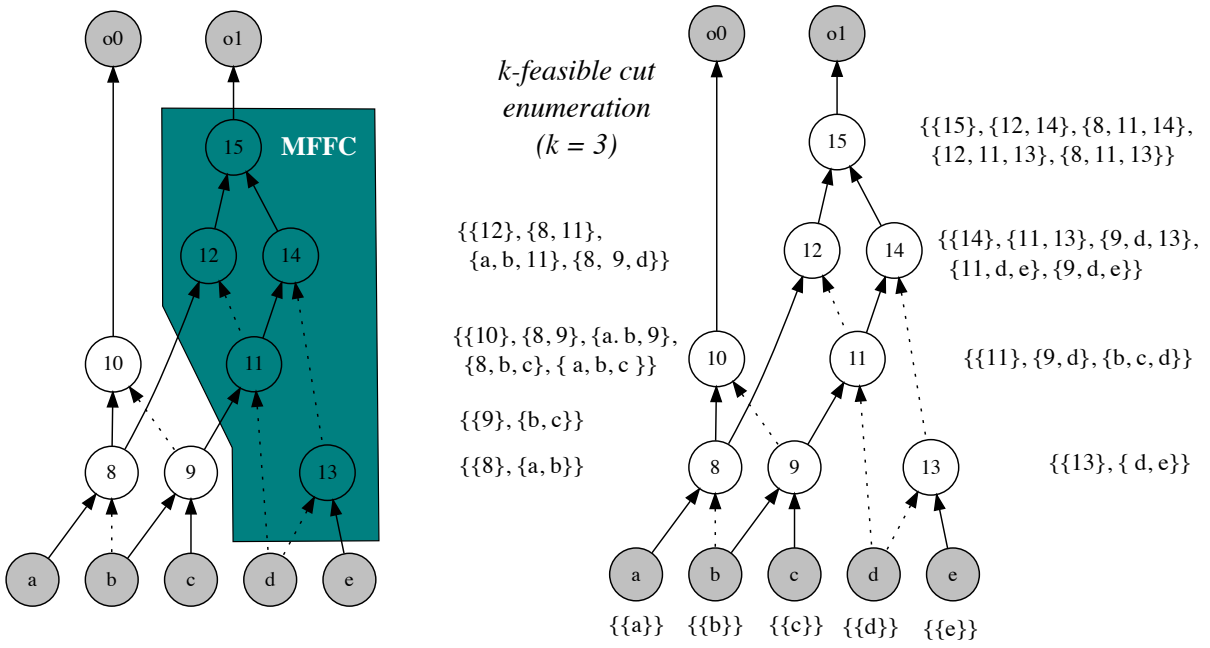


Figure 3.1: AIG example: (a) MFFC of node 15 and (b) k-cut enumeration with $k = 3$.

3.1.6 K-Feasible Cuts

A *cut* c rooted in a given AIG node n is a set of nodes, also called *leaves*, such that every path from the PIs to the root n contains at least one *leaf* $\in c$. A cut c_1 is dominated by another cut c_2 if $c_2 \subseteq c_1$, *i.e.*, c_1 is a redundant cut. A cut is *k-feasible* if it contains k nodes or less; such a cut is known as a *k-cut*. We review the standard k-cut enumeration defined in previous works (PAN; LIN, 1998; CONG; WU; DING, 1999; MISHCHENKO; CHATTERJEE; BRAYTON, 2007). Let A and B be two sets of cuts and the auxiliary operation $\diamond$ be defined as follows:

$$A \diamond B = \{u \cup v | u \in A, v \in B, |u \cup v| \leq k\} \tag{3.1}$$

The set of k-cuts of a given node n is defined recursively according to the following function $\Phi(n)$:

$$\Phi(n) = \begin{cases} \{\{n\}\} & : n \in PI \\ \{\{n\}\} \cup [\Phi(n_1) \diamond \Phi(n_2)] & : otherwise \end{cases} \tag{3.2}$$

When n is an AND2 node its two fanins are represented as n_1 and n_2. Conventionally, all k-cuts are computed by a single pass from the PIs to the POs of the AIG. The k-cuts of an AND2 node are computed by performing the Cartesian product (merging) between the

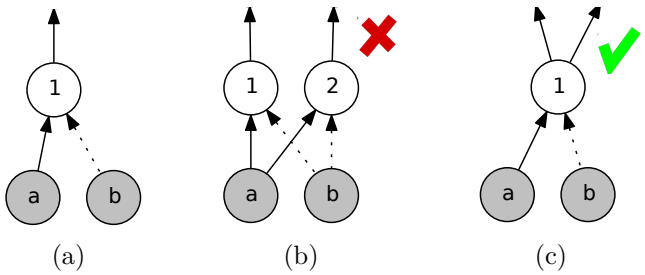
cut sets of its two fanin nodes. Moreover, each node has its *trivial cut*, which is defined by the node itself denoted as $\{n\}$ in the function Φ presented in Equation 2.2. Fig. 3.1(b) illustrates an example of the 3-input cut enumeration from the PIs to the POs.

A k-cut is associated to a local Boolean function, which is defined by the logic cone of the root node n and expressed in terms of the cut *leaves*. Boolean functions of cuts up to 16 inputs can be efficiently represented by truth tables, implemented using bit-strings, and manipulated using bitwise operations.

3.1.7 Structural Hashing

In the context of AIGs, *structural hashing* is a technique adopted to ensure that there is no AND2 gates with the same pair of fanins, even up to input permutation (MISHCHENKO; BRAYTON, 2006). Conventionally, the AIG manager stores each AND2 node in a global hash table by building a *key* in terms of its two fanin edges and nodes. Before creating a new node, the AIG manager performs a lookup in the hash table to check whether an equivalent node with the same fanin exists. If such a node exists, it is reused in a logic sharing way. Otherwise, a new node is created and inserted into the hash table that represents the structural hashing.

For instance, consider the AIG implementing an AND2 node ($a*!b$), as shown in Fig. 3.2(a). Suppose that, due to a given logic optimization in the AIG, a new AND2 node ($a*!b$) need to be created. Therefore, instead the AIG manager directly creates the new (redundant) node as shown in Fig. 3.2(b), the manager first lookup the hash table using a key generate as $key = hashFunction(a, !b)$. In this case, as such an equivalent node already exists in the AIG, then the node is shared by creating a new edge in node fanout, as illustrated in Fig. 3.2(c). An efficient structural hashing implementation is fundamental during AIG rewriting and other multi-level logic optimizations.



(a) (b) (c)

Figure 3.2: (a) AND2 node already in the AIG, (b) creation of an equivalent (redundant) AND2 node and (c) the logic sharing figured out by using structural hashing.

3.1.8 NPN Classification

A Boolean function f is NPN-equivalent to another f' if one of them can be obtained from the other by applying negations/permutations of its inputs/output. In the context of logic rewriting, NPN classification is commonly used to find out logically equivalent but structurally different implementations used for rewriting a cut in the logic network (MISHCHENKO; BRAYTON, 2006). Moreover, NPN classification is a conventional technique employed for Boolean matching (cell binding) in standard cell technology mapping (CHATTERJEE et al., 2006b; TOGNI et al., 2002).

3.2 Related Works on Logic Optimization

Since the early days of logic synthesis, different data structures have been adopted to represent logic networks and, consequently, several algorithms have been developed for optimizing and mapping such networks. Therefore, the representation and the manipulation of logic networks have evolved together in the last decades. For instance, Boolean expressions such as sum-of-products (SOP) were widely used for logic representation and optimization in the past. The nature of SOPs defines a two-level representation comprising a first level of AND operations and a second level of OR operations. Algorithms such as Quine (JQUINE, 1955), McCluskey (MCCLUSKEY, 1956) and ESPRESSO (BRAYTON P.C. MCGEER, 1993; SANGIOVANNI-VINCENTELLI, 1993) were proposed targeting two-level optimizations by minimizing the number of literals and cubes in a given SOP (RUDELL; SANGIOVANNI-VINCENTELLI, 1987). However, such logic representation and algorithms do not present scalability as the complexity of digital IC designs increases.

Directed-acyclic graphs (DAGs) were adopted as a multi-level representation of larger networks, where each graph node can represents an arbitrary Boolean function and graph edges define the relationship among such intermediary functions. For instance, in algebraic methods, each DAG node represents Boolean function in the SOP form and then algebraic optimizations are performed on the network by applying *elimination*, *decomposition*, *extraction* and *substitution* (BRAYTON; HACHTEL; SANGIOVANNI-VINCENTELLI, 1990). Other Boolean optimizations on DAGs are based on more complex and powerful techniques such as kernel-based Boolean division (MICHELI, 1994). Similar techniques applied for optimizing multi-level logic networks can be applied for transistor-level logic optimization as well (DA ROSA JR et al., 2007; POSSANI et al., 2016; POSSANI et al., 2017).

In the aforementioned DAG-based networks, the data structure is an heterogeneous representation in the sense that each node can represent a distinct Boolean function, making the graph manipulation complex. Therefore, homogeneous representation such as AIGs, were gradually becoming a standard structure for logic synthesis and verification, commonly used nowadays. Two motivations to adopt the AIG representation are related to its moderate space complexity in terms of memory footprint and its easiness to be manipulated by logic synthesis and verification algorithms.

In the last two decades, several AIG-based optimizations were introduced, presenting good design quality and scalability. Commonly techniques used in recent multi-level

logic optimization are *refactoring*, *balancing*, *rewriting*, proposed respectively in (BRAYTON; MCMULLEN, 1982), (CORTADELLA, 2003) and (MISHCHENKO; CHATTERJEE; BRAYTON, 2006). Several other approaches, such as Boolean decomposition, can be used to manipulate Boolean functions and to derive optimized AIGs (BERTACCO; DAMIANI, 1997; MARTINS; RIBAS; REIS, 2012). Even though Boolean function decomposition and factorization are general solutions, logic rewriting are often more suitable for multi-level optimization of large designs. Nowadays, AIG rewriting becomes a standard in both academic and industrial logic synthesis tools (AMARÚ et al., 2017).

Recently, majority-inverter graphs (MIGs) has becoming popular as an alternative data structure for multi-level logic optimization. A MIG is an homogeneous and universal representation analogous to an AIG. Amarú *et al.* proposed a set of axioms and native operations to efficiently manipulate and optimize MIGs (AMARÚ; GAILLARDON; MICHELI, 2014; AMARÚ; GAILLARDON; MICHELI, 2016). Moreover, several standard techniques applied to AIGs are being reformulated targeting to exploit interesting properties of MAJ operator for logic optimization in MIGs.

In the next subsection, we revisit the DAG-aware AIG rewriting proposed by Mishchenko *et al.* (MISHCHENKO; CHATTERJEE; BRAYTON, 2006) that has become a standard technique in modern logic synthesis. In the sequence, we show that such a rewriting technique has been used as reference for designing other recent AIG and MIG rewriting methods. We briefly highlight the main innovation that each work introduced.

3.2.1 DAG-Aware AIG Rewriting

In (MISHCHENKO; CHATTERJEE; BRAYTON, 2006), Mishchenko *et al.* proposed an AIG rewriting algorithm that extends the prior work (BJESSE; BORALV, 2004) and is implement in ABC command *rewrite* as follows.

A hash table of precomputed structures serves as a database to represent 4-input AIGs used for subgraph replacements. The set of all 4-input Boolean functions can be grouped in 222 NPN classes. The authors have empirically observed that many of these classes appear rarely in practical designs and therefore only about one hundred of the NPN classes are used in logic optimization. A hash table containing optimized AIG implementations for this useful subset is precomputed in advance and loaded into the rewriting manager. During the rewriting process, the hash table is used to retrieve subgraphs that are candidates to replace part of the original AIG.

The overall procedure for AIG rewriting works as follows. For each AIG node n, in the topological order, the algorithm computes *4-input cuts* and their respective truth tables for selecting parts of the graph to be rewritten. The Boolean functions of each 4-input cut is mapped into its NPN class and it is used to looking up a new cut implementation in the hash table of precomputed structures. The gain obtained by rewriting a given cut, rooted in the node n, is calculated in terms of the number of nodes that will be deleted and added into the AIG. Thus, the cut/implementation that leads to the best local improvement is greedily selected to replace the old structure of the cut. The number of deleted nodes is related to the MFFC of n. The number of added nodes is related to the amount of nodes already in the AIG that can be reused (shared) to express the new implementation of the cut. The structural hashing technique is usually applied for detecting such a logic sharing.

For instance, let us consider a didactic example of how this rewriting algorithm works. Given a possible AIG cut as shown in Fig. 3.3(a), the Boolean function related to this cut is computed and canonized using NPN classification, as illustrated in Fig. 3.3(b). Thus, the canonized function is used for looking up the hash table of precomputed subgraphs and retrieving different implementations for the function $f_i = a.b.c$, see Fig. 3.3(c). These three sugbraphs are tried and the subgraph that leads to the best node count reduction is selected to replace the original cut. In this example, the *subgraph 1* was selected, since the structural hashing enabled to reuse (share) other nodes already in the AIG to express the logic of the *subgraph 1*, as presented in Fig. 3.3(d).

In the last decade, this AIG rewriting method has been demonstrated an interesting balance between QoR and runtime. In general, the method does not require much hand-tuning to produce good results and it is orders of magnitude faster than traditional flow in MVSIS (MVSIS Group, UC Berkeley,) with comparable or better quality. Although the rewriting is local, it is faster than on-the-fly synthesis methods and can be applied many times to explore the solution space. In ABC scripts, this *rewrite* command is usually interleaved with two other techniques such as *refactoring* and *balancing*.

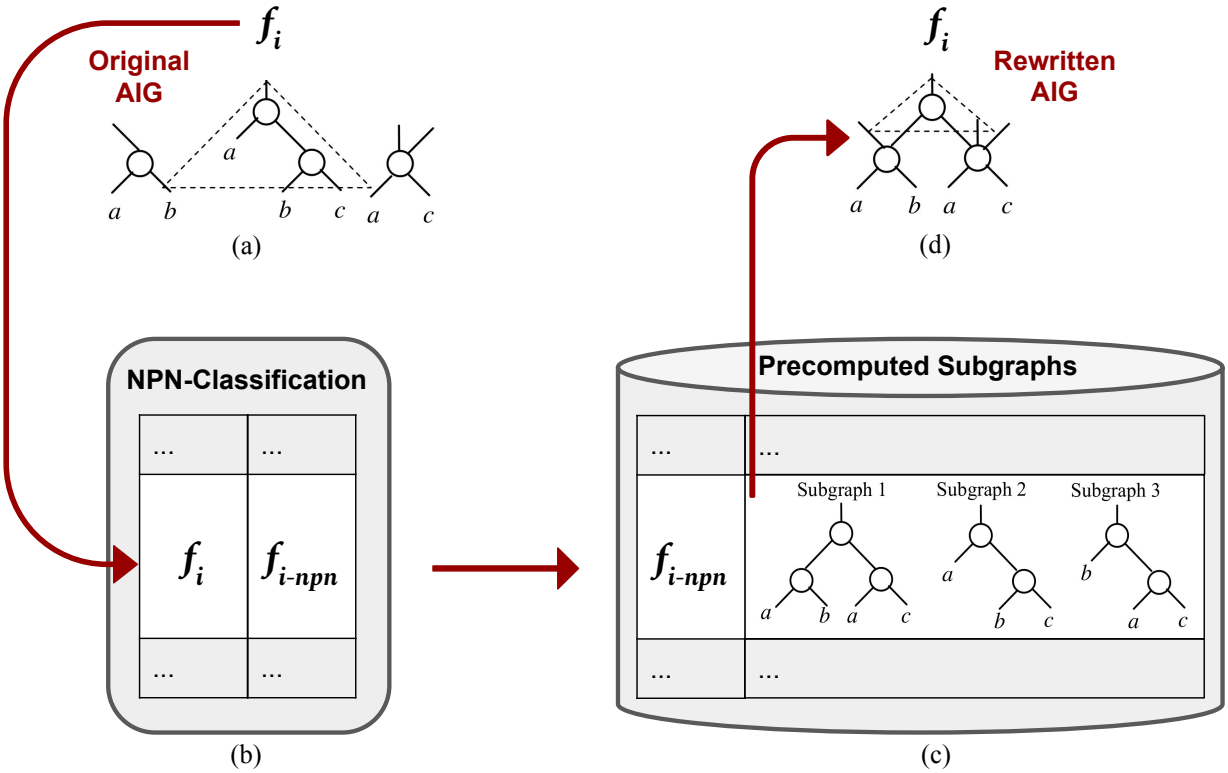


Figure 3.3: AIG rewriting example: (a) cut extraction; (b) cut function classification into NPN classes; (c) precomputed subgraph lookup; (d) cut replacement reducing one AIG node. Figure redesigned from (MISHCHENKO; CHATTERJEE; BRAYTON, 2006).

3.2.2 Other Recent Methods on Logic Rewriting

The local scope of 4-input cuts and the reduced set of 4-input Boolean functions, used in the previous method (MISHCHENKO; CHATTERJEE; BRAYTON, 2006), motivated Li *et al.* to extend this approach to 5-input cuts as presented in (LI; DUBROVA, 2011). The authors introduced a technique to build a library of precomputed subgraphs that can implement 1,185 NPN classes of 5-input Boolean functions. The 5-input cut enumeration and subgraph replacement is analogous to the previous work.

Another work presents an alternative to increase the size of k-cuts and explore larger databases with more complex Boolean functions, as introduced by Yang *et al.* in (YANG; WANG; MISHCHENKO, 2012). The paper presents an approach to extract several optimized subnetworks from designs already optimized with different techniques. Similarly to the previous methods, the extracted subneworks are stored and viewed as a library of structures to be used for logic rewriting.

Soeken *et al.*, in (SOEKEN et al., 2016), proposed an approach for MIG rewriting in three different graph traversals, top-down, bottom-up and based on fanout-free regions. Moreover, the authors proposed an exact MIG synthesis method, which was used to build

a hash table of precomputed structures to be used during rewriting.

Recently, a new method for *XOR Majority Graphs* (XMG) rewriting was proposed by Haaswijk *et al.* in (HAASWIJK et al., 2017). This approach uses a LUT-based technology mapper to mine useful Boolean functions to be considered during the rewriting process. Then, the exact MIG synthesis proposed in (SOEKEN et al., 2016) is used to compute MIG implementations for the collected functions. The MIG implementations are stored in a database and used for logic rewriting, similarly to the previous works.

The most recent approach, proposed by Amarú *et al.* in (AMARÚ et al., 2017), is based on DAG-aware AIG rewriting. However, instead of replacing k-cuts by improved AIG structures, the method replaces k-cuts by combinations of standard cells from a precomputed database. The work introduces the concept of *equioptimizable arrival times* that is used together with the cut Boolean function to retrieve the combination of cells that minimizes the delay at the cut output.

We argue that, by understand how to unlock the parallelism in the reference DAG-aware AIG rewriting method, we can exploit parallelism in those state-of-art logic synthesis methods based on that. In other words, it is possible to extend the parallel rewriting proposed in this work to incorporate the best characteristics of previous AIG- and MIG-based methods as well as investigate novel solutions in this direction. Moreover, parallel logic synthesis enables intensive and iterative optimizations while keeping a moderated computation time (ELBAYOUMI et al., 2014; LIU; ZHANG, 2017). Therefore, it reinforces our motivations on keeping rewriting methods running fast to optimize current and upcoming generations of large digital IC designs.

3.3 Proposed Fine-Grain Parallel AIG Rewriting

In this work we are introducing a set of principles to unlock fine-grain parallelism for rewriting multiple nodes of the graph at the same time, rather than to apply graph partitioning and rewriting each partition sequentially. The proposed approach is based on the AIG rewriting introduced in (MISHCHENKO; CHATTERJEE; BRAYTON, 2006). Therefore, our method employs 4-input cuts, NPN-equivalence, and the same set of pre-computed subgraphs used in the ABC command *rewrite*. All shared data structures were rethought in a novel proposal based on the *operator formulation* and the Galois system (PINGALI et al., 2011), in a multicore environment with shared memory.

Initially, a package for AIG representation and manipulation was implemented using the Galois graph representation *FirstGraph*, which has abstract locks for graph elements and supports changes in the graph topology. The top-level view of the parallel AIG rewriting is presented in Algorithm 1. This routine receives as input the AIG and other parameters such as the number of cuts stored per AIG node, the maximum number of precomputed subgraphs tried per cut and other optional flags.

The rewriting manager and all the other necessary modules are instantiated in line 3 of Algorithm 1. We follow the Galois formulation based on *active nodes* and *neighborhoods* to exploit parallelism. The initial set of active nodes is defined by all the primary inputs in the AIG, as shown in lines 6-7 of Algorithm 1. When the AIG comprises sequential elements, all latches can also start as active nodes, similarly to the primary inputs. This way, the threads are directly launched to rewrite all combinational logic clouds between barriers of sequential elements.

Active nodes are stored in a Galois thread-safe worklist *PerSocketChunkBag<500>*, maintaining distributed chunks of the worklist in the multi-core processor sockets (ISS Group, The University of Texas at Austin,). The items in this worklist are distributed along the machine cores to local worklists comprising chunks of 500 items each one. The chunk size of 500 items was determined based on some empirical experiments. In cases that threads need to push new items into the worklist, they can push it directly in its local worklist. When a local worklist becomes empty a new chunk of 500 items is picked up from the global worklist. In line 9 of Algorithm 1, the parallel *for_each* dynamically assigns active nodes to available threads as the computation proceeds. The activities are performed by executing the operator implemented in the rewrite manager.

Algorithm 3.1: Parallel AIG Rewriting

```
1 Function parallelAIGRewriting()
      Input  : AIG, nCuts, nStr, useZero, nThreads
      Output: a rewritten AIG or the original one
2     GaloisSetThreads( nThreads);
3     // Instantiate all necessary components
4     RewritingManager rwMan( AIG, nStr, useZero, cutMan(4, nCuts),
        npnMan(), strMan() );
5     // All primary inputs and latches are the initial active nodes
6     galois_for_each( AIG.PIs, rwMan ); // Parallel for
7     return AIG;
```

3.3.1 Rewriting Manager

Algorithm 2 presents an overview of the rewriting manager, which implements the operator executed at each active node. Essentially, the rewriting operator works just like the reference method in the sequential code. However, its main internal routines were rethought to support concurrent operations. Notice that the rewriting manager is constructed based on references to other shared-memory objects, *e.g.* AIG, $cutMan$, $npnMan$ and $strMan$. These auxiliary managers are discussed in the next subsection.

The operator receives as argument an *active node* and the *GaloisCtx*, which provides access to the Galois context such as worklists and customized memory allocators. In order to minimize the computation lost due to thread conflicts, all logical locks in the neighborhood of an active node must be acquired in advance. In the proposed approach, such a neighborhood is defined by a window containing the fanouts of the active node and the logic cones rooted into the active node and expressed in terms of its k-cut leaves. The operations performed in lines 4-7 of Algorithm 2 make the operator cautious. In case of conflict, the operator is aborted as soon as possible and does not lose any logic optimizations already done. This way, only the k-cuts of the active node are discarded (lost) because another thread can change the graph structure and make such cuts inconsistent.

In the sequence, all 4-input cuts are evaluated in order to figure out the best cut/structure to be used for rewriting. The number of precomputed structures tried per cut can be limited by using the $nStr$ parameter, shown in line 15 of Algorithm 2. When the operator reaches line 22, it means that all necessary locks were acquired and

the subgraph replacement can be committed safely. In other words, the operator cannot make the AIG inconsistent by aborting during the subgraph replacement.

Finally, the last task of the operator is to evaluate the fanouts of the active node and determine the ones that may become active. We adopted other two auxiliary labels, beyond the label *active*, to define the state of each AIG node. Therefore, a node can be unprocessed (*inactive*), under processing (*active*) or already processed (*done*). As the k-cut enumeration is based on the cuts of previous levels, an AND node can become active and pushed into the worklist only when its two fanin nodes were already processed (*done*). Such constraint is required to avoid inconsistencies in k-cuts, which may be introduced when the operator aborts its execution due to thread conflicts. The routine $pushFanoutNodes$ called in line 24 of Algorithm 2, is responsible for evaluating and pushing the fanout nodes that are ready to become active.

Since the hash table of precomputed structures does not change according to AIG being rewritten, the time complexity of the AIG rewriting method is bounded by the time complexity of enumerating all k-cuts. Considering an AIG with n nodes, in the worst case, the number of k-cuts is $O(n^k)$ (MISHCHENKO; CHATTERJEE; BRAYTON, 2007). Let C be the maximum number of k-cuts stored per AIG node, which can be defined by an user parameter. The time complexity for computing the k-cuts for a single node is $O(k.C^2)$ due to the Cartesian Product applied for merging two sets of k-cuts from the fanin nodes. However, as the rewriting method computes only 4-input cuts, *i.e.*, $k = 4$, each AIG node stores only few and small cuts. Since the rewriting visits each AIG node once, enumerating 4-feasible cuts, its overall time complexity is bounded by $O(n.k.C^2)$, which can be simplified to $O(n.4.C^2) = O(n.C^2)$. In practice, the AIG rewriting method has a runtime behavior almost linear in the number of AIG nodes.

The main challenges to enable a fine-grain parallel AIG rewriting are related to efficiently manage the access to shared-data structures without creating bottlenecks when threads are competing for a shared resource. Therefore, in the following subsections, we present how we designed necessary data structures to work efficiently in a multi-threaded environment. Figure 3.4 shows how the data structures are organized and how threads communicate.

Algorithm 3.2: Rewriting Manager Operator

```
1  RewritingManager begin
2      constructor( AIG, nStr, useZero, cutMan, npnMan, strMan);
3      Function operator( node, GaloisCtx)
4          C = cutMan.computeKCuts( node);
5          lockFanoutNodes( node);
6          for each 4-input cut c in C do
7              lockFaninCone( node, c);
8          bestS = null;
9          bestG = -1;
10         for each 4-input cut c in C do
11             nSaved = computeMFFC( node, c);
12             f = cutMan.getCutFunction( node, c);
13             f_npn = npnMan.getRepresentative( f);
14             S = strMan.lookupStructures( f_npn);
15             for each structure s in S up to nStr do
16                 nAdded = countAddedNodes( s, c);
17                 gain = nSaved − nAdded;
18                 if ( (gain > 0) || ( (gain == 0) && (useZero) ) ) then
19                     if ( (bestS == null) || (bestG < gain) ) then
20                         bestS = s;
21                         bestG = gain;
22         if (bestS ≠ null) then
23             updateAIG( node, bestS);
24         pushFanoutNodes( node, GaloisCtx);
```

3.3.2 Cut Manager

Cut Manager is responsible for providing all necessary routines and data structures for k-cut computation and storage. When it comes to high-performance computing, it is important to handle memory allocation efficiently. As the k-cut enumeration creates a large set of cuts, it is wise to use pre-allocated memory for cut storage. In this sense, we designed the cut manager so that each thread has its own memory pool. When a thread needs to compute k-cuts, a thread-local pointer is used to get access to its own memory pool, as shown in Fig. 3.4. This way, we unlock the parallelism by avoiding dependencies among threads and avoiding call to the operating system for memory allocation.

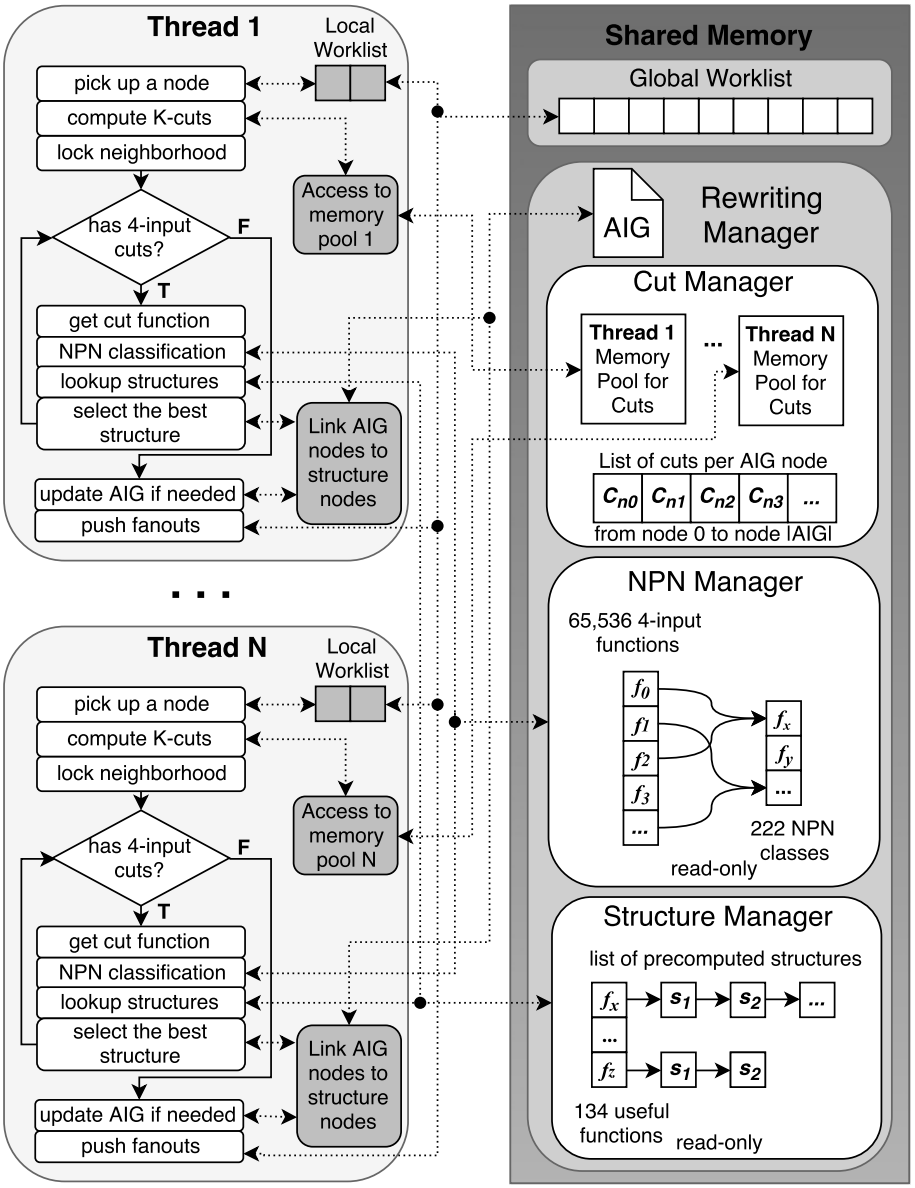


Figure 3.4: Relationship among rewriting operator, thread-local, and shared-memory data structures.

The initial size of each memory pool is defined according to the initial AIG size and each memory pool is resized as necessary. In several cases, k-cuts need to be recomputed for a second passing of rewriting, technology mapping or due to a thread conflict. An interesting feature is that our memory management allows a given thread to reuse the memory of old cuts without any logical locks, even when such cuts are allocated in the memory pool of another thread. When a thread is recomputing k-cuts at a node with an old set of cuts, the new cuts are stored by overwriting the memory region of the old ones. If extra memory is need, then such a thread uses its own memory pool to store the remaining cuts. It is worth to mention that, even though we are using 4-input cuts for rewriting, the proposed parallel k-cuts were designed to compute cuts for any k.

3.3.3 NPN Manager

NPN Manager provides a fast NPN classification tailored to work with 4-input Boolean functions. It provides NPN classification in constant time as all of the 65,536 4-input Boolean functions and their respective 222 classes are stored in a hash table, including the inputs/output phase assignments and permutations of each function. This manager is strongly based on the ABC code (Berkeley Logic Synthesis and Verification Group,). The only modification is that the hash table must be read-only, designed in such way that all threads can access the data without any kind of locks or synchronization.

3.3.4 Parallel-aware structural hashing

Parallel-aware structural hashing is a reformulation of the conventional AIG structural hashing, making it more suitable to work in a parallel environment. Conventionally, structural hashing is performed using a global hash table. Each AIG node is added to this table using its fanins as a hash key. This hash table is built when the AIG is parsed and used during the operations that change the AIG structure. For instance, when a synthesis method is about to add an AND node to the AIG, the hash table is used to ensure that there is no replicated AND nodes with exactly the same fanins. Therefore, if an equivalent node is found in the table, it can be reused (shared). Otherwise, a new AND is created and added to the table. However, notice that handling a global hash table in a multi-threaded environment may lead to a bottleneck.

Structural hashing plays an important role in AIG rewriting, allowing the identification and sharing of equivalent AND nodes during subgraphs replacement. In line 14 of Algorithm 2, the structural hashing *lookup* is intensively called as an internal routine to determine how many nodes can be reused if the current structure is used for rewriting the AIG. In this context, we propose to use a decentralized scheme of hash tables, which works efficiently for performing parallel *lookup*, *insertion* and *deletion* of AIG nodes. The proposed approach is based on the observation that structural hashing relies on:

Given a pair of nodes (n_1, n_2) and a pair of edge polarities (e_1, e_2), check if there exists a two-input AND node n_3, which is connected to the ***fanout*** *of n_1 and n_2 through the polarities e_1 and e_2, respectively.*

Therefore, this task can be solved by searching for node n_3 directly in the fanout of nodes n_1 and n_2, avoiding the use of a global hash table. However, a linear search in

the fanout of n_1 and n_2 may be a time consuming task due to high fanout nodes. In this sense, we are using a local hash table at each AIG node in order to lookup its fanout nodes as fast as in conventional structural hashing. Concerning the memory usage, we have adopted a rule to ensure that each two-input AND node is stored exactly once, only in the hash table of its fanin node with the *smallest identifier* (id). For instance, assume that node n_3 has fanins n_1 and n_2 and assume that $n_1.id < n_2.id$. In this case, n_3 is stored only in the hash table of node n_1. This rule saves memory without losing the property of conventional structural hashing.

In the context of the algorithm, it is only needed to perform a simple comparison of node IDs to access the right hash table and then to apply the conventional operations for node *lookup*, *insertion* and *deletion*. Moreover, if a given thread owns the locks for the pair of nodes n_1 and n_2, it means that such thread has exclusive access to the hash tables of these nodes, ensuring mutual exclusion. In other words, this approach fits well with the Galois strategy to handle logical locks.

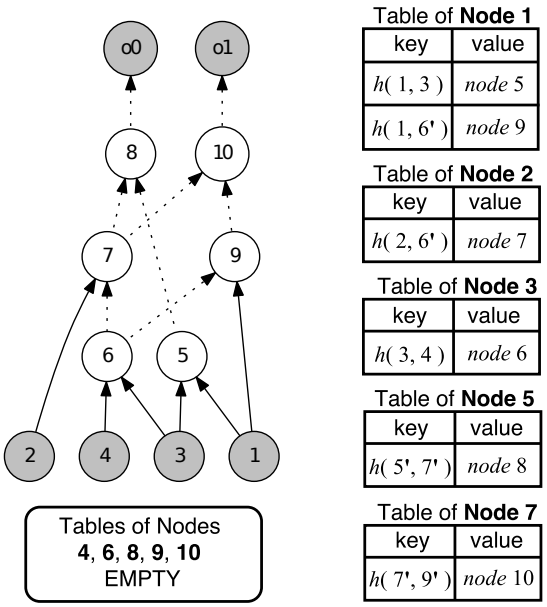


Figure 3.5: Proposed approach for decentralized structural hashing. Assuming h as a hash function applied on the pair of edges/nodes, where complemented edges are represented as apostrophes (') added to node ids.

Figure 3.5 illustrates an example of the decentralized approach for structural hashing. In this case, the structural hashing for the AIG presented in the left side of Fig. 3.5 leads to the set of non-empty hash tables of nodes 1, 2, 3, 5 and 7. In this case, nodes 4, 6, 8, 9 and 10 have empty tables because their respective fanouts are already registered in the hash table of some node with smaller identifier. Notice that, such AIG has

six AND nodes and the hash tables, altogether, also store only six nodes. This way, we efficiently manage memory usage, compared to the conventional structural hashing. The benefits of the proposed approach become more evident because logic sharing increases when rewriting is applied to large AIGs.

3.3.5 Structure Manager

Structure Manager stores a set of precomputed structures containing efficient implementations for a subset of 134 useful NPN classes selected out of all 222 NPN classes. In (MISHCHENKO; CHATTERJEE; BRAYTON, 2006), the authors defined as useful all 4-input Boolean functions appearing as functions of 4-input cuts selected during AIG rewriting on a set of benchmarks. We consider the same set of structures as used in the ABC command *rewrite* but change the method used to try a structure for a given cut.

Generally speaking, the precomputed structures act as templates to guide the construction of improved and logically equivalent arrangements of AIG nodes. In this sense, when a given structure is tried as a candidate for replacement, it is needed to keep a temporary one-to-one assignment between each node of the precomputed subgraph (template nodes) and its corresponding AIG node (real nodes). This one-to-one assignment is used to figure out how many AIG nodes can be reused or must to be created for implementing the new subgraph. The structural hashing technique presented before is used for detecting the equivalent nodes between the AIG and the precomputed subgraphs.

In the reference method, such an one-to-one assignment is done using pointers from the nodes of precomputed subgraphs to their corresponding nodes in the AIG, as shown in Fig. 3.6(a). In other words, when a precomputed subgraph is tried, it is necessary to write information in such a structure. Therefore, the main challenge in this step is that many threads can try to use the same set of precomputed subgraphs simultaneously, requiring an strategy to ensure mutual exclusion.

We solved this issue by making the table of precomputed subgraphs read-only and using a thread-local data structure for tracking the one-to-one assignment of nodes. The IDs of nodes from precomputed subgraphs are used to index a thread-local map, which contains pointers to temporary copies of the corresponding AIG nodes, as shown in Fig. 3.6(b). This approach enables parallelism in AIG rewriting, since many threads can use the same precomputed subgraphs simultaneously without using locks.

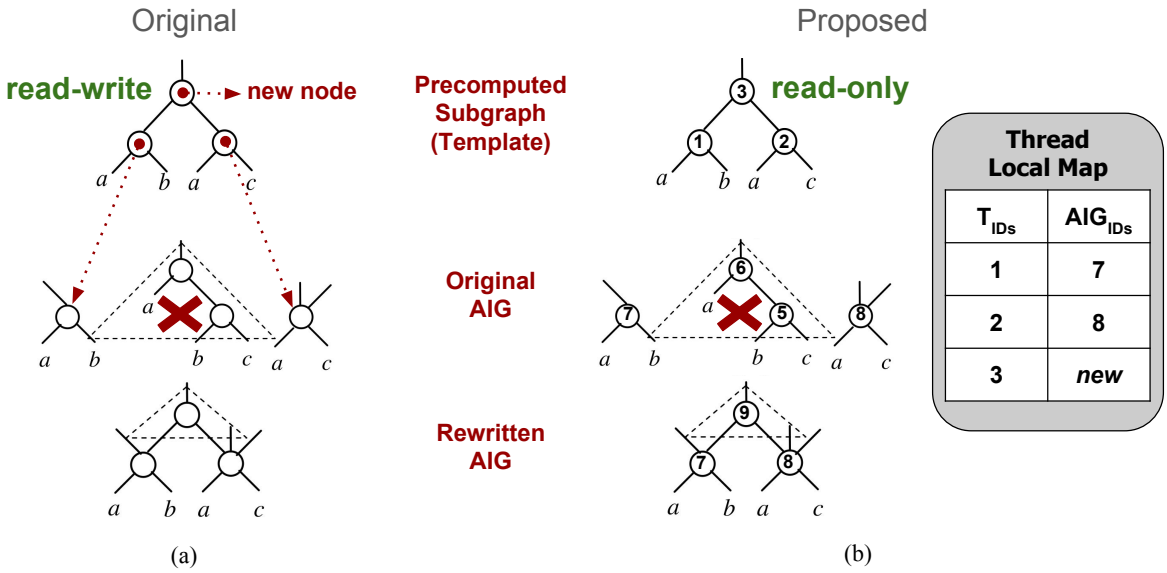


Figure 3.6: In (a), original tracking between AIG nodes and subgraph nodes and in (b) the proposed lock-free solution for tracking nodes. Figure redesigned from (MISHCHENKO; CHATTERJEE; BRAYTON, 2006).

3.3.6 Fine Tuning in Galois Graph Structure

Galois system provides a set of thread-safe graph representations which can be chosen according to the target application demands. However, all these graph structures are generic and can represent any information through user specified data stored into the graph vertices and edges. Moreover, the graphs can represent directed edges or not and are generic in terms of the number of incoming and outgoing edges connected at each vertex.

It is expected some performance degradation due to these general abstractions. In this sense, we have reviewed the code and executed performance profiling in order to improve the Galois graph structure used for representing AIGs. Since an AIG has a particular topology in which the most part of the graph nodes represent AND2 operators and edges represent a Boolean value (true or false), it is wise to perform a fine tuning in the graph implementation to get better performance. We performed a code refactoring in the Galois *First-Graph* class, which implements the graph structure supporting topology modifications. Our modifications improved the access to incoming and outgoing edges of each graph vertex and also enabled the use of efficient bit masks to represent Boolean values of graph edges.

3.4 Experimental Results

In the following experiments we compare the proposed parallel AIG rewriting to the rewriting methods implemented in the ABC commands *rewrite* and *drw*. The comparison is based on the method runtimes and QoR in terms of number of nodes and levels in the rewritten AIGs.

The parallel AIG rewriting was implemented in C++ 11 using Galois system (PINGALI et al., 2011; LENHARTH; NGUYEN; PINGALI, 2016). Both the proposed method and ABC were compiled using GNU g++ version 6.1.0 and executed in a 64-bit Linux distribution. The results were collected on a server with 128GB of shared RAM and 4 processors Intel®Xeon®CPU E7- 4860 at 2.27GHz, where each processor has 10 physical cores. We have ran the command &*cec* in ABC to check the correctness of our parallel rewriting on a large set of MCNC, ISCAS and EPFL benchmark circuits.

In all of the experiments, we ran ABC and the parallel rewriting five times for each design and for each number of threads in order to minimize the effects of external noise on the runtime. Therefore, the results under comparison were obtained by computing the average among five executions. The runtime of both ABC and the proposed parallel AIG rewriting were measured considering the exclusive time elapse of the rewriting methods, discarding the parser runtime. We have observed negligible variations in runtime among several executions of the methods. Since the Galois scheduling for handling thread conflicts is non-deterministic, we compute the average *size* and *depth* of the AIGs optimized by the parallel rewriting.

3.4.1 Benchmarks

The parallel rewriting is designed to target large AIGs. The largest designs available in public benchmarks suites are three AIGs with *more than ten million* (MtM) nodes from the EPFL benchmark suite (AMARÚ; GAILLARDON; MICHELI, 2015). The MtM node circuits comprise AIGs with sixteen, twenty and twenty three million nodes, representing random Boolean functions with complex implementation cost. These synthetic circuits contain structures that are quite different from those found in practical designs. Therefore, we considered the three MtM circuits and selected ten additional designs with

Table 3.1: The set of ten circuits obtained by applying 10x (8x) the ABC command *double* and the three MtM AIG nodes circuits from the EPFL benchmark suite.

Circuit	# PI	# PO	# AND2	# Levels
sin_10xd	24,576	25,600	5,545,984	225
arbiter_10xd	262,144	132,096	12,123,136	87
voter_10xd	1,025,024	1,024	14,088,192	70
square_10xd	65,536	131,072	18,927,616	250
sqrt_10xd	131,072	65,536	25,208,832	5,058
mult_10xd	131,072	131,072	27,711,488	274
log2_10xd	32,768	32,768	32,829,440	444
mem_10xd	1,232,896	1,260,544	47,960,064	114
hyp_8xd	65,536	32,768	54,869,760	24,801
div_10xd	131,072	131,072	58,620,928	4,372
sixteen	117	50	16,216,836	140
twenty	137	60	20,732,893	162
twentythree	153	68	23,339,737	176

more realist structures: seven largest AIGs among the arithmetic and three largest control circuits from EPFL benchmark (AMARÚ; GAILLARDON; MICHELI, 2015).

In order to derive larger AIGs from these ten arithmetic and random control circuits, we applied the ABC command *double* 10 times for each design. For the largest design "hyp", we applied the command *double* 8 times. This command doubles the size of a given AIG by creating a copy of the original design. The test cases generated using the *double* command are still synthetic but they are arguably more realistic than the MtM designs. In the real designs, synthesis tools are usually applied to a combinational logic cloud located between flip-flops in multiple design blocks. Since design blocks are often unrelated to each other, the resulting logic cloud may look somewhat similar to a set of copies of the same design used in the adopted benchmarks. Table 3.1 presents the circuits derived by applying the *double* command and the three MtM circuits from the EPFL benchmark suite.

3.4.2 Parallel k-Cut Computation Scalability

The cut enumeration is in the core of several novel logic synthesis algorithms. Therefore, we present an evaluation of the standalone parallel k-cut computation. In

Table 3.2: Runtimes, in seconds, of the k-cut methods under comparison.

	-K 4 -M unbounded		*-K 6 -M 20*	
Circuit	ABC *cut*	Parallel *40 threads*	ABC *cut*	Parallel *40 threads*
sin_10xd	24.59	1.50	41.11	1.38
arbiter_10xd	31.79	1.78	54.47	2.87
voter_10xd	50.54	2.03	88.64	3.92
square_10xd	81.40	3.94	143.13	5.88
sqrt_10xd	81.77	3.63	122.50	3.63
mult_10xd	124.84	5.00	207.94	5.79
log2_10xd	151.46	8.88	246.68	12.37
mem_10xd	126.63	7.39	220.29	8.98
hyp_8xd	228.32	10.20	516.25	21.04
div_10xd	193.06	7.94	301.31	13.85
sixteen	49.08	5.17	72.87	10.30
twenty	63.81	5.28	93.23	11.64
twentythree	71.60	5.09	105.51	7.18

order to perform a fair comparison, we compared the performance of the parallel k-cut enumeration with the command *cut* in ABC, using 4-input and 6-input cuts. The number of cuts per AIG node is limited to 20 for 6-input cuts and unbounded for 4-input cuts, *i.e.*, ABC commands: *"cut -K 6 -M 20 -a -t -x"* and *"cut -K 4 -a -t -x"*. The remaining flags are used to disable some extra computations in order to perform a fair comparison between the methods. It is worth to mention that the parallel k-cut enumeration produces exactly the same set of k-cuts delivered by the ABC command *cut*. Therefore, the parallel and the sequential methods are equivalent in terms of QoR.

The parallel k-cut computation is executed using a given number of threads, going from 1 to 40. Fig. 3.7 and Fig. 3.8 present the scalability of the 4-input and 6-input cut enumeration, respectively. The parallel k-cut enumeration is up to 25x and 36x faster than the sequential version when using $k = 4$ and $k = 6$, respectively. It can be observed that the parallel k-cut enumeration scales reasonably well, which demonstrates a trend that our parallel technology mapping will scale too. Table 3.2 presents the absolute runtimes, in seconds, of the ABC command *cut* and the proposed parallel k-cut enumeration running with 40 threads.

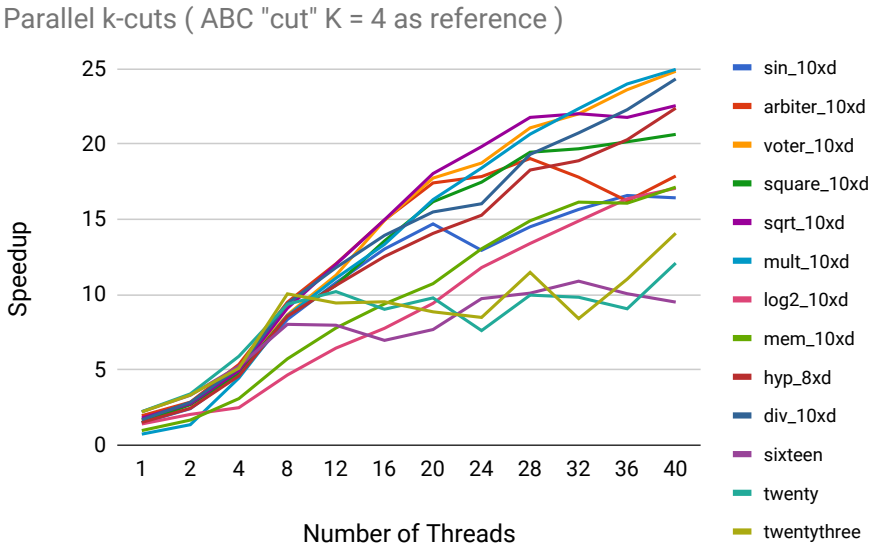


Figure 3.7: Speedups and scalability of parallel 4-input cut enumeration compared to the ABC command *cut -K 4*.

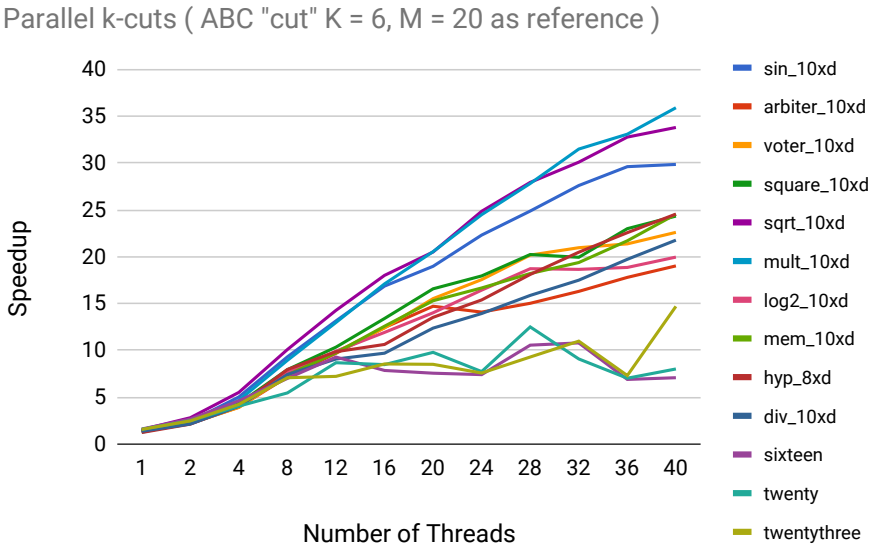


Figure 3.8: Speedups and scalability of parallel 6-input cut enumeration compared to the ABC command *cut -K 6*.

3.4.3 Parallel AIG Rewriting Scalability

In the following experiments, we compare the proposed parallel AIG rewriting to the *rewrite* and *drw* commands in ABC. In most of the practical applications, rewriting is applied to the same AIG several times. However, we are concerned to compare the parallel rewriting against the sequential one and measure its scalability. Therefore, a single AIG rewriting pass is used for each circuit and a given number of threads.

By default, the *rewrite* command does not limit the number of cuts per node and the number of precomputed structures tried per cut. Therefore, we have ran the parallel rewriting with these two parameters *unbounded*. Moreover, preserving logic level during rewriting was disabled using the switch *-l*, *i.e., rewrite -l*. To measure the parallel rewriting scalability as the number of threads increases, we ran the parallel method with 1 to 40 threads. Fig. 3.9 shows the parallel rewriting scalability. Overall, rewriting with one thread is approximately 2x faster than the *rewrite* command whereas rewriting with 40 threads is up to 36x faster. We have observed a poor scalability for the the three MtM designs due to their syntactical structures. The best speedup observed for the MtM circuits was 5.8x faster than the ABC command *rewrite -l*. In the next subsection,s we present a structural analysis of such benchmarks, justifying the poor scalability. Table 3.3 presents the runtimes for *rewrite -l* and for parallel rewriting with 40 threads *unbounded*.

The *drw* command is an improved version of the *rewrite* command. One of the main differences is the larger set of precomputed structures used during rewriting. Moreover, *drw* accepts some parameters to trade off QoR and runtime, making it possible to limit the number of cuts per node and the number of precomputed structures per cut. In this experiment, both methods under comparison were limited (*bounded*) to eight cuts per node and five precomputed structures per cut, which are the default values in *drw*. We have also ran both methods without preserving logic level and increasing the number of threads from 1 to 40. Fig. 3.10 presents the speedups of parallel rewriting *bounded* compared to the *drw* command.

Overall, the parallel rewriting *bounded* running with one thread is approximately 2.5x faster than the sequential command *drw* whereas rewriting with 40 threads is up to 50x faster. However, notice that the scaling curve for the "sqrt_10xd" circuit presents a particular behavior shown in Fig. 3.10. In this test case, the rewriting with one thread is approximately 6x faster than *drw*, reaching a speedup peak of 68x with 28 threads. As *drw* uses a more complete table of precomputed structures containing 222 Boolean

functions, it is expected to have variations in both runtime and QoR when comparing the parallel method to *drw*. The runtimes of *drw* command and parallel method with 40 threads *bounded* are presented in Table 3.3. The largest runtime of *drw* was observed when rewriting the "sqrt_10xd" circuit, meaning that *drw* performed many more attempts to replace subgraphs in the AIG. The parallel rewriting still presenting poor scalability for the MtM benchmarks, reaching an maximum speedup of 3.7x faster than the *drw* command.

Table 3.3: Runtimes, in seconds, of the rewriting methods under comparison.

Circuit	ABC *rewrite -l*	Parallel *unbounded* *40 threads*	ABC *drw*	Parallel *bounded* *40 threads*
sin_10xd	155.96	6.54	99.13	2.86
arbiter_10xd	227.73	8.17	177.90	6.89
voter_10xd	425.49	13.86	238.05	10.29
square_10xd	492.96	14.09	302.75	6.49
sqrt_10xd	784.64	32.82	1,368.16	22.18
mult_10xd	770.70	21.38	460.28	11.39
log2_10xd	1,077.12	34.37	602.86	17.31
mem_10xd	623.16	23.41	611.56	19.91
hyp_8xd	1,451.22	40.93	988.28	19.74
div_10xd	1,575.80	81.15	1,223.83	58.39
sixteen	819.97	225.01	362.91	138.06
twenty	1,155.85	288.41	476.80	177.54
twentythree	1,394.55	324.91	573.41	200.26

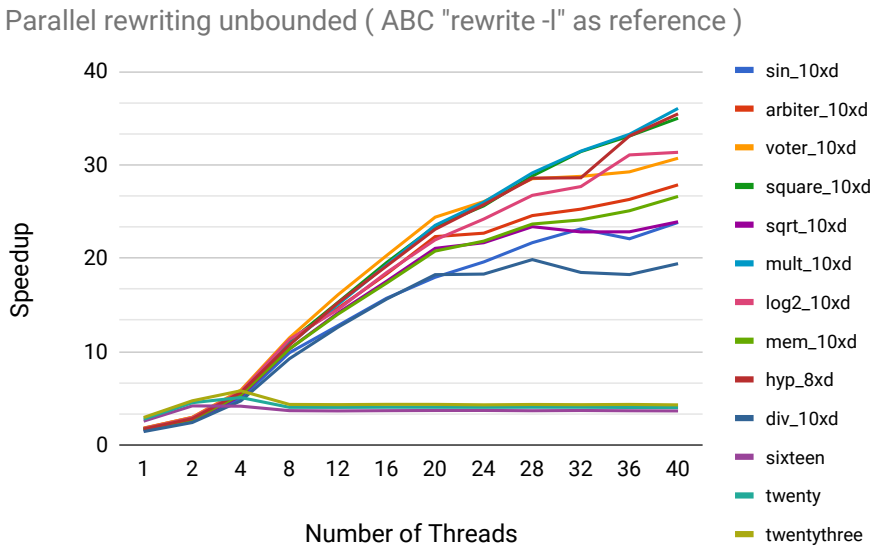


Figure 3.9: Speedups and scalability behavior of parallel rewriting *unbounded* compared to ABC command *rewrite -l*.

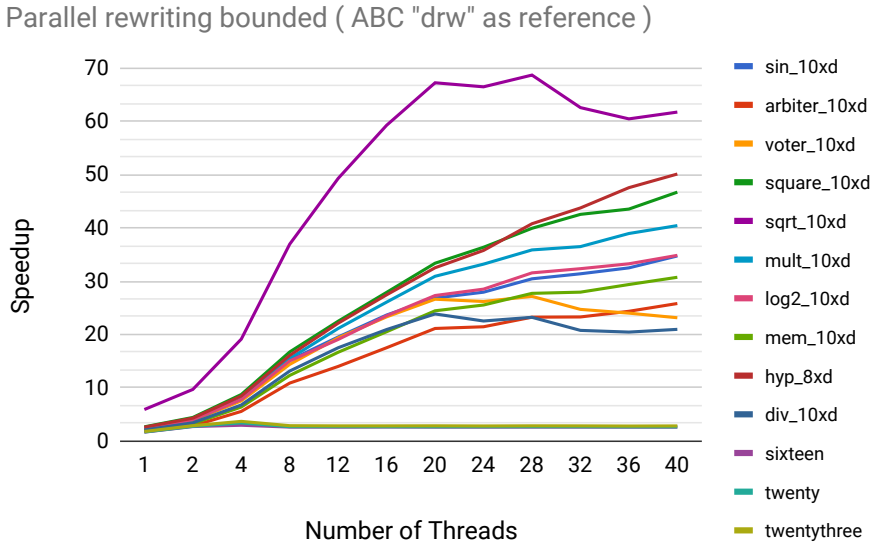


Figure 3.10: Speedups and scalability behavior of parallel rewriting *bounded* compared to ABC command *drw*.

3.4.4 Discussion About MtM Benchmark and Scalability

In order to get a better understanding about the weak scalability of the parallel rewriting when optimizing the MtM nodes AIGs from EPFL benchmark suite, we performed a profiling of such AIGs and observed a kind of *structural bottleneck* in all of them. For instance, the "sixteen" circuit has 117 primary inputs, 50 primary outputs and 16 million AND2 nodes. From these 16 million nodes, more than 7 million of nodes are in the first five levels of the graph, as show in Fig. 3.11. Moreover, primary inputs in these designs present a large fanout count. In the "sixteen" circuit, a single PI has 86,286 fanout edges and the number is even larger for other circuits as "twenty" and "twentythree". In other words, there are millions of nodes connected to only 117 PIs.

In this scenario, when threads are processing active nodes in the first five levels of the AIG, the transitive fanin cones defined by 4-input cuts have to be locked, potentially converging to the 117 PIs. Thus, the number of conflicts grows up due to many threads competing for node locks closer to each other. When considering MtM nodes designs, the parallel rewriting stops scaling closer to 4-6 threads due to the structural bottleneck in such circuits. The maximum speedups observed in such cases were 5.8x faster than the *rewrite -l* command and 3.7x faster than the *drw* command.

As discussed in the beginning of Section 3.3, the proposed parallel AIG rewriting uses a worklist to store active nodes in chunks of 500 nodes per processor socket, i.e., *PerSocketChunkBag<500>*. However, we have observed that, by increasing the chunk size from 500 to 5,000 it is possible to spread the threads and minimize conflicts when processing the MtM circuits. This way, we have enabled speedups up to 9x faster than the *rewrite -l* command and 6x faster than the *drw* command. It is worth to highlight that, even when optimizing these very large designs where it is hard to exploit parallelism, the proposed parallel rewriting has been able to deliver solutions almost one order of magnitude faster than the state-of-the-art method.

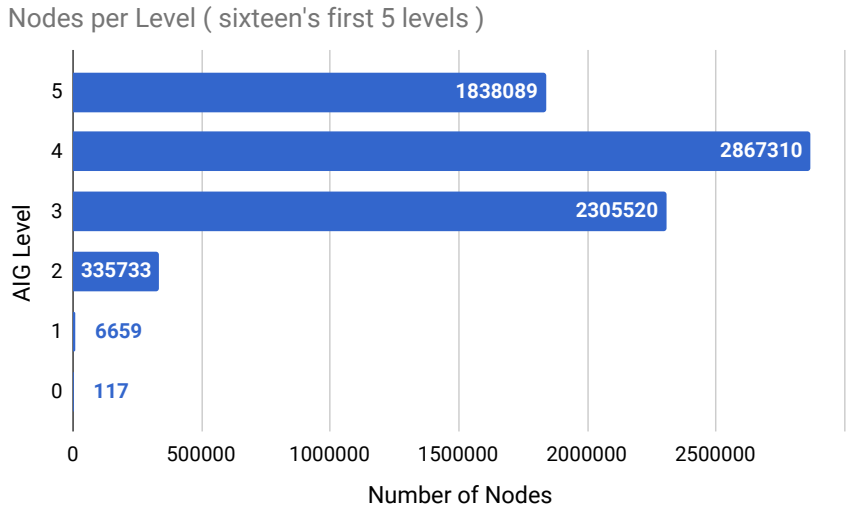


Figure 3.11: Structural bottleneck in the first five levels of the "sixteen" circuit from EPFL benchmark suite.

3.4.5 Parallel AIG Rewriting QoR

We also report the number of AIG nodes (*size*) and levels (*depth*) obtained at each execution, allowing the comparison of the parallel rewriting to *rewrite* and *drw* commands in terms of AIG *size* and *depth*. We consider the *size* and *depth* of the rewriting with 40 threads as the baseline because it produced the largest speedup. Table 3.4 shows that the parallel rewriting presents a small variation in the AIG *size* and *depth* when compared to command *rewrite* (reference method). As shown in Table 3.5, the *drw* command was able to reduce the AIG *size* and *depth* probably due to its more complete set of precomputed structures. It is expected that, using the same set of precomputed structures, the parallel rewriting and the *drw* command present the same quality in both the AIG *size* and *depth*.

Table 3.4: QoR ratio of rewriting with 40 threads *unbounded* over ABC command *rewrite*.

	ABC *rewrite -l*		Parallel *unbounded* *40 threads*	
Circuit	*size*	*depth*	*size*	*depth*
sin_10xd	5,465,088	223	1.00	1.00
arbiter_10xd	12,123,136	87	1.00	1.00
voter_10xd	11,553,792	63	0.99	1.06
square_10xd	18,803,712	250	1.00	1.00
sqrt_10xd	18,923,520	6,048	1.00	1.00
mult_10xd	27,551,744	274	1.00	1.00
log2_10xd	32,304,128	443	1.00	1.00
mem_10xd	47,885,312	114	1.00	1.00
hyp_8xd	54,854,144	24,801	1.00	1.00
div_10xd	42,138,624	4,406	1.00	1.00
sixteen	12,177,639	109	1.00	1.00
twenty	15,511,594	111	1.00	1.01
twentythree	17,373,362	129	1.00	1.00

Table 3.5: QoR ratio of rewriting with 40 threads *bounded* over ABC command *drw*.

	ABC *drw*		Parallel *bounded* *40 threads*	
Circuit	*size*	*depth*	*size*	*depth*
sin_10xd	5,313,053	223	1.04	1.01
arbiter_10xd	12,123,136	87	1.00	1.00
voter_10xd	10,640,603	68	1.12	1.07
square_10xd	18,184,830	250	1.04	1.00
sqrt_10xd	18,924,544	6,048	1.00	1.00
mult_10xd	25,355,760	273	1.08	1.00
log2_10xd	30,475,035	423	1.05	1.05
mem_10xd	47,401,984	115	1.01	0.99
hyp_8xd	54,566,159	24,801	1.01	1.00
div_10xd	42,186,609	4,406	1.00	1.00
sixteen	12,279,623	103	1.00	1.06
twenty	15,652,199	107	1.00	1.06
twentythree	17,568,563	111	1.00	1.16

Since the proposed parallel AIG rewriting is non-deterministic due to the Galois scheduling to deal with thread conflicts, we are presenting an analysis on QoR variation. We have performed previous experiments considering 13 different circuits, 12 different thread counts and 5 executions of the same circuit for each thread count. This way, we produced 780 samples of runtime, AIG *size* and *depth*, *i.e.*, 60 samples per design. For each circuit, we considered as the baseline the minimum *size* and *depth* observed among the 60 samples. Thus, we calculated the maximum percentage of variation from the minimum values.

Table 3.6 presents a comparison of QoR variation for many executions of parallel rewriting, demonstrating that the variations in AIG *size* have been negligible. Besides, variations in terms of AIG *depth* were zero for all the executions on top of the first eleven designs under optimization. Some small variations in the circuit *depth* were observed for the MtM designs. Overall, even when different thread counts are used, the proposed approach produces similar solutions in terms of AIG *size* and *depth*.

Table 3.6: QoR variation considering 780 executions of the parallel AIG rewriting *unbounded*.

	size		*depth*	
Circuit	*min*	*max var.*	*min*	*max var.*
sin_10xd	5,460,989	0.0001%	223	0.0%
arbiter_10xd	12,123,136	0.0000%	87	0.0%
voter_10xd	11,387,187	0.0124%	67	0.0%
square_10xd	18,775,062	0.0005%	250	0.0%
sqrt_10xd	18,923,520	0.0000%	6,048	0.0%
mult_10xd	27,520,256	0.0011%	274	0.0%
log2_10xd	32,302,022	0.0001%	443	0.0%
mem_10xd	47,884,313	0.0005%	114	0.0%
hyp_8xd	54,859,255	0.0000%	24,801	0.0%
div_10xd	42,154,768	0.0009%	4,407	0.0%
sixteen	12,189,609	0.0527%	109	0.0%
twenty	15,525,965	0.0403%	111	2.6%
twentythree	17,389,858	0.0418%	129	0.7%

3.5 Summary

In this chapter we introduced a set of principles to unlock the parallelism for AIG rewriting. We have rethought both algorithms and shared data structures to work in a massive parallel environment. Several techniques such as k-cut enumeration, structural hashing and AIG subgraph replacement were designed in a lock-free fashion to fit into the Galois programming model without inserting extra logical locks and overheads.

Experimental results have demonstrated the scalability of the proposed fine-grain parallel AIG rewriting to optimize designs comprising millions of nodes. Usually, AIG rewriting techniques are applied many times on top of the same design. Therefore, our fast and scalable rewriting has demonstrated potential for speeding up the multi-level optimization as well as improving its quality within a moderated runtime. In other words, one can use the saved time provided by our parallel solution to perform extra iterations of intensive optimizations.

It is worth to mention that, we can also exploit the hierarchy modularization of very large designs to combine our shared-memory approach with distributed computing. Therefore, with some additional work, the proposed approach can be deployed in a cloud computing platform to fully exploit the computing power of such massive parallel environments. This is an interesting features since EDA vendors are gradually migrating the tools to cloud computing services and opening new challenges on the research side.

4 LUT-BASED TECHNOLOGY MAPPING

In this chapter, we are revisiting the problem of technology mapping into k-input LUTs for FPGA-based digital designs. Nowadays, FPGAs are extensively used for hardware acceleration in data centers dealing with a wide range of applications such as artificial intelligence, data analytics and financial (PUTNAM et al., 2014). Therefore, there is an increasing demand for efficient design automation tools to improve QoR and time to market of FPGAs applications.

In the context of QoR, it is known that the initial topology of the logic network dictates and restricts the mapping possibilities, i.e. *structural bias* (CHATTERJEE et al., 2006b; MISHCHENKO; CHATTERJEE; BRAYTON, 2007). Moreover, a recent study demonstrated a *miscorrelation* between the cost metrics adopted during multi-level logic optimization and technology mapping (LIU; ZHANG, 2017). Therefore, advanced techniques for optimizing the logic network topology during technology mapping are applied to minimize the structural bias and miscorrelation effects. However, such advanced techniques have a direct impact in the synthesis runtime due to complex Boolean optimizations or iterative try and error *ad-hoc* optmizations.

This work brings two main contributions to deal with QoR and runtime issues during LUT-based technology mapping. Firstly, we propose a fine-grain parallel mapper based on the priority cuts techniques that scales for many threads. In the sequence, we introduce a set of principles for extending our fast parallel mapper to make tech-independent optimizations driven by tech-dependent costs. Our novel approach synergistically integrates logic rewriting and LUT-based mapping in a parallel environment.

The background section introduces some definitions for a better understating of this work, comprising a detailed view of the standard cost functions used in LUT-based technology mapping. In the sequence, we revisit the state-of-the art methods on LUT-based mapping. Finally, we introduce our parallel mappers followed by the section of experimental results.

4.1 Background

4.1.1 Structural and Functional Mappers

Technology mappers can be divided in structural mappers and functional mappers (MISHCHENKO; CHATTERJEE; BRAYTON, 2007). Structural mappers perform the mapping without changing the structure of the underlying logic network received as input, usually called *subject graph*. Functional mappers are allowed to change the subject graph structure targeting to improve the mapping quality. Typically, functional mappers trade off runtime for better QoR, since they apply Boolean decomposition and multi-level logic optimization during the mapping.

4.1.2 K-input LUT and K-feasible Cuts

A k-input look up table (LUT) is a programmable cell which is able to implement any k-variable Boolean function. LUTs are commonly used as the elementary devices to implement the combinational logic in Field-Programmable Gate Arrays (FPGAs).

In the context of LUT-based mapping, it is not efficient to apply pattern matching from the subject graph against to k-input LUTs because the number of possible patterns increases according to the number of existing k-variable Boolean functions, which is 2^{2^k}. In this sense, instead of using graph pattern matching, LUT-based mappers are based on k-feasible cut computation to decompose the given logic network into a set of k-bounded sugbraphs. Since the Boolean function associate to each k-feasible cut comprises at most k-input variables, such a function can be directly implement into a k-input LUT (PAN; LIN, 1998; CONG; WU; DING, 1999).

In this chapter, we assume that the reader is familiarized with the k-cut computation procedure since it is defined and employed in our rewriting method proposed in the Chapter 3. Therefore, for a detailed definition and review on k-cut computation, we refer the reader to the Background Subsection 3.1.6 in Chapter 3. In the sequence, we present the main criteria used for guiding the k-cut computation during depth- and are-oriented mapping.

4.1.3 Depth-Oriented Cost Function

In 1994, Cong and Ding demonstrated that depth-optimal technology mapping in k-LUTs can be solved in polynomial time on a DAG (CONG; DING, 1994a). Modern *depth-oriented* mapping enumerates and sorts k-cuts according to their potential for reducing logic *depth* (*delay*) of the LUT network (MISHCHENKO et al., 2007). Usually, it is considered that each k-cut can be implemented into a LUT which has a unit cost in the depth of the mapped network. Therefore, the *depth* cost of a given k-cut rooted in a node n and expressed in terms of its set of leaves $L = \{l_1, l_2, ..., l_m\}$, being k the LUT size and $m \leq k$, is the following:

$$depth(n) = \begin{cases} 0 & : n \in PI \\ 1 + max(depth(l_1), depth(l_2), ..., depth(l_m)) & : otherwise \end{cases} . \quad (4.1)$$

The *depth* cost given by Equation 4.1 is considered for sorting k-cuts of each AIG node during the cut enumeration performed in topological order (MISHCHENKO et al., 2007). For each AIG node n, the k-cut that leads to the smallest logic depth is marked as the best cut of n. The *arrival time* of node n is given by the logic *depth* of its best cut.

4.1.4 Area-Oriented Cost Functions

Area-oriented mapping performs the k-cut enumeration using heuristics for sorting the cuts according to their potential for reducing area of the final LUT network. Mishchenko *et al.* demonstrated that it is convenient to apply two different heuristics, *global view* and *local view* for area recovering after depth-oriented mapping (MISHCHENKO; CHATTERJEE; BRAYTON, 2007).

For a better understanding of the area cost functions, we first introduce the definition of *reference counters*. The reference counter of a given AIG node n represents how many times such node is referred as a leaf in the best cuts of other nodes. In other words, the reference counters determine how many times each node is used in the current mapping. If the reference counter of node n becomes zero, it means that the best cut of n is no longer used in the current mapping. For instance, considering the mapping illustrated in Fig. 4.1, the best cut of node 10 is referred two times because it appears as a leaf in the best cuts of node 11 and node 14. The other nodes used in the current mapping are

referred once, comprising reference counters equal to 1. The remaining nodes, internal to the cut cones (LUTs), have reference counters zero which are omitted for the sake of simplicity. Both the global view and the local view heuristics for area recovering are based on such reference counters.

The *global view* heuristic is based on the *area flow* (MANOHARARAJAH; BROWN; VRANESIC, 2006) or *effective area* (CONG; WU; DING, 1999) concept. Basically, the area flow aims to provide an estimated notion of the area of each k-cut considering the logic sharing in terms of its root node n and leaves $L = \{l_1, l_2, ..., l_m\}$, with $m \leq k$. Therefore, the area flow (AF) is defined as follows:

$$AF(n) = \begin{cases} 0 & : n \in PI \\ [Area(n) + \sum_{i=1}^{m} AF(l_i)]/ReferenceCounter(n) & : otherwise \end{cases}. \quad (4.2)$$

The $Area(n)$ is the number of LUTs needed to implement the best cut rooted in n and the $AF(l_i)$ is the area flow cost of the i_{th} leaf of such a best cut. The $ReferenceCounter(n)$ can be viewed as the number of fanout nodes of node n in the current mapped solution, which is used for dividing the area cost according to the node logic sharing. The standard approach is to apply area flow during the k-cut computation in the topological order.

The *local view* heuristic is based on the area added or deleted to the mapped network if a given node n is selected or discarded in the graph covering, also called *exact area* of a cut. The exact area of a given k-cut c rooted at node n is defined by summing up the number of LUTs needed to implement the MFFC of node n in terms of the cut c. A local depth-first search (DFS) starts at node n, decreases the reference counters of the leaves in the best cut of n and expands recursively through the leaves whose reference counters become zero (MISHCHENKO; CHATTERJEE; BRAYTON, 2007). The number of LUTs in the current mapping that are exclusively used by the node n are summed up to the exact local area of n in terms of cut c.

For example, when computing the exact area for the best cut of node 14 in Fig. 4.1, the DFS starts at node 14 and the reference counters of nodes 10 and 12 are decreased, becoming one and zero, respectively. Since the reference counter of the node 12 becomes zero, it means that the LUT represented by the node 12 is exclusively used by the best cut of the node 14. Therefore, this LUT is summed up to the exact area of the node 14, leading to a local area of two LUTs.

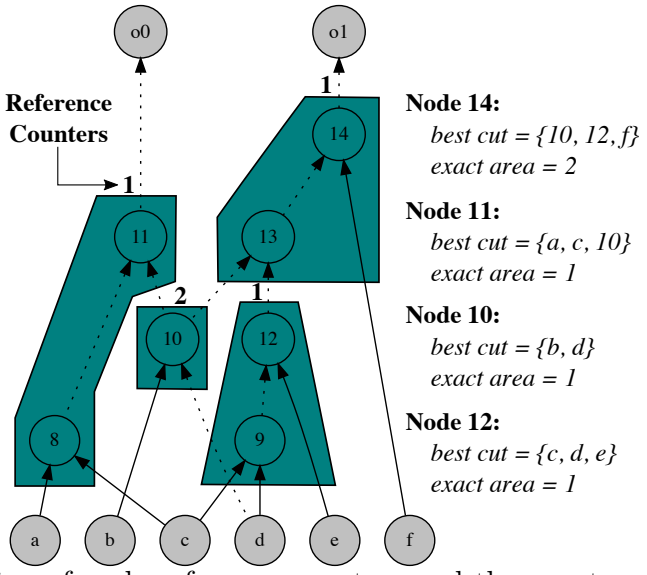


Figure 4.1: Illustration of node reference counters and the exact area computation when considering k-cuts with $k = 3$.

4.1.5 Computing Required Times

The concept of *required times* is a standard technique applied to enable area optimization without losing the depth-optimal mapping produced based on the Equation 4.1. To compute the required times, the primary output with the critical (worst) delay is detected. Such worst case delay is propagated in the reverse topological order to adjust the required time for each AIG node as presented in Algorithm 4.1 (MISHCHENKO et al., 2007). Therefore, during the area-oriented mapping, the best cut of a given node n is replaced by a new cut that minimizes area only if such a new cut respects the required time of node n.

4.1.6 Computing Graph Covering

The process of determining (selecting) a subset of the best cuts to compose the final solution is also known as the subject graph covering. The covering is computed in the reverse topological order and it can be done recursively or iteratively (MISHCHENKO et al., 2007). Algorithm 4.2 describes how the covering is computed starting from the circuit POs towards the circuit PIs. Each node n added into the covering corresponds to a LUT in the final network, where n represents the LUT output and the best cut best cut B_{cut_n} of n represents the LUT inputs.

Algorithm 4.1: Computing Required Times

```
1  Function computeRequiredTimes()
       Input  : AIG, CutMan
       Output: Updated required times
2      // find the global required times
3      T_max = findLatestPoArrivalTime( AIG);
4      // initialize the required times
5      for each AIG.node n do
6          setRequiredTime( n, ∞);
7      for each AIG.PO po do
8          setRequiredTime( po, T_max);
9      // propagate the required times
10     for each AIG.node n in reverse topological order do
11         T_req_new = getRequiredTime( n) – 1;
12         c = CutMan.getBestCut( n);
13         for ( each leaf l ∈ c ) do
14             T_req_old = getRequiredTime( l);
15             setRequiredTime( l, MIN( T_req_old, T_req_new));
```

Algorithm 4.2: Computing Graph Covering

```
1  Function computeCovering()
       Input  : AIG, CutMan
       Output: A set of nodes representing the Covering
2      S = AIG.POs; // Set of nodes to expand the covering
3      Covering = ∅; // Set of nodes in the covering
4      while ( S ≠ ∅ ) do
5          n = S.extract();
6          c = CutMan.getBestCut( n);
7          Covering.insert( n);
8          for ( each leaf l ∈ c ) do
9              if ( ( l ∉ Covering ) and ( l ∉ AIG.PIs ) ) then
10                 S = S ∪ l;
11     return Covering;
```

4.2 Related Works on LUT-Based Technology Mapping

The LUT mapping problem has been addressed in several previous works (CONG; DING, 1994a; PAN; LIN, 1998; CONG; DING, 1994b; CONG; WU; DING, 1999), which contributed to the state-of-the-art solutions employed in modern mappers. Since it is not practical to present a detailed view of all of them, we are focusing on the most recent techniques which are related to this work. In the next subsection, we pass through two important techniques from the literature, the *priority cuts* (MISHCHENKO et al., 2007) and the *lossless synthesis* (MISHCHENKO; CHATTERJEE; BRAYTON, 2007). We are giving attention to these techniques because they are the basis to other recent LUT-based mappers as well as to the parallel mappers proposed in this thesis.

4.2.1 Mapping with Priority Cuts and Choices

The concept of *priority cuts* was introduced in (MISHCHENKO et al., 2007). A structural mapper based on priority cuts stores a list S comprising a subset with up to C cuts and elects the best cut for each AIG node. The exact logic depth and heuristics for area minimization are commonly used for sorting and updating the list of cuts S, iteratively. Typically, only 8-10 cuts are stored per AIG node, contributing for reducing runtime and memory during technology mapping with competitive QoR when compared to exhaustive cut enumeration. For instance, the default configuration of the mapper implemented in ABC command $\&if$ works by computing priority cuts according to the following steps:

- Depth-optimum mapping with cut edges as tie-break;
- Depth-optimum mapping with area flow as tie-break;
- Area recovering based on area flow (2x);
- Area recovering based on exact area (2x);
- Graph covering for generating the final LUT network.

The list of cuts S is updated at each mapping pass, according to the current cost function. The first mapping pass aims to compute a depth-optimum solution according to the Eq. 4.1, which is related to the LUT network timing (delay). In the next, a second depth-oriented mapping pass is performed considering the area flow as tie-brake to sort cuts. The subsequent passes aim to optimized (recovery) area without losing the depth-

optimal solution. At each pass, in topological order, the mapper updates the best cut for each AIG node according to the current cost function, *i.e., area flow* or *exact area*, if the best cut does not increases the logic depth. The final LUT network is obtained in the reverse topological order by applying the standard technique described in the Algorithm 4.2.

The concept of *lossless synthesis* was introduced in (CHATTERJEE et al., 2006a) and in (MISHCHENKO; CHATTERJEE; BRAYTON, 2007) to reduce the structural bias during the technology mapping. This approach is based on the concept of choice nodes introduced in (LEHMAN et al., 1997). The lossless synthesis aims to combine many different logic networks produced during the multi-level logic optimization into a single network, since a given intermediary network can enable better mapping quality. The lossless synthesis can be divided into two main steps:

- Creating choices;
- Mapping with choices.

The process for *creating choices* relies on detecting functionally equivalent subgraph in a set of networks by using equivalence checking techniques and combining them into a single network with choices. The set of nodes proved to be equivalent are collected in equivalent classes comprising *choice nodes*. Let N be an equivalence class of choice nodes, and let n_r be the representative node of this class. The representative node $n_r \in N$ is an AIG node comprising a linked list pointing to the other nodes (subgraphs) belonging to N. These subgraphs are structurally different but functionally equivalent to n_r. In a network with choices, only the representative nodes of each class have fanouts. For example, consider the Fig. 4.2, where the *Network 1* and the *Network 2* were combined into a network with choices represented by the nodes x_1 and x_2. In this example, the node x_1 is the representative of the class comprising a pointer to the next choice of the list x_2.

The process for *mapping with choices* requires modifications only in the k-cut computation to consider the cuts related to the choice subgraphs during the mapping. The k-cut computation is still the same for all the AIG nodes, except for those representative nodes of each equivalence class. For each choice node $n \in N$, compute the k-cut set $\Phi(n)$ of n and then merge all the cut sets into a single cut set of the class $\Phi(N)$, as follows:

$$\Phi(N) = \bigcup_{n \in N} \Phi(n) \tag{4.3}$$

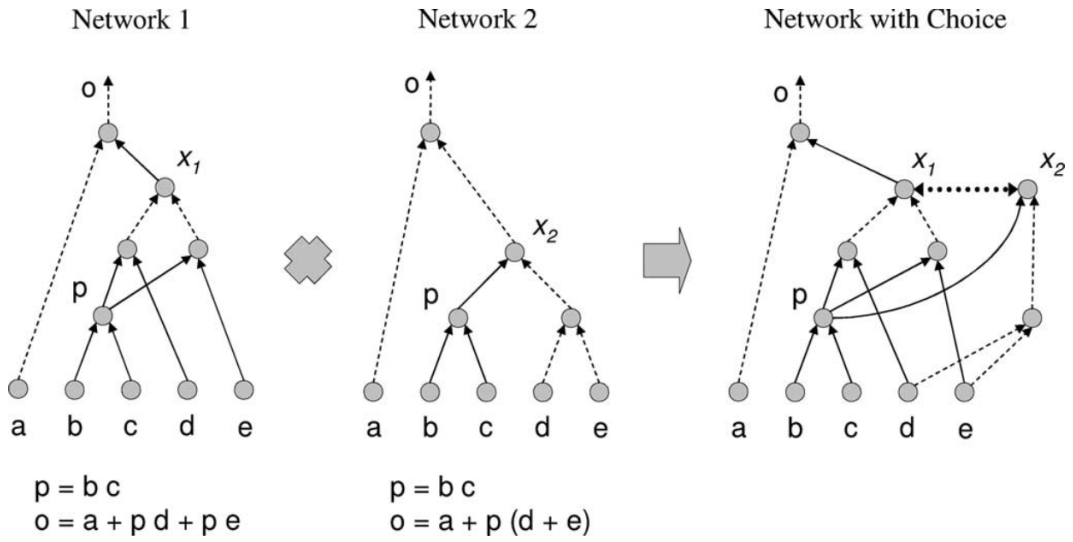


Figure 4.2: Creating a network with choices. Source: (CHATTERJEE et al., 2006a; MISHCHENKO; CHATTERJEE; BRAYTON, 2007).

Therefore, the resulting cut set $\Phi(N)$ is used to define the cut set of the representative node $\Phi(n_r)$, *i.e.,* $\Phi(n_r) = \Phi(N)$. This way, when evaluating the node n_r, the mapper can consider all available choices for implementing such a node. Moreover, all the cuts derived from choices are considered when computing the k-cuts of the nodes connected in the fanout of n_r.

Mapping with choices and priority cuts can be easily integrated to work in a cooperative way for improving the mapping quality. The main limitation of this approach is the high runtime during the k-cut computation when the AIG has many choices. These techniques are employed in ABC tool as the *&dch* command for creating choices and the $\&if$ command for LUT-based mapping. The default configuration of the $\&if$ command does not use choices, it needs to be explicitly enabled by running firstly the *&dch* command. It is worth mentioning that these mapping techniques are flexible and can reformulated to work with emerging technologies as proposed in (NEUTZLING et al., 2013; NEUTZLING et al., 2015).

4.2.2 Other Recent Approaches on LUT-Based Mapping

When considering parallel approaches for LUT-based technology mapping, a parallel LUT-based mapper was proposed by Kenning and Ravishankar, using the Intel Thread Build Block (TBB) (KENNINGS; RAVISHANKAR, 2011). In this work, the authors exploit the data parallelism existing during k-cut enumeration. Since k-cuts are gradually

computed in topological order throughout the AIG levels, the cuts for all the nodes at the same level can be computed in parallel. Although the authors present a parallel mapper which is up to 3.2x faster than their reference method, such an approach does not support logic optimization by changing the subject graph during the mapping. Parallel operations for improving the mapping quality require graph partitioning or a more sophisticated parallelization strategy.

Recently, in (LIU; ZHANG, 2017), Liu and Zhang proposed a parallel and iterative method for performing multi-level logic optimization together with LUT-based technology mapping. In order to enable parallelism, the initial logic network is partitioned, where sub-networks are distributed and synthesized in different machines. The method applies logic transformations, *e.g.* *rewriting*, *balancing* and *refactoring*, as stochastic moves to change the sub-networks during technology mapping. However, the graph partitioning tends to introduce negative effects during the logic optimization process, since the logic represented in the partition boundaries cannot be optimized. Therefore, in this method, the partitions are recomputed at each iteration in order to mitigate such a negative effect. This approach leads to improvements due to the ability to escape from the local minima and minimize structural bias effects. Although the method has presented interesting QoR, its runtime is high to synthesize large designs due to several iterations of partitioning and trial and error optimization in an *ad hoc* process.

Remapping techniques are commonly applied for improving mapped networks (MACHADO et al., 2012). In (SCHMITT; MISHCHENKO; BRAYTON, 2018), the authors proposed a structural SAT-based area recovery technique for improving LUT mapping. Given a mapped network, the method iterates over LUTs to incrementally compute and remap windows, which are relatively small subcircuits extracted from the mapped network. Each window is remapped by computing k-cuts and formulating the covering (cut selection) problem as a SAT instance. Thus, incremental SAT solving is used for determining whether there exist a covering which improves the mapping in terms of LUT count or logic depth. If such better covering exists, then the best cuts of the nodes inside the window are updated and the algorithm repeats this process iteratively. This method has demonstrated potential as a post-processing for refining the solution delivered by a conventional technology mapper, such as those based on priority cuts. However, since this method is structural, it is limited to the initial mapping network topology. Moreover, the number of LUTs considered in each window must to be carefully controlled due to expensive runtime of SAT solving. Therefore, there is room for improvements in both

QoR and runtime when considering this remapping approach.

In (MACHADO; CORTADELLA, 2018), the authors introduced a Boolean function decomposition based on support reduction and employ it in a functional mapper trying to minimize the structural bias effect. The proposed decomposition is considered during a recursive remapping method that processes the logic cone of each PO independently by applying collapsing, decomposition and remapping. The partial solutions obtained independently for each PO are combined back into a single network by using structural hashing. Finally, a similar remapping process is applied by extracting and remapping shared sub-networks. This approach has presented promising results since it is able to create an initial subject graph for LUT-based mapping with better structural characteristic that minimizes the structural bias effect. However, the main limitation of this method is the runtime and memory costs related to the network collapsing, which is unfeasible for complex designs. Therefore, there is much room for integrating synthesis and mapping techniques based on more complex Boolean optimizations like this as well as based on lightweight optimizations such as logic rewriting approaches.

4.3 Proposed Parallel Technology Mapping

In this section, we propose a novel solution which enables fine-grain parallelism for both structural and functional technology mapping. We start designing a priority cut manager for parallel cut computation, which is in the core of both the structural and the functional technology mappers. Thus, we propose a parallel structural mapping flow which employs only the priority cut technique. In the sequence, we introduce an alternative to extend our mapper to a parallel functional mapper by synergistically integrating AIG rewriting, priority cuts and choice nodes in a parallel environment. The proposed mapping approaches are based on the Galois system and the concept of active nodes. The mappers rely on the same AIG package implemented using the Galois *FirtGraph* that we developed for the AIG rewriting proposed in the Chapter 3.

4.3.1 Parallel-Aware Priority Cut Manager

The priority cut manager was designed to efficiently deal with memory allocation and to consider the depth- and area-oriented cost functions while computing k-cuts in parallel. Algorithm 4.3 presents the priority cut manager and its *operator*. Each thread executes the operator on top of a given active node and its neighborhood. In this case, neighbors of an active node are only its immediate fanin and fanout nodes. Therefore, the first task of the operator is to acquire the locks of neighbors as shown in lines 4-5 of Algorithm 4.3. The nested loops of lines 7-8 perform the combination (Cartesian product) between the cut sets from the two fanin nodes.

In the core of the algorithm, auxiliary routines merge pairs of cuts to produce new ones and to compute the depth/area costs associated to each k-cut. Each non-redundant cut that does not exceed k is inserted and sorted in the cut set S. The cut set S stores only the C best cuts sorted according to the current mapping goal. The first cut of the sorted set S is the best cut of the node. Since many cuts are created and discarded during the enumeration, each thread has its own memory pool with preallocated memory for cuts storage, similarly to the cut manager designed for the parallel AIG rewriting proposed in Chapter 3. Auxiliary functions help in this memory management for cut storage.

Since our parallel algorithms target to synthesize design with millions of AIG nodes, it is important to keep a moderate memory usage. Therefore, when all the fanouts of a given node n were already processed, the memory corresponding to the k-cuts of n are

returned back to the cut pool. The function *deleteFaninCuts* in line 26 of Algorithm 4.3 evaluates fanin nodes and returns their memory back to the pool when such cuts are no longer needed. The last task of the operator is to schedule the fanout nodes to be processed. A fanout node becomes active and is pushed into the worklist only if its two fanin nodes were already processed. The function *pushFanoutNodes* evaluates the fanouts of the active node and schedules those nodes that become active.

Parallel-Aware Area Heuristics: k-cut computation presents intrinsic data independence, since the cut set of each AIG node is produced by reading cuts from its fanin nodes and combining them. However, data dependencies are introduced when computing the exact local area of k-cuts and updating node reference counter. As presented in Subsection 4.1.4, the exact area heuristic is based on a local DFS to determine how many LUTs will be added to the network if a given cut c is selected as the best one. Such a DFS is guided by reading and writing into the reference counters of visited AIG nodes.

For instance, consider that multiple threads are performing such a DFS on overlapping subgraphs simultaneously. In this scenario, threads will compete for logical locks of AIG nodes, leading to potential conflicts. Moreover, it can lead to inconsistencies if a thread changes the reference counter of a given node and aborts its execution due to a conflict. This issue was also reported in a related work on parallel LUT-based mapping (KENNINGS; RAVISHANKAR, 2011).

We are proposing the following solution to deal with the data dependence when manipulating reference counters of AIG nodes:

- Making reference counters read-only during each mapping pass.
- Employing thread-local maps for tracking and changing reference counters during the DFS performed to compute the exact local area of a k-cut.
- Resetting and updating the reference counters based on the graph covering that express the current mapping, before to start the next mapping pass.

We have empirically observed that to make the nodes reference counters read-only during each mapping pass does not have a significant impact in the mapping quality. Moreover, it is also observed that it is convenient to compute the current mapping covering and to update the reference counters based on those values that express the current covering. In the next subsection, we present a detailed view on how to update the reference counters.

Algorithm 4.3: Parallel-Aware Priority Cut Manager Operator

```
1  PriorityCutManager begin
2      constructor( AIG, k, C);
3      Function operator( node, GaloisCtx)
4          lockFanoutNodes( node);
5          lockFaninNodes( node);
6          S = ∅; // S is the sorted list of cuts
7          for each cut c1 in node.fanin1 do
8              for each cut c2 in node.fanin2 do
9                  newCut = getMemoryFromPool();
10                 mergeCuts( newCut, c1, c2);
11                 if ( |newCut| > k ) then
12                     returnMemoryToPool( newCut);
13                     continue;
14                 if ( cutFilter( newCut) ) then
15                     returnMemoryToPool( newCut);
16                     continue;
17                 mappingGoal = getMappingGoal();
18                 computeCutCost( newCut, mappingGoal);
19                 sortedInsertion ( S, newCut );
20                 if ( |S| > C ) then
21                     lastCut = pruneLastCut( S);
22                     returnMemoryToPool( lastCut);
23         C_best = getFirstCut( S);
24         if ( C_best.delay <= node.reqTime ) then
25             setBestCut( C_best);
26         deleteFaninCuts( node);
27         pushFanoutNodes( node, GaloisCtx);
```

The proposed parallel cut manager supports the computation of k-cuts in AIGs with choices. Therefore, both the proposed parallel structural and functional mappers can be benefited from choices during technology mapping. The support to choices is an interesting feature of the proposed approach, since the runtime of k-cut computation tends to increase significantly when many choices are available in the AIG (MISHCHENKO; CHATTERJEE; BRAYTON, 2007). Therefore, our parallel methods can provide an moderated runtime in this task.

This section is closed presenting the time complexity of the k-cut computation. Considering an AIG with n nodes, k-input cuts and C cuts stored per AIG node, the worst case time complexity for one pass of priority cuts computation is $O(n.k.C^2)$. Our parallel k-cut enumeration does not change this time complexity, but it enables practical speedups during the technology mapping by processing many nodes in parallel.

4.3.2 Parallel Structural Mapper

Algorithm 4.4 presents a top-level view of the proposed parallel structural mapper. The method receives as input an AIG, the LUT size k, the number of cuts C stored per AIG node, the desired number of threads and the number of area recovering iterations. The proposed parallel-aware priority cut manager is instantiated and the initial set of active nodes is defined by the AIG PIs and latches. During the parallel processing, active nodes are stored into a worklist *PerSocketChunkBag<1000>*, which distributes chunks of 1000 nodes to the multi-core processor sockets. The Galois parallel for_each consumes and produces new active nodes while executing the operator specified by the cut manager.

Before starting the next mapping pass, the required times of AIG nodes are computed in order to preserve the depth-optimal solution obtained in the first mapping pass. This is a standard technique in timing-driven technology mapping. However, to deal with challenges introduced by the parallel k-cut computation, we are proposing a top-level mapping flow that fits better to a parallel environment. The main differences of our top-level flow compared to the ABC mapper $\&if$ are:

- Interleaving mapping passes based on *exact area* and *area flow* in order to escape from local minimum during area recovering;
- Computing the current *covering* after each mapping pass to use the covering information to guide the area recovering heuristics;
- Updating the node *reference counters* only after each mapping pass by using the reference counting that express the current covering.

By handling the node reference counters according to the proposed approach, we ensure that the parallel structural mapper has a deterministic behavior independent of the number of threads used when running the mapper. Experimental results have demonstrated that, besides the runtime improvements, the proposed solution avoids significant QoR degradation as well as enables better area results in some cases, when compared to the state-of-the-art method implemented in ABC command $\&if$.

For a better understanding, we are explicitly presenting the pseudocode responsible for updating the reference counter in lines 22-27 of Algorithm 4.4. However, in practice, reference counters are reset to zero during the required time computation (Algorithm 4.1) and then they are directly incremented during the covering computation (Algorithm 4.2). This way, we are avoiding several iterations over the cuts in the current covering.

Algorithm 4.4: Parallel Structural LUT-Based Mapping

```
1  Function parallelStructuralMapping()
       Input  : AIG, k, C, nThreads, nRecovery
       Output: a network mapped into k-LUTs
2      PriotityCutManager CutMan(AIG, k, C);
3      GaloisSetThreads( nThreads);
4      // All primary inputs and latches are the initial active nodes at each pass
5      setMappingGoal( depthEdge); //  Cut edge as tie-break
6      galois_for_each( AIG.PIs, CutMan ); //  Parallel for
7      prepareNextPass( AIG, CutMan);
8      setMappingGoal( depthFlow); //  Cut area flow as tie-break
9      galois_for_each( AIG.PIs, CutMan ); //  Parallel for
10     //Two iterations of area recovering by default
11     for ( i = 0 ; i < nRecovery ; i++ ) do
12         prepareNextPass( AIG, CutMan);
13         setMappingGoal( exactArea);
14         galois_for_each( AIG.PIs, CutMan ); //  Parallel for
15         prepareNextPass( AIG, CutMan);
16         setMappingGoal( areaFlow);
17         galois_for_each( AIG.PIs, CutMan ); //  Parallel for
18     return computeCovering();
19 Function prepareNextPass()
       Input  : AIG, CutMan
       Output: updated data structures
20     coputeRequiredTimes( AIG, CutMan);
21     Covering = computeCovering( AIG, CutMan);
22     for each AIG.node n do
23         n.refCounter = 0;
24     for ( each node n ∈ Covering ) do
25         c = CutMan.getBestCut( n);
26         for ( each leaf l ∈ c ) do
27             l.refCounter++;
```

4.3.3 Parallel Functional Mapper

In this work, we are introducing a novel approach for synergistically integrating logic rewriting and LUT-based mapping. In this section, we are presenting our contributions from a theoretical and conceptual standpoint whereas a practical implementation of the proposed approach is under development.

Before introducing the proposed approach, we revisit the discussion on the motivational example presented in the introduction of this work. It is known that the AIG structure has a strong influence in the quality of the technology mapping (CHATTERJEE et al., 2006b). Moreover, it is hard to deal with this challenge since there is a gap between multi-level logic optimization and technology mapping due to the different QoR metrics considered during these synthesis steps (LIU; ZHANG, 2017).

For instance, in conventional area-oriented multi-level logic optimization, algorithms aim to minimize logic node count and logic levels without any information on LUT count and delay (MISHCHENKO; CHATTERJEE; BRAYTON, 2006), (LI; DUBROVA, 2011), (YANG; WANG; MISHCHENKO, 2012), (SOEKEN et al., 2016), (HAASWIJK et al., 2017) and (POSSANI et al., 2018). However, in many practical cases, an AIG with larger number of nodes enables a mapping with fewer LUTs. Therefore, we need more sophisticated metrics for guiding multi-level logic optimizations.

A recent attempt to decrease the gap between multi-level logic optimization and LUT mapping is the method proposed in (LIU; ZHANG, 2017). However, this method still performes logic optimization guided by AIG node count expecting to minimize the LUT count in an iterative process of try-and-error stochastic optimization. Therefore, to the best of our knowledge, there is not an effective method in the literature to compute logic optimization possibilities and to make the best decisions based on technology dependent costs such as LUT count and mapped network delay.

We are proposing a *rewriting-aware functional mapper* where many threads rewrite and remap different subgraphs, simultaneously. The main novelties are established on:

- Expanding the solution space exploration by using rewriting subgraphs;
- Transferring the rewriting decisions to the technology mapper;
- Making the logic optimization decisions based on technology dependent costs.

The proposed approach relies on combining the parallel AIG rewriting introduced in Chapter 3 with the structural mapper based on priority cuts introduced in the previous section.

This solution is promising to minimize the structural bias of the initial AIG structure as well as to minimize the miscorrelation between tech-independent and tech-dependent cost metrics. The rewriting and mapping integration creates more opportunities for improving QoR whereas the multi-threading provides a moderate execution time. A top-level view of the proposed parallel functional mapper is presented in Algorithm 4.5.

Constructing Initial Solution: Firstly, the method produces an initial mapping solution by simple executing the proposed parallel structural mapper, as shown in line 4 of Algorithm 4.5. This initial mapping provides a good start point for applying successive mapping refinements. Moreover, the next steps of the proposed algorithm work on an AIG with a virtual LUT network annotation to enable more accurate decisions during the rewriting-aware tech mapping. In practice, this initial solution is represented by the AIG plus the subset of the best cuts selected in the mapping covering. Typically, each AIG node n has a pointer to its best cut B_{cut_n}. Therefore, if a given node n has a *non-null* pointer to its best cut, it means that n and B_{cut_n} are representing the output and the inputs of a LUT in the current mapping, respectively. Otherwise, it means that n is an internal node to a given LUT in the current mapping. This way, it is possible to track the current mapping during the multi-level logic optimization and to compute how many LUTs are deleted/added in the network at each rewriting attempt.

Algorithm 4.5: Parallel Functional LUT-Based Mapping

```
1  Function parallelFunctionalMapping()
       Input  : AIG, k, C, S, nThreads, nRecovery, nRefinement
       Output: a network mapped into k-LUTs
2      GaloisSetThreads( nThreads);
3      //Produce and initial mapping solution
4      parallelStructuralMapping( AIG, k, C, nThreads, nRecovery);
5      //The AIG with mapping annotation is passed to the RWMap manager
6      RWMapManager RWMapMan(AIG, k, C, S, nRecovery);
7      //Iterative rewriting and mapping refinement
8      for ( i = 0 ; i < nRefinement ; i++ ) do
9          galois_for_each( AIG.PIs, RWMapMan ); //  Parallel for
10     return computeCovering();
```

RWMap Manager: The functional mapper core is defined by the manager operator presented in Algorithm 4.6. This operator specifies the task executed by each thread on top of a given active node. The RWMap manager is instantiated in line 6 of Algorithm 4.5 and it is constructed based on the given AIG, LUT size k, number of cuts per AIG node C, window size S and other parameters. Internally, the manager has instances of all those auxiliary managers previously introduced in this work for enabling parallel rewriting and mapping, i.e., *Parallel-Aware Structural hashing*, *CutManager*, *NPNManager*, *StructureManager* and *PriorityCutManager*. The Galois parallel loop is executed in line 9 of Algorithm 4.5 to launch the parallel execution of the RWMap manager operator.

Window Computation: The first task of the operator in Algorithm 4.6 is to compute a window W around of the given active node. A window can be viewed as multi-output subcircuit used to delimit a perimeter of optimization in the logic network. Typically, the window computation starts from a given node n in the network and collects some nodes in the l levels of the transitive fanin and fanout cones of n. The window complexity can be controlled in terms of the number of visited levels l, AIG nodes or LUTs inside the window. This complexity can be easily controlled by input parameters and it is directly related to the desired logic synthesis effort. Windowing is a well-established technique with several variations to fit in its target application (MISHCHENKO; BRAYTON, 2006; SCHMITT; MISHCHENKO; BRAYTON, 2018). In the proposed approach, the threads try to acquire the necessary logical lock of AIG nodes during the window computation. If all necessary locks are successfully acquired, it means that the operator can proceed to rewrite and remap the window safely. Otherwise, the thread aborts its execution and the active node being processed is rescheduled to be processed latter on.

Exploring the Solution Space: After computing the window W, the operator executes a modified version of AIG rewriting procedure, shown in lines 8-16 in Algorithm 4.6. Firstly, 4-input cuts are computed inside the window in order to explore the possibilities for rewriting W. The algorithm iterates on the set of 4-input cuts computing the cut Boolean function, applying NPN classification and looking up the table of pre-computed subgraphs. However, instead of making a decision to select the pair of cut/subgraph that leads to the best AIG node reduction, the algorithm accumulates all the rewriting possibilities as choices inside the window. In other words, we are postponing the decision on what cuts/sugraphs will be select for rewriting. This way, we are transferring the rewriting decisions to the technology mapper, which is able to perform more accurate decisions by considering the LUT counting and delay instead of considering AIG nodes and levels.

Algorithm 4.6: Rewriting-Aware Mapping Operator

```
1  RWMapManager begin
2      constructor( AIG, k, C, S, nRecovery);
3      Function operator( node, GaloisCtx)
4          //All necessary logical locks are acquired when computing the window
5          W = computeWindow( node, S);
6          for each node n in W do
7              //4-input cuts for rewriting
8              Cuts = cutMan.computeKCuts( n);
9              //Adding all the rewriting possibilities as choices
10             for each 4-input cut c in Cuts do
11                 f = cutMan.getCutFunction( n, c);
12                 f_npn = npnMan.getRepresentative( f);
13                 S = strMan.lookupStructures( f_npn);
14                 for each structure s in S up to nStr do
15                     //Transferring rewriting decisions to tech mapper
16                     appendStructureAsChoice( n, c, s);
17         //Making rewriting decisions based on technology costs (LUTs & delay)
18         M = structuralMapping( W, k, C, nRecovery);
19         if ( W.cost > M.cost ) then
20             commitRewritingAndMapping( W, M);
21         cleanupUselessChoices( W);
22         pushFanoutNodes( node, GaloisCtx);
```

Rewriting-Aware Tech Mapping: The next step is to execute a structural mapping with choices on the window in order to evaluate the potential improvement of all the rewriting possibilities at the same time. This task can be solved by a simple variation of the proposed structural mapper introduced in previous section. The mapper is responsible for executing multiple passes of depth-optimal mapping and area recovering considering all available choices inside the window. Finally, the area and delay costs of the resulting mapping M are compared to the initial costs of the window W. If there is improvement, then the window is updated to express the new mapping in terms of the

rewriting subgraphs represented as choices. The choices that are needed to express the new mapping are committed by properly reconnecting fanout edges in the AIG whereas the useless choices are removed from the graph. The last step of the operator is to schedule the fanouts of the active node to be processed in the next iterations.

Discussion on Flexibility: Notice that we are proposing a novel and flexible concept for synergistically integrating logic rewriting algorithms to LUT-based mapping ones. Therefore, the proposed approach for parallel functional mapping is not limited to the combination of the parallel AIG rewriting and the parallel structural mapper previously proposed in this work. The proposed approach presented in Algorithm 4.6 is generic and can employ any logic rewriting method such as those proposed in (LI; DUBROVA, 2011), (YANG; WANG; MISHCHENKO, 2012), (SOEKEN et al., 2016) and (HAASWIJK et al., 2017). Analogously, different LUT-based mappers can be applied to solve the task presented in line 18 of Algorithm 4.6. For instance, we consider that the SAT LUT mapping proposed in (SCHMITT; MISHCHENKO; BRAYTON, 2018) is a promising approach for evaluating all the rewriting possibilities represented as choices and making an accurate decision. We intend to adapt SAT LUT method to handle choice nodes during the SAT-based covering computation and integrate it in our functional mapper.

Discussion on Related Works: Our parallel functional mapper relies on a set of techniques previously introduced in the literature, and it is important to highlight the differences of the proposed approach to previous works considering mapping with choices (LEHMAN et al., 1997; STOK; IYER; SULLIVAN, 1999; CHATTERJEE et al., 2006b; MISHCHENKO; CHATTERJEE; BRAYTON, 2007).

The methods presented in (LEHMAN et al., 1997) and (STOK; IYER; SULLIVAN, 1999) apply some algebraic transformations based on *associative* and *distributive* rules in order to produce different structures for the subject graph. These local transformations are applied for creating choices during standard-cell design methodology. However, this approach is limited to such a primitive transformation which are strongly dependent on the initial structure of the subject graph. Notice that our proposal for integrating rewriting and mapping is flexible, allowing to take advantage of large data bases of precomputed structures. Moreover, powerful synthesis techniques can be applied for computing optimized structures in advance whereas the proposed approach can choose the most appropriate structures based on tech-dependent costs during the mapping process.

Regarding the traditional mapping flow based on *lossless synthesis* (CHATTERJEE et al., 2006b; MISHCHENKO; CHATTERJEE; BRAYTON, 2007), this method

aims to execute scripts for multi-level logic optimization in a conventional way and to collect some intermediary solutions to created choices. Therefore, notice that in such an approach the logic optimizations are applied totally independent of the technology mapping and still guided by the AIG node and level counting. On the other hand, we are proposing a novel alternative to effective combining such a techniques in a fine-grain level of integration that enables multi-level logic optimization driven by technology mapping cost functions. This way, every local step of logic optimization is made based on the target technology costs.

We are not arguing that the proposed approach substitutes the original lossless synthesis proposed in (CHATTERJEE et al., 2006b; MISHCHENKO; CHATTERJEE; BRAYTON, 2007). We are suggesting that both techniques are complementary and there is much room for applying both of them in different steps of logic synthesis. For instance, a promising synthesis scenario to improve both QoR and runtime is:

- Applying the proposed parallel AIG rewriting and other logic optimization methods based on the original lossless synthesis approach to accumulate choices.
- Applying the proposed parallel structural mapper to produce a good initial solution quickly, benefiting from choices created by the original lossless synthesis;
- Applying the proposed parallel functional mapper to refine the initial solution based on the synergistic integration of rewriting and mapping.

4.4 Experimental Results

In the following experiments, we compare the proposed parallel structural mapper to the state-of-the-art LUT-based mapper implemented in the ABC command $\&if$. The parallel technology mapper was implemented in C++ 11 using Galois system (PINGALI et al., 2011; LENHARTH; NGUYEN; PINGALI, 2016). Both the proposed method and ABC tool (Berkeley Logic Synthesis and Verification Group,) were compiled using GNU g++ version 6.1.0 and executed in a 64-bit Linux distribution. The results were collected on a server with 128GB of shared RAM and 4 processors Intel®Xeon®CPU E7- 4860 operating at 2.27GHz, where each processor has 10 physical cores. We ran command $\&cec$ in ABC to check the correctness of the solutions provided by the parallel mapper on a large set of MCNC, ISCAS and EPFL benchmarks.

4.4.1 Benchmarks

To evaluate our parallel mapper, we have adopted the same set of circuits comprising millions of AIG nodes used to evaluate the proposed parallel AIG rewriting method in the previous chapter of this work. Table 4.1 provides a fresh view about the adopted benchmark circuits. The LUT mapping methods under comparison were executed directly on the circuits presented in Table 4.1 without any previous optimization. This comparison considers as criteria the runtime and QoR provided by each method in terms of number of LUTs and logic depth of the mapped networks. We are considering the runtime used exclusively in the technology mapping task, discarding the parsing runtime of both ABC command $\&if$ and the proposed method. Moreover, the runtime comparison is based on a single execution for each circuit and thread configuration, since multiple executions could take some weeks.

4.4.2 Parallel Structural Mapper Scalability

We start evaluating the scalability of the proposed parallel structural mapper by running it at 1 thread up to 40 threads. The mapper was evaluated under two different scenarios by mapping the circuits into 6-input and 8-input LUTs, storing 20 cuts per AIG

Table 4.1: The set of ten circuits obtained by applying 10x (8x) the ABC command *double* and the three MtM AIG nodes circuits from the EPFL benchmark suite.

Circuit	# PI	# PO	# AND2	# Levels
sin_10xd	24,576	25,600	5,545,984	225
arbiter_10xd	262,144	132,096	12,123,136	87
voter_10xd	1,025,024	1,024	14,088,192	70
square_10xd	65,536	131,072	18,927,616	250
sqrt_10xd	131,072	65,536	25,208,832	5,058
mult_10xd	131,072	131,072	27,711,488	274
log2_10xd	32,768	32,768	32,829,440	444
mem_10xd	1,232,896	1,260,544	47,960,064	114
hyp_8xd	65,536	32,768	54,869,760	24,801
div_10xd	131,072	131,072	58,620,928	4,372
sixteen	117	50	16,216,836	140
twenty	137	60	20,732,893	162
twentythree	153	68	23,339,737	176

node. The ABC mapper was executed in an equivalent mode by using the commands "$\&if$ -K 6 -C 20" and "$\&if$ -K 8 -C 20". Both methods were executed considering the same number of mapping passes, two passes for depth-optimal mapping followed by four passes of area recovering (two of area flow and two of exact area).

Table 4.2 presents the absolute runtimes for each mapping scenario when executing the single-threaded $\&if$ mapper and the proposed parallel structural mapper running at 40 threads. The proposed approach has enabled significant runtime reductions, bringing the runtime down from more than 1h to only few minutes in several cases. The runtime of the ABC mapper was taken as reference to compute the speedups. Our parallel mapper presents a promising scalability as shown in Fig. 4.3 and in Fig. 4.4. We have observed speedups up to 21x and 25x when mapping into 6-input and 8-input LUTs, respectively.

In most cases, the parallel mapper keep scaling as the thread count increases, except for the three MtM circuits in which our mapper has presented a similar scalability behavior to the parallel AIG rewriting. This weak scalability is due to many thread conflicts generated by the structural bottleneck present in these circuits, as previously discussed in Subsection 3.4.4 of this work. However, even for these hard cases, the proposed approach was able to accelerate the mapping process up to 5x, bringing the runtime down from 30min to only 7.7min.

Table 4.2: Runtimes of the technology mapping methods under comparison, in seconds.

	-K 6 -C 20		*-K 8 -C 20*	
Circuit	ABC *&if*	Parallel *40 threads*	ABC *&if*	Parallel *40 threads*
sin_10xd	647.47	38.69	1,004.18	54.80
arbiter_10xd	489.63	53.79	782.74	68.60
voter_10xd	1,232.35	84.41	2,187.62	127.10
square_10xd	2,326.20	111.15	3,690.48	149.70
sqrt_10xd	2,447.40	181.26	4,781.64	277.52
mult_10xd	2,922.03	180.06	5,387.81	251.77
log2_10xd	3,754.37	218.44	7,522.44	335.28
mem_10xd	3,109.24	232.84	5,256.27	304.80
hyp_8xd	6,609.16	387.17	10,576.95	544.14
div_10xd	6,378.03	376.79	10,974.52	548.74
sixteen	1,307.07	414.31	1,565.11	581.02
twenty	1,850.07	463.44	2,180.85	827.02
twentythree	2,251.62	478.95	2,748.72	957.21

It is challenge to get speedups when compared to the *&if* command due to the different AIG representations used by each method. On the one hand, the proposed mapper works on the AIG package we designed based on the generic data structure provided by Galois *FirstGraph*. This AIG representation has some overheads to support concurrent operations and graph topology modifications. On the other hand, the *&if* mapper works on a very compact and tuned AIG representation called GIA. The GIA package represents the graph as a single array where each position stores the data related to a given AIG node. The nodes are natively stored in a topological order and the fanout edges of each node are not represented. Therefore, the mapper can traverse the graph in the topological order quickly by executing a *for* iterator on the array. This data structure is very efficient to implement structural mappers, since the graph topology is not modified during the mapping process.

However, we highlight that the proposed parallel mapper has prensented promising speedups and is more flexible by supporting parallel modifications into the graph topology towards our parallel functional mapper. Besides that, runtime improvement can be achieved by fine tuning the generic Galois *FirstGraph* to fit better into logic synthesis tasks.

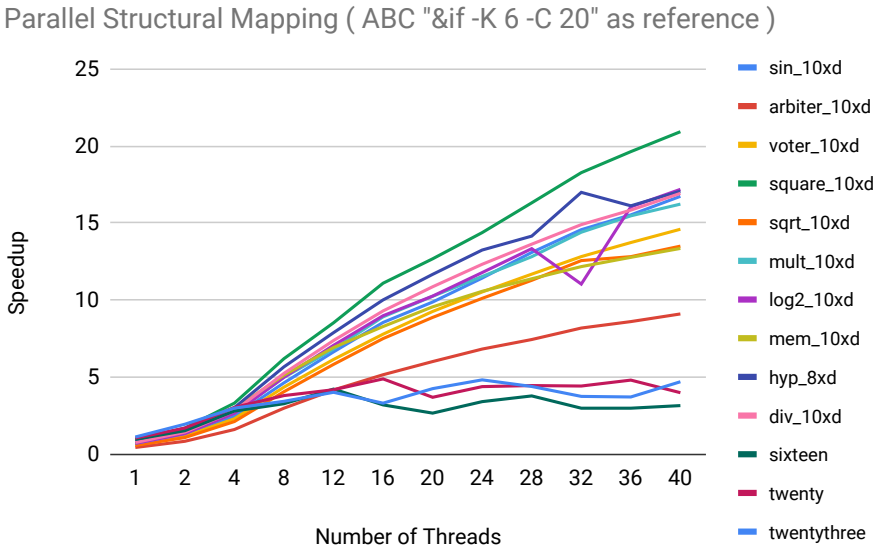


Figure 4.3: Speedups and scalability behavior of parallel structural mapper compared to ABC command *&if -K 6 -C 20*.

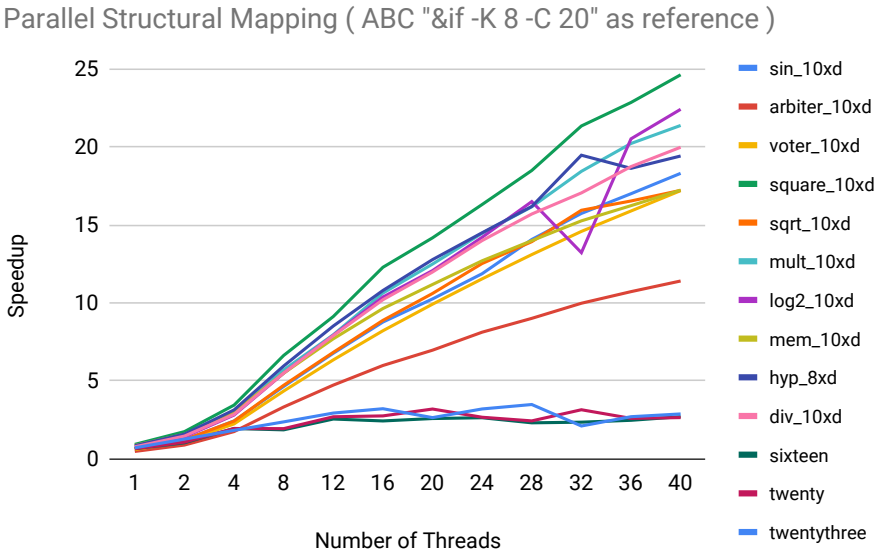


Figure 4.4: Speedups and scalability behavior of parallel structural mapper compared to ABC command *&if -K 8 -C 20*.

4.4.3 Parallel Structural Mapper QoR

The experimental results have demonstrated that the proposed parallel mapper delivers solutions with similar quality when compared to the $\&if$ mapper. A comparative analysis in terms of LUT count and logic depth is presented in Table 4.3 and in Table 4.4. The absolute values of the solutions delivered by $\&if$ mapper were taken as reference to compute the quality ratio provided by our parallel mapper.

Both methods are able to deliver the depth-optimal solution considering a fixed topology of the AIG received as input. Even though we have introduced different heuristics to update node reference counters when computing k-cut area, the proposed approach has presented competitive quality when compared to the state-of-the-art mapper $\&if$. Since the area recovering techniques are based on heuristics, there is much room for delivering solutions with different area cost. For the mapping into 6-input LUTs, we have achieved improvements in LUT count up to 14% and degradation up to 11%. Whereas, for the mapping into 8-input LUTs, we have achieved improvements in LUT count up to 18% and degradation up to 22%.

Table 4.3: QoR ratio of the parallel mapping over ABC command $\&if$ -K 6 -C 20.

	ABC $\&if$		Parallel	
Circuit	*LUTs*	*depth*	*LUTs*	*depth*
sin_10xd	1,482,752	42	0.98	1.00
arbiter_10xd	2,787,328	18	1.00	1.00
voter_10xd	2,306,048	16	1.07	1.00
square_10xd	4,072,448	50	0.92	1.00
sqrt_10xd	6,529,024	1,024	0.86	1.00
mult_10xd	6,014,976	53	0.98	1.00
log2_10xd	8,015,872	76	1.01	1.00
mem_10xd	12,057,600	25	1.02	1.00
hyp_8xd	11,405,568	4,195	1.02	1.00
div_10xd	22,825,984	864	0.93	1.00
sixteen	4,065,972	29	1.08	1.00
twenty	5,034,882	33	1.08	1.00
twentythree	5,641,898	36	1.11	1.00

Table 4.4: QoR ratio of the parallel mapping over ABC command &if -K 8 -C 20.

	ABC &if		Parallel	
Circuit	*LUTs*	*depth*	*LUTs*	*depth*
sin_10xd	1,257,472	30	0.96	1.00
arbiter_10xd	2,118,656	13	1.00	1.00
voter_10xd	2,166,784	12	0.99	1.00
square_10xd	3,445,760	36	0.96	1.00
sqrt_10xd	5,234,688	692	0.82	1.00
mult_10xd	4,162,560	40	0.97	1.00
log2_10xd	4,533,248	48	0.89	1.00
mem_10xd	8,468,480	19	1.05	1.00
hyp_8xd	10,920,704	2,818	0.94	1.00
div_10xd	19,206,144	617	1.02	1.00
sixteen	2,950,996	22	1.16	1.00
twenty	4,046,997	24	1.19	1.00
twentythree	4,485,663	27	1.22	1.00

4.4.4 Parallel Functional Mapper Case Study

Since the proposed parallel functional mapper is under development, we are presenting a case study in order to present evidence that the proposed approach can bring practical benefits in QoR. We selected the $i2$ circuit, from the MCNC benchmark (YANG, 1991), that is a small combinational circuit with similar size of a window extracted from a larger circuit. Therefore, in this case study, this circuit play the role of a given window under optimization. We are presenting four different scenarios for optimizing and mapping the $i2$ considering three alternative flows in ABC tool as well as a partial version of the proposed *RWMap*. These four scenarios are illustrated in Fig. 4.5 and defined as:

- **ABC 1**: applies just the LUT mapping on the original AIG, commands "&if -k 6".
- **ABC 2**: applies a single pass of AIG rewriting without level preservation followed by LUT mapping, commands "*rewriting* -l ; &if -k 6".
- **ABC 3**: applies *compress* and *compress*2 scripts for multi-level logic optimization and collects choices followed by LUT mapping, commands "&dch ; &if -k 6".
- **RWMap**: applies the two main steps presented in Algorithm 4.6, *explore* by adding the rewriting structures as choices and *decide* based on technology mapping costs.

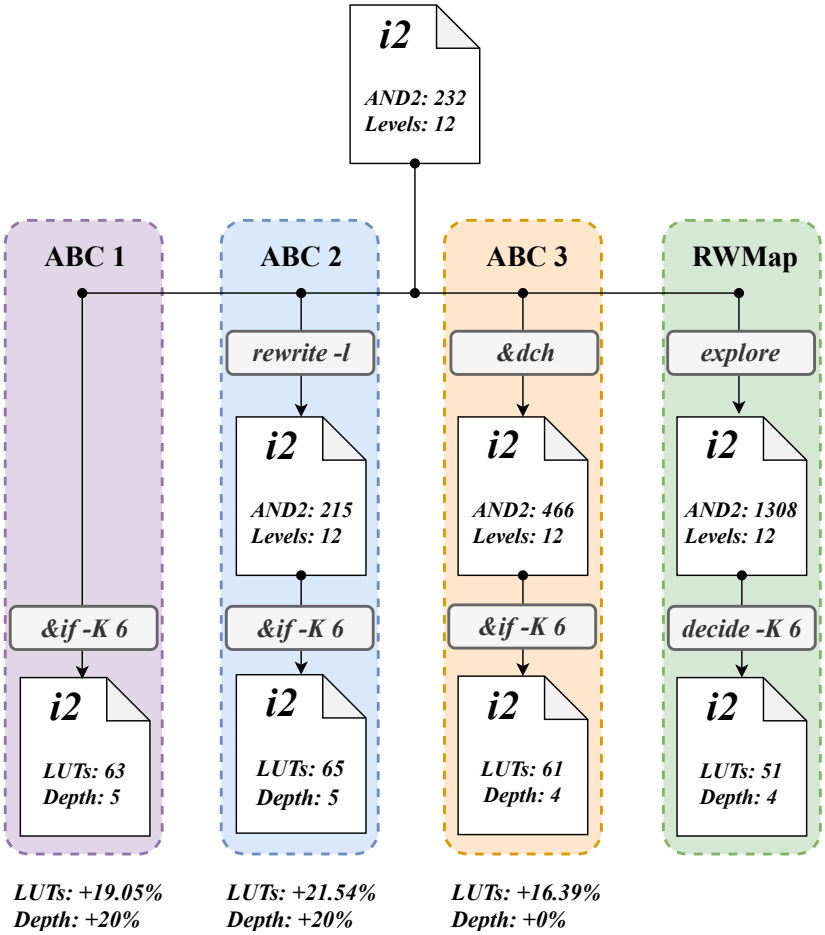


Figure 4.5: Case study on the *i*2 circuit from MCNC benchmark: QoR analysis by integrating rewriting (*explore*) and mapping (*decide*) as proposed in *RWMap* manager of Algorithm 4.6.

All the flows under comparison were executed in a single-thread version, since the main objective is to evaluate the QoR. The runtime of each flow was around of 1 second or less, due to the small instance of the problem. In Fig. 4.5, we present the initial and the intermediary solutions in terms of AIG node and level counting whereas the final solutions are presented in number of LUTs and logic depth. The three ABC flows significantly increase the synthesis effort from **ABC 1** to **ABC 3**. However, there is a small difference in the practical benefit attained in the final solutions when comparing these three ABC flows. In turn, the proposed approach was able to go further by performing multi-level logic optimizations driven by tech-dependent costs. Therefore, we conjecture that in a real synthesis scenario, the proposed approach can make better decisions when resynthesizing circuit windows in an incremental and iterative process.

One can argue that the proposed flow presented in Fig. 4.5 produced better results by employing more complex and intensive optimizations than the other three ABC flows.

However, the proposed flow has a computing cost as small as the cost of the flow described in **ABC 2**. Our approach only executes a single pass adding rewriting choices and makes decision based on the proposed structural mapper which works similarly to the ABC $\&if$. When compared to **ABC 2** flow, the proposed approach has an extra cost to consider choices during mapping. This extra cost is not a bottleneck because our approach can ensure scalability by controlling the windows size and computing k-cuts in parallel.

ABC 3 represents the most complex flow among the four evaluated scenarios. This flow employs the *&dch* command that internally executes the script *compress* followed by the script *compress*2. These two scripts are based on sequences of algorithms for multi-level logic optimization like *rewriting* (*rw* and *rwz*), *balancing* (*b*) and *refactoring* (*rf* and *rfz*). The switch *-l* is used for disabling level preservation since these scripts perform area-oriented optimizations as follows:

- *compress*: "*b -l* ; *rw -l* ; *rwz -l* ; *b -l* ; *rwz -l* ; *b -l*"
- *compress*2: "*b -l* ; *rw -l* ; *rf -l* ; *b -l* ; *rw -l* ; *rwz -l* ; *b -l* ; *rfz -l* ; *rwz -l* ; *b -l*"

Notice that the *&dch* command applies *rewriting* 7x, *balancing* 7x and *refactoring* 2x by internally executing the scripts *compress* and *compress*2. Moreover, after executing these scripts, *&dch* executes SAT solving to prove equivalences between intermediary networks in order to create choices. Therefore, the *&dch* command is significantly more expensive where thousands of optimization attempts are tried and rejected during there 16x optimization passes guided by AIG node and level counting.

Although, the *&dch* command is well established and has presented good results, we argue that there is room for improvement in both runtime and QoR by considering a more effective integration for creating and making choices in the technology mapping. Our intuition is that the proposed *RWMap* will be able to leverage the quality of conventional rewriting methods to another level just by considering more accurate decisions. However, since we do not have experimental results to support this claim yet, such an improvement remains hypothetical.

4.5 Summary

In this chapter, we discussed the main challenges on LUT-based technology mapping and proposed two complementary methods for accelerating and improve the mapping process. The proposed parallel structural mapper has presented promising speedups as well as competitive QoR when compared to the state-of-the-art structural mapper implemented in ABC command $\&if$. Experimental results have demonstrated speedups of up to 25x, representing practical runtime reductions from more than one hour to few minutes.

Besides that, we introduced a set of principles for integrating logic rewriting and technology mapping in a flexible way, leading to a parallel functional mapper. The main novelty of the proposed approach is to explore the solution space based on rewriting subgraphs while making optimization decisions totally based on the tech-dependent costs. We are introducing a novel concept for a fine-grain integration of multi-level logic optimization and mapping. Therefore, several other optimization and mapping algorithms from the literature can be employed to work based on the proposed concept.

5 COMBINATIONAL EQUIVALENCE CHECKING

Fast and scalable techniques for combinational equivalence checking (CEC) are essential in modern electronic design automation (EDA) environments. In a typical scenario, the logic function implemented by an optimized digital integrated circuit (IC) is checked for equivalence to the original specification after multi-level logic synthesis (BRAYTON; HACHTEL; SANGIOVANNI-VINCENTELLI, 1990). Moreover, scalable CEC techniques are quite useful in several key logic synthesis processes, which depend on efficient computation of equivalence classes of internal circuit nodes. A non-exhaustive list of these tasks is the following:

- Removal of functionally equivalent logic in the design during logic synthesis and optimization;
- A number of utility packages, *e.g.* node name transfer across netlists before and after synthesis;
- Sequential equivalence checking based on register and signal correspondence (BJESSE; CLAESSEN, 2000; MISHCHENKO et al., 2008);
- Bridging circuits for the implementation and the specification in engineering change orders (ECOs) (KRISHNASWAMY et al., 2009);
- Computation of structural choices, enabling circuit area and signal delay improvement after technology mapping step (CHATTERJEE et al., 2006a).

In recent years, equivalence checking has become more critical due to the increasing in complexity of current and upcoming system-on-chip (SoC) and VLSI designs. In the introduction of this thesis, we highlighted the complexity of CEC when dealing with large designs by presenting and instance where the modern CEC engine from ABC tool took more than 24 hours to prove logic equivalences. In order to enable the next rounds of technology innovation, there is a demand for massively parallel EDA tools running on the cloud (STOK, 2013; STOK, 2014). Considering this scenario, traditional EDA algorithms and data structures, in particular, those dealing with CEC, need to be rethought to benefit from parallel computing platforms.

In typical EDA environment, algorithms work on a sparse graph representing the circuit. We reinforce that it is often challenging to exploit parallelism of graph-based algorithms due to irregular nature of the graphs implemented using pointer-based data-structures. Moreover, the parallelization challenge increases when the graph-based algo-

rithm relay on logic simulation and Boolean satisfiability (SAT) (MISHCHENKO et al., 2006)

In this chapter, we are unlocking massive parallelism for CEC by applying graph (miter) partitioning in order to balance data sharing and data independence during SAT solving. Three complementary models for enabling parallelism in CEC are proposed, addressing different design and verification scenarios. Experimental results have demonstrated speedups up to 63x when compared to a single-threaded implementation of similar CEC engine. The practical impact of such a speedup represents a runtime reduction from 19 hours to only 18 minutes. Therefore, the proposed solution presents great potential for improving current EDA environments.

We start providing a background on the main concepts and techniques used in modern CEC engines, before to review recent related works and to introduce the proposed parallel CEC approach.

5.1 Background

5.1.1 Boolean Satisfiability

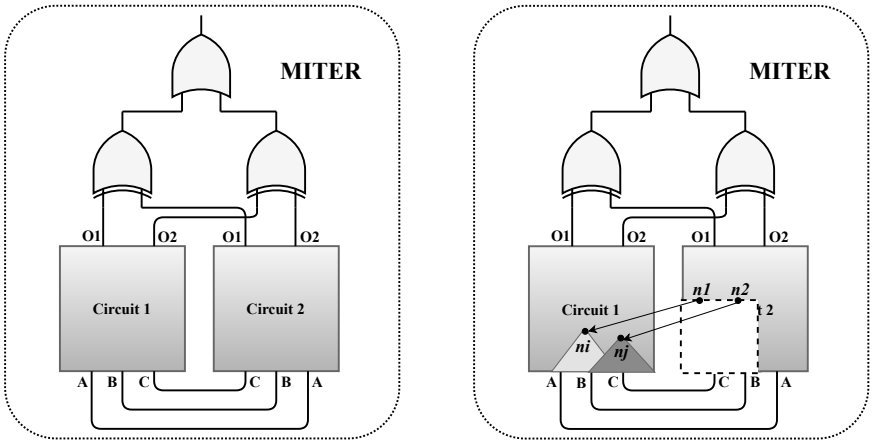
Boolean satisfiability is the decision problem to determine whether there exist an assignment to the input variables that makes the output of a given Boolean formula evaluates to *true*. If such an assignment exists, then the formula is $satisfiable$ (sat). Otherwise, the formula is $unsatisfiable$ ($unsat$). Conventionally, SAT problem instances are represented by a formula in the conjunctive normal form (CNF). SAT solvers are tools implementing advanced techniques for deciding whether a given SAT instance is *sat* or *unsat* (EéN; SöRENSSON, 2004). Several techniques are used for speeding up SAT solving such as the cube-and-conquer method (HEULE et al., 2018), the dual SAT solving (AMARù et al., 2015), and others. Many practical problems can be modeled using SAT and efficiently solved by modern SAT solvers.

5.1.2 Mitering

A typical CEC engine transforms two circuits under verification into a single circuit called *miter* (BRAND, 1993). The miter is constructed by pair-wise connecting the inputs

with the same name and by pair-wise comparing the outputs with the same name using exclusive-OR (EXOR) operator. At the top of the miter, an OR operator connects all the outputs of EXORs, representing the output of the comparator for the primary outputs, as shown in Fig. 5.1(a).

In practice, a miter can be represented by a single AIG, where each output is a separate decision problem. The AIG is viewed as an instance of a multi-output SAT problem that can be easily converter to CNF form by using the Tseitin transformation (TSEITIN, 1983), and then given to the SAT solver. If the SAT solver returns *unsat*, all pairs of outputs under comparison are equivalent. Otherwise, the solver returns *sat*, *i.e.*, at least one pair of outputs is different. This process for proving equivalence using miter and SAT solving is also referred as *mitering*.



(a) (b)

Figure 5.1: Miter structure (a) with reduced logic (b) by merging equivalent internal nodes.

5.1.3 BDD and SAT Sweeping

The techniques known as *BDD sweeping* (KUEHLMANN et al., 2002) and *SAT sweeping* (KUEHLMANN, 2004) are used to detect functionally equivalent nodes internally in AIG or miter. In this approach, pairs of internal nodes are checked for equivalence in a topological order. If the equivalence is proved, the corresponding nodes can be merged to simplify the miter, as shown int Fig. 5.1(b). BDD and SAT sweeping are useful since to prove equivalence by directly constructing a BDD of a miter for the entire circuits under

verification is often not practical (BRYANT, 1986). In many cases, BDD construction and SAT solving for all the outputs require a lot of memory and computation resources.

5.1.4 Logic Simulation

Logic simulation is a common technique used to quickly detect non-equivalent nodes, helping to reduce the number of SAT calls during SAT sweeping. Typically, random vectors and counter-examples returned by SAT solvers are used as input patterns for simulation (KUEHLMANN, 2004; LU et al., 2003). A *counter-example* is an assignment of input variables of the Boolean formula representing the SAT problem instance, proving that a pair of nodes is not equivalent. This way, simulation is used to group potential-equivalent nodes into classes whereas SAT solving is used for checking equivalences among nodes belonging to the same class.

5.2 Related Works on CEC

In the last decades, several works addressed the CEC problem by proposing solutions based on BDDs (MATSUNAGA, 1996; MOONDANOS et al., 2001; MOON; PIXLEY, 2004) or hybrid BDD/SAT-based engines (KUEHLMANN et al., 2002). Most part of earlier CEC engines prove logic equivalence based ROBBDs (BRYANT, 1986). In many cases, the construction of ROBDDs is not practical due to the increasing size and complexity of current and upcoming designs. Such approaches are less scalable when running on a single thread and harder to parallelize than those based on simulation and SAT (LU et al., 2003; MISHCHENKO et al., 2006).

In the following, we revisit previous works that are most related to the proposed approach or that are complementary to it somehow. We start revisiting the state-of-the-art CEC engine available in ABC tool, which is based on simulation and SAT solving. Besides that, we revisit other recent approaches introduced to accelerate CEC or to accelerate SAT solving that has directly impact on the performance of CEC engines.

5.2.1 CEC Engine in ABC Tool

Typically, modern CEC engines alternate between miter solving to prove equivalences for outputs and miter simplification to prove equivalence for internal nodes. Simulation and SAT sweeping are used for gradually simplifying the miter complexity. Therefore, for the sake of simplicity, we have adopted the terms *main miter* and *internal miter* when referring to the data being processed in different steps of the CEC engine core.

- The *main miter* refers to the miter created once at the beginning of the verification process. This miter comprises all the logic of the two circuits under comparison. During the CEC, the main miter is gradually simplified by merging internal equivalent nodes from both circuits.
- The *internal miter* refers to the miter created temporarily at each integration of SAT sweeping in the CEC core. This miter comprises subgraphs from the main miter that represent a set of CEC subproblems for determining those internal equivalent nodes.

Fig. 5.2 presents a scheme of the CEC core implemented in the ABC command *&cec* (Berkeley Logic Synthesis and Verification Group,). This command implements

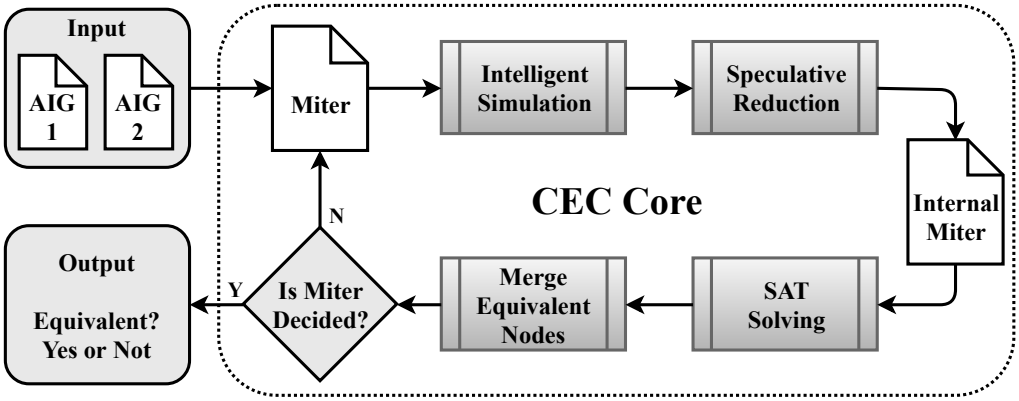


Figure 5.2: CEC engine of ABC command &*cec*.

an improved version of the method proposed in (MISHCHENKO et al., 2006). This engine is based on the SAT solver *MiniSat* (EéN; SöRENSSON, 2004) and employs several techniques introduced in previous work such as *intelligent simulation* and *functionally reduced AIGs (FRAIGs)* (MISHCHENKO et al., 2005) .

Initially, intelligent simulation is used to group nodes that appear to be equivalent into classes. The input patterns used for simulation are computed based on SAT counter-examples that contribute to distinguish non-equivalent nodes belonging to candidate equivalence classes. An extended set of *distance-1* simulation vectors are produced by flipping one bit at a time from the counter-example (MISHCHENKO et al., 2006).

While simplifying the main miter in the &*cec* command, pairs of nodes from each class are represented in the temporary internal miter and checked by the SAT solver. All those nodes proved equivalent are merged in order to simplify the main miter under verification and the counter-examples are used to disprove equivalences in further iterations. This iterative process of miter refinement and checking is executed until: (i) the miter is fully solved (proved *unsat*); (ii) a counter-example is detected (proved *sat*); or (iii) a resource limit is reached.

By carefully investigating the ABC CEC engine, we have observed that the most advantageous strategy to accelerate &*cec* is to apply main miter and internal miter partitioning for enabling parallel SAT solving. The parallel CEC we are proposing in this work is based on this observation and exploits trade-offs between data sharing and data independence to solve SAT problems in multiple solver instances.

5.2.2 Other Recent Related Works

In (CHATTERJEE; BERTACCO, 2010), a different method called EQUIPE was proposed for parallelizing CEC based on a hybrid solution that combines CPU and GPU. The authors exploit parallelism in GPU to perform signature-based analysis and structural matching in order to minimize the number of SAT calls. The method evaluates the circuits under verification trying to prove speculatively the equivalence between internal nodes. When the equivalence cannot be proved by the GPU-based solution, a SAT solver is then executed in the host processor (CPU) to check equivalence of individual nodes. However, even when using 14 GPU-cores and 4 CPU-cores, the speedup provided by EQUIPE is limited to a factor of three compared to the non-parallel CEC engine available in ABC tool.

In (AMARù et al., 2015), the authors exploit properties related to Boolean function duality to speed up SAT solving. In several practical SAT application, problems are solved by proving the input CNF is a tautology (constant one) or a contradiction (constant zero). Therefore, the authors exploit the duality between the tautology and contradiction by switching logic operators in the circuit representation in order to produce two equivalent-solvable versions of the same problem. This way, the two dual instances of the problem are solved in parallel by two processor cores and the one which is solved first is used as the final solution whereas the other one is stopped.

This strategy is able to accelerate the SAT solving since one of the instances can be solved much faster than its dual version. Experiments have demonstrated an average runtime reduction of 25% by using both the regular and the dual instance of a given problem compared to use only the regular instance. The solution proposed in (AMARù et al., 2015) and the parallel approach we are proposing in this work are complementary and both solutions can be combined for creating more opportunities to speed up SAT solving and, consequently, CEC engines.

5.3 Proposed Parallel CEC

In this section, we propose three strategies to enable parallelism during equivalence checking. Our parallel CEC is based on the state-of-the-art CEC engine available in ABC command *&cec*, which exploits the synergy between logic simulation and SAT (MISHCHENKO et al., 2006). As discussed before, *&cec* solves many SAT problems appearing in the main miter and in the internal miter derived during SAT sweeping. The proposed approach relies on graph partitioning to exploit data independence during miter simplification and solving. Therefore, we start defining the proposed strategy for graph partitioning. In the sequence, we introduce two models to employ graph partitioning for enabling parallelism during main minter and internal miter processing. Finally, we present how both models can be combined in a third hybrid model that allows to fully exploit the power of parallel computing.

5.3.1 Graph Partitioning

The SAT problem arising in the proof of equivalence between pairs of POs or internal nodes are intrinsically independent of each other. These problems are encoded together in the same graph due to the natural logic sharing introduced during logic synthesis and optimization. We can exploit such an intrinsic independence of the problems to solve them in separate batches. Since each AIG output represents a SAT problem, we have three main motivations for performing the graph partitioning based on POs, as follows:

- The SAT problems for checking pairs of POs or internal nodes are independent from each other.
- We have empirically observed that adjacent POs tend to present more shared logic than randomly selected groups of POs because RTL elaborators, which translate word-level design description into bit-level circuit, place bit-level flops next to each other.
- The set of SAT problems formulated for checking equivalent classes during SAT sweeping are encoded as subsequent POs in the internal miter.

Miter partitioning provides a trade-off between data sharing and data independence during equivalence checking. On one extreme side, we have several SAT problems encoded

into a single miter, which can be incrementally solved at the same SAT solver by exploiting shared clauses (all together). On the other extreme side, we can apply graph partitioning to extract the TFI cones related to the pairs of outputs under comparison and check each pair using independent SAT calls (all separated). In order to achieve an equilibrium between data sharing and data independence, we are proposing to create relatively large partitions comprising a subset of SAT problems. The partitioning enables to solve the subset of SAT problems in parallel by independent SAT solver instances. Algorithm 5.1 presents a top-level view of the graph partitioning procedure.

Algorithm 5.1: Top-level View of Graph Partitioning

```
1  Function GraphPartitioning()
2      input: N (number of partitions), miter ( AIG )
3      output: P ( partitions )
4      int S; // partition size
5      int R; // division reminder
6      if ( miter.nPO ≥ N ) then
7          S = miter.nPO / N;
8          R = miter.nPO % N;
9      else
10         S = 1;
11         R = 0;
12         N = miter.nPO ;
13     if ( R > 0 ) then
14         S++; // to treat remaining POs
15     AIG * P [ N ]; // partitions as an array of AIGs
16     int i = 0, j = 0;
17     for each po in miter do
18         appendTFICone( po, miter, P[ i ]);
19         j++;
20         if ( j == S) then
21             i++; // move to the next partition
22             j = 0; // reset the counter
23             // if all remaining POs were processed
24             if ( −−R == 0 ) then
25                 S−−; // go back to the original size
26     return P;
```

Initially, Algorithm 5.1 checks whether it is feasible to decompose the given miter (AIG) into the desired number of partitions N, as shown in lines 6-12. This checking ensures that the graph partitioning will be consistent by assigning feasible values to the number of partitions N and partition size S. When the division of $miter.nPO$ by N produces a remainder R, the algorithm distributes the set of remaining POs by placing one more PO to the first R partitions. This distribution contributes to produce a workload balance among partitions.

In line 15 of Algorithm 5.1, all partitions start empty and the transitive fanin cones of subsequent POs are gradually appended to the current partition until the partition size is reached, as shown in the loop of lines 17-25. The routine *appendTFICone* collects all the logic that a given PO depends on towards the PIs, recursively. The proposed graph partitioning uses structural hashing and preserves logic sharing inside the partitions with some logic duplication among the partitions. However, since the CEC is a decision problem, logic duplication does not affect the solution quality. In other words, the CEC engine is not sensitive to logic duplication unlike other optimization problems, such as multi-level logic optimization and technology mapping.

The proposed graph partitioning guarantees the soundness and completeness by ensuring that, for each output pair to be checked for equivalence in a given partition, the partition also contains all the logic needed to prove or disprove equivalence (soundness) and each output pair belongs to some partition (completeness). The same graph partition can be applied for both the main miter and the internal miter partitioning processes. The soundness and completeness properties hold for both cases since the internal miter comprises smaller instances of the same problem represented in the main miter.

5.3.2 Main Miter Partitioning

We are introducing the *Model P*, depicted in Fig. 5.3, to decompose the main miter into a set of P independent sub-problems (partitions) and to verify them in parallel. This model receives the pair of circuits under verification and pre-processes them by applying mitering construction and the graph partitioning introduced in Algorithm 5.1. The graph partitioning delivers a list of AIGs representing the miter partitions, which are mapped to threads by using the POSIX Threads standard (*Pthreads*). Each thread receives a miter partition as argument and checks this miter by executing the CEC core

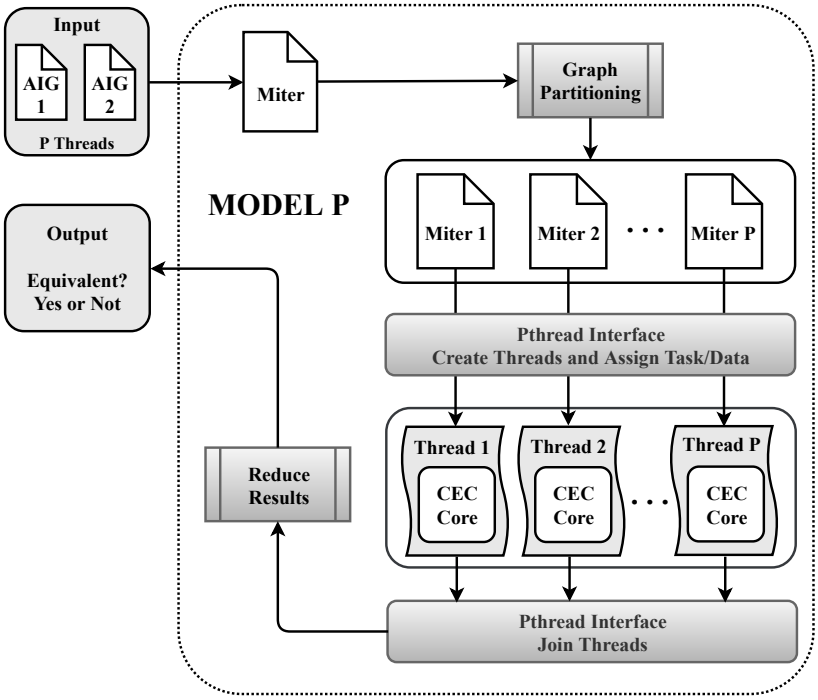


Figure 5.3: Scheme of parallel CEC by main miter partitioning.

illustrated in Fig. 5.2. By applying the proposed graph partitioning, the *Model P* is able to explore massive parallelism where each thread solves part of the problem without any dependencies and conflicts to other threads. Finally, all threads are joined to the main thread and the intermediate solutions are combined to deliver the final solution. If all the threads return the answer *equivalent*, then it means that the whole circuits under verification are equivalent. However, when at least one pair of outputs is proved *non-equivalent*, the *Model P* enables earlier termination to the verification process by exploring the solution space quickly. For instance, if a given thread proves that at least one pair of POs is non-equivalent, then that thread can report the input/output patterns and broadcast a stop signal to other threads.

The proposed approach lies closer to the middle of the spectrum between data sharing and data independence, enabling faster equivalence checking. This solution works like a divide-and-conquer approach to decompose a problem into smaller ones, enabling efficient parallel processing on multi-core platforms with shared memory. Moreover, since miters are completely independent to each other, the proposed approach can be easily extended to exploit the advantages of distributed computing in cloud.

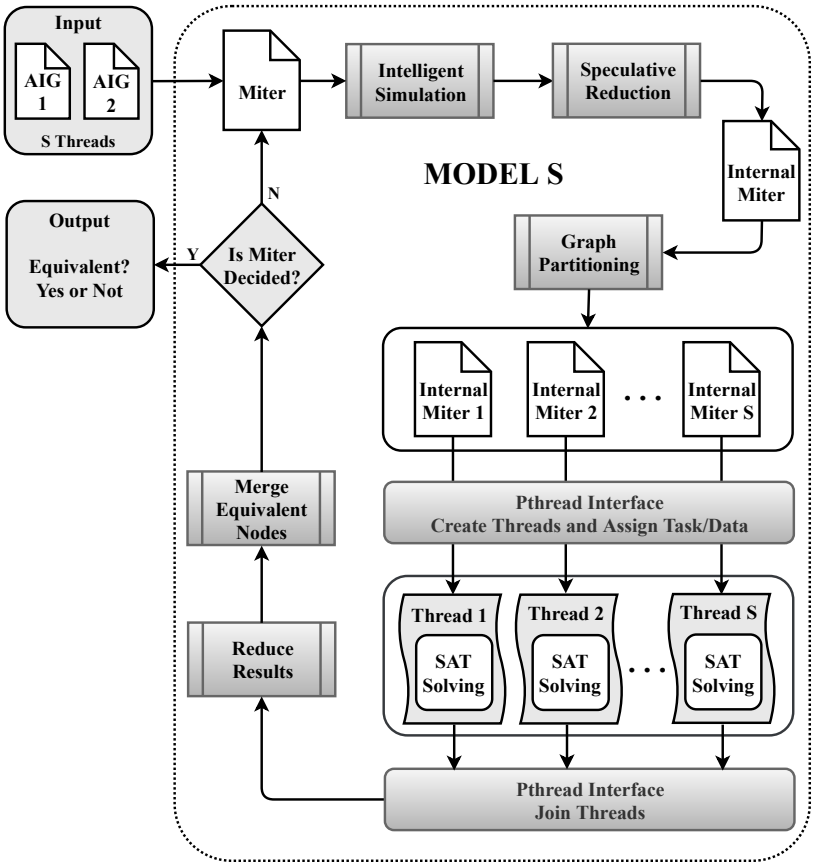


Figure 5.4: Scheme of parallel CEC by internal miter partitioning.

5.3.3 Internal Miter Partitioning

We are introducing the *Model S*, depicted in Fig. 5.4, to speed up SAT sweeping while checking equivalences of internal nodes in the CEC core. The same principle of graph partitioning presented in Algorithm 5.1 can be applied to decompose the internal miter constructed during SAT sweeping. After applying internal miter partitioning, the *Pthreads* interface is used to launch S SAT solver calls for checking equivalences of internal nodes in parallel, as illustrated in Fig. 5.4. In this approach, each thread receives a partition of the internal miter and executes an independent instance of the Minisat (EéN; SöRENSSON, 2004) for checking the miter.

Notice that the proposed approach interleaves serial and parallel sections of the code at each iteration, as shown in Fig. 5.4. Therefore, before starting the next iteration of the CEC core loop, all threads must finish and return partial results. For each iteration, a set of SAT counter-examples is collected from several SAT calls to produce simulation vectors for the next iteration. Moreover, the pairs of nodes proved to be equivalent are collected to refine the main miter. Therefore, auxiliary routines are used for reducing

back all these partial results representing counter-examples and equivalent nodes. This way, the runtime of the latest thread defines the delay before moving to the next iteration of CEC core.

The amount of SAT problems encoded in the internal miter and the complexity of these problems are strongly dependent on the circuit structure. Typically, the main miter comprises several complex SAT problems whereas the internal miter comprises a lot of small and easy SAT subproblems. Considering this scenario, it is a challenge to find a good trade-off between data sharing and data independence of the internal miter to perform a partitioning that leads to a good workload balance among the threads.

It is worth to mention that several other applications of SAT sweeping can be benefited from the proposed *Model S*. In optimization problems based on SAT sweeping, such as redundancy removal and signal correspondence (MISHCHENKO et al., 2011), it is not wise to partition the input graph because optimization opportunities tend to be lost due to the negative bias of partition boundaries. Therefore, in such optimization problems, the proposed *Model P* is less useful. On the other hand, the proposed *Model S* can enable parallelism to solve internal problems encoded in the temporary miter, without adversely impacting the quality of results of such optimization problems.

5.3.4 Combined Miter Partitioning

We have introduced the models P and S for speeding up equivalence checking by exploiting parallelism on different levels of the CEC engine. A strong feature of our approach is that both models can work together, cooperating to improve the CEC runtime. It means that each thread created in *Model P* can create a set of subthreads in *Model S* to speedup the computation. The amount of threads working at each model can be specified by the parameters P and S. This way, based on the circuit characteristics, one can fine tune the engine to get the best thread workload balance.

Since the amount of parallelism available in the main miter and the internal miter partitioning processes is strongly related to the circuit structure, the ability to run *Model P* and *Model S* together helps to deal with a wide range of design characteristics. It has been confirmed in our experiments where the combined execution of models P and S achieves higher speedups, leading to the best results for some benchmarks. Moreover, when considering massively parallel environments used in cloud computing, both models allow for a synergistic integration to fully exploit the power of distributed and shared

memory computing. For instance, *Model P* can be used to partition the initial problem and distribute tasks to many nodes in the cloud whereas *Model S* can exploit the potential of multi-core architectures at each computing node.

5.4 Experimental Results

The proposed methods have been implemented in C programming language using *Pthreads* and incorporated in the ABC command *&cec* (Berkeley Logic Synthesis and Verification Group,). The three ways of parallelizing CEC proposed in Section III can be enabled in the *&cec* command using the switches $-P$ and $-S$, together or separately, followed by the desired number of threads applied in each model. In the following experiment, the proposed parallel CEC has been compared to a commercial verification tool. In the sequence, we present a scaling analysis of the proposed models in a massive parallel environment when comparing to the reference single-threaded method available through the ABC command *&cec*.

5.4.1 Benchmarks

Since this work is focusing on speeding up CEC for large circuits, we have selected the MtM AIG node circuits from EPFL suite (AMARÚ; GAILLARDON; MICHELI, 2015) and a subset of six out of ten circuits obtained by applying the ABC command *double* 10x, as presented in the previous chapters of this work. The four largest circuits used in the experiments of the previous chapters were not considered herein due to the extreme high runtime needed for several verification runs on such large circuits. However, all the circuits considered in our experiments comprise AIGs with millions of nodes being large enough to present challenges for CEC engines. Table 5.1 presents a refresh on characteristics of the MtM node AIGs from EPFL benchmark suite as well as the circuits derived by applying *double* command.

In a typical scenario, CEC is used to check equivalence between original and optimized versions of the same circuit after logic synthesis and/or technology-dependent optimization. To reproduce this scenario, we have first applied the script *dc2* available in ABC to the circuits shown in Table 5.1. The script performs area-driven, delay-constrained, multi-level optimization process using *rewriting*, *balancing* and *refactoring* algorithms. In the following experiments, we have checked the equivalence between the original circuit and the *dc2*-optimized ones using a commercial verification tool, the original single-threaded version of the ABC command *&cec* and the proposed models for parallel CEC.

Table 5.1: The set of six circuits obtained by applying the ABC command *double* 10x and the three MtM AIG nodes circuits from the EPFL benchmark suite.

Circuit	# PI	# PO	# AND2	# Levels
sin_10xd	24,576	25,600	5,545,984	225
arbiter_10xd	262,144	132,096	12,123,136	87
voter_10xd	1,025,024	1,024	14,088,192	70
square_10xd	65,536	131,072	18,927,616	250
sqrt_10xd	131,072	65,536	25,208,832	5,058
mult_10xd	131,072	131,072	27,711,488	274
sixteen	117	50	16,216,836	140
twenty	137	60	20,732,893	162
twentythree	153	68	23,339,737	176

5.4.2 Comparing to Commercial Verification Tool

In this experiment we are comparing the proposed models for parallel CEC to a commercial verification tool. The results were collected in a server with 64GB of shared RAM and a processor Intel®Core®i7-7700K CPU at 4.20GHz, where the processor has four physical cores. The tools under comparison were executed in a 64-bit Linux distribution and the runtimes were measured using the Linux bash command *time* (real).

Both the commercial tool and the proposed models were executed using four threads to exploit the potential of the four physical cores available on the server. We have executed Model P and Model S separately and combined. Table 5.2 presents the absolute runtimes for each method in the format ($hours : minutes : seconds$). The experimental results have demonstrated that the proposed models for parallel CEC present significant smaller runtimes than the commercial verification tool when running at the same number of threads. Notice that, for several cases, the commercial tool took more than a day for verifying the circuits, whereas the proposed approach took only few minutes/hours. The last column of the Table 5.2 presents the best speedups provided by the proposed approach when comparing to the commercial tool. Moreover, a detailed view on the speedups provided by each configuration of the proposed approach is presented in Fig. 5.5. The "voter_10xd" circuit was remove from this plot since it does not presented speedups.

Table 5.2: Runtime comparison among the commercial tool and the proposed models running at four threads, in (h:m:s).

Circuit	Commercial (4 threads)	-P 4	-S 4	-P 2 -S 2	Speedup
sin_10xd	1:02:09	**0:07:09**	0:11:15	0:08:29	8.68x
arbiter_10xd	0:51:53	**0:00:51**	0:01:36	0:01:06	60.56x
voter_10xd	**1:19:37**	4:19:08	4:54:53	4:09:05	0.32x
square_10xd	2:57:02	**0:03:28**	0:07:45	0:04:52	51.13x
sqrt_10xd	69:03:25	**3:31:50**	6:47:06	6:38:47	19.56x
mult_10xd	5:20:41	**0:09:04**	0:20:15	0:12:46	35.34x
sixteen	21:21:26	4:22:54	3:34:31	**1:45:06**	12.19x
twenty	38:08:40	3:35:22	6:27:38	**2:19:40**	16.39x
twentythree	49:46:06	3:56:46	8:11:20	**2:52:52**	17.27x

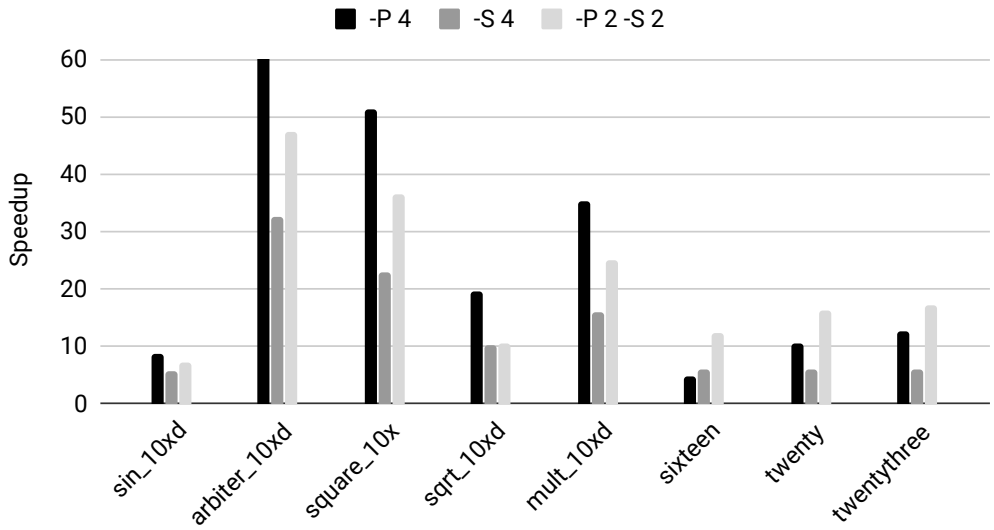


Figure 5.5: Speedups of the proposed models when compared to a commercial verification tool.

5.4.3 Parallel CEC Scalability

In order to asses the scalability of the proposed parallel models to many threads, a set of experiments has been carried out using a server with 128GB of shared RAM and four processors Intel®Xeon®CPU E7- 4860 operating at 2.27GHz, where each processor has 10 physical cores, corresponding to the total of 40 cores. ABC tool was compiled using GNU g++ version 6.1.0 and executed in a 64-bit Linux distribution. The runtimes were measured using the Linux bash command *time* (real). Unfortunately, we do not have a commercial tool available on this server with greater computing power. Therefore, in the following experiments, we are comparing the proposed parallel CEC approach to the state-of-the-art serial CEC implemented in ABC Command &*cec*.

In the first experiment, we compare the proposed models P and S separately. The goal of this experiment is to measure the speedup and the scalability that each model can bring to the verification process as the thread count increases. Table 5.3 presents the runtime for the original single-thread CEC and for parallel models P and S running at 4 up to 40 threads. The speedup introduced by each model according to the thread count is shown in Fig. 5.6 and in Fig. 5.7.

Regarding the benefits of the proposed approaches, *Model P* leads to more significant improvement than *Model S*. For most circuits, *Model P* works 30x faster than the single-thread CEC. Moreover, for the circuits "twenty" and "twentythree", we have observed super-linear speedups of 60.73x and 63.08x, respectively. In these cases, graph partitioning enabled a good balance between data sharing and data independence. Besides that, the CEC engine is based on incremental SAT solving by sharing clauses among successive SAT calls (MISHCHENKO et al., 2006; EéN; SöRENSSON, 2004). Therefore, the miter partitioning can lead to a better incremental behavior in the SAT solving heuristics, since each thread is applied to a smaller set of problems using separate SAT solvers.

It is harder to get high speedups in *Model S* because many smaller SAT problems are created in the interleaved sections of sequential and parallel codes in the CEC core. Actually, the SAT problem size and complexity depends on the characteristics of the circuits under verification. Therefore, one should not discard *Model S* since it can be advantageous for certain designs, and it can also be combined to work together with *Model P*.

Table 5.3: Runtime comparison among the original ABC command &*cec* and the proposed models P and S running separately, in (h:m:s).

Circuit	ABC &*cec*	-P 4	-S 4	-P 10	-S 10	-P 20	-S 20	-P 30	-S 30	-P 40	-S 40
sin_10xd	1:04:52	0:15:33	0:23:56	0:06:28	0:16:27	0:03:09	0:12:55	0:02:12	0:11:48	**0:02:05**	0:12:14
arbiter_10xd	0:05:17	0:01:35	0:03:36	**0:01:18**	0:03:18	0:01:38	0:03:16	0:02:08	0:03:22	0:02:41	0:03:28
voter_10xd	24:13:00	5:56:16	7:34:12	2:51:05	3:25:12	1:17:48	1:46:06	0:50:16	1:12:14	**0:39:12**	1:02:42
square_10xd	0:33:53	0:07:58	0:17:28	0:03:23	0:14:23	0:01:56	0:13:02	0:01:28	0:12:41	**0:01:16**	0:12:56
sqrt_10xd	21:34:04	7:45:45	14:17:51	4:05:54	12:57:21	2:18:04	12:19:09	1:34:57	12:09:34	**1:21:22**	12:10:35
mult_10xd	1:21:10	0:19:37	0:43:36	0:08:27	0:36:30	0:04:35	0:33:28	0:03:25	0:32:42	**0:03:03**	0:33:26
sixteen	8:27:33	7:07:47	7:17:48	1:43:48	7:49:08	0:29:03	7:08:11	0:15:21	7:10:25	**0:13:28**	7:24:37
twenty	14:35:59	4:14:36	14:53:13	3:48:59	13:39:38	0:27:43	14:32:43	0:15:27	13:55:20	**0:14:25**	13:50:52
twentythree	18:51:30	5:08:32	17:50:37	2:01:59	17:09:33	0:50:28	18:41:32	0:33:01	18:46:43	**0:17:56**	17:03:48

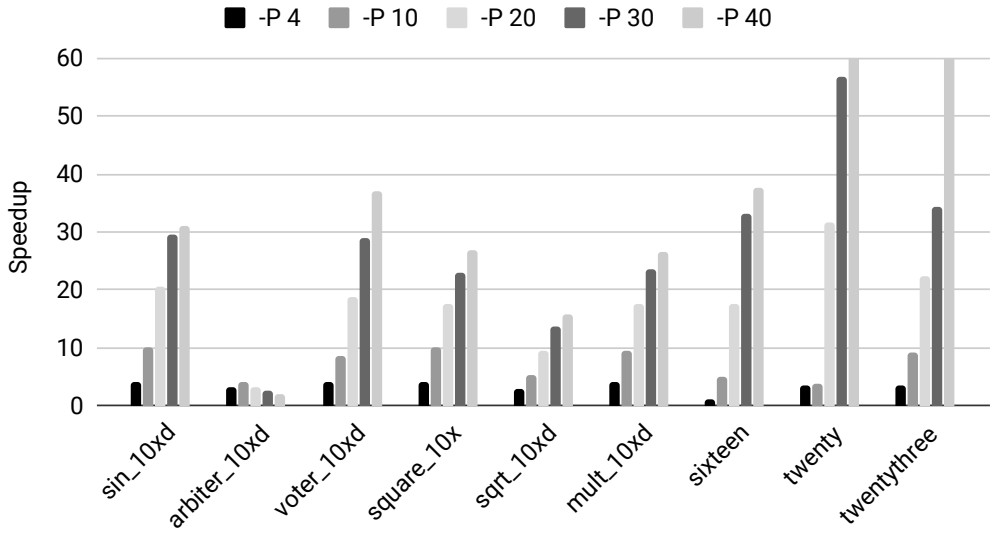


Figure 5.6: Speedups by the main miter partitioning (*Model P*).

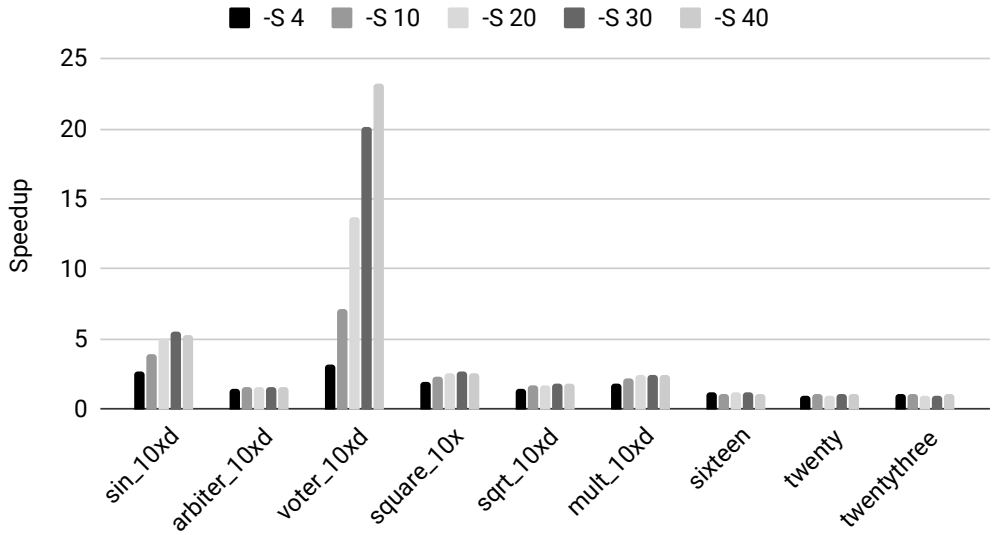


Figure 5.7: Speedups by the internal miter partitioning (*Model S*).

By combining models P and S, we enable extra opportunities to trade-off data sharing and data independence in the CEC engine. In this experiment, we consider three different thread configurations for combining both models using the total of 40 threads. In the first configuration ($-P$ 4 $-S$ 10), we are considering fewer partitions of the main miter and more partitions of the internal miter. In the second configuration ($-P$ 10 $-S$ 4), we have swapped the thread count between the main miter and the internal miter partitionings. In the last configuration ($-P$ 20 $-S$ 2), we have increased the thread count for the main miter partitioning since it has presented the best results in previous experiments. The absolute runtime values of each configuration are presented in Table 5.4 and the respective speedups are shown in Fig. 5.8.

To demonstrate the advantages of combining both models, consider the results for circuits "voter_10xd" and "arbiter_10xd" shown in Fig. 5.8. The configuration ($-P$ 10 $-S$ 4) resulted in extra speedups for both circuits when comparing to the models P and S running independently. For "arbiter_10xd", we observed the speedup of 6.7x whereas in the previous experiments the speedups were 4x for *Model P* and 1.5x for *Model S*. Moreover, for "voter_10xd" circuit, we observed the speedup of 39x which is practically linear (optimal) in the number of threads. In the previous experiments, the speedups for "voter_10xd" were 37x and 23x for models P and S, respectively.

Table 5.4: Runtime comparison between original ABC command &*cec* and the combined models P and S running at 40 threads, in (h:m:s).

Circuit	ABC &*cec*	-P 4 -S 10	-P 10 -S 4	-P 20 -S 2
sin_10xd	1:04:52	0:03:55	0:02:32	**0:02:04**
arbiter_10xd	0:05:17	0:01:03	**0:00:47**	0:01:20
voter_10xd	24:13:00	0:39:45	**0:37:05**	0:37:26
square_10xd	0:33:53	0:03:21	0:01:54	**0:01:27**
sqrt_10xd	21:34:04	5:33:28	2:38:45	**1:44:48**
mult_10xd	1:21:10	0:09:09	0:04:51	**0:03:36**
sixteen	8:27:33	1:54:34	1:14:02	**0:28:07**
twenty	14:35:59	2:17:06	1:42:43	**0:27:01**
twentythree	18:51:30	2:21:32	2:22:09	**1:06:56**

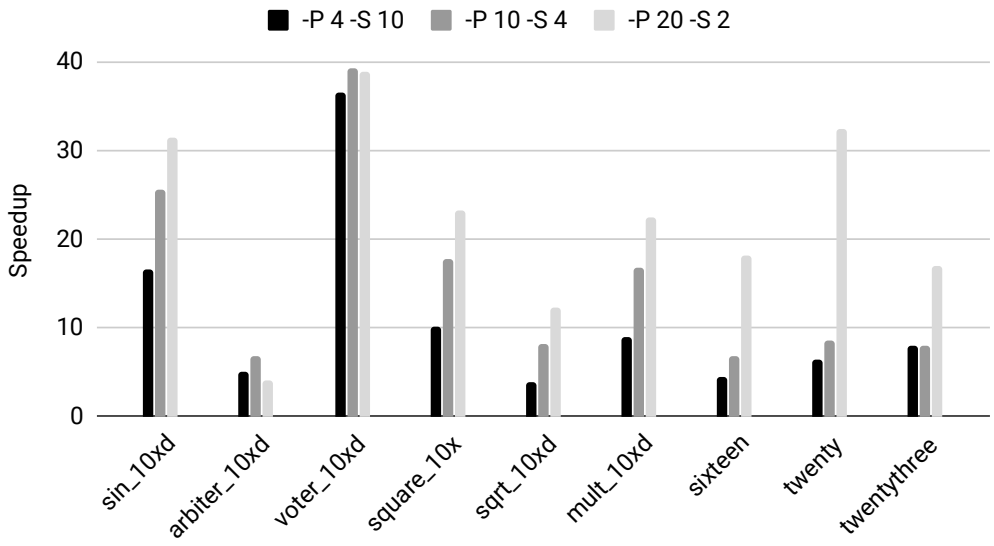


Figure 5.8: Speedups by combining models P and S at 40 threads.

5.4.4 Discussion

Table 5.5 presents a summary of the best results and the respective thread configuration. In general, the configurations $-P$ 40 and $-P$ 10 $-S$ 4 lead to the best results. Notice that the proposed approach can significantly reduce the runtime of CEC when comparing to the single-thread method. One great improvement has taken place when the CEC runtime went down from 24h13min to only 37min and from 18h51min to only 18min. Overall, the proposed solutions have the potential to improve existing EDA environments.

Proposed models are based on simple principles making them easy to reproduce and deploy in a standard EDA flow. Therefore, several other important tasks that depend on CEC, can be benefited from the speedup enabled by the proposed scalable solution. Moreover, with additional customization to perform proper handling of fanout nodes during equivalence checking the proposed CEC is also applicable in: (i) SAT sweeping under observability *don't-cares* (ZHU et al., 2006); (ii) node minimization with satisfiability, observability and external *don't-cares* (MISHCHENKO; BRAYTON, 2005); (iii) high-effort resynthesis for circuit delay, area, power dissipation and wiring congestion reduction using Boolean resubstitution (MISHCHENKO et al., 2011); (iv) various traversal-based computations comparing functions of the nodes in terms of primary inputs under observability conditions (ATPG, redundancy removal, false path detection and removal), and others.

Table 5.5: Summary of the best results and configurations for each circuit, in (h:m:s).

Circuit	ABC *&cec*	Parallel	Speedup	Config.
sin_10xd	1:04:52	0:02:04	31.33x	-*P* 20 -*S* 2
arbiter_10xd	0:05:17	0:00:47	6.76x	-*P* 10 -*S* 4
voter_10xd	24:13:00	0:37:05	39.17x	-*P* 10 -*S* 4
square_10xd	0:33:53	0:01:16	26.72x	-*P* 40
sqrt_10xd	21:34:04	1:21:22	15.90x	-*P* 40
mult_10xd	1:21:10	0:03:03	26.59x	-*P* 40
sixteen	8:27:33	0:13:28	37.67x	-*P* 40
twenty	14:35:59	0:14:25	60.73x	-*P* 40
twentythree	18:51:30	0:17:56	63.08x	-*P* 40

Regarding the previous parallel CEC method EQUIPE, it is quite difficult to perform a direct comparison to such a method (CHATTERJEE; BERTACCO, 2010). At first, the methods are running on different platforms with the technology gap of almost a decade. Besides, it is not clear whether the authors are comparing against *cec* or *&cec* command, being that *&cec* is significantly faster than *cec*. Moreover, it is hard to ensure that we are using exactly the same pairs of original and optimized circuits as input to CEC. Therefore, in order to avoid an unfair comparison, we are presenting an analysis based on our solution and on the general information presented in the EQUIPE paper.

The results produced by the EQUIPE method were collected on a platform containing a CUDA-enabled 8800GT GPU with 14 multiprocessors operating at 600 MHz and a CPU Intel Core 2 Quad operating at 2.4 GHz. The authors reported average speedups of one order of magnitude when comparing to a commercial tool and only up to 3.22x speedup when comparing to ABC tool. In the latter case, even using the 4 CPU cores and the 14 auxiliary GPU cores, the method was not able to speedup CEC beyond 3.22x. It is worth to notice that the CPU and GPU cores work at different frequencies and there is an additional cost related to the communication between CPU and GPU. Moreover, the authors mention that in some cases the internal miters need to be reconstructed, leading to runtime degradation, as shown in (CHATTERJEE; BERTACCO, 2010). On the other hand, the parallel models P and S proposed in this work are able to verify circuits with millions of AIG nodes. It scales to many cores and achieves significant improvement when comparing to *&cec* command, the latest CEC engine in ABC tool. Therefore, the proposed approach is more promising than the EQUIPE method when it comes for improving CEC for large designs.

5.5 Summary

In this chapter we proposed a novel approach comprising three different models to enable parallelism and to speedup modern CEC engine. The models trade off data sharing and data independence by applying graph partitioning during mitering and SAT sweeping. Experiments lead to promising results in speeding up the verification task for large designs. In several cases, where the ABC or the commercial verification tool took more than one day for checking circuits, the proposed approach has finished this task in a few minutes/hours. It is an interesting contribution since CEC is known to be a co-NP-complete problem and even the advance techniques used in modern CEC engines have an exponential time complexity, being sensitive to break down for large designs. Moreover, the proposed models can be easily adapted to exploit massively parallel environments of cloud computing.

The proposed solution has additional practical benefits, in particular, the potential to improve the runtime and scalability of other applications in current EDA environments. It should be noted that scalable CEC techniques are used as an important building block in other applications, which depend on efficient computation of equivalence classes of internal nodes. Given so many uses of the scalable CEC, parallelizing it as proposed in this work will help to improve a number of other important applications in a typical EDA flow.

6 CONCLUSIONS

This thesis addressed three challenging problems from the logic synthesis and verification fields. This work presents investigations and contributions across the multi-level logic optimization, LUT-based technology mapping and combinational equivalence checking problems. The problems addressed in this work are not trivially parallelizable, since advanced algorithms rely on complex graph-based data structures and SAT solving. Therefore, we adopted the most promising parallelization strategy to get speedups and scalability according to the problem/solution characteristics.

We carefully dissected these problems and rethought related algorithms and data structures by introducing novel techniques to unlock the parallelism in the core of logic synthesis and verification algorithms. The main challenge in this direction is to rethink shared-data structures and routines to fit and to work in a parallel environment. We conclude that, it is wise to consider the following list of principles for enabling scalability to parallel algorithms, always it is possible:

- Avoiding to insert extra synchronization based on logical locks;
- Rethinking parallel algorithms and data structures to work in a lock-free way;
- Making shared data structure read-only and writing into thread-local structures;
- Managing memory allocation to avoid bottlenecks and thread conflicts;
- Exploring data sharing and data independence trade-offs;
- Choosing the parallelization strategy based on intrinsic properties of the problem.

All the parallel-aware algorithms and data structures proposed in this work employed these principles to achieve the significant speedups supported by experimental results.

6.1 Summary of Contributions

The main contributions and impact of this thesis are summarized in the following:

- A scalable parallel AIG rewriting which is able to optimize AIGs with millions of nodes in few seconds. It is the first work in the literature exploiting fine-gain parallelism for AIG rewriting, which is a standard technique for both academy and industry. This work introduced techniques to unlock parallelism in well established logic synthesis algorithms with potential to impact in novel parallel algorithms.

We have demonstrated significant runtime improvements without QoR degradation when compared to the stated-of-the-art method.

- A parallel LUT-based structural mapper that scales to many threads and has potential for a deep integration to our parallel AIG rewriting as well as to other logic synthesis algorithms. We conjecture that the fully integration of this parallel methods can mitigate the gap between multi-level and technology mapping by making rewriting decisions based on mapping costs in a.
- A parallel CEC engine which provides three different levels for exploiting parallelism during design verification. The proposed approach improves the scalability of the state-of-the-art CEC engine from ABC tool, making the verification of designs with millions of AIG nodes expressively faster. The proposed parallel CEC is able to reduce the verification time from more than one day to few minutes, demonstrating potential for improving productivity in a real design flow. There are several applications which can be benefited from the proposed parallel and scalable CEC, since checking for logic equivalence is an often problem in different domains.
- The majority contributions of this work can be directly applied to both ASIC and FPGA design flows. The proposed parallel AIG rewriting and parallel CEC rely on technology independent optimizations and verification, respectively. Therefore, these two methods can be directly applied to both design flows.
- It is worth to mention that all the proposed solutions are suitable to be deployed in a cloud computing service with some reformulations to exploit the benefits from the cloud technology and infrastructure. Since there is an increasing migration of EDA software to the cloud, the solutions proposed in this thesis can bring insights and directions on how to parallelize logic synthesis and verification algorithms in the cloud.
- We are bringing contributions to the logic synthesis and verification field by collaborating with two important opensource projects, the ABC tool and the Galois system. We have improved the CEC engine in ABC, which is an important component in this tool. Moreover, we designed an initial parallel logic synthesis package in Galois system. All the contributions proposed in this works will be integrated and distributed as part of ABC tool and Galois system in GitHub, allowing others to use and improve the proposed methods.

6.2 Open Problems and Future Work

The algorithms proposed in this work can be extended and improved by:

- Concluding the implementation of the proposed parallel functional mapper.
- Rethinking the proposed parallel technology mappers to a standard-cell design flow.
- Designing a deterministic version of proposed parallel AIG rewriting.
- Improving the algorithms performance by fine tuning the Galois data structures.
- Increasing the CEC speedups by exploiting duality for solving each partition.
- Adding one more level of parallelism to CEC by employing a parallel SAT solver.

REFERENCES

AMARÚ, L.; GAILLARDON, P.-E.; MICHELI, G. D. Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization. In: Proc. of Design Automation Conference - DAC'14. **Proceedings...** [S.l.: s.n.], 2014.

AMARÚ, L.; GAILLARDON, P.-E.; MICHELI, G. D. The EPFL Combinational Benchmark Suite. In: Proc. of International Workshop on Logic and Synthesis - IWLS'15. **Proceedings...** [S.l.: s.n.], 2015.

AMARÚ, L.; GAILLARDON, P. E.; MICHELI, G. D. Majority-inverter graph: A new paradigm for logic optimization. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 35, n. 5, p. 806–819, May 2016.

AMARÚ, L. et al. Enabling exact delay synthesis. In: Proc. of International Conference on Computer Aided Design - ICCAD'17. **Proceedings...** [S.l.: s.n.], 2017. p. 352–359.

AMARù, L. et al. Exploiting Circuit Duality to Speed up SAT. In: 2015 IEEE Computer Society Annual Symposium on VLSI. **Proceedings...** [S.l.: s.n.], 2015. p. 101–106.

AMDAHL, G. M. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: Proc. of the April 18-20, 1967, Spring Joint Computer Conference - AFIPS '67 (Spring). **Proceedings...** [S.l.: s.n.], 1967. p. 483–485.

Apache Giraph. Apache Giraph. In: . [s.n.]. Available from Internet: <http://giraph.apache.org>.

Apache Spark GraphX. Spark GraphX. In: . [s.n.]. Available from Internet: <https://spark.apache.org/docs/latest/graphx-programming-guide.html#property-operators>.

Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. In: . [s.n.]. Available from Internet: <http://www.eecs.berkeley.edu/~alanmi/abc/>.

BERTACCO, V.; DAMIANI, M. The disjunctive decomposition of logic functions. In: Proc. International Conference on Computer Aided Design - ICCAD'97. **Proceedings...** [S.l.: s.n.], 1997. p. 78–82.

BJESSE, P.; BORALV, A. DAG-Aware Circuit Compression for Formal Verification. In: Proc. of International Conference on Computer Aided Design - ICCAD'04. **Proceedings...** [S.l.: s.n.], 2004. p. 42–49.

BJESSE, P.; CLAESSEN, K. SAT-Based Verification Without State Space Traversal. In: Proc. of Formal Methods in Computer-Aided Design - FMCAD'00. **Proceedings...** [S.l.: s.n.], 2000. p. 372–389.

BRAND, D. Verification of large synthesized designs. In: International Conference on Computer-Aided Design - ICCAD. **Proceedings...** [S.l.: s.n.], 1993. p. 534–537.

BRAYTON P.C. MCGEER, J. S. A. S.-V. R. **A New Exact Minimizer for Two-level Logic Synthesis. In Logic Synthesis and Optimization**. [S.l.]: T. Sasao (Ed.). Kluwer Academic Publishers, Dordrecht, 1993. 1–31 p.

BRAYTON, R.; HACHTEL, G.; SANGIOVANNI-VINCENTELLI, A. Multilvel Logic Synthesis. **Proc. IEEE**, v. 78, Feb 1990.

BRAYTON, R.; MCMULLEN, C. The Decomposition and Factorization of Boolean Expressions. In: Proc. of International Symposium on Circuits and Systems - ISCAS'82. **Proceedings...** [S.l.: s.n.], 1982. p. 29–54.

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.

BRYANT, R. Graph-Based Algorithms for Boolean Function Manipulation. **IEEE Transactions on Computers**, C-35, n. 8, p. 677–691, Aug 1986.

BUTENHOF, D. R. **Programming with POSIX Threads**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

BUTZEN, P. F. et al. Standby power consumption estimation by interacting leakage current mechanisms in nanoscaled cmos digital circuits. **Microelectronics Journal**, Elsevier, v. 41, n. 4, p. 247–255, 2010.

CHATTERJEE, D.; BERTACCO, V. EQUIPE: Parallel equivalence checking with GP-GPUs. In: Proc. of International Conference on Computer Design - ICCD'10. **Proceedings...** [S.l.: s.n.], 2010. p. 486–493.

CHATTERJEE, S. et al. Reducing Structural Bias in Technology Mapping. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 25, n. 12, p. 2894–2903, Dec 2006.

CHATTERJEE, S. et al. Reducing structural bias in technology mapping. **IEEE Trans. on Comput.-Aided Design of Integr. Circuits and Syst.**, v. 25, n. 12, 2006.

CHEN, D.; CONG, J.; PAN, P. FPGA Design Automation: A Survey. **Found. Trends Electron. Des. Autom.**, v. 1, n. 3, p. 139–169, jan. 2006. ISSN 1551-3939.

COFFMAN, E. G.; ELPHICK, M.; SHOSHANI, A. System Deadlocks. **ACM Comput. Surv.**, v. 3, n. 2, p. 67–78, jun. 1971.

CONG, J.; DING, Y. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. **IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems**, v. 13, n. 1, 1994.

CONG, J.; DING, Y. On area/depth trade-off in LUT-based FPGA technology mapping. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 2, n. 2, 1994.

CONG, J.; WU, C.; DING, Y. Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution. In: Proc. of International Symposium on Field-Programmable Gate Arrays - FPGA'99. **Proceedings...** [S.l.: s.n.], 1999.

CORTADELLA, J. Timing-driven logic bi-decomposition. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 22, n. 6, p. 675–685, 2003.

DA ROSA JR, L. S. et al. A comparative study of cmos gates with minimum transistor stacks. In: ACM. PROCEEDINGS OF THE 20TH ANNUAL CONFERENCE ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN. **Proceedings...** [S.l.], 2007. p. 93–98.

DA ROSA JR, L. S. et al. Scheduling policy costs on a java microcontroller. In: SPRINGER. ON THE MOVE TO MEANINGFUL INTERNET SYSTEMS 2003: OTM 2003 WORKSHOPS. **Proceedings...** [S.l.], 2003. p. 520–533.

DAGUM, L.; MENON, R. OpenMP: an industry standard API for shared-memory programming. **IEEE Computational Science and Engineering**, v. 5, n. 1, p. 46–55, Jan 1998.

EIJKHOUT, V. **Introduction to High Performance Scientific Computing**. [S.l.]: Lulu, 2014.

ELBAYOUMI, M. et al. TACUE: A timing-aware cuts enumeration algorithm for parallel synthesis. In: Proc. of Design Automation Conference - DAC'14. **Proceedings...** [S.l.: s.n.], 2014.

EéN, N.; SöRENSSON, N. An Extensible SAT-solver. In: E. Giun-chiglia and A. Tacchella (Eds.): SAT 2003, LNCS 2919. **Proceedings...** [S.l.: s.n.], 2004. p. 502–518.

GRAMA, A.; GUPTA, G. K. A.; KUMAR, V. **Introduction to parallel computing**. [S.l.]: Pearson Education, 2003.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI: portable parallel programming with the message-passing interface**. [S.l.]: MIT press, 1999.

HAASWIJK, W. et al. A novel basis for logic rewriting. In: Proc. of Asia and South Pacific Design Automation Conference - ASP-DAC'17. **Proceedings...** [S.l.: s.n.], 2017. p. 151–156.

HASSOUN, S.; SASAO, T. **Logic Synthesis and Verification**. [S.l.]: Springer US, 2002.

HEULE, M. J. H. et al. Cube-and-conquer for satisfiability. In: _____. HANDBOOK OF PARALLEL CONSTRAINT REASONING. **Proceedings...** Cham: Springer International Publishing, 2018. p. 31–59.

HWANG, K.; JOTWANI, N. **Advanced Computer Architecture, 3e**. [S.l.]: McGraw-Hill Education, 2011.

ISS Group, The University of Texas at Austin. Galois System. In: . [s.n.]. Available from Internet: <http://iss.ices.utexas.edu/projects/galois/api/current/index.html>.

JQUINE, W. V. A Way to Simplify Truth Functions. **The American Mathematical Monthly**, v. 62, n. 9, p. 627–631, Nov 1955.

KENNINGS, A.; RAVISHANKAR, C. Parallel FPGA technology mapping using multi-core architectures. In: Proc. of Canadian Conference on Electrical and Computer Engineering - CCECE'11. **Proceedings...** [S.l.: s.n.], 2011. p. 274–279.

KRISHNASWAMY, S. et al. DeltaSyn: An efficient logic difference optimizer for ECO synthesis. In: Proc. of International Conference on Computer Aided Design - ICCAD'09. **Proceedings...** [S.l.: s.n.], 2009. p. 789–796.

KUEHLMANN, A. Dynamic transition relation simplification for bounded property checking. In: Proc. of International Conference on Computer Aided Design - ICCAD'04.. **Proceedings...** [S.l.: s.n.], 2004. p. 50–57.

KUEHLMANN, A. et al. Robust Boolean reasoning for equivalence checking and functional property verification. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 21, n. 12, p. 1377–1394, Dec 2002.

LEHMAN, E. et al. Logic decomposition during technology mapping. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 16, n. 8, p. 813–834, 1997.

LENHARTH, A.; NGUYEN, D.; PINGALI, K. Parallel graph analytics. **Communications of the ACM**, v. 59, n. 5, 2016.

LENHARTH, A.; PINGALI, K. Scaling Runtimes for Irregular Algorithms to Large-Scale NUMA Systems. **Computer**, v. 48, n. 8, p. 35–44, 2015.

LI, N.; DUBROVA, E. AIG rewriting using 5-input cuts. In: Proc. of Iternational Conference on Computer Design - ICCD'11. **Proceedings...** [S.l.: s.n.], 2011. p. 429–430.

LIU, G.; ZHANG, Z. A Parallelized Iterative Improvement Approach to Area Optimization for LUT-Based Technology Mapping. In: Proc. of International Symposium on Field-Programmable Gate Arrays - FPGA'17. **Proceedings...** [S.l.: s.n.], 2017. p. 147–156.

LU, F. et al. A circuit SAT solver with signal correlation guided learning. In: Proc. of Design, Automation and Test in Europe - DATE'03. **Proceedings...** [S.l.: s.n.], 2003. p. 892–897.

MACHADO, L.; CORTADELLA, J. Support-Reducing Decomposition for FPGA Mapping. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, 2018.

MACHADO, L. et al. Kl-cut based digital circuit remapping. In: IEEE. NORCHIP, 2012. **Proceedings...** [S.l.], 2012. p. 1–4.

MALEWICZ, G. et al. Pregel: A System for Large-scale Graph Processing. In: Proc. of International Conference on Management of Data - SIGMOD '10. **Proceedings...** [S.l.: s.n.], 2010. p. 135–146.

MANOHARARAJAH, V.; BROWN, S. D.; VRANESIC, Z. G. Heuristics for Area Minimization in LUT-Based FPGA Technology Mapping. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 25, n. 11, p. 2331–2340, Nov 2006.

MARTINS, M. G.; RIBAS, R. P.; REIS, A. I. Functional composition: A new paradigm for performing logic synthesis. In: IEEE. QUALITY ELECTRONIC DESIGN (ISQED), 2012 13TH INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.], 2012. p. 236–242.

MATSUNAGA, Y. An efficient equivalence checker for combinational circuits. In: Proc. of Design Automation Conference - DAC'96. **Proceedings...** [S.l.: s.n.], 1996. p. 629–634.

MCCLUSKEY, E. J. Minimization of Boolean functions. **The Bell System Technical Journal**, v. 35, n. 6, p. 1417–1444, Nov 1956.

MICHELI, G. D. **Synthesis and Optimization of Digital Circuits**. 1st. ed. [S.l.]: McGraw-Hill Higher Education, 1994.

MISHCHENKO, A.; BRAYTON, R. SAT-based complete don't-care computation for network optimization. In: Proc. of Design, Automation and Test in Europe - DATE'05. **Proceedings...** [S.l.: s.n.], 2005. p. 412–417.

MISHCHENKO, A. et al. Scalable Don't-care-based Logic Optimization and Resynthesis. **ACM Transactions on Reconfigurable Technology and Systems - TRETS**, v. 4, n. 4, p. 34:1–34:23, dec. 2011.

MISHCHENKO, A.; BRAYTON, R. K. Scalable logic synthesis using a simple circuit structure. In: Proc. of International Workshop on Logic and Synthesis - IWLS'06. **Proceedings...** [S.l.: s.n.], 2006.

MISHCHENKO, A. et al. Scalable and scalably-verifiable sequential synthesis. In: Proc. of International Conference on Computer Aided Design - ICCAD'08. **Proceedings...** [S.l.: s.n.], 2008. p. 234–241.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In: Proc. of Design Automation Conference - DAC'06. **Proceedings...** [S.l.: s.n.], 2006. p. 532–535.

MISHCHENKO, A. et al. Improvements to Combinational Equivalence Checking. In: Proc. of International Conference on Computer Aided Design - ICCAD'06. **Proceedings...** [S.l.: s.n.], 2006. p. 836–843.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. K. Improvements to technology mapping for LUT-based FPGAs. **IEEE Trans. on Comput.-Aided Design of Integr. Circuits and Syst.**, v. 26, n. 2, 2007.

MISHCHENKO, A. et al. **FRAIGs: A unifying representation for logic synthesis and verification**. [S.l.], 2005.

MISHCHENKO, A. et al. Combinational and sequential mapping with priority cuts. In: Proc. of International Conference on Computer Aided Design - ICCAD'07. **Proceedings...** [S.l.: s.n.], 2007.

MOCTAR, Y. O. M.; BRISK, P. Parallel FPGA routing based on the operator formulation. In: Proc. of Design Automation Conference - DAC'14. **Proceedings...** [S.l.: s.n.], 2014. p. 1–6.

MOON, I.-H.; PIXLEY, C. Non-miter-based Combinational Equivalence Checking by Comparing BDDs with Different Variable Orders. In: Proc. of Formal Methods in Computer-Aided Design - FMCAD'04. **Proceedings...** [S.l.: s.n.], 2004. p. 144–158.

MOONDANOS, J. et al. CLEVER: Divide and Conquer Combinational Logic Equivalence VERification with False Negative Elimination. In: Proc. of International Conference on Computer Aided Verification - CAV'01. **Proceedings...** [S.l.: s.n.], 2001. p. 131–143.

MOREIRA, M. et al. Semi-custom ncl design with commercial eda frameworks: Is it possible? In: INTERNATIONAL SYMPOSIUM ON ASYNCHRONOUS CIRCUITS AND SYSTEMS (ASYNC), POTSDAM. **Proceedings...** [S.l.: s.n.], 2014.

MVSIS Group, UC Berkeley. MVSIS: Multi-Valued Logic Synthesis System. In: . [s.n.]. Available from Internet: <http://www-cad.eecs.berkeley.edu/mvsis/>.

NEUTZLING, A. et al. Synthesis of threshold logic gates to nanoelectronics. In: IEEE. 2013 26TH SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). **Proceedings...** [S.l.], 2013. p. 1–6.

NEUTZLING, A. et al. Threshold logic synthesis based on cut pruning. In: IEEE PRESS. PROCEEDINGS OF THE IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.], 2015. p. 494–499.

OpenMP. OpenMP. In: . [s.n.], 1997. Available from Internet: <https://www.openmp.org>.

PAN, P.; LIN, C.-C. A New Retiming-based Technology Mapping Algorithm for LUT-based FPGAs. In: Proc. of International Symposium on Field Programmable Gate Arrays - FPGA '98. **Proceedings...** [S.l.: s.n.], 1998. p. 35–42.

PINGALI, K. et al. **Amorphous data-parallelism in irregular algorithms**. [S.l.], 2009.

PINGALI, K. et al. The tao of parallelism in algorithms. In: ACM Sigplan Notices. **Proceedings...** [S.l.: s.n.], 2011. v. 46, n. 6.

POSSANI, V. et al. Unlocking fine-grain parallelism for aig rewriting. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.: s.n.], 2018. (ICCAD '18), p. 87:1–87:8.

POSSANI, V. N. et al. Graph-based transistor network generation method for supergate design. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 24, n. 2, p. 692–705, Feb 2016.

POSSANI, V. N. et al. Transistor count optimization in ig finfet network design. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 36, n. 9, p. 1483–1496, Sep. 2017.

PUTNAM, A. et al. A reconfigurable fabric for accelerating large-scale datacenter services. **SIGARCH Comput. Archit. News**, ACM, v. 42, n. 3, p. 13–24, jun. 2014.

REINDERS, J. **Intel Threading Building Blocks**. First. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

RUDELL, R. L.; SANGIOVANNI-VINCENTELLI, A. Multiple-Valued Minimization for PLA Optimization. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 6, n. 5, p. 727–750, September 1987.

SANGIOVANNI-VINCENTELLI, A. L. M. P. S. J. B. R. ESPRESSO-SIGNATURE: A New Exact Minimizer for Logic Functions. In: . [S.l.: s.n.], 1993. p. 618–624.

SCHMITT, B.; MISHCHENKO, A.; BRAYTON, R. SAT-based area recovery in structural technology mapping. In: 23rd Asia and South Pacific Design Automation Conference (ASP-DAC). **Proceedings...** [S.l.: s.n.], 2018. p. 586–591.

SOEKEN, M. et al. Optimizing Majority-Inverter Graphs with functional hashing. In: Proc. of Design, Automation and Test in Europe - DATE'16. **Proceedings...** [S.l.: s.n.], 2016. p. 1030–1035.

STOK, L. Developing Parallel EDA Tools [The Last Byte]. **IEEE Design Test**, v. 30, n. 1, p. 65–66, 2013.

STOK, L. The Next 25 Years in EDA: A Cloudy Future? **IEEE Design & Test**, v. 31, n. 2, 2014.

STOK, L. **EDA 3.0: Implications to Logic Synthesis. In Advanced Logic Synthesis**. [S.l.]: Reis, André Inácio, Drechsler, Rolf (Eds.). Springer., 2018.

STOK, L.; IYER, M. A.; SULLIVAN, A. J. Wavefront Technology Mapping. In: Proc.s of Design, Automation and Test in Europe - DATA'99. **Proceedings...** [S.l.: s.n.], 1999.

TOGNI, J. et al. Automatic generation of digital cell libraries. In: IEEE. PROCEEDINGS. 15TH SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN. **Proceedings...** [S.l.], 2002. p. 265–270.

TSEITIN, G. S. **On the complexity of derivation in propositional calculus, Automation of reasoning**. [S.l.]: Springer Berlin Heidelberg, 1983.

WANG, L. T.; CHANG, Y. W.; CHENG, K. T. T. **Electronic design automation: synthesis, verification, and test**. [S.l.]: Elsevier. Morgan Kaufmann., 2009.

WILTGEN, A. et al. Power consumption analysis in static cmos gates. In: IEEE. 2013 26TH SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). **Proceedings...** [S.l.], 2013. p. 1–6.

XIN, R. S. et al. GraphX: A Resilient Distributed Graph System on Spark. In: International Workshop on Graph Data Management Experiences and Systems - GRADES '13. **Proceedings...** [S.l.: s.n.], 2013. p. 2:1–2:6.

YANG, S. **Logic Synthesis and Optimization Benchmarks User Guide: version 3.0**. [S.l.], 1991.

YANG, W.; WANG, L.; MISHCHENKO, A. Lazy man's logic synthesis. In: Proc. of International Conference on Computer Aided Design - ICCAD'12. **Proceedings...** [S.l.: s.n.], 2012. p. 597–604.

ZHU, Q. et al. SAT sweeping with local observability don't-cares. In: Proc. of Design Automation Conference - DAC'06. **Proceedings...** [S.l.: s.n.], 2006. p. 229–234.